SSL and TLS

Theory and Practice

Third Edition

Rolf Oppliger

SSL and TLS

Theory and Practice

Third Edition

SSL and TLS

Theory and Practice

Third Edition

Rolf Oppliger

ARTECH
HOUSE

BOSTON | LONDON
artechhouse.com

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library.

Cover design by Mark Bergeron

ISBN 13: 978-1-68569-015-1

10 9 8 7 6 5 4 3 2 1

To Lara

Contents

Foreword

The TLS protocol, and its forerunner SSL, is a cornerstone of internet security. TLS provides a secure communications facility, ensuring strong confidentiality and integrity guarantees for application layer data via a streaming interface that mimics TCP. Its security relies on strong cryptographic mechanisms being combined in the right way to build a secure protocol. TLS is used—usually invisibly—by billions of people and devices on a daily basis. It is the secure protocol of choice for many developers, since it is widely supported in cryptographic and networking libraries via simple interfaces and provides a fairly intuitive abstraction of what a "secure channel" should achieve. (Misinterpretations of this abstraction are of course possible, leading to insecure deployments of the protocol!)

TLS was always of interest to security researchers. But due to its growing importance, TLS was subjected to more intense scrutiny by the research community over the last decade or so. This activity uncovered significant weaknesses in earlier versions of the protocol, in its implementations, and in the supporting public key infrastructure. Notably, numerous vulnerabilities were found both in the TLS record protocol (which uses symmetric cryptographic algorithms to provide confidentiality and integrity for application data) and in the TLS handshake protocol (which perform entity authentication of protocol participants and establishes secure key material for use in the record protocol). Indeed, in the period 2011–2016, it felt like barely a month went by without some new vulnerability related to TLS being announced via a backronym, a logo, and a website. The practical impact of these vulnerabilities was and remains debatable. However, they collectively indicated a protocol that was starting to show signs of age and in need of a revamp.

At the same time, new protocols like Google's quick UDP internet connections (QUIC) began to emerge, with a superior performance profile compared to TLS. Specifically, QUIC offered a "Zero-Round-Trip-Time" (0-RTT) mode, allowing data to be securely transported from a client to a server in the first flow of communication. By contrast, TLS as standardized required two full communication round trips before the first client data could be sent securely (though extensions of TLS could reduce this to one round trip). In networking and in commercial applications of it like e-commerce, every millisecond of latency matters, so the superior performance of QUIC presented a challenge to TLS.

These two factors prompted the IETF to begin work on a new version of the protocol in early 2014. After 4 years, 28 drafts, and thousands of in-person hours of effort in design, analysis, implementation, and experimental deployment, TLS 1.3 finally emerged as RFC 8446 in August 2018. TLS 1.3 represents a major redesign of the protocol, while previous iterations were more incremental in their approach. TLS 1.3 strips out many legacy features and outmoded cryptographic algorithms while continuing to support the most important use cases. It improves overall security; for example, more of the handshake protocol is now encrypted, meaning less information is available for fingerprinting users. And it offers its own 0-RTT mode, competing with QUIC. An important aspect of the TLS 1.3 development process, and another contrast with previous design iterations, was the active involvement of the scientific community. This meant that security analysis was conducted on intermediate designs of the protocol using a diverse set of tools and by independent teams. The results of these analyses were in turn used to guide the further development of the protocol. This was something of a departure for the IETF, which had largely developed previous versions of the protocol without this two-way interaction. (This process was not perfect, in the sense that not all security and functionality requirements were apparent at the beginning of the process. Moreover, one vulnerability in preshared key modes of TLS 1.3 was missed by these analyses, the so-called Selfie attack.) Today, about 60% of websites support TLS 1.3, and all the major web browsers do too. As a result, most web traffic is now protected using TLS 1.3 (though other application domains where TLS is used probably lag behind). The transition to TLS 1.3 is well underway and is proceeding at a much faster pace than earlier protocol version transitions.

This third edition of Rolf Oppliger's book tells the complete story of TLS, from its earliest incarnation (SSL 1.0 in 1994), all the way up to and including TLS 1.3. It describes each version of the protocol in great depth, all the way down to the byte level, explaining why the protocol looked like it did, and why it now looks like it does. It gives perhaps the most accessible introduction to TLS 1.3 that is currently available. It also provides a similar level of detail on DTLS, a close sibling of TLS that is designed to operate over UDP instead of TCP. Stepping away from cryptographic protocol details, the book provides a broader context for TLS by discussing how TLS works with firewalls (and "network middleboxes" more generally). And it covers the key (!) topic of public key infrastructures and their role in securing TLS. Finally, the book gives an in-depth yet accessible treatment of many of the vulnerabilities in earlier versions of TLS referred to above.

Given its depth and breadth of coverage, this book will be an invaluable resource to practitioners and researchers alike. Its structured presentation of the historical evolution of TLS will also be of great value to scholars: After all, we can only properly understand where we are going by first appreciating from whence we have come. Readers of this book will surely enjoy the journey.

Kenny Paterson
Professor, ETH Zurich
June 2023

Preface

In theory, theory and practice are the same.
In practice, they are not.

— Albert Einstein

Theory is when you know everything and nothing works;
Practice is when everything works and nobody knows why;
Here we combine theory with practice: Nothing works and nobody knows why.

— Anonymous[1]

Terms like electronic commerce (e-commerce), electronic business (e-business), and electronic government (e-government) are omnipresent today. When people use these terms, they often refer to stringent security requirements that must be satisfied in one way or another. If they want to manifest that they are tech-savvy, then they bring in acronyms like SSL and TLS. Since SSL stands for *secure sockets layer* and TLS stands for *transport layer security*, it seems that adding SSL or TLS to applications automatically makes them secure and magically solves all of their security problems. This is arguably not the case and largely exaggerates the role the SSL/TLS protocols can play in the security arena.

But SSL/TLS is by far the most widely used and most important technology to secure internet applications or some parts of them. This is certainly the case for all applications on the World Wide Web (WWW) based on the Hypertext Transfer Protocol (HTTP), but it is increasingly also true for many other internet applications, such as electronic mail (email), instant messaging, file transfer, terminal access, internet banking, money transfer, remote internet voting, online gaming, as well as consumer electronics connected to the Internet of Things (IoT). Many of these

1 This quote seems to have first appeared in [1].

applications and respective protocols are nowadays routinely layered on top of SSL, TLS, and datagram TLS (DTLS) to provide basic security services to their respective users.

Considering the wide deployment of the SSL/TLS protocols, it is important to teach application designers and developers about the fundamental principles and the rationale behind the various versions of the protocols. Simply invoking security software libraries and respective function calls from an application programming interface (API) is not enough to design and develop secure applications.[2] In fact, it is fairly common today to invoke such libraries and function calls from otherwise exploitable code. The resulting application is not going to be secure—whether SSL/TLS is in place or not. Against this background, secure programming and secure software development techniques are of the utmost importance when it comes to building secure applications. Also, a thorough understanding of a security technology is required to correctly apply it and properly complement it with other security technologies. This rule of thumb also applies to SSL/TLS. It is necessary to fully understand what the SSL/TLS protocols can do and what they cannot do in order to apply them correctly. Otherwise, mistakes and omissions are likely to occur and hurt badly. The SSL/TLS protocols are not a panacea. They enable applications to be only as secure as the underlying infrastructural components, such as the computer systems and networks in use. If these components are vulnerable and susceptible to attacks, then the security the SSL/TLS protocols provide may be questionable or even illusive—I occasionally use the term *cryptollusion* to emphasize this point [2].

When I started to compile a teaching module about SSL/TLS more than two decades ago, I used two reference books [3, 4] that nicely introduced the topic and provided a comprehensive overview about SSL 3.0 and the then newly specified TLS 1.0 protocol. As my lecture notes matured and the two books grew old and slowly became out of date, I decided to take my notes and compile a new book that would not only address the fundamental principles of the SSL/TLS protocols,

2 Sometimes the libraries themselves contain bugs that allow adversaries to attack the systems that employ them. In 2014, for example, it was revealed that the SSL/TLS implementation of Apple iOS contained a serious bug that was later named the "double goto fail" bug (because it was caused by a goto fail statement that was erroneously written twice) and that the GnuTLS implementation contained a similar bug named "GnuTLS bug." The bugs were similar in the sense that they both allowed invalid public key certificates to pass certificate validation checks, and hence that an adversary could use such certificates to mount a man-in-the-middle (MITM) attack (for the purpose of this book, we continue to use the term MITM, although some people prefer the term person-in-the-middle that would then be abbreviated as PITM). Maybe most importantly, it was revealed in the same year that some older versions of the OpenSSL library contained a very severe bug in the implementation of the Heartbeat extension of the (D)TLS protocol(s). As further addressed in Section 3.8, this bug allowed an adversary to read out the server memory, possibly compromising cryptographic keys stored therein. The bug became known as *Heartbleed*, and it casted a damning light on the security of OpenSSL.

but would also explain the rationale behind their designs. The resulting book was entitled *SSL/TLS: Theory and Practice* and it first appeared in 2009.

The triumphant success of SSL/TLS and many developments made it necessary to update the book and publish a second edition sooner than originally anticipated. This seems to be the price to pay when addressing a timely and rapidly evolving field: Books outdate sooner than is normally the case. So, I took the opportunity to update the book and bring it in line with some cryptanalytical results and developments. Some of these results were already known when the first edition of the book went to press. However, they were not properly addressed, because there was no evidence that the underlying vulnerabilities could actually be exploited in real-world attacks. This changed fundamentally, and terms like BEAST, CRIME, TIME, BREACH, POODLE, FREAK, Logjam, and Lucky 13 made press headlines and frightened both application developers and users.

Against this background, I wrote the second edition of *SSL/TLS: Theory and Practice* that was released in 2016. In addition to providing a thorough introduction into the SSL/TLS and DTLS protocols, this book also explained the attacks mentioned above (and many more), and tried to discuss the patches and countermeasures that were available at that time—sometimes together with the respective counterattacks. Around the same time, another book appeared [5][3] that addressed complementary topics (that are not directly related to security), such as implementation, deployment, performance optimization, and configuration issues—sometimes even related to specific software packages and products. For the reasons discussed below, I took a different approach and didn't delve into implementation issues and configuration details for particular implementations.

Following the old (and anonymous) saying that "attacks always get better; they never get worse," many of the attacks mentioned above were refined and severely hit the SSL/TLS ecosystem. Most of these attacks and the underlying vulnerabilities and shortcomings were finally addressed and mitigated in an entirely new version 1.3 of both the TLS protocol and the DTLS protocol that appeared in 2018 and 2022. These new protocol versions brought so many new ideas and points to address that it was again necessary to update the book and come up with a third edition. This is the book you have in your hands or see on your screen: It is the third edition of the book that was originally published in 2009, and that tries to explain the SSL and TLS protocols from today's perspective, also explaining all innovative ideas that have been incorporated into TLS 1.3. Again, the book elaborates on some complementary topics, like DTLS, firewall traversal, and public key certificates and internet public key infrastructure (PKI).

In addition to providing a basic introduction and discussion of the SSL, TLS, and DTLS protocols, another major goal of the book is to provide enough

3 More recently, a second edition of this book was released.

background information to understand, discuss, and put into perspective even the latest attacks, countermeasures, and counterattacks. This goal is ambitious, because it requires a lot of background knowledge mainly in applied cryptography. There are at least two consequences that need to be mentioned here:

- First, I have to assume that readers who want to truly understand the attacks (and hence also the book) have some basic knowledge about cryptography. This includes secret key cryptography, public key cryptography, and some cryptographic protocols to do things like a key exchange. Fortunately, there are many books that introduce these topics and can be used as an entry point. For obvious (and selfish) reasons, I recommend [2] here.
- Second, I have changed the outline of the book (as compared to the previous two editions): Instead of explaining the attacks in the main text, I have compiled the respective explanations in Appendix A. This makes it possible for the reader to acquire the knowledge about a particular attack, whenever he or she comes across the respective attack name. This should make the main text more readable and still provide the material necessary to understand what is going on. The resulting book is not intended to be read from the start to the end in a linear fashion. Instead, readers regularly have to jump to the appendix to learn how a particular attack actually works, and then jump back to the main text to learn about the implications of the attack with regard to countermeasures and counterattacks, as well as the design of newer protocol versions (i.e., in the aftermath of the attack).

The bottom line is that this book requires some basic knowledge about cryptographic technologies and techniques, and that it tries to explain how they are used in the SSL/TLS ecosystem. It mainly addresses theoretical aspects, but it also explains to some extent how the protocols can withstand the test of practice. It is hoped that the result is something that contradicts both quotes stated above: With regard to Einstein's quote it is hoped that theory does not deviate too far away from practice, and with regard to the anonymous quote it is hoped that the SSL/TLS protocols not only work in practice, but that it is also understood in theory why this is the case. Needless to say that these goals are ambitious.

As mentioned before, implementation issues and respective details are not addressed or only addressed superficially in this book. There are so many implementations of the SSL/TLS protocols, both freely and commercially available, that it makes no sense to address them in a (static) book; they are modified and updated too frequently. The most popular open-source implementations are OpenSSL,[4]

4 https://www.openssl.org.

GnuTLS,[5] and Bouncy Castle,[6] but there are many more. Some of these implementations are available under a license that is compatible with the GNU General Public License (GPL), such as GnuTLS, whereas the licenses of some other implementations are special and slightly deviate from the GPL, such as the OpenSSL license.

Because this book targets more technicians than lawyers, I do not further address the many issues regarding software licenses. Instead, I emphasize the fact that some open-source implementations have a bad track record when it comes to security (remember the Heartbleed bug), and hence there are also a few open-source implementations that have forked from OpenSSL, such as LibreSSL from OpenBSD,[7] BoringSSL from Google,[8] and s2n[9] from Amazon.[10] More interestingly, there are open source SSL/TLS implementations that make it possible to perform a formal verification of the resulting implementation, such as miTLS.[11] Furthermore, there are SSL/TLS implementations that are dual-licensed, meaning that they are available either as an open source or under a commercial license. Examples of this type include the Rambus TLS Toolkit[12] (formerly known as MatrixSSL), mbed TLS[13] (formerly known as PolarSSL), wolfSSL[14], and cryptlib.[15]

In addition, all major software manufacturers have SSL/TLS implementations and libraries of their own that they embed in their products, such as Secure Channel (SChannel) from Microsoft, Secure Transport from Apple, the Java Secure Socket Extensions (JSSE) from Oracle, and the Network Security Services (NSS) from Mozilla.[16] Due to their origin, these implementations are particularly widely deployed in the field and hence used by many people in daily life on a regular basis. If you want to use the SSL/TLS protocols practically (e.g., to secure an e-$*$ application), then you may have to delve into the documentation and technical specification of the application or development environment that you are working with. This book is not a replacement for these documents; it is only aimed at providing the basic knowledge to properly understand them—you still have to capture and read them. In the case of OpenSSL, for example, you may use [6] or Chapter 11 of [5] as a handy reference. Keep in mind, though, that, due to Heartbleed, the security of

5 https://www.gnutls.org.
6 https://www.bouncycastle.org.
7 https://www.libressl.org.
8 https://boringssl.googlesource.com/boringssl.
9 The acronym s2n stands for "signal to noise," referring to the fact that the signals generated by legitimate visitors of websites may be hidden from noise by the use of strong cryptography.
10 https://github.com/awslabs/s2n.
11 https://www.mitls.org.
12 https://www.rambus.com/security-ip/software-protocols/secure-communication-toolkits/tls-toolkit.
13 https://tls.mbed.org.
14 https://www.wolfssl.com.
15 https://www.cryptlib.com.
16 Note that the NSS is also available as open source under a special Mozilla Public License (MPL).

OpenSSL has been improved considerably. In the case of another library or development environment, you are condemned to read the original documentation or manual anyway—there is no bypass or shortcut here.

In addition to cryptography, this book also assumes some basic familiarity with TCP/IP networking. Again, this assumption is reasonable, because anybody not familiar with TCP/IP is well-advised to first get in touch and try to comprehend TCP/IP networking, before moving on to the SSL/TLS protocols. Only trying to understand SSL/TLS is not likely to be fruitful. Readers unfamiliar with TCP/IP networking can consult one (or even several) of the many books about the topic. Among these books, I particularly recommend the book of Douglas Comer [7] and the trilogy coauthored by Richard Stevens [8–10], but there are many other (complementary) books also available in the shelves of the bookstores.

To properly understand the current status of the SSL/TLS protocols, it is useful to be familiar with the internet standardization process. Again, this process is likely to be explained in any book on TCP/IP networking. It is also explained in RFC 2026 [11]—that also represents a Best Current Practice (BCP) document—and updated on a web page hosted by the Internet Engineering Task Force (IETF).[17] For many protocols specified in RFC documents, we are going to say whether they have been submitted to the internet standards track or specified for experimental, informational, or historic use only. This distinction is important and relevant in the field.

When we discuss the practical use of the SSL/TLS protocols, it is quite helpful to visualize things with a network protocol analyzer, such as Wireshark[18] or any other software tool that provides a similar functionality. Wireshark is freely available and open-source. With regard to SSL/TLS, it is sufficiently complete, meaning that it can be used to capture and properly analyze data referring to the SSL/TLS protocols. We don't reproduce screenshots in this book, mainly because the graphical user interfaces (GUIs) of tools like Wireshark are nested, and the corresponding screenshots are difficult to read and interpret if only single screenshots are allowed. When we use Wireshark output, we provide it in purely textual form. This is visually less stimulating, but generally more appropriate and therefore more useful for the purpose of this book. For those who want to go one step further and experiment with SSL/TLS implementations and even try out specific attacks, TLS-Attacker[19] may be a tool—or rather a Java-based framework for analyzing TLS libraries—to consider. This is particularly true for experimentally minded and hands-on people.

The third edition of *SSL/TLS: Theory and Practice* is again organized and structured in seven chapters, described as follows:

17 https://www.ietf.org/about/standards-process.html.

18 http://www.wireshark.org.

19 https://github.com/tls-attacker/TLS-Attacker.

- Chapter 1, *Introduction*, prepares the ground for the topic of this book and provides the fundamentals and basic principles that are necessary to understand the SSL/TLS protocols properly.
- Chapter 2, *SSL Protocol*, introduces, overviews, and puts into perspective the SSL protocol.
- Chapter 3, *TLS Protocol*, does the same for the TLS protocol. Unlike Chapter 2, it does not start from scratch but focuses on the main differences between the SSL and the various versions of the TLS protocol, including TLS 1.3.
- Chapter 4, *DTLS Protocol*, elaborates on the DTLS protocol, which is basically a UDP version of the TLS protocol. Again, the chapter mainly focuses on the differences between the SSL/TLS protocols and the DTLS protocol.
- Chapter 5, *Firewall Traversal*, addresses the practically relevant and nontrivial problem of how the SSL/TLS protocols can (securely) traverse a firewall (or middlebox, respectively).
- Chapter 6, *Public Key Certificates and Internet PKI*, elaborates on the management of public key certificates used for the SSL/TLS protocols, for example, as part of an internet PKI. This is a comprehensive topic that deserves a book of its own. Because the security of the SSL/TLS (and DTLS) protocols depends on and is deeply interlinked with the public key certificates in use, we have to say a little bit more than what is usually found in introductory texts about this topic.[20]
- Chapter 7, *Concluding Remarks*, rounds off the book with some recommendations.

In addition, the book includes a comprehensive Appendix A, which explains in detail and puts into perspective the various attacks mentioned throughout the book; Appendix B, a list of registered cipher suites; Appendix C, a list of registered TLS extensions with respective explanations; a list of abbreviations and acronyms; my biography; and an index.

At various places, the book refers to common vulnerabilities and exposures (CVEs) entries. CVE refers to a common enumeration scheme for publicly known problems that should make it easier to share data across separate vulnerability capabilities (e.g., tools, repositories, and services). For a representative repository for CVEs that is widely used in the field, please see https://cve.mitre.org. Sometimes, CVEs come along with respective common vulnerability scoring system (CVSS)

20 In particular, this also includes the Let's Encrypt initiative and the automatic certificate management environment (ACME) and protocol.

values. Due to the fact that these values do not follow a clear and commonly agreed metric, we don't use CVSS values in this book.

Again, I hope that *SSL/TLS: Theory and Practice, Third Edition*, serves your needs. The ultimate goal of a technical book is to provide insight and save readers time, and I hope that this applies to this book, too. Also, I would like to take the opportunity to invite readers to let me know your opinions and thoughts. If you have something to correct or add, please let me know. If I have not expressed myself clearly, please let me know, too. I appreciate and sincerely welcome any comments or suggestions to update the book in future editions and turn it into a reference book that can be used for educational purposes. The best way to reach me is to send a message to rolf.oppliger@esecurity.ch. You can also visit the book's home page at https://www.esecurity.ch/Books/ssltls3e.html; I use this page to post errata lists, additional information, and complementary material (e.g., slide decks to teach classes). I look forward to hearing from you in one way or another.

References

[1] Grint, K., *Management: A Sociological Introduction*, Polity Press, Cambridge, UK, 1995.

[2] Oppliger, R., *Cryptography 101: From Theory to Practice*, Artech House, Norwood, MA, 2021.

[3] Rescorla, E., *SSL and TLS: Designing and Building Secure Systems*, Addison-Wesley, Reading, MA, 2000.

[4] Thomas, S.A., *SSL and TLS Essentials: Securing the Web*, John Wiley & Sons, New York, NY, 2000.

[5] Ristić, I., *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*, Feisty Duck Limited, London, UK, 2014.

[6] Viega, J., M. Messier, and P. Chandra, *Network Security with OpenSSL*, O'Reilly, Sebastopol, CA, 2002.

[7] Comer, D.E., *Internetworking with TCP/IP Volume 1: Principles, Protocols, and Architecture*, 6th edition, Addison-Wesley, New York, NY, 2013.

[8] Fall, K.R., and W.R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, 2nd edition, Addison-Wesley Professional, New York, NY, 2011.

[9] Stevens, W.R., and G.R. Wright, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley Professional, New York, NY, 1995.

[10] Stevens, W.R., and G.R. Wright, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP and the Unix Domain Protocols*, Addison-Wesley Professional, New York, NY, 1996.

[11] Bradner, S., "The Internet Standards Process—Revision 3," RFC 2026 (BCP 9), October 1996.

Acknowledgments

Many people have been involved and contributed to the writing and publication of this book. First, I thank the designers and developers of the SSL/TLS protocols. They have turned theory into practice and provided the basis for many security solutions in use today. Second, I thank the buyers and readers of the first two editions of the book, especially the ones who provided feedback and approached me personally with error notifications, questions, comments, and hints for additional material. It is always a pleasure to see a book being used in the field. Third, I thank all colleagues who have patiently answered questions, reviewed (parts of) the manuscript, and discussed interesting topics with me. All of them have provided invaluable and highly appreciated contributions to the book. Representative for all of them, I want to express my deepest thanks to Kenny Paterson, who was kind enough to provide the foreword for the book (even a few hours before the deadline ended at New Year's Eve), and Juraj Somorovsky, who has done an outstanding job as a reviewer to uphold the quality of the book. Again, the staff at Artech House has been enormously helpful in producing and promoting the book. Among these people, I am particularly grateful to Casey Gerard, who supervised the development process, and Bill Bazzy, who has not only enabled this book but also the more than fifty other books that have been published in the information security and privacy series since its launch in the 1990s. Last but not least, I am indebted to my family—my beloved wife, Isabelle, and our adult children, Lara (to whom I dedicate this book again) and Marc. They have again supported the project, and without their encouragement, patience, and love, this book would not have come into existence.

Chapter 1

Introduction

This introductory chapter prepares the ground for the topic of the book: It starts with a brief outline of information and network security in Section 1.1, delves more deeply into transport layer security in general, and the evolution of the secure sockets layer (SSL) and transport layer security (TLS) protocols in Section 1.2, and concludes with some final remarks in Section 1.3.

1.1 INFORMATION AND NETWORK SECURITY

According to the internet security glossary that is available in the informational FC 4949 [1], *information security* (INFOSEC) refers to "measures that implement and assure security services in information systems, including in computer systems (computer security) and in communication systems (communication security)." In this context, *computer security* (COMPUSEC) refers to security services provided by computer systems (e.g., access control services), whereas *communication security* (COMSEC) refers to security services provided by communication systems that exchange data (e.g., data confidentiality, authentication, and integrity services). It goes without saying that in a practical setting, COMPUSEC and COMSEC must go hand in hand to provide a reasonable level of INFOSEC. If, for example, data is cryptographically protected while being transmitted, then that data may be susceptible to attacks on either side of the communication channel while it is being stored or processed. So COMSEC—which is essentially what this book is all about—must always be complemented by COMPUSEC; otherwise all security measures are not particularly useful and can be circumvented at will. It also goes without saying that the formerly clear distinction between COMPUSEC and COMSEC has been blurred to some extent by recent developments in the realm of service-oriented architectures

(SOAs) and cloud computing. The distinction is no longer precise, and hence it may not be as useful as it used to be in the past.

INFOSEC (as well as COMPUSEC and COMSEC) comprises all technical and nontechnical measures aimed at implementing and assuring security services in information systems. So it is not only about technology, and in many cases, legal, organizational, and personnel measures are more effective than purely technical ones. This should be kept in mind, despite the fact that this book is only about technology. Any technical measure that is not complemented by adequate nontechnical measures is likely to fail—at least in the long term.

To illustrate this point, let us consider ransomware as an example. Technically defeating ransomware seems to be difficult, if not impossible given the current state of computer security. But nontechnically defeating ransomware is simple: Just make sure that nobody is paying ransom. If nobody is paying ransom, then it is very likely that criminals move away from ransomware to find other criminal activities that are more profitable (remember that crime always follows the money). Making sure that nobody is paying ransom is clearly a nontechnical measure, but we leave aside the question how to implement and enforce it.

For the purpose of this book, we use the term *network security* as a synonym for communication security. So we try to invoke measures that can be used to implement and assure security services for data in transmission. When we talk about security services, it makes a lot of sense to introduce them (together with mechanisms that are qualified to provide them) in some agreed-upon and standardized way.

A standard that can be used here is the *security architecture* that is appended as part two of the formerly famous (but nowadays almost entirely ignored) open systems interconnection (OSI) basic reference model specified by the Joint Technical Committee 1 (JTC1) of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) in 1989 [2]. Without delving into the details, we only mention that the OSI model has seven layers, whereas the more familiar TCP/IP model has only four of them. Figure 1.1 illustrates the layers and the relationship between them. Most importantly, the physical and data link layers of the OSI model are merged in a single network access layer in the TCP/IP model, and the session, presentation, and application layers of the OSI model are merged in a unique application layer in the TCP/IP model. There are pros and cons on either side, but in the end they are only models that should help to understand the real world.

According to its title, ISO/IEC 7498-2 claims to provide a security architecture, but it is more a terminological framework than a true architecture (you may refer to [3] for a discussion of what a security architecture really is). Nevertheless, we still use the term *OSI security architecture* to refer to ISO/IEC 7498-2. In 1991,

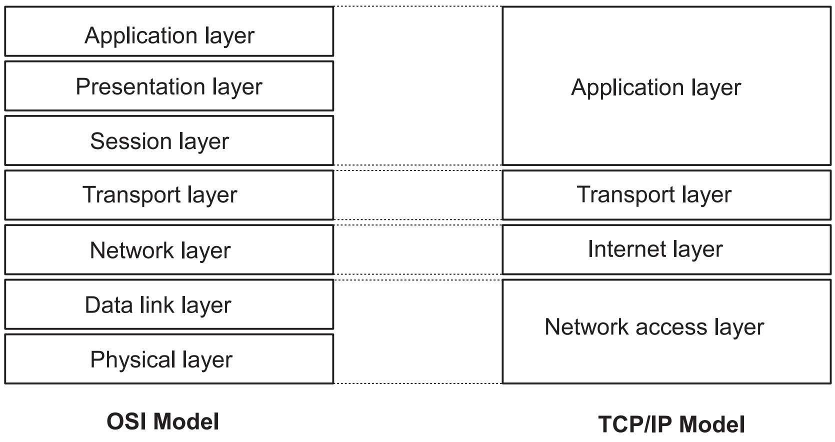

Figure 1.1 Layers of the OSI and TCP/IP models.

the Telecommunication Standardization Sector of the International Telecommunication Union (ITU), also known as ITU-T, adopted the OSI security architecture in its recommendation X.800 [4]. Also in the early 1990s, the no longer existing Privacy and Security Research Group (PSRG) of the Internet Research Task Force (IRTF[1]) preliminarily adopted the OSI security architecture in a corresponding internet security architecture published as an Internet-Draft.[2] In essence, ISO/IEC 7498-2, ITU-T X.800, and the internet security architecture draft all describe the same security standard, and in this book we use the term *OSI security architecture* to refer to all of them collectively. Contrary to the OSI basic reference model, the OSI security architecture has been in widespread use in the past few decades—at least for referential purposes. It provides a general description of security services and related security mechanisms and discusses their relationships and dependencies. Let us briefly introduce the security services and mechanisms that are mentioned in the OSI security architecture. Note that these services and mechanisms are neither comprehensive nor are they intended to be so. For example, anonymity and pseudonymity services

1 The IRTF is a sister group to the IETF. Its stated mission is "to promote research of importance to the evolution of the future internet by creating focused, long-term and small research groups working on topics related to Internet protocols, applications, architecture and technology." Its website is available at https://irtf.org.

2 This work has been abandoned.

are not addressed at all, but in a real-world application setting, these services may still be important. Take digital cash and e-voting as examples to make this point.

Table 1.1
Classes of OSI Security Services

1	Peer entity authentication service
	Data origin authentication service
2	Access control service
3	Connection confidentiality service
	Connectionless confidentiality service
	Selected field confidentiality service
	Traffic flow confidentiality service
4	Connection integrity service with recovery
	Connection integrity service without recovery
	Selected field connection integrity service
	Connectionless integrity service
	Selected field connectionless integrity service
5	Nonrepudiation with proof of origin
	Nonrepudiation with proof of delivery

1.1.1 Security Services

As shown in Table 1.1, the OSI security architecture distinguishes between five classes of security services (i.e., authentication, access control, data confidentiality, data integrity, and nonrepudiation[3] services). Just as layers define functionality in the OSI reference model, so do services in the OSI security architecture. A security service hence provides a particular security functionality or set of functionalities that may be relevant in a particular application setting.

1.1.1.1 Authentication Services

As its name suggests, an *authentication service* provides authentication for something, such as a peer entity or the origin of data. The respective services are slightly different and can be described as follows.

3 There is some controversy in the community regarding the correct spelling of the term *nonrepudiation*. In fact, the OSI security architecture uses *non-repudiation* instead of *nonrepudiation*, and there are many people still using this spelling. In this book, however, we use the more modern spelling of the term without a hyphen.

- A *peer entity authentication service* provides authentication for a peer entity, such as a user, client, or server, meaning that it allows an entity in an association to verify that the peer entity is what it claims to be. This, in turn, provides assurance that the peer entity is not attempting to masquerade or mounting an unauthorized replay attack. A peer entity authentication is typically performed either during a connection establishment phase or, occasionally, during a data transfer phase.
- A *data origin authentication service* provides authentication for the origin of data, meaning that it allows an entity in an association to verify that the source of data received is as claimed. A data origin authentication service is typically performed during a data transfer phase. Note, however, that such a service only protects the origin of data, and that an adversary may still duplicate or modify the data at will. To protect against such attacks, a data origin authentication service may be complemented with a data integrity service (as discussed in Section 1.1.1.4). Hence, data origin authentication and integrity services are usually combined with each other and go hand in hand.

Authentication services are very important in practice, and they often represent a prerequisite for the provision of authorization, access control, accountability, and respective services. Authorization refers to the process of granting rights, which includes the granting of access based on access rights. Access control refers to the process of enforcing these rights, and accountability refers to the property that actions of an entity can be traced uniquely to a particular entity. All of these services are important for the overall security of a system.

1.1.1.2 Access Control Services

As mentioned above, *access control services* enforce access rights, meaning that they protect system resources against unauthorized use. The use of a system resource is unauthorized if the entity that seeks to use it does not have the proper privileges or permissions to do so. As such, access control services are typically the most commonly thought of services in computer and network security. However, again as mentioned above, access control services are closely tied to authentication services: A user or process acting on the user's behalf must usually be authenticated before access can be controlled and an access control service can be put in place. Authentication and access control services therefore usually go hand in hand—this is why people sometimes use terms like *authentication and authorization infrastructure* (AAI), *identity management*, or *identity and access management* (IAM) to refer to an infrastructure that provides support for both authentication and authorization in terms of access control. Such infrastructures are very important in the field.

1.1.1.3 Data Confidentiality Services

In general parlance, data confidentiality refers to the property that data is only available to authorized entities, and hence *data confidentiality services* protect data from unauthorized disclosure. There are several types of such services:

- A *connection confidentiality service* provides confidentiality for all data transferred over a connection (i.e., it applies to a connection-oriented setting).
- A *connectionless confidentiality service* provides confidentiality for individual data units (i.e., it applies to a connectionless setting where data units are protected even though they are not transferred over a connection).
- A *selective field confidentiality service* provides confidentiality for certain fields within individual data units or data transmitted over a connection.
- A *traffic flow confidentiality service* provides confidentiality for traffic flows, meaning that it attempts to protect all data that is associated with and communicated in a traffic flow against traffic analysis.

The first three confidentiality services can be provided by standard cryptographic techniques and mechanisms, whereas the provision of traffic flow confidentiality services requires some complementary and more involved security mechanisms that are not addressed in this book.

1.1.1.4 Data Integrity Services

Data integrity refers to the property that data is not modified (or even destroyed) in an unauthorized way, and hence *data integrity services* protect data from unauthorized modification. Again, there are several types of such services:

- A *connection integrity service with recovery* provides integrity for all data transmitted over a connection, where the loss of integrity can be recovered.
- A *connection integrity service without recovery* is similar to a connection integrity service with recovery, except that the loss of integrity can only be detected but not recovered.
- A *selected field connection integrity service* provides integrity for specific fields within the data transmitted over a connection.
- A *connectionless integrity service* provides integrity for individual data units that are not transmitted over a connection.

- A *selected field connectionless integrity service* provides integrity for specific fields within individual data units.

To secure a connection, it is usually the case that a peer entity authentication service provided at the start of the connection is combined with a connection integrity service provided during the connection. This ensures that all data received by the recipient is authentic and has not been tampered with during its transmission. To keep things as simple as possible, connection integrity services are often provided without recovery. This also applies to the SSL/TLS protocols.

1.1.1.5 Nonrepudiation Services

A *nonrepudiation service* protects an entity from having a peer entity (that is also involved in the communication) deny that it has participated in all or part of the communication. In general, there are two types of nonrepudiation services that are relevant in practice:

- A *nonrepudiation service with proof of origin* provides the recipient of data with a proof of origin, meaning that the sender cannot later deny having sent the data.
- A *nonrepudiation service with proof of delivery* provides the sender of data with a proof of delivery, meaning that the recipient of the data cannot later deny having received it.[4]

Nonrepudiation services are key for internet-based e-commerce (e.g., [5]). Consider, for example, the situation in which an investor communicates with his or her stockbroker over the internet. If the investor decides to sell a large number of stocks, then he or she may send a corresponding request to the stockbroker. If the prices are about to change only moderately, then everything works fine. However, if the stock price rises sharply, then the investor may deny having sent the order to sell the stocks in the first place. Conversely, it is possible that under reversed circumstances the stockbroker may deny having received the order to sell the stocks. In situations like these, it seems to be the case that the ability to provide nonrepudiation services is key to the success of the whole endeavor.

Not all security services are equally important in a given application setting. In Section 2.1, we elaborate on what services are relevant for the SSL/TLS protocols.

4 Note, however, that a proof of delivery only proves the successful delivery of the data. It does not necessarily mean that the recipient has also read and properly understood the data.

1.1.2 Security Mechanisms

In addition to the security services mentioned above, the OSI security architecture also itemizes security mechanisms that may be used to implement the services. A distinction is made between specific security mechanisms and pervasive ones. While a specific security mechanism can be used to implement a specific security service, a pervasive security mechanism is generally not specific to a particular service and can be used to implement—or rather complement—several security services at the same time.

Table 1.2
Specific Security Mechanisms

1	Encipherment
2	Digital signature mechanisms
3	Access control mechanisms
4	Data integrity mechanisms
5	Authentication exchange mechanisms
6	Traffic padding mechanisms
7	Routing control mechanisms
8	Notarization mechanisms

1.1.2.1 Specific Security Mechanisms

As summarized in Table 1.2, there are eight specific security mechanisms enumerated in the OSI security architecture. Some of them are cryptographic in nature, meaning that their proper implementation and use requires some basic knowledge in cryptography. As already mentioned in the preface, this knowledge is assumed and not repeated here (you may refer to [6] to acquire it). The specific security mechanisms can be characterized as follows:

1. *Encipherment* can be used to protect the confidentiality of data or to support other security mechanisms.
2. *Digital signature mechanisms* can be used to provide digital signatures and respective nonrepudiation services. A digital signature is the electronic analog of a handwritten signature, and as such it is applicable to electronic documents. Like handwritten signatures, digital signatures must not be forgeable; a recipient must be able to verify the signature, and the signatory must not be able to repudiate it later. But unlike handwritten signatures, digital signatures

incorporate the data (or a hash value of the data) that is signed. Different data therefore results in different signatures, even if the signatory remains the same.

3. *Access control mechanisms* can be used to control access to system resources. Traditionally, a distinction is made between a discretionary access control (DAC) and a mandatory access control (MAC).[5] In either case, the access control is described in terms of subjects, objects, and access rights:

   - A subject is an entity that attempts to access objects. This can be a host, a user, or an application.
   - An object is a resource to which access needs to be controlled. This can range from an individual data field in a file to a large program.
   - Access rights specify the level of authority for a subject to access an object, so access rights are defined for each subject-object pair. Examples of UNIX-style access rights are read, write, and execute.

   More recently, people have introduced the notion of a role and have developed role-based access controls (RBACs) to make the assignment of access rights to subjects more simple and straightforward, and hence also more flexible [7]. Today, people are elaborating on using attributes to simplify the assignment of access rights in so-called attribute-based access controls (ABACs) [8].

4. *Data integrity mechanisms* can be used to protect the integrity of data—be it individual data units or fields within them or sequences of data units or fields within them. Note that data integrity mechanisms, in general, do not protect against replay attacks that work by recording and replaying previously sent data units. Also, protecting the integrity of a sequence of data units and fields within these data units generally requires some form of explicit ordering, such as sequence numbering, time-stamping, or cryptographic chaining.

5. *Authentication exchange mechanisms* can be used to verify the claimed identities of entities. A broad distinction is made between *weak* authentication exchange mechanisms that are vulnerable to passive wiretapping and replay attacks and *strong* authentication exchange mechanisms that protect against these attacks. Strong authentication exchange mechanisms usually employ sophisticated cryptographic techniques and sometimes even rely on dedicated hardware as a second factor (e.g., smartcards). Sometimes, the use of biometrics yields another authentication exchange mechanism for human users.

5 This acronym is important in the security literature. In the rest of this book, however, a MAC always refers to a message authentication code.

6. *Traffic padding mechanisms* can be used to protect against traffic analysis. It works by having one entity generate and transmit randomly composed data hand in hand with the actual data. Only the originator and intended recipient(s) know how this data is transmitted; thus, an unauthorized party who captures and attempts to replay the data cannot distinguish the randomly generated data from meaningful data. This, in turn, means that traffic analysis doesn't provide meaningful results.
7. *Routing control mechanisms* can be used to choose—either dynamically or by prearrangement—specific routes for data transmission. Communicating systems may, on detection of persistent passive or active attacks, wish to instruct the network service provider to establish a connection via a different route. Similarly, data carrying certain security labels may be forbidden by policy to pass through certain networks or links. Routing control mechanisms are not always available, but if they are they tend to be very effective.
8. *Notarization mechanisms* can be used to assure certain properties of the data communicated between two or more entities, such as its integrity, origin, time, or destination. The assurance is provided by a trusted party—sometimes also called a trusted third party (TTP)—in a testable manner.

As explained later, all specific security mechanisms except access control and routing control mechanisms are employed by the SSL/TLS protocols. Access control mechanisms need to be invoked above the transport layer (typically at the application layer), whereas routing control mechanisms are better placed underneath it. Traffic padding mechanisms would also be better placed underneath the transport layer, but since version 1.2 the TLS protocol at least provides some support for it.

Table 1.3
Pervasive Security Mechanisms

1	Trusted functionality
2	Security labels
3	Event detection
4	Security audit trail
5	Security recovery

1.1.2.2 Pervasive Security Mechanisms

Contrary to the specific security mechanisms outlined in Section 1.1.2.1, pervasive security mechanisms are generally not specific to a particular security service. Some of these mechanisms can even be regarded as aspects of security management. As shown in Table 1.3, the OSI security architecture enumerates five security mechanisms that are pervasive:

1. As its name suggests, *trusted functionality* refers to functionality that is trusted to perform as intended. From a security perspective, any functionality (provided by a service and implemented by a mechanism) should be trusted, and hence trusted functionality is a pervasive security mechanism that is orthogonal to all specific security mechanisms.
2. System resources may have *security labels* associated with them, for example, to indicate a sensitivity level. This allows the resources to be treated in an appropriate way. For example, it allows data to be encrypted transparently (i.e., without user invocation) for transmission. In general, a security label may be additional data associated with the data or it may be implicit (e.g., implied by the use of a specific key to encipher data or implied by the context of the data such as the source address or route). In an SSL/TLS setting, however, security labels are not typically used and all data transmitted is protected equally instead.
3. It is increasingly important to complement preventive security mechanisms with detective and corrective ones. This basically means that security-related events must be detected in one way or another. This is where *event detection* as another pervasive security mechanism comes into play. Event detection basically depends on heuristics that are applied to network traffic.
4. A security audit refers to an independent review and examination of system records and activities to test for adequacy of system controls, to ensure compliance with established policy and operational procedures, to detect breaches in security, and to recommend any indicated changes in control, policy, and procedures. Consequently, a *security audit trail* refers to data collected and potentially used to facilitate a security audit. Needless to say, this a very fundamental and important pervasive security mechanism.
5. As previously noted, corrective security mechanisms are getting more and more important. *Security recovery* is about implementing corrective security mechanisms and putting them in appropriate places. For all practical purposes, security recovery is a very important topic, as being unable to recover from a security incident is almost always fatal.

The SSL/TLS protocols do not prescribe any pervasive security mechanism. Instead, it is up to a particular implementation to support one or several of these mechanisms. It goes without saying that SSL/TLS alert messages at least provide a basis for event detection, security audit trail, and security recovery.

Last but not least, we recapitulate the fact that the OSI security architecture has not been developed to solve a particular network security problem, but rather to provide the network security community with a terminology that can be used to consistently describe and discuss security-related problems and corresponding solutions. This book follows this tradition and uses the OSI security architecture for exactly this purpose.

1.2 TRANSPORT LAYER SECURITY

When the WWW began its triumphal success in the first half of the 1990s, many people started to purchase items electronically. As is still the case, the predominant electronic payment systems were credit cards and respective credit card payments and transactions. Because people had reservations about the transmission of credit card information as being part of a web transaction, many companies and researchers looked into possibilities to provide web transaction security and corresponding services to the general public. The greatest common denominator of all these possibilities was the use of cryptographic techniques to provide basic security services. Other than that, however, there was hardly any consensus about what techniques to use and at what layer to invoke them.

In general, there are many possibilities to invoke cryptographic techniques at various layers of the TCP/IP model and protocol stack. In fact, all internet security protocols overviewed in [9] and [10] can be used to secure web transactions. In practice, however, the combined use of the IP security (IPsec) and the internet key exchange (IKE) protocols [11] on the internet layer, the SSL/TLS protocols on the transport layer,[6] and some variation of a secure messaging scheme [12] on the application layer are the most appropriate possibilities.

There is a famous *end-to-end argument* in system design that strongly speaks in favor of providing security services at a higher layer [13]. The argument basically says that any nontrivial communications system involves intermediaries, such as network devices, relay stations, computer systems, and software modules that are, in principle, unaware of the context of the communication being involved. This, in turn, means that these intermediaries are incapable of ensuring that the data is processed correctly. It thus follows that, whenever possible, communications

6 Strictly speaking, the SSL/TLS protocols operate at an intermediate layer between the transport layer and the application layer. To keep things simple and in line with the TCP/IP model, however, we refer to this layer as a part of the transport layer. This is not perfectly correct, but it simplifies things considerably and is therefore justified.

protocol operations should be defined to occur at the end points of a communications system, or as close as possible to the resource being controlled. The end-to-end argument applies generally (i.e., for any type of functionality), but as pointed out in [14], it particularly applies to the provision of network security services. This is still true, even after four decades of experience in the field.

Following the end-to-end argument and the respective design principles, the IETF chartered a web transaction security (WTS) working group (WG) in the early 1990s.[7] The WG was tasked with the specification of requirements and respective security services for web transactions (i.e., transactions using HTTP). The outcome of the WG is documented in [15–17].[8] Most importantly, the *Secure Hypertext Transfer Protocol* (S-HTTP or SHTTP) was a security enhancement for HTTP that could be used to encrypt and/or digitally sign documents or specific parts thereof [17]. As such, S-HTTP was conceptually similar to the specifications provided by the Extensible Markup Language (XML) Security WG of the World Wide Web Consortium (W3C).[9] They also provide support for XML encryption and XML signatures. S-HTTP was submitted to the web transaction discussion in 1994, and—due to its strong initial support from almost the entire software industry—it seemed to be only a question of time until it would become a standard and key player in the field.

But things evolved differently. Independent from the end-to-end argument and the S-HTTP proposal, the software developers at Netscape Communications[10] had prosecuted the claim that transport layer security provides an interesting trade-off between low-layer and high-layer security. In fact, they had taken the viewpoint of the application developer and wanted to enable him or her to establish secure connections (instead of normal connections) in a way that is as simple as possible. To achieve this goal, they inserted an intermediate layer between the transport layer and the application layer. This layer was named *secure sockets layer*, and it was to handle security, meaning that it had to establish secure connections and to transmit data over these secure connections. Because its functionality is deeply intertwined with the transport layer, it can be technically assigned to this layer, meaning that the SSL/TLS protocols can be assigned to the class of transport layer security protocols. More specifically, the SSL protocol is layered on top of a connection-oriented and reliable transport layer protocol like TCP. The connectionless best effort datagram delivery protocol that operates at the transport layer is named User

7 https://datatracker.ietf.org/wg/wts/about.

8 While [15] is an informational RFC document, [16] and [17] are experimental ones.

9 https://www.w3.org/2008/xmlsec.

10 Netscape Communications (formerly known as Netscape Communications Corporation and commonly known as Netscape) was an American software company that was best known for its web browser, Netscape Navigator. The company was acquired by AOL Inc. (previously known as America Online) in 1999. Today, AOL is a subsidiary of Verizon Communications, Inc.

Datagram Protocol (UDP),[11] and it was not until recently that the TLS protocol had been adapted to be used on top of UDP, as well. This is the realm of the DTLS protocol that is further addressed in Chapter 4.

Netscape Communications started to develop the SSL protocol soon after the National Center for Supercomputing Applications (NCSA[12]) released Mosaic 1.0—the first popular web browser—in 1993. Eight months later, in the middle of 1994, Netscape Communications had already completed the design of SSL version 1 (SSL 1.0). This version circulated only internally (i.e., inside Netscape Communications), since it had several shortcomings and flaws. For example, it didn't protect the integrity of data. In combination with the use of the stream cipher RC4 for data encryption, this allowed an adversary to make predictable changes to the plaintext data. Also, SSL 1.0 did not use sequence numbers, so it was vulnerable to various types of replay attacks. Later on, the designers of SSL 1.0 added sequence numbers and checksums, but still used an overly simple cyclic redundancy check (CRC) instead of a cryptographically strong hash function to protect data integrity.

This and a few other problems had to be resolved before the SSL protocol could be deployed in the field. Toward the end of 1994, Netscape Communications came up with SSL version 2 (SSL 2.0). Among other changes, the CRC was replaced with MD5 that was still assumed to be secure at this time.[13] Netscape Communications then released the Netscape Navigator that implemented SSL 2.0 together with a few other products that supported SSL 2.0. The official SSL 2.0 protocol specification was written by Kipp E.B. Hickman from Netscape Communications, and as such it was submitted to the IETF as an Internet-Draft entitled "The SSL Protocol" in April 1995.[14] Four months later, in August 1995, Netscape Communications also filed a patent application entitled "Secure Socket Layer Application Program Apparatus and Method," which basically refers to the SSL protocol (hence the patent is also called the *SSL patent*[15]). In August 1997, it was assigned to Hickman and Taher

11 It is sometimes argued that TCP is connection-oriented and reliable, whereas UDP is connectionless and unreliable. This characterization is imprecise, mainly because the term *unreliable* suggests that UDP was intentionally designed to lose packets. This was clearly not the case. Instead, a best-effort delivery protocol has no built-in functions to detect or correct for packet loss but relies on underlying protocols to provide this service. Over a modern LAN, for example, loss is nearly zero, and hence a best-effort delivery protocol is sufficient for many applications. A key benefit from providing no loss detection is that the resulting protocol is efficient to process and introduces no latency to the delivery. The bottom line is that it is more appropriate to say that UDP is a best-effort datagram delivery protocol than an unreliable one.

12 https://www.ncsa.illinois.edu.

13 The first successful collision attacks against MD5 were published in 2005. In 2016, they were used in the so-called *SLOTH attacks* against MD5 and SHA-1, where SLOTH is an acronym standing for "security loss due to the use of obsolete and truncated hash constructions."

14 http://tools.ietf.org/html/draft-hickman-netscape-ssl-00.

15 U.S. Patent No. 5,657,390 (https://patents.google.com/patent/US5657390A/en).

Elgamal—the inventor of the Elgamal public key cryptosystem [18]—on behalf of Netscape Communications. Because Netscape Communications had filed the patent for the sole purpose of preventing others from moving into the space, it was given away to the community for everyone to use for free (until it expired in 2015).

With the release of the Netscape Navigator (supporting the newly specified SSL 2.0 protocol), the internet and the WWW really started to take off. This made some other companies nervous about the potential and the lost opportunities of not being involved. Most importantly, Microsoft decided to become active and came up with Internet Explorer in the second half of 1995. In October of the same year, Microsoft published a protocol named *Private Communication Technology* (PCT) that was conceptually and technically very similar and closely related to SSL 2.0.[16] In particular, the two protocols used a compatible record format, meaning that a server could easily support both of them. Because the most significant bit of the protocol version number is set to one in the case of PCT (instead of zero as in the case of SSL), a server supporting both protocols could easily be triggered to use either of the two protocols. From today's perspective, the PCT protocol is only historically relevant. Some Microsoft products supported it, but outside the world of Microsoft, the PCT protocol has never been used and has silently sunk into oblivion. All one needs to know is the acronym and what it stands for, namely a Microsoft version of the SSL protocol.

The few improvements that had been suggested by PCT were retrofitted into SSL version 3 (SSL 3.0), which was officially released soon after the publication of PCT in 1995. The SSL 3.0 protocol was specified by Alan O. Freier and Philip Karlton from Netscape Communications with the support of Paul C. Kocher. Also, around this time, Netscape Communications hired several security professionals, including Elgamal (as mentioned above), that helped to make SSL 3.0 more robust and secure. The specification of SSL 3.0 was finally published as an Internet-Draft entitled "The SSL Protocol Version 3.0" in November 1996. More recently, the SSL 3.0 specification was released in a historic RFC document [19] that is used for referential purposes.

SSL 2.0 had a few shortcomings and security problems that were corrected in SSL 3.0:

- SSL 2.0 permitted the client and server to send only one public key certificate to each other. This basically means that the certificate had to be directly signed by a trusted root CA. Contrary to that, SSL 3.0 allows clients and servers to have arbitrary-length certificate chains.
- SSL 2.0 used the same keys for message authentication and encryption, which led to problems for certain ciphers. Also, if SSL 2.0 was used with RC4 in

16 http://graphcomp.com/info/specs/ms/pct.htm.

export mode, then the message authentication and encryption keys were both based on 40 bits of secret data. This is in contrast to the fact that the message authentication keys could potentially be much longer (the export restrictions only applied to encryption keys). In SSL 3.0, different keys are used, and hence even if weak ciphers are used for encryption, mounting attacks against message authenticity and integrity is still infeasible (because the respective keys are much longer by default).

- SSL 2.0 exclusively used MD5 for message authentication. In SSL 3.0, however, MD5 is complemented with SHA-1, and the construction to generate and verify a message authentication code (MAC) is more sophisticated (we come to this construction in Section 2.2.1.3).

Due to these shortcomings and security problems, the IETF recommended the complete replacement of SSL 2.0 with SSL 3.0 in 2011 [20]. Note, however, that due to some even more recent developments (in particular, regarding the POODLE attack), the IETF deprecated SSL 3.0 and the entire SSL protocol only four years later in 2015 [21].

After the publication of SSL 3.0 and PCT, there was a lot of confusion in the security community. On one hand, there was Netscape Communications and a large part of the internet and web security community pushing SSL 3.0. On the other hand, there was Microsoft with a large installed base pushing PCT. The company also had to support SSL for interoperability reasons. To make things worse, Microsoft had even come up with yet another protocol proposal, named *Secure Transport Layer Protocol* (STLP), which was basically a modification of SSL 3.0, providing additional features that Microsoft considered to be critical, such as support for UDP, client authentication based on shared secrets, and some performance optimizations (many of these features have been built into the TLS protocol meanwhile). In this situation, an IETF TLS WG[17] was formed in 1996 to resolve the issue and standardize a unified TLS protocol. This task was technically simple, because the protocols to begin with—SSL 3.0 and PCT/STLP—were already technically similar. However, it still took the IETF TLS WG a long time to accomplish its mission. There are at least three reasons for this delay:

- First, the internet standards process [22] requires that a statement be obtained from a patent holder indicating that a license will be made available to applicants under reasonable terms and conditions. This also applied to the SSL patent (such a statement was not included in the original specification of SSL 3.0).

17 https://datatracker.ietf.org/group/tls/about.

- Second, at the April 1995 IETF meeting in Danvers, Massachusetts, the IESG adopted the so-called *Danvers Doctrine* [23], which basically says that the IETF should design protocols that embody good engineering principles, regardless of exportability issues. With regard to the yet-to-be-defined TLS protocol, the Danvers Doctrine implied that the data encryption standard (DES) needed to be supported at a minimum and that 3DES was the preferred choice (remember that the export of DES and 3DES from the United States was still regulated at this time).
- Third, the IETF had a longstanding preference for unencumbered algorithms when possible. So when the Merkle-Hellman patent (covering many public key cryptosystems) expired in 1998, but RSA was still patented, the Internet Engineering Steering Group (IESG) began pressuring working groups to adopt the use of the unpatented public key cryptosystems.

When the IETF TLS WG finished its work in late 1997, it sent the first version of the TLS protocol specification to the IESG. The IESG, in turn, returned the specification with a few instructions to add other cryptosystems, namely DSA for authentication, Diffie-Hellman for key exchange (note that the respective patent was about to expire), and 3DES for encryption, mainly to solve the two last issues mentioned above. The first issue could be solved by adding a corresponding statement in the TLS protocol specification. Much discussion on the mailing list ensued, with Netscape Communications in particular resisting mandatory cryptographic systems in general and 3DES in particular. After some heated discussions between the IESG and the IETF TLS WG, a grudging consensus was reached and the protocol specification was resubmitted with the appropriate changes in place.

Unfortunately, another problem had appeared in the meantime: The IETF Public Key Infrastructure X.509 (PKIX) WG had been tasked to standardize a profile for X.509 certificates on the internet, and this WG was just winding up its work. From the very beginning, the TLS protocol depended on X.509 certificates and hence on the outcome of the IETF PKIX WG. In the meantime, the rules of the IETF forbid protocols advancing ahead of other protocols on which they depend. PKIX finalization took rather longer than originally anticipated and added another delay.

The bottom line is that it took almost three years until the IETF TLS WG could officially release its security protocol.[18] In fact, the first version of the TLS protocol (i.e., TLS 1.0) was specified in RFC 2246 [24] and released in January 1999. The required patent statement was included in Appendix G of this document. Despite the name change, TLS 1.0 is nothing more than a new version of SSL 3.0. In fact,

18 The protocol name had to be changed from SSL to TLS to avoid the appearance of bias toward any particular company.

there are fewer differences between TLS 1.0 and SSL 3.0 than there are differences between SSL 3.0 and SSL 2.0. TLS 1.0 is therefore sometimes also referred to as SSL 3.1 (and this viewpoint is still visible in the protocol header's version field). In addition to the TLS 1.0 specification, the IETF TLS WG also completed a series of extensions to the TLS protocol that are documented elsewhere.

After the 1999 release of TLS 1.0, work on the TLS protocol continued inside the IETF TLS WG. In April 2006, the TLS protocol version 1.1 (i.e., TLS 1.1) was specified in Standards Track RFC 4346 [25], making RFC 2246 [24] obsolete. As discussed in this book, there were quite a few cryptographic problems that needed to be solved in TLS 1.1. Most importantly, the emergence of padding oracle attacks—both in the symmetric and asymmetric settings—and some issues with the CBC mode of operation had to mitigated. In addition to TLS 1.1, the IETF TLS WG also released the first version of the DTLS protocol to secure UDP-based applications (i.e., DTLS 1.0) in Standards Track RFC 4347 [26].

After another two-year revision period, the TLS protocol version 1.2 (i.e., TLS 1.2) was specified in Standards Track RFC 5246 in 2008 [27]. This document not only made RFC 4346 obsolete, but also a few other RFC documents (most of them referring to TLS extensions). The biggest changes in this protocol version were related to the consolidation of TLS extensions, the specification of protocol parameters in registries maintained by the Internet Assigned Numbers Authority (IANA[19]), and the optional use on modes of operation for symmetric encryption that provide authenticated encryption. Four years after the release of TLS 1.2 (i.e., in January 2012), version 1.2 of the DTLS protocol was made available in Standards Track RFC 6347 [28].

In spite of the fact that TLS 1.2 had been around since 2008, there was no rush in migrating to this new protocol version. If servers supported this protocol version at all, most of them still supported SSL 3.0, TLS 1.0, and TLS 1.1—mainly for the sake of backward compatibility. The bottom line was that many attacks were published that exploited the fact that TLS 1.2 servers still supported elder protocol versions. Consequently, the community became aware of the fact that security alone does not provide enough incentive to migrate to newer protocol versions, and that a new protocol must provide added value beyond security. So when the IETF TLS WG started to design the next TLS protocol version, it also wanted to improve the efficiency of the protocol (in addition to its security). This led to major changes in the structure of the protocol, and these changes immediately led to a dispute about the correct name of this protocol version. While some people argued in favor of TLS 1.3, there were quite a few other people who opted for TLS 2.0, TLS 3.0, or TLS 4.0. In the end, the proponents of TLS 1.3 were able to assert themselves, and the official

19 The IANA is responsible for the global coordination of the domain name system (DNS) root, IP addressing, and other Internet Protocol resources.

name became TLS 1.3. But the big structural changes in the protocol also led to a situation in which many TLS proxy servers and middleboxes were overextended and broke when they had to handle TLS 1.3 traffic. So the designers of TLS 1.3 had to introduce a compatibility mode that makes TLS 1.3 traffic look like TLS 1.2 traffic. This was not nice from a theoretical viewpoint, but it was almost unavoidable from a practical viewpoint. In the end, it took ten years after the release of TLS 1.2 in 2008 that TLS 1.3 could be specified in Standards Track RFC 8447 [29]. This occurred in 2018, and it took another four years to incorporate the changes from TLS 1.3 into DTLS 1.3. In 2022, the respective protocol version was officially released in Standards Track RFC 9147 [30].

TLS 1.3 and DTLS 1.3 clearly mark the state of the art when it comes to transport layer security. But it is certainly not the end of the story, as the protocols continue to evolve in the future. One such area of evolution is, for example, related to a compact version of TLS and DTLS known as *compact TLS* (cTLS). The cTLS protocol is being specified within the IETF TLS WG (currently as an Internet-Draft[20]). It is intended to be logically equivalent to ordinary (D)TLS, but can be implemented more compactly and is therefore more suitable for space-constrained environments and applications, such as the ones in the realm of IoT (that may implement the profiles specified in [31]).

1.3 FINAL REMARKS

This chapter has prepared the ground for the topic of this book, and we are now ready to delve more deeply into the SSL/TLS protocols. These protocols are omnipresent on the internet today, and there are only a few competitors, such as the many versions[21] of the Secure Shell (SSH), the Noise protocol framework,[22] WireGuard,[23] and the Application Layer Transport Security (ALTS) that is yet similar to mutually authenticated TLS, but still kept as simple as possible to serve the needs of Googles production infrastructure. The TLS handshake protocol is also used in a new general-purpose transport layer protocol named QUIC[24] [32, 33]. All of these protocols are not further addressed in this book.

There are supposedly two major reasons for the tremendous success of the SSL/TLS protocols:

20 Use https://datatracker.ietf.org/doc/draft-ietf-tls-ctls/ to retrieve the latest status of work on cTLS.
21 This includes OpenSSH, available at https://www.openssh.com.
22 https://noiseprotocol.org.
23 https://www.wireguard.com.
24 The name was initially proposed as the acronym for "quick UDP internet connections." However, the name is no longer used as an acronym—at least in the realm of the IETF. Instead, it is the proper name of the protocol today.

- On one hand, they can be used to secure any application layer protocol that is layered on top of them. This means that any TCP-based application can potentially be secured with the SSL/TLS protocols. As explained later, there is even a possibility to secure any UDP-based application with the DTLS protocol.
- On the other hand, they can operate nearly transparently for users, meaning that users need not be aware of the fact that the SSL/TLS protocols are in place.[25] This simplifies the deployment of the protocols considerably.

The SSL/TLS protocols are cryptographic security protocols, meaning that they aim to provide security services with cryptographic means. In any security discussion, it is mandatory to nail down the threats model and to specify against whom one wants to protect oneself. In the cryptographic literature, the threats model that is most frequently used is the *Dolev-Yao model* [34]. It basically says that the (active) adversary is yet able to control the communications network used to transmit all messages but that he or she is not able to compromise the end systems. This basically means that the adversary can mount all kinds of (passive and active) attacks. Roughly speaking, a passive attack “attempts to learn or make use of information from the system but does not affect system resources,” whereas an active attack “attempts to alter system resources or affect their operation” [1]. Obviously, passive and active attacks can (and will) be combined to effectively invade a computing or networking environment. For example, a passive wiretapping attack can be used to eavesdrop on the authentication information that is transmitted in the clear (e.g., username and password), and this information can then be used to masquerade the user and to actively attack the system accordingly.

The SSL/TLS protocols have been designed to be secure in the Dolev-Yao model, but the model has its limitations and shortcomings, too. For example, many contemporary attacks are either based on malware or employ sophisticated techniques to spoof the user interface of the client system. Such attacks are outside the scope of the Dolev-Yao model, and hence the SSL/TLS protocols do not natively provide protection against them. In the future, it is possible and very likely that we will have to adapt and extend the threats model a little bit. This is a current research topic in network security, and its results will have a strong impact on the way the TLS protocol fits into the security landscape. The bottom line is that one always has to be careful when people make claims about the security of a protocol. One has to look behind the scenes and assess the threats model against which the claims are made. This is always important, and it is particularly important in the realm of the

25 The only place where user involvement is ultimately required is when the user must verify the server certificate. This is also one of the Achilles’ heels of SSL/TLS.

SSL/TLS protocols. Everything that is being said should be taken with a grain of salt, and this certainly also applies to the rest of this book.

References

[1] Shirey, R., “Internet Security Glossary, Version 2,” RFC 4949 (FYI 36), August 2007.

[2] ISO/IEC 7498-2, Information Processing Systems—Open Systems Interconnection Reference Model—Part 2: Security Architecture, 1989.

[3] Oppliger, R., “IT Security: In Search of the Holy Grail,” *Communications of the ACM,* Vol. 50, No. 2, February 2007, pp. 96–98.

[4] ITU X.800, Security Architecture for Open Systems Interconnection for CCITT Applications, 1991 (CCITT is the acronym of “Comité Consultatif International Téléphonique et Télégraphique,” which is the former name of the ITU).

[5] Zhou, J., *Non-Repudiation in Electronic Commerce*, Artech House, Norwood, MA, 2001.

[6] Oppliger, R., *Cryptography 101: From Theory to Practice*, Artech House, Norwood, MA, 2021.

[7] Ferraiolo, D.F., D.R. Kuhn, and R. Chandramouli, *Role-Based Access Controls*, 2nd edition, Artech House, Norwood, MA, 2007.

[8] Hu, V.C., D.F. Ferraiolo, and R. Chandramouli, *Attribute-Based Access Control*, Artech House, Norwood, MA, 2017.

[9] Oppliger, R., *Internet and Intranet Security*, 2nd edition, Artech House, Norwood, MA, 2002.

[10] Oppliger, R., *Security Technologies for the World Wide Web*, 2nd edition, Artech House, Norwood, MA, 2003.

[11] Frankel, S., *Demystifying the IPsec Puzzle*, Artech House, Norwood, MA, 2001.

[12] Oppliger, R., *End-to-End Encrypted Messaging*, Artech House, Norwood, MA, 2020.

[13] Saltzer, J.H., D.P. Reed, and D.D. Clark, “End-to-End Arguments in System Design,” *ACM Transactions on Computer Systems,* Vol. 2, No. 4, November 1984, pp. 277–288.

[14] Voydock, V., and S.T. Kent, “Security Mechanisms in High-Level Network Protocols,” *ACM Computing Surveys,* Vol. 15, 1983, pp. 135–171.

[15] Bossert, G., S. Cooper, and W. Drummond, “Considerations for Web Transaction Security,” RFC 2084, January 1997.

[16] Rescorla, E., and A. Schiffman, “Security Extensions For HTML,” RFC 2659, August 1999.

[17] Rescorla, E., and A. Schiffman, “The Secure HyperText Transfer Protocol,” RFC 2660, August 1999.

[18] Elgamal, T., “A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithm,” *IEEE Transactions on Information Theory,* IT-31(4), 1985, pp. 469–472.

[19] Freier, A., P. Karlton, and P. Kocher, “The Secure Sockets Layer (SSL) Protocol Version 3.0,” RFC 6101, August 2011.

[20] Turner, S., and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0," RFC 6176, March 2011.

[21] Barnes, R., et al., "Deprecating Secure Sockets Layer Version 3.0," RFC 7568, June 2015.

[22] Bradner, S., "The Internet Standards Process—Revision 3," RFC 2026 (BCP 9), October 1996.

[23] Schiller, J., "Strong Security Requirements for Internet Engineering Task Force Standard Protocols," RFC 3365 (BCP 61), August 2002.

[24] Dierks, T., and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, January 1999.

[25] Dierks, T., and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, April 2006.

[26] Rescorla, E., and N. Modadugu, "Datagram Transport Layer Security," RFC 4347, April 2006.

[27] Dierks, T., and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, August 2008.

[28] Rescorla, E., and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, January 2012.

[29] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8447, August 2018.

[30] Rescorla, E., H. Tschofenig, and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3," RFC 9147, April 2022.

[31] Tschofenig, H., and T. Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things," RFC 7925, July 2016.

[32] Iyengar, J., and M. Thomson (Eds.), "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021.

[33] Thomson, M., and S. Turner (Eds.), "Using TLS to Secure QUIC," RFC 9001, May 2021.

[34] Dolev, D., and A.C. Yao, "On the Security of Public Key Protocols," *Proceedings of the IEEE 22nd Annual Symposium on Foundations of Computer Science,* 1981, pp. 350–357.

Chapter 2

SSL Protocol

This chapter introduces, discusses, and puts into perspective the first transport layer security protocol[1] spelled out in the title of this book (i.e., the SSL protocol that is specified with retroactive effect in RFC 6101) [1].[2] More specifically, it provides an introduction in Section 2.1, overviews the SSL protocol and its subprotocols in Section 2.2, outlines the transcript of a protocol execution in Section 2.3, briefly analyzes its security in Section 2.4, and concludes with some final remarks in Section 2.5. The bottom line is that—due to the POODLE and RC4 NOMORE attacks—the SSL protocol should no longer be used. So you may wonder why we spend so much time on introducing and explaining in detail a now deprecated protocol. The answer is related to the fact that the TLS and DTLS protocols are very similar (and in many regards even identical) to the SSL protocol. So most things we say about the SSL protocol similarly apply to some versions of the (D)TLS protocols. This allows us to shorten the respective chapters considerably, and to put everything into the right perspective. The SSL protocol is really key to understanding the TLS and DTLS protocols—it somehow refers to their little brother.

1 As mentioned in Section 1.2, the SSL protocol does not, strictly speaking, operate at the transport layer. Instead, it operates at an intermediate layer between the transport layer and the application layer.

2 Note that this RFC is historic and was published in August 2011. Before this date, there was only a draft specification that could be used to serve as a reference. Also note that this chapter focuses on version 3.0 of the SSL protocol, and that its predecessors are only mentioned for the sake of completeness.

2.1 INTRODUCTION

Chapter 1 looked back into the 1990s and explained why Netscape Communications proposed SSL and how the SSL protocol has evolved in three versions—SSL 1.0, SSL 2.0, and SSL 3.0—to become what is currently known as the TLS protocol. Referring to the terminology introduced in Section 1.1, SSL is a client/server protocol that provides the following basic security services to the communicating peers:

- Authentication (both peer entity and data origin authentication) services;
- Connection confidentiality services;
- Connection integrity services (without recovery).

In spite of the fact that the SSL protocol uses public key cryptography (for authentication and key exchange), it does not provide nonrepudiation services—neither nonrepudiation with proof of origin nor nonrepudiation with proof of delivery—for data that is transmitted. This is because the data is authenticated with a MAC (instead of a digital signature) and all data is treated equally (i.e., there is no possibility to distinguish between data parts that need to be protected in terms of nonrepudiation services and some other parts). This is in sharp contrast to S-HTTP and XML signatures that can be used to digitally sign specific data parts.

As its name suggests, the SSL protocol is sockets-oriented, meaning that all or none of the data that is sent to or received from a network connection (identified by a so-called socket) is cryptographically protected in exactly the same way. This means that if data must be treated differently or nonrepudiation services are needed, then the application-layer protocol must take care of it (in the case of web traffic, this is HTTP). This, in turn, suggests that there is and will always be room for other—mostly complementary—security protocols in addition to SSL/TLS.

SSL can be best viewed as an intermediate layer between the transport and the application layer that serves two purposes:

- On one hand, it has to establish a secure (i.e., authentic and confidential) channel or connection between the communicating peers.
- On the other hand, it has to use this connection to securely transmit higher-layer protocol data from the sender to the recipient. It therefore fragments the data into manageable pieces (called fragments) and processes each fragment individually. More specifically, each fragment may be compressed, authenticated, encrypted,[3] prepended with a header, and transmitted to the recipient.

3 Depending on the mode of operation, encryption may also include padding.

Each data fragment prepared this way is sent in a distinct SSL record.[4] On the recipient's side, the SSL records are decrypted,[5] authenticated, and decompressed, before the data is reassembled and delivered to the higher-layer—typically the application-layer—protocol.

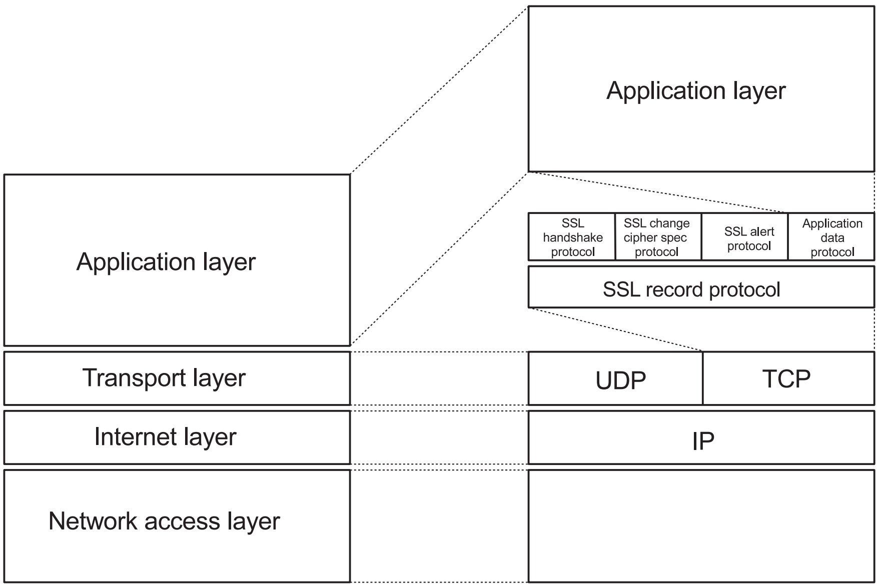

Figure 2.1 The SSL with its (sub)layers and (sub)protocols.

The placement of the SSL layer is illustrated in Figure 2.1. In accordance to the two functions mentioned above, it consists of two sublayers and a few subprotocols.

- The lower sublayer is stacked on top of some connection-oriented and reliable transport layer protocol, such as the Transmission Control Protocol (TCP) in the case of the TCP/IP protocol suite.[6] This layer basically comprises the *SSL*

4 To be precise, an SSL record consists of four fields: a type field, a version field, a length field, and a fragment field. The fragment field, in turn, comprises the higher-layer protocol data. The exact format of an SSL record is addressed below.
5 Decryption may also include padding verification.
6 This is in contrast to the DTLS protocol that is stacked on top of UDP. The DTLS protocol is further addressed in Chapter 4.

record protocol that is mainly used for the second function mentioned above, namely the secure transmission of the higher-layer protocol data.

- The higher sublayer is stacked on top of the SSL record protocol and comprises four subprotocols.
  - The *SSL handshake protocol* is the core subprotocol of SSL. It is used for the first function mentioned above, namely the establishment of a secure connection. More specifically, it allows the communicating peers to authenticate each other, to exchange some keying material, and to negotiate a cipher suite and a compression method. As their names suggest, the cipher suite is used to cryptographically protect data in terms of authenticity, integrity, and confidentiality, whereas the compression method is used to optionally compress data.[7]
  - The *SSL change cipher spec protocol* allows the communicating peers to signal to each other a cipher spec change (i.e., a change in the ciphering strategy and the way data is cryptographically protected). While the SSL handshake protocol is used to negotiate the parameters of a secure connection, the SSL change cipher spec protocol is used to put these parameters in place and make them effective. As discussed later, it is questionable whether a separate protocol is required to serve this purpose, and in the current version of the TLS protocol (i.e., TLS 1.3) the change cipher spec protocol does no longer exist (or exists only for the sake of backwards compatibility).
  - The *SSL alert protocol* allows the communicating peers to signal indicators of potential problems and—as its name suggests—send respective alert messages to each other.
  - Together with the SSL record protocol, the *SSL application data protocol* is used for the second function mentioned above, namely the secure transmission of application data. As such, it is the workhorse of SSL. It takes higher-layer—typically application-layer—data and feeds it into the SSL record protocol for cryptographic protection and optional compression.

In spite of the fact that SSL consists of several subprotocols, we use the term *SSL protocol* to refer to all of them collectively. When we refer to a specific subprotocol, we usually employ its full name (but sometimes also leave the prefix "sub" aside).

7 Note that recent attacks (Section A.6) have shown that compressing data in SSL/TLS is dangerous and should be avoided in the first place.

Like most protocols layered on top of TCP, the SSL protocol is self-delimiting, meaning that it can autonomously (i.e., without the assistance of TCP) determine the beginning and ending of an SSL record that is transported in a TCP segment, as well as the beginning and ending of each SSL message that may be included in a record. The SSL protocol therefore employs multiple length fields. More specifically, each SSL record is tagged with a (record) length field that refers to the length of the entire record, whereas each SSL message included in such a record is additionally tagged with a (message) length field that refers to the length of this particular message. Note again and keep in mind that multiple SSL messages may be transported in a single SSL record.

A major advantage of the SSL protocol is that it is application-layer protocol-independent, meaning that any TCP-based application protocol can be layered on top of it to provide the basic security services mentioned above. In order to accommodate connections from clients that do not support SSL, servers should be prepared to accept both secure and nonsecure versions of any given application-layer protocol. In general, there are two strategies to achieve this goal:

- In a *separate port strategy*, two different port numbers are assigned to the secure and nonsecure versions of the application-layer protocol. This suggests that the server has to listen on both the original (unsecure) port and the new (secure) port. SSL can then be invoked automatically and transparently for any connection that arrives on the secure port.
- In contrast, in an *upward negotiation strategy*, a single port is used for either version of the application-layer protocol. This protocol, in turn, must be extended to support a message indicating that one side would like to upgrade to SSL. If the other side agrees, then SSL can be invoked and a secure channel can be established for the application-layer protocol.

Both strategies have advantages and disadvantages, and hence, in principle, either strategy can be pursued. For example, in the case of HTTP, the upward negotiation strategy is used in RFC 2817 [2],[8] whereas the separate port strategy is used in the more widely deployed RFC 2818 [3]. Both RFC documents can be characterized as follows:

- RFC 2817 explains how to use the upgrade mechanism of HTTP/1.1 to initiate SSL/TLS over an existing TCP connection. This mechanism can be invoked by either the client or the server, and upgrading can be optional or mandatory. In either case, the HTTP/1.1 upgrade header must be employed. This is a hop-by-hop header, and hence care must be taken to upgrade across all proxy

8 Note that this RFC is written for the TLS protocol, but the same mechanism applies to the SSL protocol, as well.

servers in use. The bottom line is that the upgrade mechanism of HTTP/1.1 allows unsecured and secured HTTP traffic to share the same server port (typically port 80).

- In contrast, RFC 2818 elaborates on using a different server port for the SSL/TLS-secured HTTP traffic (typically port 443). This is comparably simple and straightforward (also because no protocol support is required), and hence it is more widely deployed in the field.

In general, it is up to the designer of an application-layer protocol to make a choice between the separate port and upward negotiation strategies. Historically, most protocol designers have made a choice in favor of the separate port strategy. For example, until the SSL 3.0 protocol specification was released in 1996, the IANA had already reserved the port number 443 for use by HTTP over SSL (`https`), and was about to reserve the port numbers 465 for use by the simple mail transfer protocol (SMTP) over SSL (`ssmtp`) and 563 for the network news transfer protocol (NNTP) over SSL (`snntp`). Later on, the IANA decided to consistently append the letter "s" after the protocol name, so `snntp` became `nntps`.

Table 2.1
Port Numbers Reserved for Application Protocols Layered over SSL/TLS

Protocol	Description	Port #
`nsiiops`	IIOP Name Service over SSL/TLS	261
`https`	HTTP over SSL/TLS	443
`nntps`	NNTP over SSL/TLS	563
`ldaps`	LDAP over SSL/TLS	636
`ftps-data`	FTP Data over SSL/TLS	989
`ftps`	FTP Control over SSL/TLS	990
`telnets`	Telnet over SSL/TLS	992
`imaps`	IMAP4 over SSL/TLS	993
`ircs`	IRC over SSL/TLS	994
`pop3s`	POP3 over SSL/TLS	995
`tftps`	TFTP over SSL/TLS	3713
`sip-tls`	SIP over SSL/TLS	5061
...	...	...

Today, there are several port numbers reserved by the IANA for application protocols layered on top of SSL/TLS.[9] The most important examples are summarized in Table 2.1—together with their respective port number. Among these examples, `ldaps`, `ftps` (and `ftps-data`), `imaps`, and `pop3s` are particularly

9 https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

important and widely used in the field. In contrast, there are only a few application layer protocols that implement an upward negotiation strategy. We have mentioned the HTTP/1.1 upgrade mechanism above. However, by far the most prominent example is SMTP with its STARTTLS feature specified in RFC 2487 [4] that invokes SSL/TLS to secure the messages that are exchanged between SMTP servers. For the sake of completeness, we note that STARTTLS is based on the SMTP extensions mechanism originally specified in RFC 1869 [5], and that—like other upward negotiation strategies and respective mechanisms—it is susceptible to downgrade attacks, where an attacker may force a victim to use an insecure connection.[10]

The separate port strategy has the disadvantage that it effectively halves the number of available ports on the server side (because two ports are required for each application-layer protocol). During an IETF meeting in 1997, the area directors and the IESG therefore affirmed that the upward negotiation strategy would be the way to go and that the separate port strategy should be deprecated. This is also why the respective RFC 2818 is only informational. In reality, however, we observe the opposite trend—at least in the realm of HTTP. In spite of the fact that RFC 2817 (specifying an upgrade mechanism for HTTP/1.1) has been available for almost a decade and has been submitted to the IETF standards track, there has hardly been interest in implementing alternatives to port 443. This may change for other—especially future—application-layer protocols. For HTTP, however, implementing the separate port strategy and using port 443 is still the preferred choice, and this is not likely to change anytime soon.[11]

The SSL protocol was designed with interoperability in mind. This means that the protocol is intended to make the probability that two independent SSL implementations can interoperate as large as possible. As such, the design of the SSL protocol is simpler and more straightforward than the design of many other security protocols, including, for example, IPsec/IKE and even some versions of the TLS protocol (as explained in Chapter 3). However, the simple and straightforward design of the SSL protocol is also slightly stashed away by the fact that the specification of SSL 3.0 and the RFC documents that specify the various versions of the TLS protocol use a special presentation language. For the purpose of this book, we neither introduce this language nor do we actually use it. Instead, we use plain English text to describe the protocols with only a few bit-level details where necessary and

10 https://www.usenix.org/conference/usenixsecurity21/presentation/poddebniak.

11 The upward negotiation strategy has two major disadvantages: (1) As mentioned above, it is susceptible to downgrade attacks, and (2) it is more easily subject to misconfiguration, meaning that data is sent over an unprotected channel without being noticed. Against this background, the IETF has officially changed its preference with regard to HTTP in Section 3.2 of RFC 9325 (Chapter 7).

appropriate. We hope that this makes the book more readable than the respective RFC documents.

The SSL protocol (and its TLS and DTLS successors) are block-oriented with a block size of one byte (i.e., eight bits). This means that multiple-byte values can be seen as a concatenation of bytes. Such a concatenation is written from left to right and from top to bottom, but keep in mind that the resulting strings are just byte strings transmitted over the wire. The byte ordering—also known as *endianness*—for multiple-byte values is the *network byte order* or *big endian* format. This means that the first-appearing bytes refer to the higher values. For example, the sequence of hexadecimally written bytes 0x01, 0x02, 0x03, and 0x04 (i.e., 0x01020304) refers to the following decimal value:

$$\begin{aligned} 1 \cdot 16^6 + 2 \cdot 16^4 + 3 \cdot 16^2 + 4 \cdot 16^0 &= 16,777,216 + 131,072 + 768 + 4 \\ &= 16,909,060 \end{aligned}$$

The aim of the SSL protocol is to securely transmit application-layer data between communicating peers. The protocol therefore establishes and employs SSL connections and SSL sessions. Both terms are required to properly understand the working principles of the SSL protocol. Unfortunately, the terms, described as follows, are not consistently used in the literature.

- An *SSL connection* is used to actually transmit data between two communicating peers, typically a client and a server, in some cryptographically protected and optionally compressed form. Hence, there are some cryptographic (and other) parameters that must be put in place and applied to the data transmitted over the SSL connection. One or several SSL connections may be associated with an SSL session.
- Similar to an IPsec/IKE security association,[12] an *SSL session* refers to an association between two communicating peers that is established by the SSL handshake protocol. The SSL session defines a set of cryptographic (and other) parameters that are commonly used by the SSL connections associated with the session to cryptographically protect and optionally compress data.

12 There are still a few conceptual and subtle differences between an IPsec/IKE security association and an SSL session: (1) An IPsec/IKE security association is unidirectional, whereas an SSL session is bidirectional. (2) An IPsec/IKE security association identifier—also known as a *security parameter index* (SPI)—is intentionally kept as short as 32 bits (as it is being transmitted in each IP packet), whereas the length of an SSL session identifier does not really matter and need not be minimized. (3) IPsec/IKE do not really represent client/server protocols, mainly because clients and servers do not really exist at the internet layer (instead the terms *initiator* and *responder* are used in this context). In contrast, the SSL protocol in general and the SSL handshake protocol in particular represent true client/server protocols.

Hence, an SSL session can be shared among multiple SSL connections, and SSL sessions are primarily used to avoid the necessity to perform a computationally expensive negotiation of new cryptographic parameters for each connection individually.

Between a pair of entities, there may be multiple SSL connections in place. In theory, there may also coexist multiple simultaneous SSL sessions, but this is seldom used in practice.

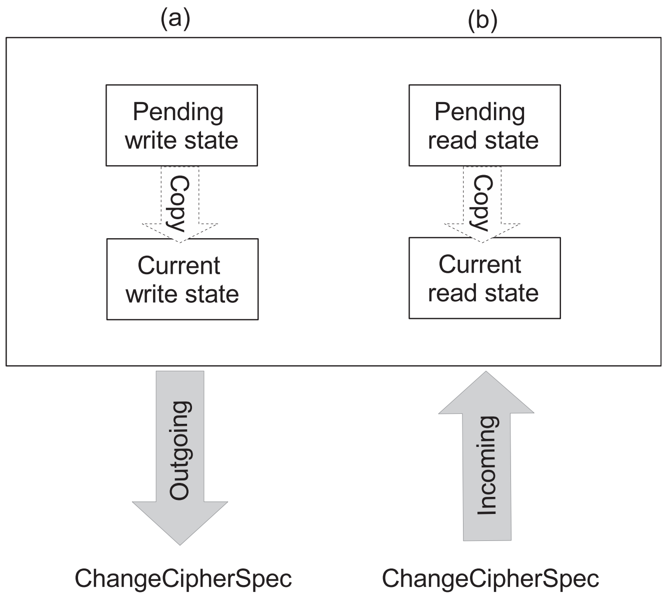

Figure 2.2 The SSL state machine.

SSL sessions and connections are stateful, meaning that the client and the server must keep some state information. It is the responsibility of the SSL handshake protocol to establish and coordinate (as well as possibly synchronize) the state on the client and server sides, thereby allowing the SSL state machines on either side to operate consistently. Logically, the state is represented twice, once as the *current state* and once as the *pending state*. Also, separate *read* and *write* states are maintained. So there is a total of four states that need to be managed on either side.

As illustrated in Figure 2.2, the transition from a pending to a current state occurs when a CHANGECIPHERSPEC message is sent or received during an SSL handshake. The respective rules are as follows:

- If an entity (i.e., client or server) sends a CHANGECIPHERSPEC message, then it copies the pending write state into the current write state. The read states remain unchanged. This case refers to (a) in Figure 2.2 (on the left).
- If an entity receives a CHANGECIPHERSPEC message, then it copies the pending read state into the current read state. In this case, the write states remain unchanged. This case refers to (b) in Figure 2.2 (on the right).

When the SSL handshake negotiation is complete, the client and server exchange CHANGECIPHERSPEC messages,[13] and hence the pending states are copied into the current states. This means that they can now use the newly agreed-upon cryptographic (and other) parameters. As outlined below, the FINISHED messages are then the first SSL handshake messages that are protected according to the new parameters.

Table 2.2
SSL Session State Elements

`session identifier`	Arbitrary byte sequence chosen by the server to identify an active or resumable session state (the maximum length is 32 bytes).
`peer certificate`	X.509v3 certificate of the peer (if available).
`compression method`	Data compression algorithm used (prior to encryption).
`cipher spec`	Data encryption and MAC algorithms used (together with cryptographic parameters, such as the length of the hash values).
`master secret`	48-byte secret that is shared between the client and the server.
`is resumable`	Flag indicating whether the SSL session is resumable, meaning that it can be used to initiate new connections.

For each session and connection, the SSL state machine must hold some information elements. The respective session state and connection state elements are summarized in Tables 2.2 and 2.3. We revisit some of these elements when we go through the SSL protocol in more detail. At this point in time, we only want to introduce the elements together with their proper names.

Where appropriate and available (in the 1990s when the protocol was originally designed), the developers of the SSL protocol tried to follow standards. For

13 As mentioned above and further addressed below, the CHANGECIPHERSPEC messages are not part of the SSL handshake protocol. Instead, they are the single messages that are exchanged as part of the SSL change cipher spec protocol.

Table 2.3
SSL Connection State Elements

`server and client random`	Byte sequences that are chosen by the server and client for each connection.
`server write MAC key`	Secret used in MAC operations on data written by the server.
`client write MAC key`	Secret used in MAC operations on data written by the client.
`server write key`	Key used for data encrypted by the server and decrypted by the client.
`client write key`	Key used for data encrypted by the client and decrypted by the server.
`initialization vectors`	If a block cipher in CBC mode is used for data encryption, then an IV must be maintained for each key. This field is first initialized by the SSL handshake protocol. Afterward, the final ciphertext block from each SSL record is preserved to serve as IV for the next record.
`sequence numbers`	SSL message authentication employs sequence numbers. This means that the client and server must maintain a sequence number for the messages that are transmitted or received on a particular connection. Each sequence number is 64 bits long and ranges from 0 to $2^{64}-1$. It is set to zero whenever a CHANGECIPHERSPEC message is sent or received. Since it cannot wrap, a new connection must be negotiated when the number reaches $2^{64}-1$.

example, RSA signatures are always performed using public key cryptography standard (PKCS) #1 block type 1,[14] whereas RSA encryption employs PKCS #1 block type 2. The PKCS #1 version that was relevant when the SSL protocol was specified was 1.5 [6].[15] As discussed in Section 2.4, PKCS #1 version 1.5 was shown to be susceptible to an adaptive chosen-ciphertext attack (CCA2) in 1998. At the same time, PKCS #1 was updated to version 2.0 [7], but the old version was not deprecated and prevailed. Later on, more subtle vulnerabilities and respective attacks made it necessary to update PKCS #1 again to version 2.1 [8]. The full story is provided in Section A.1. We now continue with a more detailed overview and explanation of the SSL protocol and its subprotocols.

14 There is another block type 0 specified in PKCS #1. This type, however, is not used in the SSL protocol specification.

15 Note that PKCS #1 had been published in a document series of RSA Laboratories, before it was specified in an informational RFC document in 1998.

2.2 PROTOCOLS

As mentioned above, the SSL protocols comprise the SSL record protocol, the SSL handshake protocol, the SSL change cipher spec protocol, the SSL alert protocol, and the SSL application data protocol. We walk through each of these protocols in Sections 2.2.1–2.2.5.

2.2.1 SSL Record Protocol

We already said that the SSL record protocol is used for the transmission of higher-layer protocol data, and that it therefore fragments the data into manageable pieces—called fragments—that are processed individually. More specifically, each fragment is optionally compressed and cryptographically protected according to the compression method and cipher spec of the SSL session state (Table 2.2) and the cryptographic parameters and elements of the SSL connection state (Table 2.3). The result is sent to the recipient in the fragment field of an SSL record.

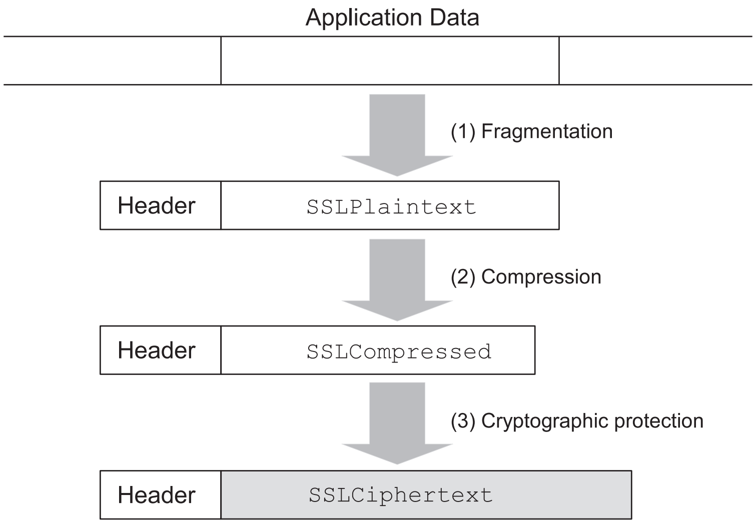

Figure 2.3 The SSL record processing (overview).

The SSL record processing is overviewed in Figure 2.3. It consists of three steps: *fragmentation*, *compression*, and *cryptographic protection* (including message authentication and encryption). According to the SSL protocol specification, the data structure that results after fragmentation is called `SSLPlaintext`, the one after compression is called `SSLCompressed`, and the one after cryptographic protection is called `SSLCiphertext`. Each of these data structures is embedded in the fragment field of an SSL record. Also, it is prepended with a 5-byte header that comprises the following fields (not separately illustrated in Figure 2.3):

- A 1-byte *type* field that refers to the higher-layer SSL protocol. There are four predefined values:
  - 20 (0x14) refers to the SSL change cipher spec protocol;
  - 21 (0x15) refers to the SSL alert protocol;
  - 22 (0x16) refers to the SSL handshake protocol;
  - 23 (0x17) refers to the SSL application data protocol.
- A 2-byte *version* field that refers to the version of the SSL protocol in use. It consists of a major and a minor version number.[16] Hence, the value of SSL 3.0 is 0x0300, where byte 0x03 refers to the major version number and byte 0x00 refers to the minor version number. This value can also be written in decimal notation with a comma separating the major and minor version numbers. So 0x0300 corresponds to 3,0. As we will see later, TLS 1.0 refers to 0x0301 (3,1), TLS 1.1 to 0x0302 (3,2), TLS 1.2 to 0x0303 (3,3), and TLS 1.3 to 0x0304 (3,4). Hence, in all versions of the SSL/TLS protocols, the major version number is 3.
- A 2-byte *length* field that refers to the byte-length of the higher-layer protocol messages that are transmitted in the fragment part of the SSL record (remember that multiple higher-layer protocol messages that belong to the same type can be packed in a single SSL record). The length field comprises two bytes, so in theory an SSL record can be $2^{16} - 1 = 65{,}535$ bytes long. According to the SSL protocol specification, however, the maximum record length should not exceed $2^{14} = 16{,}384$ bytes plus some expansion from compression and protection,[17] but there are always implementations that do not follow this recommendation and use longer records.

16 Remember from Section 1.2 that the PCT protocol's record format was compatible with that of the SSL protocol, and that in the case of PCT the most significant bit of the protocol version field was set to one.

17 The length of this expansion depends on the protocol version. Until TLS 1.2, it can be up to 2,048 bytes (though typically only 16 bytes are used), but in TLS 1.3, it is reduced to 256 bytes.

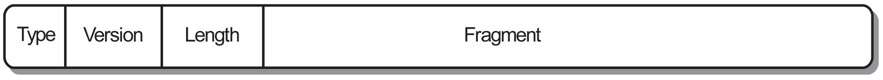

Figure 2.4 The outline of an SSL record.

The resulting outline of an SSL record is illustrated in Figure 2.4. Again note that the fragment may comprise a `SSLPlaintext`, a `SSLCompressed`, or a `SSLCiphertext` data structure. The SSL record that comprises the `SSLCiphertext` data structure in the fragment field is the one that is actually transmitted to the recipient. Let us briefly go through the three steps of SSL record processing next.

2.2.1.1 Fragmentation

In the first step, the SSL record protocol fragments the higher-layer protocol data—typically application-layer data—into blocks of at most $2^{14} = 16{,}384$ bytes. Each such block is then packed into the fragment field of an SSL record (or `SSLPlaintext` data structure, respectively), and the record header is completed with appropriate values for the type, version, and length fields.

2.2.1.2 Compression

In the second step, the SSL record protocol may optionally compress the fragment field of the `SSLPlaintext` structure and write it into the fragment field of the `SSLCompressed` structure. Whether compression takes place or not depends on the compression method specified for the SSL session. In the case of SSL 3.0, it is initially set to null, so no compression is usually invoked by default.

Before we delve more deeply into the topic of using compression, we start with the following observation: If data needs to be compressed and encrypted, then the order of the operations matters. In fact, the only order that makes sense is first compress and then encrypt. If data is first encrypted, then compression does not make a lot of sense, because the resulting ciphertext looks like random data and cannot be compressed anymore. So any protocol that requires data to be compressed and encrypted must ensure that compression is invoked first. This is why compression is addressed in the SSL protocol. Once data is encrypted, it cannot be compressed anymore—even not on lower layers in the protocol stack. So the designers of the SSL protocol made a reasonable decision when they enabled the SSL record protocol to compress data before it is encrypted.

Today, the situation is more involved and support for compression in the SSL record protocol is known to be a dual-edged sword: On the one hand, support for compression makes sense (due to the line of argumentation given above), but on the other hand, the combined use of compression and encryption is also known to be dangerous, as it introduces new vulnerabilities that may be exploited by some distinct attacks (Section A.6). So the use of compression has changed several times. In SSL 3.0, for example, it was theoretically possible to invoke compression, but nobody did so. The only requirement was that the compression is lossless and does not increase the length of the fragment field by more than 1,024 bytes.[18] However, no compression method other than null was defined for SSL 3.0. This has since changed, and a few compression methods have been specified for TLS 1.2 (Section 3.4.5). Due to the compression-related attacks mentioned above, support for compression has again disappeared. Today, most security practitioners recommend not to invoke compression when using SSL (or TLS) in the first place, and support for compression has even been abandoned completely in TLS 1.3. This means that the state of the art in protocol design prohibits the use of compression in combination with encryption.

In either case (i.e., independent from whether data is compressed or not), the output of this step is an `SSLCompressed` data structure with distinct type, version, length, and fragment fields. The type and version fields are inherited from the `SSLPlaintext` structure. If the SSL protocol uses null compression, then the compression method represents the identity operation, and the length and fragment fields remain the same. Otherwise (i.e., if compression is invoked), then the fragment field changes and the respective length field carries a different (typically smaller) value. But again, this is purely hypothetical and nobody is using a compression method other than null today.

2.2.1.3 Cryptographic Protection

In the third and final step, the SSL record protocol applies the cryptographic protection defined in the cipher spec of the SSL session state to the `SSLCompressed` data structure. According to Table 2.2, a *cipher spec* refers to a pair of algorithms that are used to cryptographically protect data (i.e., a data encryption and a MAC algorithm), but it does not comprise a key exchange algorithm. The key exchange algorithm is used to establish an SSL session and a respective master secret (that represents another SSL session state element). It is not part of the cipher spec. In order to refer to a cipher spec and a key exchange algorithm, the term *cipher suite* is used. When profiling the use of SSL/TLS and making respective recommendations, people thus usually refer to cipher suites.

18 Of course, one hopes that compression shrinks rather than expands the fragment. However, for very short fragments, it is possible, because of formatting conventions, that the compression method actually provides output that is longer than the input.

Table 2.4
SSL Cipher Suites*

Name	Key Exchange	Cipher	Hash
SSL_NULL_WITH_NULL_NULL	NULL	NULL	NULL
SSL_RSA_WITH_NULL_MD5	RSA	NULL	MD5
SSL_RSA_WITH_NULL_SHA	RSA	NULL	SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5	RSA_EXPORT	RC4_40	MD5
SSL_RSA_WITH_RC4_128_MD5	RSA	RC4_128	MD5
SSL_RSA_WITH_RC4_128_SHA	RSA	RC4_128	SHA
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSA_EXPORT	RC2_CBC_40	MD5
SSL_RSA_WITH_IDEA_CBC_SHA	RSA	IDEA_CBC	SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA_EXPORT	DES40_CBC	SHA
SSL_RSA_WITH_DES_CBC_SHA	RSA	DES_CBC	SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA	RSA	3DES_EDE_CBC	SHA
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	DH_DSS_EXPORT	DES40_CBC	SHA
SSL_DH_DSS_WITH_DES_CBC_SHA	DH_DSS	DES_CBC	SHA
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH_DSS	3DES_EDE_CBC	SHA
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	DH_RSA_EXPORT	DES40_CBC	SHA
SSL_DH_RSA_WITH_DES_CBC_SHA	DH_RSA	DES_CBC	SHA
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH_RSA	3DES_EDE_CBC	SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	DHE_DSS_EXPORT	DES40_CBC	SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA	DHE_DSS	DES_CBC	SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE_DSS	3DES_EDE_CBC	SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	DHE_RSA_EXPORT	DES40_CBC	SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA	DHE_RSA	DES_CBC	SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE_RSA	3DES_EDE_CBC	SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	DH_anon_EXPORT	RC4_40	MD5
SSL_DH_anon_WITH_RC4_128_MD5	DH_anon	RC4_128	MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	DH_anon_EXPORT	DES40_CBC	SHA
SSL_DH_anon_WITH_DES_CBC_SHA	DH_anon	DES_CBC	SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	DH_anon	3DES_EDE_CBC	SHA
SSL_FORTEZZA_KEA_WITH_NULL_SHA	FORTEZZA_KEA	NULL	SHA
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA	FORTEZZA_KEA	FORTEZZA_CBC	SHA
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA	FORTEZZA_KEA	RC4_128	SHA

* According to [1].

In the case of SSL 3.0, the protocol specification comes along with 31 predefined cipher suites that are summarized in Appendix C of [1]. For the sake of completeness, they are also itemized in Table 2.4. The cipher suites written in italics used to be exportable from the United States.[19] In fact, they were exportable only if the length of the public key was not longer than 512 bits, and the key length of the block cipher was not longer than 40 bits. The names of the cipher suites are shown in the first column of Table 2.4. In the remaining three columns, each cipher suite is decomposed into a key exchange algorithm, a cipher (i.e., symmetric encryption system), and a cryptographic hash function. For example, SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA refers to the cipher suite that

19 This criterion was important until the end of the 1990s.

comprises an RSA-authenticated fixed Diffie-Hellman key exchange, Triple DES (3DES) in CBC mode for encryption,[20] and SHA-1 for message authentication. Each cipher suite is encoded in two bytes: the first byte is 0x00 and the second byte refers to the hexadecimal representation of the cipher suite number as it occurs in Table 2.4 (starting with 0x00). This is partly in line with the encoding of the cipher suites specified for TLS (Appendix B). There are a few differences though. For example, the FORTEZZA key exchange algorithm KEA is no longer supported in TLS.[21] So the respective cipher suites from Table 2.4 do no longer exist, and the respective places are now occupied by some cipher suites.

Let us raise a terminology issue that applies to the entire book: While the original specifications of the SSL/TLS protocols used the acronym DSS (standing for "digital signature standard") universally (i.e., to refer to the digital signature algorithm (DSA) and the respective National Institute of Standards and Technology (NIST) standard), the more recent specifications of the TLS protocol make a distinction and use the acronym DSA to refer to the algorithm and the acronym DSS to refer to the NIST Federal Information Processing Standard (FIPS) PUB 186-4 that represents the standard. Note, however, that the DSS as it is available today is broader and comprises other algorithms than only the DSA (e.g., RSA and ECDSA). The distinction between DSA and DSS is made explicit in the specification of TLS 1.3. In this book, we try to adopt it and also make a distinction whenever needed.

There is always an active cipher suite. It is initially set to SSL_NULL_WITH _NULL_NULL, which does not provide any security (in terms of cryptographic protection) at all. In fact, this cipher suite refers to no key exchange (because no key exchange is needed in the first place), the identity operation for encryption, and no message authentication (referring to a MAC size of zero). It basically leads to a situation in which the fragment fields of the `SSLCompressed` and `SSLCiphertext` structures are identical. The SSL_NULL_WITH _NULL_NULL cipher suite is the first item in Table 2.4. After the execution of the SSL handshake protocol, it is possible and very likely that another cipher suite is put in place, one that is able to cryptographically protect the record.

If cryptographic protection comprises message authentication and encryption, then one of the first questions that pops up is related to the order of the respective operations. In theory, there are three approaches:

20 We ignore the acronym EDE here. It basically means that the 3DES cipher applies an encryption, a decryption, and another encryption operation in this order. So the letters "E" and "D" refer to encryption and decryption.

21 The FORTEZZA key exchange algorithm (KEA) dates back to the 1990s, when the U.S. government tried to deploy a key escrow system using a cipher named Skipjack. The FORTEZZA KEA was declassified in 1998. Because the original SSL 3.0 specification was released in 1996, the details of the FORTEZZA KEA could not be included in the first place. Instead, the FORTEZZA KEA was treated as a black box in the specification of SSL 3.0.

1. In the *authenticate-then-encrypt* (abbreviated AtE) approach, the message is first authenticated (by appending a MAC) and then encrypted. In this case, the encryption includes the MAC that is appended to the message for authentication.
2. In the *encrypt-then-authenticate* (abbreviated EtA) approach, the message is first encrypted and then authenticated. In this case, the encryption does not include the MAC.
3. In the *encrypt-and-authenticate* (abbreviated E&A) approach, the message is encrypted and authenticated simultaneously, meaning that the pair consisting of a ciphertext and a MAC is sent to the recipient.

Different internet security protocols follow different approaches. IPsec, for example, follows the EtA approach, whereas SSH follows the E&A approach. In the case of the SSL protocol, the developers of the original protocol followed the AtE approach. All of these protocols were developed in the 1990s, and at this time it was not yet clear what approach is best from a security perspective. It was not until 2001 that it was finally shown that EtA is the generically secure method of combining message authentication and encryption [9, 10]. So if one designed the SSL protocol today, then one would certainly follow the EtA approach or combine authentication and encryption in what is known as *authenticated encryption* (AE) or *authenticated encryption with associated data* (AEAD). However, this is clearly not the case here, and changing the order of the message authentication and encryption operations in retrospect is difficult (to say the least). So people have stayed with the AtE approach even for the first versions of the TLS protocol. This should not be a problem though, because it was also shown that—at least in theory and if properly implemented—AtE is secure if a block cipher in CBC mode or a stream cipher is used for encryption. So people have thought that cipher suites that comprise either a block cipher in CBC mode or a stream cipher are equally secure—even if AtE is used instead of EtA. Unfortunately, this result only applies in theory. In practice, it has been shown that there are several possibilities to attack an AtE implementation, and padding oracle attacks bear witness to this fact (Section A.2). This is why there is an `encrypt_then_mac` extension (Section C.2.19) for TLS nowadays. If both entities support the extension, then they may switch from AtE to EtA for a particular session or connection. Unfortunately, the `encrypt_then_mac` extension has never seen widespread use in the field, so the issue has remained somehow unsolved until TLS 1.3 (that mandates the use of AEAD).

Let us now have a closer look at key exchange, message authentication, and encryption as defined for SSL 3.0.

Key Exchange

The SSL protocol employs secret key cryptography for message authentication and bulk data encryption. However, before such cryptographic techniques can be invoked, some keying material must be established between the client and the server. In the case of SSL, this material is derived from a premaster secret (that is called `pre_master_secret` in the SSL protocol specification).[22] There are three key exchange algorithms that can be used to establish such a premaster secret: RSA, Diffie-Hellman, and FORTEZZA. Some of these algorithms combine a key exchange with peer entity authentication and hence actually refer to an authenticated key exchange. To make this distinction explicit, a key exchange without peer entity authentication is called *anonymous* here. Let us have a closer look at the possibilities SSL provides to establish a premaster secret.

- If RSA is used for key exchange, then the client generates a premaster secret, encrypts it with the server's public key, and sends the resulting ciphertext to the server. The server's public key, in turn, can either be static (i.e., long-term) and retrieved from a public key certificate, or ephemeral (i.e., short-term) and provided for a particular key exchange. As outlined in Sections 2.2.2.4 and 2.2.2.5, the situation was slightly more involved, due to the fact that an RSA key exchange could also be exportable in the past (if sufficiently short keys were used). In either case, the server must use a private key to decrypt the premaster secret.
- If Diffie-Hellman is used for key exchange, then a Diffie-Hellman key exchange is performed and the resulting Diffie-Hellman value (without leading zero bytes) yields the premaster secret. Again, there are some subtleties that must be considered when the Diffie-Hellman key exchange is to be exportable (and these subtleties are also addressed in Sections 2.2.2.4 and 2.2.2.5). Except from that, the SSL protocol provides support for three versions of the Diffie-Hellman key exchange.
  - In a *fixed Diffie-Hellman key exchange* (DH), the parameters needed for the Diffie-Hellman key exchange are fixed[23] and part of the respective public key certificates. This applies to the server, meaning that the Diffie-Hellman group, the generator, and the public key are included in the server certificate. But it may also apply to the client: If client

22 The length of the premaster secret is typically 48 bytes. In the case of a Diffie-Hellman key exchange, however, its length depends on the cyclic group in use.

23 While the SSL protocol specification uses the term "fixed," another term that is frequently used in the literature is "static" here. Fixed Diffie-Hellman can thus also be called static Diffie-Hellman.

authentication is required, then the client certificate also includes a public key. Otherwise (i.e., if client authentication is not required), then the client uses the Diffie-Hellman group and generator provided by the server certificate, generates a public key pair on the fly, and provides the public key in an SSL handshake message (i.e., CLIENTKEYEXCHANGE message as outlined in Section 2.2.2.9).

– In an *ephemeral Diffie-Hellman key exchange* (DHE), the parameters needed for the Diffie-Hellman key exchange are not fixed and hence not part of any public key certificate. Instead, the parameters are generated on the fly and provided in appropriate SSL handshake messages (i.e., SERVERKEYEXCHANGE and CLIENTKEYEXCHANGE messages as outlined in Sections 2.2.2.5 and 2.2.2.9). Since the early days of the SSL protocol, people have been arguing about the pros and cons of using standard groups for the Diffie-Hellman key exchange.[24] The general trend is to go to standard groups. This is true for the newer versions of the TLS protocol, and it is particularly true for elliptic curves and respective groups. Anyway, the parameters that are used in DHE are not authenticated, and hence they must be authenticated in some way to provide an authenticated key exchange. Usually, the parameters are digitally signed with the sender's private (RSA or DSS) signing key, and the respective public key needed for verification is then provided in an appropriate public key certificate.

– An *anonymous Diffie-Hellman key exchange* (DH_anon) is similar to a DHE, but it lacks the authentication step that turns the Diffie-Hellman key exchange into an authenticated key exchange. This means that either participant of a DH_anon cannot be sure that its peer is authentic. In fact, it may be anybody spoofing a legitimate participant. The key exchange is inherently anonymous, and as such it is susceptible to a MITM attack.

DH has the problem that the keys that are generated are always the same for two participating entities, whereas DH_anon has the problem that it is highly exposed to MITM attacks (as mentioned above). This suggests that DHE is the most secure version of the Diffie-Hellman key exchange, mainly because it yields temporary keys that are authenticated in some meaningful way. This implies a property called *forward secrecy* or *perfect forward secrecy* (PFS).[25] It basically means that the compromise of any long-term key does

24 Such groups are, for example, specified in the informational RFC 5114 [11].

25 The term *forward secrecy* is preferred, mainly because it has nothing to do with information-theoretic security that is normally attributed to perfect secrecy.

not automatically compromise all session keys. Note what happens in the case of RSA (that does not provide forward secrecy): If an adversary is able to somehow retrieve a server's private RSA key, then he or she is able to forever after decrypt CLIENTKEYEXCHANGE messages and extract the respective premaster secrets. This, in turn, allows him or her to decrypt all messages that are cryptographically protected. The same is true for DH: In this case, the adversary can try to compromise the server's Diffie-Hellman key that is included in the server certificate. If he or she achieves this, then the Diffie-Hellman key exchange is broken and does not provide any security anymore. Only if DHE is used, then the adversary cannot attack a long-term key but must attack each session key individually. It goes without saying that this is much less attractive from the adversary's viewpoint, and hence that forward secrecy is a clear security advantage. It also goes without saying that the price to pay is a performance penalty (since the key exchange is ephemeral and must be performed for each session individually). The bottom line, however, is that forward secrecy is very important today,[26] and hence that the use of DHE is highly recommended. In TLS 1.3, for example, it remains the only key exchange mechanism that is still allowed.

- The FORTEZZA KEA is a specialty of SSL 3.0, and it is no longer supported in TLS. We mention it here only for the sake of completeness. The FORTEZZA KEA was typically implemented in a National Security Agency (NSA)-approved security microprocessor called Capstone chip (sometimes also known as MYK-80) that also implemented a cipher named Skipjack. The FORTEZZA KEA was actually a way to provide the keying material necessary for Skipjack in a way that allowed a TTP to retrieve the encryption keys if needed and legally justified. The technical term used in the 1990s for such an instantaneous key recovery feature was *key escrow*. Since the end of the 1990s, however, key escrow and respective techniques for key recovery have silently sunk into oblivion (and is therefore not further addressed here).

In the past, RSA was the predominant SSL key exchange method. This has changed meanwhile, mainly because—as mentioned above—DHE provides forward secrecy and has thus clear security advantages.

In either case, the result of the exchange is a premaster secret. Once a premaster secret is established, it can be used to derive a master secret (that is called `master_secret` in the SSL protocol specification), and once this master secret is derived, the premaster secret can be safely ignored and deleted from memory. This is generally a good idea, because something that is nonexistent cannot be attacked in the first place.

26 This applies also to other application areas, such as end-to-end encrypted (E2EE) messaging.

According to Table 2.2, the master secret represents an SSL session state element. Today, we would take a standardized *pseudorandom function* (PRF) or *key derivation function* (KDF) to derive a master secret, but such a cryptographic primitive was not yet available in the 1990s. Hence, the developers of the SSL protocol tried to build a PRF by mixing in different cryptographic hash functions (i.e., MD5 and SHA-1).[27] The construction looks as follows:

```
master_secret =
   MD5(pre_master_secret + SHA('A' + pre_master_secret
       + ClientHello.random + ServerHello.random)) +
   MD5(pre_master_secret + SHA('BB' + pre_master_secret
       + ClientHello.random + ServerHello.random)) +
   MD5(pre_master_secret + SHA('CCC' + pre_master_secret
       + ClientHello.random + ServerHello.random))
```

In this notation, `SHA` refers to SHA-1; `'A'`, `'BB'`, and `'CCC'` refer to the respective byte strings 0x41, 0x4242, and 0x434343; `ClientHello.random` and `ServerHello.random` refer to a pair of values that are randomly chosen by the client and server and exchanged in SSL handshake protocol messages (as addressed below); and + refers to the string concatenation operator here. An MD5 hash value is 16 bytes long, so the total length of the master secret is $3 \cdot 16 = 48$ bytes. The construction is the same for the RSA and Diffie-Hellman key exchange algorithms (in the case of the FORTEZZA KEA, the encryption keys are generated inside the Capstone chip, so the master secret is not used here).

The master secret is part of the session state and serves as a source of entropy for the generation of all cryptographic parameters (e.g., cryptographic keys and IVs) that are used in the sequel. Note that it is usually good practice in cryptography to use different keys for different purposes (i.e., algorithms and directions of a connection).

- If a key is used for different algorithms, then a successful attack against one algorithm may affect the security of another algorithm in some unwanted way. Imagine, for example, that the same key is used for encryption and message authentication, and that an adversary is able to break the MAC algorithm and extract the key in use. Because the key is also used for encryption, the adversary is now able to decrypt the message (although the encryption algorithm is secure). To make this impossible, it is necessary to use distinct keys for encryption and message authentication.

27 Due to more recent results, we know today that mixing several hash functions to achieve a more secure one is illusive [12]. Hence, all TLS protocol versions employ functions that are more standardized PRFs. This point is further explored in Section 3.1.1.

- If a key is used for different directions of a connection, then a message sent in one direction may look like a valid message sent in the other direction and may be misused accordingly. There are many attacks that work this way, including, for example, all types of reflection attacks.

To overcome these problems, one usually requires different keys to be used for different purposes, and this means that one has to generate many cryptographic parameters from the master secret. If one has a master secret, then essentially the same handcrafted function as given above can be used to generate an arbitrarily long key block, termed `key_block`. In this construction, the master secret now serves as the seed (instead of the premaster secret), whereas the client and server random values still represent the salt values that make cryptanalysis more difficult. The key block is iteratively constructed as follows:

```
key_block =
   MD5(master_secret + SHA('A' + master_secret +
       ServerHello.random + ClientHello.random)) +
   MD5(master_secret + SHA('BB' + master_secret +
       ServerHello.random + ClientHello.random)) +
   MD5(master_secret + SHA('CCC' + master_secret +
       ServerHello.random + ClientHello.random)) +
   [...]
```

Every iteration adds 16 bytes (i.e., the length of the MD5 output), and hence the construction is continued until the key block is sufficiently long to form the cryptographic SSL connection state elements from Table 2.3:

```
client_write_MAC_secret
server_write_MAC_secret
client_write_key
server_write_key
client_write_IV
server_write_IV
```

The first two values represent message authentication keys, the second two values encryption keys, and the third two values IVs that are needed if a block cipher in CBC mode is used (so they are not always needed). Any additional material in the key block is discarded. The construction equally applies to RSA and Diffie-Hellman, as well as for the MAC key construction of FORTEZZA. It does not apply to the construction of the encryption keys and IVs for FORTEZZA—these values are generated inside the security token of the client and securely transmitted to the server in a respective key exchange message. Again, since the FORTEZZA key exchange has never been widely used and is no longer an alternative, it is not discussed here. The same is true for the encryption algorithms used in exportable cipher suites. They require some additional processing to derive the final encryption keys and IVs. In 2015, two attacks, that are FREAK[28] [13] and Logjam[29] [14] (Section A.7), showed that exportable cipher suites are inherently dangerous and should no longer be supported. Both attacks represent MITM attacks, in which an adversary (acting as a MITM) tries to downgrade the key exchange method used to something that is exportable, and hence breakable. The attacks can therefore also be called key exchange downgrade attacks, or something similar. While the FREAK attack targets an RSA key exchange and exploits an implementation bug, the Logjam attack targets a DHE key exchange and does not depend on an implementation bug.

Message Authentication

First of all, we note that an SSL cipher suite specifies a cryptographic hash function (not a MAC algorithm) and that some additional information is therefore required to actually compute and verify a MAC. The algorithm used by SSL is a predecessor of the hashed MAC (HMAC) construction specified in RFC 2104 [15]. In fact, the SSL MAC algorithm is based on an original Internet-Draft for the HMAC construction that used string concatenation instead of the XOR operation. Hence, the SSL MAC algorithm is conceptually similar and its security is assumed to be the same. Using string concatenation (denoted ‖), the HMAC construction is defined as follows:

$$HMAC_k(m) \quad = \quad h(k \oplus opad \parallel h(k \oplus ipad \parallel m))$$

In this formula, h denotes a cryptographic hash function (i.e., MD5, SHA-1) or a representative from the SHA-2 family, k the secret key (used for message authentication), m the message to be authenticated, $ipad$ (standing for "inner pad") the byte `0x36` (i.e., `00110110`) repeated 64 times, $opad$ (standing for "outer pad") the byte `0x5C` (i.e., `01011100`) repeated 64 times, and $\oplus$ the bitwise addition modulo 2.

Using a similar notation, the SSL MAC construction is defined as follows:

$$\begin{aligned} &SSL\,MAC_k(SSLCompressed) = \\ &\qquad h(k \parallel opad \parallel h(k \parallel ipad \parallel seq\_number \parallel \underbrace{type \parallel length \parallel fragment}_{SSLCompressed^*})) \end{aligned}$$

Here, $SSLCompressed$ refers to the SSL data structure that is authenticated (and comprises $type$, $version$, $length$, and $fragment$ fields), $SSLCompressed^*$

28 The acronym FREAK stands for "factoring attack on RSA export keys," or something similar.
29 https://weakdh.org.

represents the same structure without the $version$ field, h denotes a cryptographic hash function, and k refers to the (server or client) MAC write key. The two values $ipad$ and $opad$ are the same bytes (as mentioned above) repeated 48 times (for MD5) or 40 times (for SHA-1)—compare this to the 64 times that are required in the normal HMAC construction that employs the XOR operation.

Instead of the $SSLCompressed$ structure's $version$ field, the SSL MAC construction employs a 64-bit sequence number seq_number that is specific for the message that is authenticated.[30] The sequence number represents an SSL connection state element (Table 2.3). In the end, the resulting SSL MAC is appended to the `SSLCompressed` data structure, before it is handed over to the encryption process (to generate the final `SSLCiphertext` data structure).

Encryption

With regard to encryption, it has to be distinguished whether a stream cipher or a block cipher is used.

- If a stream cipher is used, then no padding and IV are needed. According to Table 2.4, the only stream cipher that is employed in SSL 3.0 is RC4 with either a 40-bit or 128-bit key. It goes without saying that 40-bit keys are far too short to provide any reasonable level of security. In fact, they were used only to make the respective cipher suite exportable from the United States. As further addressed in Section 2.4, the use of RC4 is nowadays prohibited in the realm of the SSL/TLS protocols [16].
- If a block cipher is used, then things are more involved, mainly for the following two reasons:
  - First, padding is needed to force the length of the plaintext to be a multiple of the cipher's block size. If, for example, DES or 3DES is used for encryption, then the length of the plaintext must be a multiple of 64 bits or 8 bytes.[31] The padding format of SSL is slightly different from the one employed by TLS. In either case, the last byte of the padding specifies the length of the padding. In the case of SSL, the other padding bytes can be randomly chosen, whereas in the case of TLS, the last byte (specifying the length of the padding) must be repeated for all

30 The sequence number is a count of the number of messages the parties have exchanged so far. Its value is set to zero with each CHANGECIPHERSPEC message, and it is incremented once for each subsequent SSL record layer message in the connection.

31 In the case of Advanced Encryption Standard (AES), the length of the plaintext must be a multiple of 128 bits or 16 bytes. But note that the AES was not yet standardized when SSL 3.0 was specified. This is the reason why SSL 3.0 does not comprise a cipher suite with the AES.

other padding bytes. So all padding bytes are the same and refer to the padding length. Another difference is related to the fact that the padding is assumed to be as short as possible in the case of SSL, whereas this assumption is not made in the case of TLS. The devastating POODLE attack (Section A.4) exploits the overly simple padding scheme of SSL. As discussed at the end of the chapter, this attack has brought SSL to the end of its life cycle.

– Second, an IV is needed in some encryption modes. In the case of cipherblock chaining (CBC), for example, the SSL handshake protocol must provide an IV that also represents an SSL connection state element (Table 2.3). This IV is then used to encrypt the first record. Afterward, the last ciphertext block of each record serves as the IV for the encryption of the next record. This is called IV chaining, and it is actually the reason why another devastating attack, such as the BEAST attack (Section A.3), can be mounted against SSL 3.0 and TLS 1.0.

According to Table 2.4, SSL 3.0 envisioned the use of additional block ciphers, such as RC2 with a 40-bit key, DES with either a 40-bit or 56-bit key, 3DES, the international data encryption algorithm (IDEA) with a 128-bit key, and the abovementioned Skipjack cipher (named FORTEZZA here). All of them have disappeared from the market and have been replaced with the AES meanwhile.

Due to their simplicity and efficiency, many SSL implementations preferred stream ciphers and employed RC4 by default. This has been particularly true, because block ciphers operated in CBC mode have been subject to many attacks. However, due to some research efforts in the cryptanalysis of RC4, this has changed again, and—as mentioned above—the use of RC4 is prohibited today (but also keep in mind that the entire SSL protocol is deprecated).

To cut a long story short, the algorithms that are specified in the cipher suite transform an `SSLCompressed` structure into an `SSLCiphertext` structure. Encryption should not increase the fragment length by more than another 1,024 bytes (like compression), so the total length of the `SSLCiphertext` fragment (i.e., encrypted data and MAC) should be smaller than $2^{14} + 2,048$ bytes. We now shift our focus to the SSL handshake protocol that represents the centerpiece of SSL.

2.2.2 SSL Handshake Protocol

The SSL handshake protocol is layered on top of the SSL record protocol. It allows a client and a server to authenticate each other, to exchange keying material, and to negotiate issues like cipher suites and compression methods. The protocol and its message flows are illustrated in Figure 2.5. Messages that are written in light-grey and dotted-lined boxes are optional or situation-dependent, meaning that they are not always sent. Note that CHANGECIPHERSPEC is not really an SSL handshake protocol message but rather represents an SSL protocol—and hence a content type—of its own. In Figure 2.5, the CHANGECIPHERSPEC messages are therefore written in italics and illustrated in boxes marked as grey. Also note that each SSL message is typed with a one-byte value (i.e., a decimal number between 0 and 255), and that these values are appended in brackets in the explanations that follow (in hexadecimal and decimal notation). The type values are not illustrated in Figure 2.5 though.

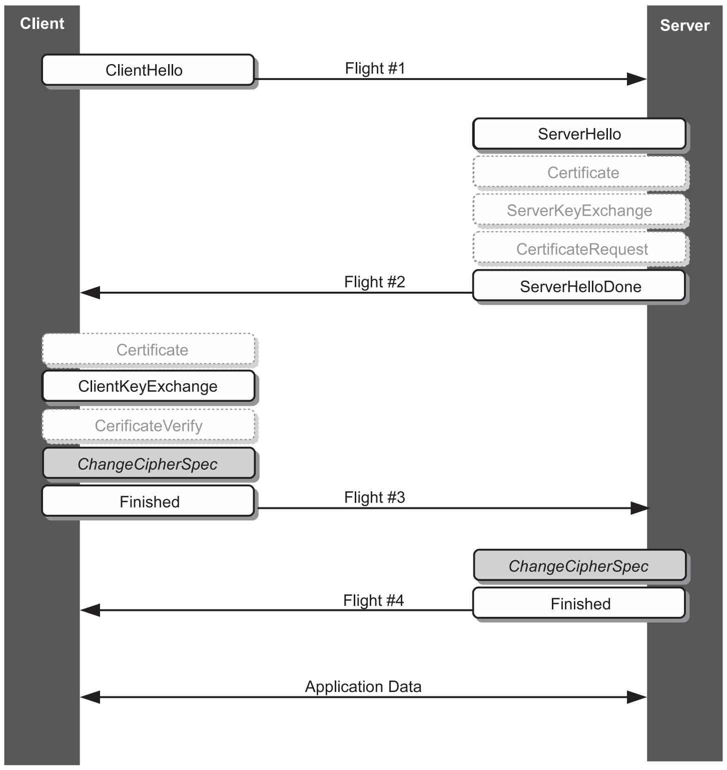

Figure 2.5 The SSL handshake protocol.

The SSL handshake protocol comprises four sets of messages—called *flights*—that are exchanged between the client and the server. All messages of the same flight may be transmitted in a single TCP segment. There may be even a fifth flight that comprises a HELLOREQUEST message (type value 0x00 or 0) and may be sent from the server to the client to initiate an SSL handshake. This message, however, is seldom used in practice. It is therefore ignored here and not shown in Figure 2.5. In either case, the messages are introduced in the order they occur in the handshake protocol. Sending messages in a different and unexpected order must always result in an error that is fatal.[32]

- The first flight comprises a single CLIENTHELLO message (type value 0x01 or 1) sent from the client to the server.
- The second flight comprises two to five messages sent back from the server to the client:

  1. A SERVERHELLO message (type value 0x02 or 2) is sent in response to a CLIENTHELLO message.
  2. If the server is to authenticate itself (which is generally the case), it sends a CERTIFICATE message (type value 0x0B or 11) to the client.
  3. Under some circumstances (discussed below), the server may send a SERVERKEYEXCHANGE message (type value 0x0C or 12) to the client.
  4. If the server requires the client to authenticate itself with a public key certificate, then it may send a CERTIFICATEREQUEST message (type value 0x0D or 13) to the client.
  5. Finally, the server sends a SERVERHELLODONE message (type value 0x0E or 14) to the client and thus closes the flight.

  After having exchanged CLIENTHELLO and SERVERHELLO messages, the client and server have negotiated a protocol version, a session identifier (ID), a cipher suite, and a compression method. Furthermore, two random values (i.e., `ClientHello.random` and `ServerHello.random`) have been generated and are now available. As mentioned above, they are used to derive the master secret from the premaster secret and the key block from the master secret.

- The third flight comprises three to five messages that are again sent from the client to the server:

32 In the realm of SSL/TLS, a fatal error must always lead to an abortion of the protocol execution. This is further addressed below.

1. If the server has sent a CERTIFICATEREQUEST message, then the client sends a CERTIFICATE message (type value 0x0B or 11) to the server.
2. In the main step of the protocol, the client sends a CLIENTKEYEXCHANGE message (type value 0x10 or 16) to the server. The contents of this message depend on the key exchange algorithm in use.
3. If the client has sent a certificate to the server, then it must also send a CERTIFICATEVERIFY message (type value 0x0F or 15) to the server. This message is digitally signed with the private key that corresponds to the client certificate's public key.
4. The client sends a CHANGECIPHERSPEC message[33] to the server (using the SSL change cipher spec protocol) and copies its pending write state into the current write state.
5. The client sends a FINISHED message (type value 0x14 or 20) to the server. As mentioned above, this is the first message that is cryptographically protected under the new cipher spec.

- Finally, the fourth flight comprises two messages that are sent from the server back to the client:

    1. The server sends another CHANGECIPHERSPEC message to the client and copies its pending write state into the current write state.
    2. Finally, the server sends a FINISHED message (type value 0x14 or 20) to the client. Again, this message is cryptographically protected under the new cipher spec.

At this point in time, the SSL handshake is complete and the client and server may start to exchange application-layer data, typically using the SSL application data protocol. This continues, until the SSL session is terminated at some later point in time.

There are two possibilities to (re)use an existing session (that may already be terminated or not): Either a not yet terminated session is used to renegotiate a new session (in a so-called *session renegotiation*), or a terminated session is resumed with a simplified handshake (in a so-called *session resumption*). This is possible, if and only if the `is resumable` flag (Table 2.2) of the respective session state is set. Both possibilities need to be clearly distinguished, as they lead to different protocol executions and respective transcripts.

33 Again, the missing type indicates that the CHANGECIPHERSPEC message is not an SSL handshake protocol message. Instead, it is an SSL change cipher spec protocol message (identified with a content type value of 20).

The SSL protocol allows a client to request a session renegotiation at any point in time simply by sending a new CLIENTHELLO message to the server. This is known as a *client-initiated renegotiation*. Alternatively, if the server wishes to renegotiate, then it can send a HELLOREQUEST message (type value 0x00 or 0) to the client (we already mentioned this message above). This is known as a *server-initiated renegotiation*, and it is basically a signal sent to the client that asks the client to initiate a new handshake.

There are many situations in which (either a client-initiated or a server-initiated) renegotiation makes sense. If, for example, a web server is configured to allow anonymous HTTP requests to most parts of its document tree but requires certificate-based client authentication for some specific parts, then it is reasonable to renegotiate a connection and request a client certificate if and only if one of these documents is requested. Similarly, the use of renegotiation is required if the strength of the cryptographic techniques needs to be changed. One such example is the use of *International Step-Up* or *Server Gated Cryptography* (SGC) that used to be important in the 1990s and is further addressed in Section 2.2.2.4. Last but not least, renegotiation may also be used if the sequence number seq_number (that represents a record counter and is also used for message authentication) is about to overflow. Needless to say, this situation seldom occurs, because the number is 64 bits long and hence very large. In any of these situations, a session renegotiation may take place. The problems hereby are that renegotiating a new session is not particularly efficient, because a full handshake must be performed, and that session renegotiation introduces new vulnerabilities that have also been exploited in specific attacks (Section A.5).

The above-described SSL handshake (to establish a session) takes two round trips. If a handshake has already been performed recently, then the respective session (that must be resumable) can be resumed in 1 round-trip time (1-RTT). So session resumption is much more efficient than a full-fledged session establishment or renegotiation. If a client and server are willing to resume a previous SSL session, then the SSL handshake protocol can be simplified considerably, and the resulting (simplified) protocol is illustrated in Figure 2.6. The client sends a CLIENTHELLO message including the session ID of the session that it wants to resume. The server checks its session cache for a match for this particular ID. If a match is found and the server is willing to reestablish a connection under the respective session state, then it sends back to the client a SERVERHELLO message with this particular session ID. The client and server can then directly move to the CHANGECIPHERSPEC and FINISHED messages. If a session ID match is not found or the session is not resumable, then the server must generate a new session ID and the client and server must go through a full SSL handshake.

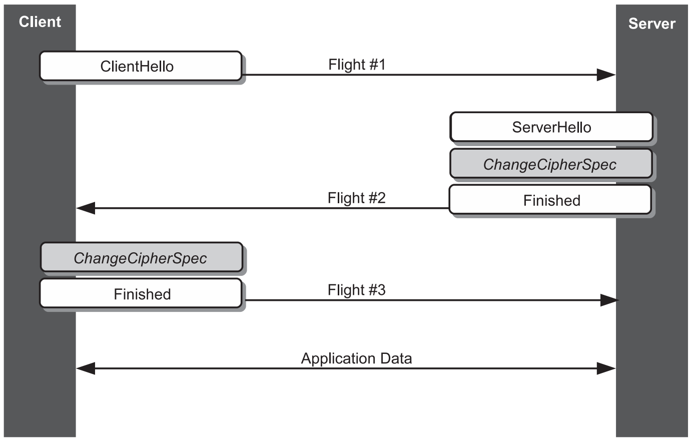

Figure 2.6 The simplified SSL handshake protocol (to resume a session).

For the sake of completeness, we note that the TLS protocol provides another 1-RTT mechanism to resume a session without requiring session-specific state on the server side. This mechanism employs a TLS extension known as a session ticket and is also addressed in Section C.2.26. It is not relevant for the SSL protocol, hence we ignore it here.

Let us have a closer look at the various messages that are exchanged in the course of an SSL handshake. Each message starts with a 1-byte *type* field that refers to the SSL handshake message and a 3-byte *length* field that refers to the byte length of the message. Remember that multiple SSL handshake messages may be sent in a single SSL record. This is illustrated in Figure 2.7. The strongly framed part refers to the SSL handshake message(s), whereas the leading 5 bytes refer to the SSL record header. This header, in turn, always comprises a 1-byte type value 22 (referring to the SSL handshake protocol), a 2-byte version value 3,0 (referring to SSL 3.0), and a 2-byte length value referring to the byte length of the remaining part of the SSL record (that comprises the actual handshake messages). So while the length field of the record header refers to the total byte length of the record, each message's length field only refers to the byte length of the particular message.

In the sequel, we focus on each SSL handshake message separately. For the sake of simplicity, we assume that each handshake message is wrapped and sent in a distinct record. However, keep in mind that this need not be the case, and that—depending on the implementation—multiple handshake messages may also be sent

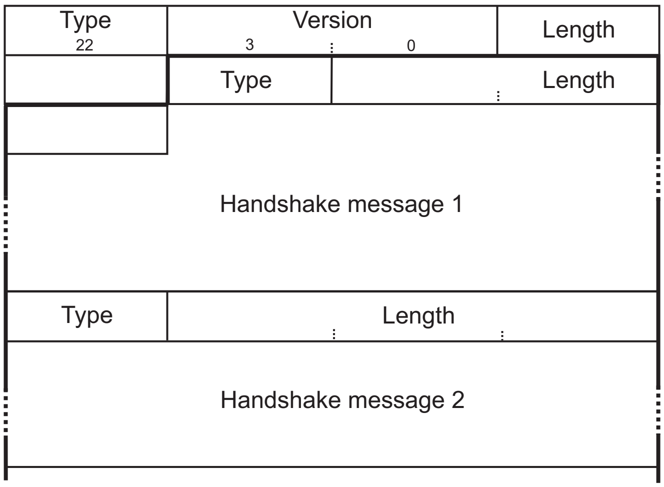

Figure 2.7 Multiple handshake messages wrapped in an SSL record.

in a single SSL record (as discussed above and illustrated in Figure 2.7). If a field value is known and fixed, then we write it in decimal notation under the name of the respective field. In Figure 2.7, for example, we know that the type must be 22 and that the version must be 3,0; both values are therefore included. Note that the length field value of a record header can only be provided if the message is sent in a record of its own (otherwise the value may be much larger).

2.2.2.1 HELLOREQUEST Message

As mentioned above, the HELLOREQUEST message allows a server to ask a client to initiate a new SSL handshake. The message is not often used in practice, but it provides some additional flexibility. If, for example, an SSL connection has been in use for so long that its security must be put in question, then the server can send a HELLOREQUEST message to force the client to renegotiate a new session and hence establish new keying material.

Figure 2.8 illustrates a HELLOREQUEST message wrapped in a distinct SSL record. As usual, the record starts with a 5-byte record header, in which the length field refers to the 4 bytes that comprise the actual HELLOREQUEST message. This message, in turn, only consists of a 4-byte header, where the first byte refers to the message type (that has a value of 0x00 or 0) and the remaining 3 bytes refer to the message length (that has a value of zero, because the message body is empty). Note

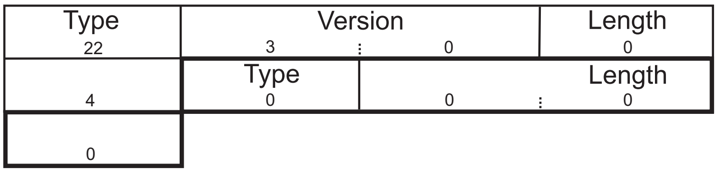

Figure 2.8 HELLOREQUEST message wrapped in an SSL record.

again that the length field of the SSL record header comprises the value 4 only if a single HELLOREQUEST message is sent in the SSL record (which is likely to be the case here, because it represents a flight on its own). If multiple messages were sent in the same record, then the respective length field value would be larger.

2.2.2.2 CLIENTHELLO Message

The CLIENTHELLO message is the first message that is sent from the client to the server in an SSL handshake. In fact, it is normally the message an SSL handshake begins with. As illustrated in Figure 2.9, an SSL CLIENTHELLO message wrapped in an SSL record starts with the usual 5-byte record header, a 1-byte type field (that has a value of 0x01 or 1), and a 3-byte message length field (with unknown value here). In addition, the body of a CLIENTHELLO message comprises the following fields:

- The 2 bytes immediately following the message length field refer to the highest SSL version supported by the client (in the case of SSL 3.0, this value is set to 3,0). In the official SSL protocol specification, this field is called `client_version`.
- The 32 bytes following the SSL version field comprise a client-generated random value. In the SSL protocol specification, this field is called `random` and its value is referenced by `ClientHello.random`. It consists of two parts:
  - A 4-byte date and time (up to the second) string in standard UNIX format that is defined as the number of seconds elapsed since midnight

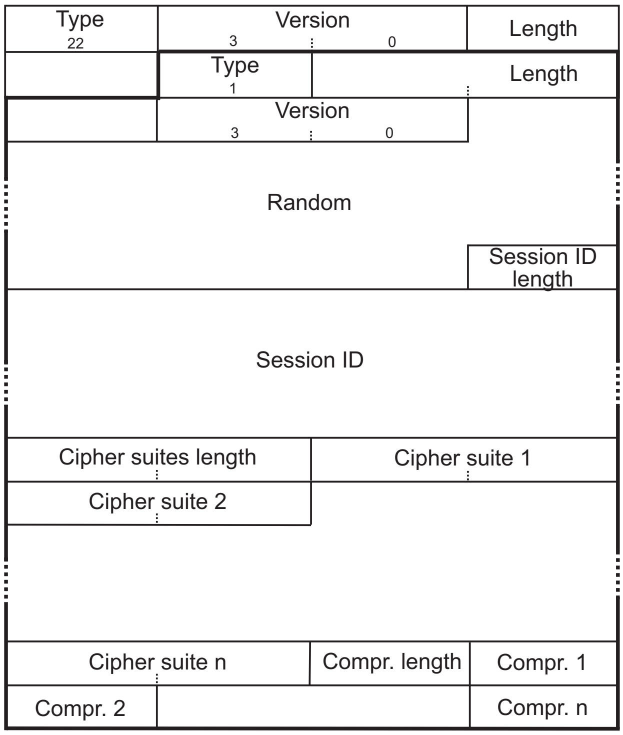

Figure 2.9 CLIENTHELLO message wrapped in an SSL record.

Coordinated Universal Time (UTC[34]) of January 1, 1970, not counting leap seconds[35] according to the sender's internal clock.[36]

– A 28-byte string that is randomly or pseudorandomly generated.

This value, together with a similar value created by the server, provides input for several cryptographic computations (Section 2.2.1.3). Consequently,

34 Note that, for historical reasons, the term used at this point is sometimes *Greenwich Mean Time* (GMT), a predecessor of UTC.

35 A leap second is a one-second adjustment that keeps broadcast standards for time of day close to mean solar time.

36 The SSL protocol specification does not require a particular level of accuracy for this value, as it is not intended to provide an accurate time indication. Instead, the specification suggests using the date and time string as a way to ensure that the client does not reuse particular values.

it is required that it is unpredictable to some extent, and hence that a cryptographically strong random or pseudorandom bit generator is used to generate the second part of it. As mentioned in Section 2.4, this has not always been the case though.

- The byte immediately following the `random` value refers to the length of the session ID. If this value is zero, then there is no SSL session to resume or the client wants to generate new security parameters. In this case, the server is going to select an appropriate ID for the session. Otherwise, if the session ID length is not equal to zero, then the client aims at resuming the session identified afterward. Because a session ID may have a variable length, it needs to be specified in one way or another.
- If the session ID length is greater than zero, then the corresponding number of bytes that follow the session ID length field represent the session ID. In the SSL protocol specification both fields are collectively referred to as `session_id`. The total length of the session ID is restricted to 32 bytes at most, but no constraints are placed on its actual content. Note that session IDs are transmitted in CLIENTHELLO messages before any encryption is put in place, so implementations should not put any information in the session ID that might, if revealed, compromise security.
- The 2 bytes immediately following the session ID refer to the number of cipher suites supported by the client. This number equals the length of the list of cipher suites that is about to follow. This list is ordered according to the client's preferences (i.e., the client's first preference appears first). Each cipher suite is represented by 2 bytes.
- For every cipher suite supported by the client, there is a 2-byte code referring to it. In the case of SSL, the first byte of the code is always set to zero, whereas the second byte of the code refers to the index in Table 2.4. For example, SSL_NULL_WITH_NULL_NULL has code 0x0000, whereas SSL_RSA_WITH_3DES_EDE_CBC_SHA has code 0x0010. These codes are appended in a variable-length cipher suites field, called `cipher_suites` in the SSL protocol specification.
- After the cipher suites, a similar scheme applies to the compression methods supported by the client. In fact, the 2 bytes immediately following the `cipher_suites` field refer to the number of compression methods supported by the client. This number equals the length of the list of compression methods that follows. The list itself is ordered according to the client's preferences (i.e., the client's first preference appears first). For every compression

method, a unique code is appended. The resulting value is written into the `compression_methods` field (as it is called in the SSL protocol specification). Due to the fact that SSL 3.0 only defines the null compression, all implementations set the compression length to one and the following byte to zero, actually referring to no compression.

In the interest of forward compatibility, it is permitted for a CLIENTHELLO message to include some extra data after the `compression_methods` field. This data must be included in the handshake hashes, but can otherwise be ignored at will. This is the only handshake message for which this is legal (i.e., for all other messages, the amount of data in the message must precisely match the description of the message). The possibility to insert data in the CLIENTHELLO message is, for example, extensively used by the extension mechanism introduced in TLS 1.2 (Section 3.4.1).

2.2.2.3 SERVERHELLO Message

After having received a CLIENTHELLO message, it is up to the server to process and verify it, and to return a SERVERHELLO message in the positive case. As Figure 2.10 illustrates, a SERVERHELLO message is structurally similar to a CLIENTHELLO message. The only significant differences are the value of the SSL handshake message type (0x02 instead of 0x01) and the fact that the server specifies a single cipher suite and a single compression method (instead of lists of cipher suites and compression methods). Remember that the server must pick from among the choices proposed by the client, and hence the values specified by the server refer to the ones that are going to be used for all SSL connections that refer to the session.

More specifically, an SSL SERVERHELLO message starts with the usual 5-byte SSL record header, a 1-byte type field (that has the value 0x02 or 2, and hence refers to a SERVERHELLO message), and a 3-byte message length field. Afterward, the body of a SERVERHELLO message comprises a few additional fields.

- The 2 bytes immediately following the message length field refer to the SSL version that is going to be used. In the SSL protocol specification, this field is called `server_version`. It basically corresponds to the lower version of that suggested by the client in the CLIENTHELLO message and the highest version supported by the server. In the case of SSL 3.0, the server version is set to 3,0.
- The 32 bytes following the server version field comprise a 32-byte server-generated random value, again called `random` in the SSL protocol specification. The structure of the random value is identical to the one generated by the

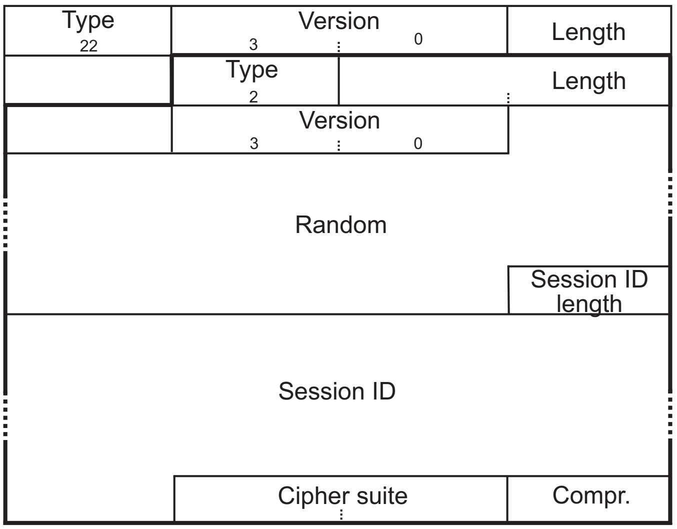

Figure 2.10 SERVERHELLO message wrapped in an SSL record.

client. Its actual value, however, must be independent and different from the client's value.

- The byte following the server random value field specifies the length of the session ID. Remember that the server may include, at its own discretion, a session ID in the SERVERHELLO message. If a session ID is included, then the server allows the client to attempt to resume the session at some later point in time. Servers that don't wish to allow session resumption can omit a session ID by specifying a length of zero.
- If the session ID length is not equal to zero, then the corresponding number of bytes after the length field represent the session ID. If the `session_id` field of the CLIENTHELLO message was not empty, then the server is asked to look in its session cache for a match. If a match is found and the server is willing to establish a new connection using the old session state, then the server must respond with the same `session_id` value as supplied by the client. This indicates a resumed session and dictates that the parties must proceed to the CHANGECIPHERSPEC and FINISHED messages (according to the simplified SSL handshake protocol illustrated in Figure 2.6). Otherwise, if no match is found or the server is not willing to establish a new connection using the old

session state, then the `session_id` field must contain a new value, and this new value then identifies the new session.

- The 2 bytes immediately following the session ID field refer to the cipher suite selected by the server. This field is called `cipher_suite` in the SSL protocol specification (note the singular form here). For resumed sessions, the value for the cipher suite field must be copied from the resumed session state.
- Finally, the last byte refers to the compression method selected by the server. This field is called `compression_method` in the SSL protocol specification (again, note the singular form). For resumed sessions, the value for the compression method field must be copied from the resumed session state, too.

After the server has sent out an SSL SERVERHELLO message, it is assumed that the client and server now have a common understanding about which SSL version and session to use, meaning that they both know which session to resume or which algorithms to use for the establishment of a new session.

2.2.2.4 CERTIFICATE Message

Most key exchange methods are nonanonymous, meaning that the server must authenticate itself to the client with a public key certificate. In fact, this applies to all key exchange methods except DH_anon. The server therefore sends a CERTIFICATE message to the client, immediately following a SERVERHELLO message (i.e., typically in the same flight). The same message type may occur later in the SSL handshake, when the server asks the client for a certificate with a CERTIFICATEREQUEST message and the client responds with another CERTIFICATE message. In either case, the aim of the CERTIFICATE message is to transfer a public key certificate, or—more generally—a set of public key certificates that form a certificate chain to the peer.

In the SSL protocol specification, the field that may comprise a certificate chain is called `certificate_list`; it includes all certificates required to form the chain. Each chain is ordered with the sender's certificate first followed by a series of CA certificates proceeding sequentially upward until the certificate of a root CA is reached. Note that support for certificate chains was introduced in SSL 3.0 (and was not present in previous versions of the SSL protocol). Anyway, the certificate types must be appropriate for the key exchange algorithm in use. Typically, these are X.509 certificates (or some modified X.509 certificates in the case of the FORTEZZA KEA). The use of raw or cached public keys, as well as other certificate formats, such as open pretty good privacy (OpenPGP), was later enabled in the TLS protocol by the use of respective TLS extensions (Appendix C).

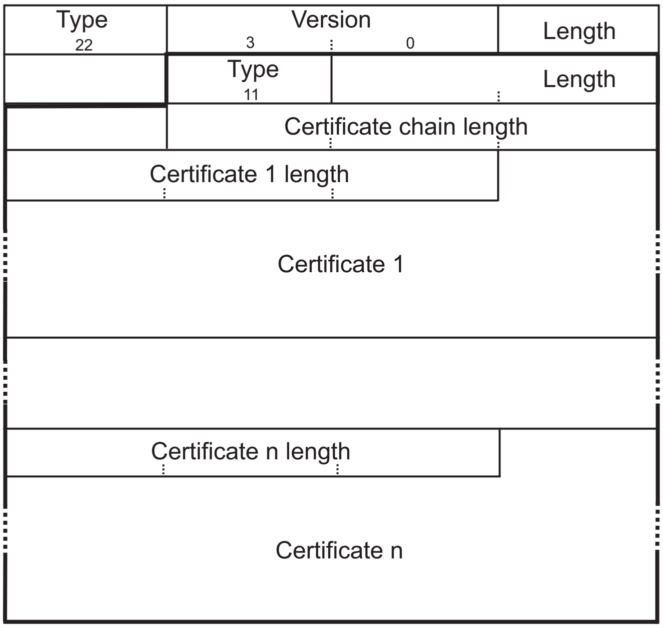

Figure 2.11 CERTIFICATE message wrapped in an SSL record.

As illustrated in Figure 2.11, an SSL CERTIFICATE message wrapped in an SSL record starts with the usual record header, a 1-byte type field with the value 0x0B or 11 (referring to an SSL CERTIFICATE message), and a 3-byte message length field. The body of the message then contains another 3-byte length field (that contains a value that is three less than the message length) and the actual certificate chain. Each certificate in the chain also begins with a 3-byte field referring to the length of this particular certificate. Depending on the length of the chain, the total byte length of the CERTIFICATE message may be very large.

Due to the U.S. export controls that were in place until the end of the 1990s, Netscape Communications and Microsoft had added features to their browsers that allowed them to use strong cryptography if triggered with specifically crafted certificates (otherwise support for strong cryptography was hidden from the server). These features were called *International Step-Up* in the case of Netscape Communications, and *SGC* in the case of Microsoft. In either case, the corresponding certificates were issued by officially approved certification authorities (CAs), such as VeriSign,[37] and contained a special attribute in the extended key usage (`extKeyUsage`) field.[38]

In order to invoke International Step-Up or SGC, a normal initial SSL handshake took place. In the CLIENTHELLO message, the client claimed to support only export-strength cipher suites. So the server had no choice but to select a corresponding cipher suite. In fact, at this point in time, the server did not even know that the client supported strong cryptography in the first place. As soon as the server provided its CERTIFICATE message, however, the client knew that the server was capable of supporting strong cryptography. In the case of SGC, the client immediately aborted the handshake and sent a new CLIENTHELLO message to the server. In this message, the client proposed full-strength cipher suites, and the server was to select one. In the case of International Step-Up, the client completed the initial handshake before it started a new handshake, in which it proposed full-strength cipher suites. The use of International Step-Up and/or SGC was a compromise between the needs of the U.S. government to limit the use of full-strength cryptography abroad and the desire of browser manufacturers to offer the strongest possible product to their customers. Controlling the use of full-strength cryptography became a matter of controlling the issuance of International Step-Up and SGC certificates. Consequently, the U.S. government regulated the possibility to purchase those certificates. Only companies that had a legitimate reason to use strong cryptography were allowed to do so. This mainly applied to financial institutions that operated globally.

Soon after International Step-Up and SGC were launched, a couple of local proxy servers for SSL were brought to market. These proxies were able to transform export-grade cryptography into strong cryptography, independent from the browser. Most interestingly, a tool named *Fortify*[39] was distributed internationally. The tool was able to patch (or rather remove) the artificial barrier that precluded a browser from using strong cryptography (independent from the server certificate in use) in the first case. This tool made International Step-Up and SGC obsolete, and the two initiatives silently sank into oblivion. They finally became obsolete when the U.S. government liberalized its export controls toward the end of the last century. So from today's perspective, International Step-Up and SGC are no longer relevant. Since

37 Note that VeriSign (https://www.verisign.com) was originally founded by RSA to provide certification services to the internet on a large-scale. The company was sold several times, and is nowadays mainly active in domain name registration and providing internet infrastructure.

38 More specifically, an International Step-Up certificate included the object identifier (OID) 2.16.840.1.113730.4.1 in the extended key usage field, whereas an SGC certificate included the OID 1.3.6.1.4.1.311.10.3.3. To keep things as simple as possible, a single certificate was typically issued that included both extended key usage objects, so the same certificate could be used to simultaneously support International Step-Up and SGC.

39 https://www.fortify.net.

the terms are sometimes still in use, it is nevertheless helpful to know what they stand for. Also, due to some recent key exchange downgrade attacks like FREAK and Logjam (Section A.7), the notion of export-grade cryptography has become important again. In fact, it is highly recommended to disable all export-grade cipher suites by default, simply because the mere existence of them yields a vulnerability that may be exploited in some unexpected ways.

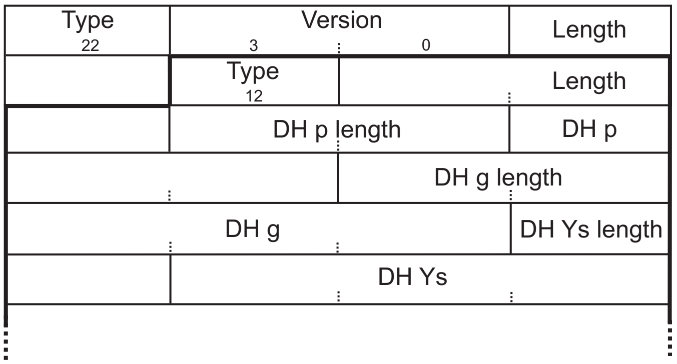

Figure 2.12 The beginning of a SERVERKEYEXCHANGE message using Diffie-Hellman (wrapped in an SSL record).

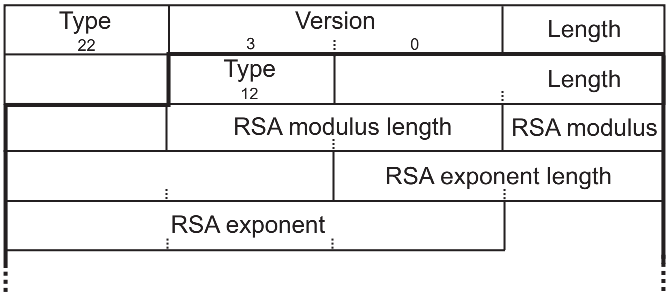

Figure 2.13 The beginning of a SERVERKEYEXCHANGE message using RSA (wrapped in an SSL record).

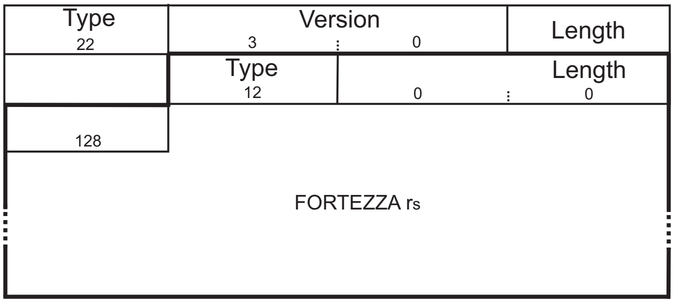

Figure 2.14 SERVERKEYEXCHANGE message using FORTEZZA (wrapped in an SSL record).

2.2.2.5 SERVERKEYEXCHANGE Message

Similar to the CERTIFICATE message, the SERVERKEYEXCHANGE message is optional, meaning that it doesn't have to be sent in all cases. If, for example, RSA is used for key exchange and the client can retrieve the appropriate public key from the server's certificate to decrypt the premaster secret (that is provided in the CLIENTKEYEXCHANGE message), then no SERVERKEYEXCHANGE message is needed. Similarly, if a fixed Diffie-Hellman key exchange is used, then the client can retrieve the server's Diffie-Hellman parameters from the server certificate, employ these parameters to perform a Diffie-Hellman key exchange, and use the result as the premaster secret. On the other hand, however, there are still a few cases in which a SERVERKEYEXCHANGE message is needed. The most important example is a DHE key exchange that requires the server to send its Diffie-Hellman parameters to the client in such a message. Consequently, the need and use of the SERVERKEYEXCHANGE message depends on the actual key exchange method in use.

As illustrated in Figures 2.12–2.14, an SSL SERVERKEYEXCHANGE message always starts with the usual 5-byte SSL record header, a 1-byte type field (that must have the value 0x0C or 12 referring to a SERVERKEYEXCHANGE message), and a 3-byte message length field. The rest of the SERVERKEYEXCHANGE message depends on the key exchange method in use (Diffie-Hellman, RSA, or FORTEZZA).

- As mentioned above, fixed Diffie-Hellman doesn't require a SERVERKEYEXCHANGE message. But if ephemeral or anonymous Diffie-Hellman is used, then the rest of the SERVERKEYEXCHANGE message comprises the server's Diffie-Hellman parameters, including a prime modulus p, a generator g, and a public exponent Y_s, as well as a digital signature for these parameters in the case of an ephemeral Diffie-Hellman key exchange (the signature is not present in an anonymous Diffie-Hellman key exchange). The beginning of such a message is illustrated in Figure 2.12 (without the signature part). Note that the fields for the Diffie-Hellman parameters have a variable length (consistently set to three in Figure 2.12).

- If normal RSA is used for key exchange, then a SERVERKEYEXCHANGE message is not needed. In this case, the client takes the public key from the server's certificate to encrypt the premaster secret. But if this key is a signature-only key, then this doesn't work, and an ephemeral RSA public key pair is still needed. This basically means that the server must generate a key pair, keep the private key on its side, and send the public key (consisting of a modulus and an exponent) to the client in digitally signed form. The beginning of such a message is illustrated in Figure 2.13 (again without the signature part). The fields for the RSA parameters have a variable length (again consistently set to three in Figure 2.13). This mechanism of introducing ephemeral RSA public key pairs allowed the use of SSL in a way that conformed to the U.S. export controls, where the maximum key length for RSA was 512 bits. In fact, this threshold only applied for RSA used for key exchange (and did not apply to RSA used for signatures). Consequently, if the cipher suite required RSA_EXPORT for key exchange, then there was an ephemeral 512-bit RSA key pair for key exchange and a larger, typically 1024-bit RSA key pair to sign the ephemeral public key and send it in the SERVERKEYEXCHANGE message to the client. Note, however, that an RSA key length of 512 bits is far too short to provide any reasonable security today, as demonstrated by the FREAK attack (Section A.7.1), and hence that exportable cipher suites prevail from the past and should not be supported anymore.
- If FORTEZZA is used, then the SERVERKEYEXCHANGE message only carries the server's r_s value that is required by the FORTEZZA KEA (Section 2.2.2.9). Since this value is always 128 bytes long, there is no need for a separate length parameter. Also, there is no need for a digital signature. A respective SERVERKEYEXCHANGE message is illustrated in Figure 2.14.

As mentioned above, the SERVERKEYEXCHANGE message may also include a signature part. This is particularly true for ephemeral Diffie-Hellman and RSA (if used with two key pairs). In these cases, the server must have provided the client with a certificate in the CERTIFICATE message. The certificate comprises a signature verification key that can be used either for RSA or DSA. If RSA is used, then the string that is signed consists of an MD5 hash value concatenated with a SHA-1 hash value. If DSA is used, then the string is only a SHA-1 hash value. In either case, the input to the hash functions is the concatenation of `ClientHello.random` (i.e., the `random` value of the CLIENTHELLO message), `ServerHello.random` (i.e., the `random` value of the SERVERHELLO message), and the server key parameters mentioned above (i.e., the Diffie-Hellman parameters of Figure 2.12 or the RSA parameters of Figure 2.13).

Table 2.5
SSL 3.0 Certificate Type Values

Value	Name	Description
1	rsa_sign	Certificate containing an RSA key
2	dss_sign	Certificate containing a DSA key
3	rsa_fixed_dh	Certificate containing a static DH key
4	dss_fixed_dh	Certificate containing a static DH key
5	rsa_ephemeral_dh	Certificate containing an ephemeral DH key
6	dss_ephemeral_dh	Certificate containing an ephemeral DH key
20	fortezza_kea	Certificate used for the FORTEZZA KEA

2.2.2.6 CERTIFICATEREQUEST Message

A nonanonymous server may optionally authenticate a client[40] by sending a CERTIFICATEREQUEST message to it. This message not only asks the client to send a certificate (and sign data using the respective signing key later on), but it also informs the client about what types of certificates it is going to accept. The possibilities to choose from are listed in Table 2.5. `Rsa_sign` refers to an RSA public key and a certificate that allows the key to be used for signing, whereas `dss_sign` refers to a DSA public key and a certificate that allows the key to be used for signing. Two certificate types (i.e., `rsa_fixed_dh` and `dss_fixed_dh`) contain static DH keys (the first signed with RSA and the second signed with DSA), whereas two certificate types (i.e., `rsa_ephemeral_dh` and `dss_ephemeral_dh`) contain ephemeral DH keys (signed with either RSA or DSA). Finally, `fortezza_kea` refers to a certificate used for the FORTEZZA KEA. Note that the meaning of the certificate types has changed over time, and that the currently valid meaning is available from the respective IANA repository.[41]

As illustrated in Figure 2.15, an SSL CERTIFICATEREQUEST message starts with the usual 5-byte SSL record header, a 1-byte type field with value 0xD or 13 (referring to a CERTIFICATEREQUEST message), and a 3-byte message length field.

40 Note that an anonymous server must not request a certificate from the client.
41 https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-2.

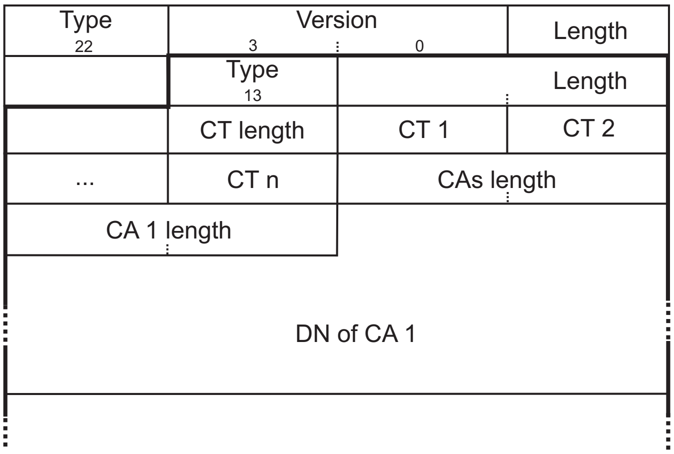

Figure 2.15 Beginning of a CERTIFICATEREQUEST message wrapped in an SSL record.

The actual body of the message begins with a list of acceptable certificate types (called `certificate_types` in the SSL protocol specification and acronymed CT in Figure 2.15). This type list has a 1-byte length field followed by a non-empty set of single-byte values that refer to certificate types from Table 2.5. After the type list, the CERTIFICATEREQUEST message also contains a list of CAs that are accepted by the server. In the SSL protocol specification, this list is called `certificate_authorities`. It starts with a 2-byte length field and then contains one or more distinguished names (DNs). Each CA (or DN, respectively) has its own 2-byte length field that is put in front of the CA's DN. In Figure 2.15, only the first CA (i.e., CA 1) is shown. However, this list may be long, and hence the entire CERTIFICATEREQUEST message may also be long.

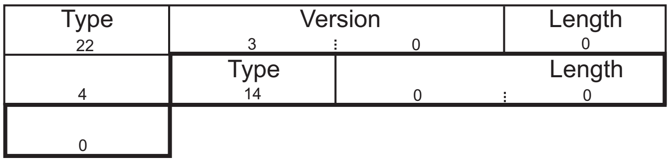

Figure 2.16 SERVERHELLODONE message wrapped in an SSL record.

2.2.2.7 SERVERHELLODONE Message

The SERVERHELLODONE message is sent by the server to indicate the end of the second flight. As illustrated in Figure 2.16, an SSL SERVERHELLODONE message starts with the usual 5-byte SSL record header, a 1-byte type field with value 0x0E or 14 (referring to a SERVERHELLODONE message), and a 3-byte message length field. Since the body of the SERVERHELLODONE message is empty, the three bytes referring to the message length are set to zero. So a HELLOREQUEST message is always 4 bytes long, and hence this value may be included in the last byte of the length field of the respective SSL record header—at least if the SERVERHELLODONE message is sent in an SSL record of its own.

2.2.2.8 CERTIFICATE Message

After having received a SERVERHELLODONE message, it is up to the client to verify the server certificate (if required) and check that the values provided by the server are in fact acceptable. If everything is fine, the client sends a couple of messages to the server in the third flight. If the server requested a certificate, then the client would first send a CERTIFICATE message to the server. This message would be structurally the same as the message sent from the server to the client (Section 2.2.2.4 and Figure 2.11). If the Diffie-Hellman key exchange algorithm is used, then the client-side Diffie-Hellman parameters must conform to the ones provided by the server, meaning that the Diffie-Hellman group and generator encoded in the client certificate must match the server's values. Note, however, that this message is only a preparatory message with regard to client authentication. The actual authentication takes place when the client sends a CERTIFICATEVERIFY message (Section 2.2.2.10) to the server and hence proves that it knows the proper private key.

2.2.2.9 CLIENTKEYEXCHANGE Message

One of the most important messages in an SSL handshake is the CLIENTKEYEXCHANGE message that is sent from the client to the server. It provides the server with the client-side keying material that is later used to secure communications. As illustrated in Figures 2.17–2.19, the format of the CLIENTKEYEXCHANGE message again depends on the key exchange method in use. In either case, it starts with the usual 5-byte SSL record header, a 1-byte type field with value 0x10 or 16 (referring to a CLIENTKEYEXCHANGE message), and a 3-byte message length field. The body of the message depends on the key exchange method in use.

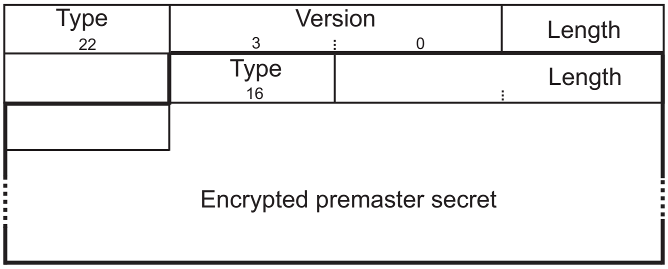

Figure 2.17 CLIENTKEYEXCHANGE message using RSA (wrapped in an SSL record).

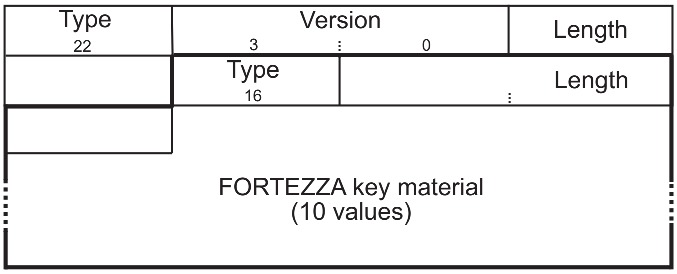

Figure 2.18 CLIENTKEYEXCHANGE message using FORTEZZA (wrapped in an SSL record).

- If RSA or FORTEZZA is used, then the body of the CLIENTKEYEXCHANGE message comprises an encrypted 48-byte premaster secret (i.e., `pre_master_secret`). To detect version downgrade attacks, the first 2 bytes of the 48 bytes refer to the latest (newest) version supported by the client and offered in the CLIENTHELLO message (note that this need not be the version that is actually in use[42]). Upon receiving the premaster secret, the server should check that this value matches the value originally transmitted by the client in the CLIENTHELLO message.
  - In the case of RSA, the premaster secret is encrypted with the public RSA key from the server's certificate or ephemeral RSA key from the SERVERKEYEXCHANGE message. A respective SSL CLIENTKEYEXCHANGE message using RSA is illustrated in Figure 2.17.

42 There are implementations that employ the version in use instead of the latest version supported by the client. This is not a severe security problem, but there are some interoperability issues involved.

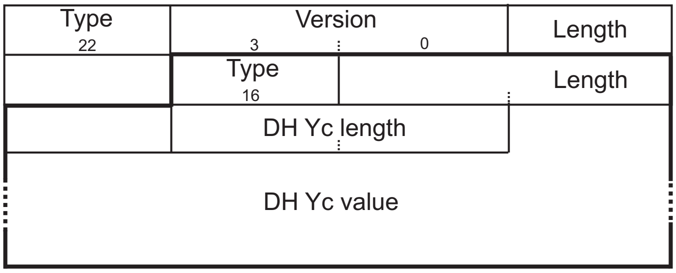

Figure 2.19 CLIENTKEYEXCHANGE message using Diffie-Hellman (wrapped in an SSL record).

– In the case of FORTEZZA, the KEA is used to derive a token encryption key (TEK), and the TEK is used to encrypt (and securely transmit) the premaster secret and a few other cryptographic parameters to the server. A corresponding SSL CLIENTKEYEXCHANGE message is illustrated in Figure 2.18. The FORTEZZA key material actually consists of 10 values, summarized in Table 2.6. Note that the client's Y_C value for the KEA calculation is between 64 and 128 bytes long and that it is empty if Y_C is part of the client certificate. Keep in mind that FORTEZZA is not used anymore, and hence this description is not important and should be taken with a grain of salt. We ignore it for the purpose of this book.

Table 2.6
FORTEZZA Key Material

Parameter	Size
Length of Y_C	2 bytes
Client's Y_C value for the KEA calculation	0–128 bytes
Client's R_C value for the KEA calculation	128 bytes
DSA signature for the client's KEA public key	40 bytes
Client's write key, wrapped by the TEK	12 bytes
Client's read key, wrapped by the TEK	12 bytes
IV for the client write key	24 bytes
IV for the server write key	24 bytes
IV for the TEK used to encrypt the premaster secret	24 bytes
Premaster secret, encrypted by the TEK	48 bytes

- If Diffie-Hellman is used, then the CLIENTKEYEXCHANGE message comprises the client's public Diffie-Hellman parameter Y_c. A respective message is illustrated in Figure 2.19.

If the server receives a CLIENTKEYEXCHANGE message, then it uses its private key to decrypt the premaster secret in the case of RSA or FORTEZZA, and it uses its own Diffie-Hellman parameter to compute a the premaster secret in the case of Diffie-Hellman. In either case, the server now knows the premaster secret and can derive all keying material from it.

2.2.2.10 CERTIFICATEVERIFY Message

If the client has provided a certificate with signing capabilities[43] in its CERTIFICATE message, then it must still prove possession of the corresponding private key. (The certificate alone does not provide client authentication, mainly because the certificate can be eavesdropped and replayed.) To do so, the client sends a CERTIFICATEVERIFY message to the server, and this message basically comprises a digital signature generated with the client's private key.

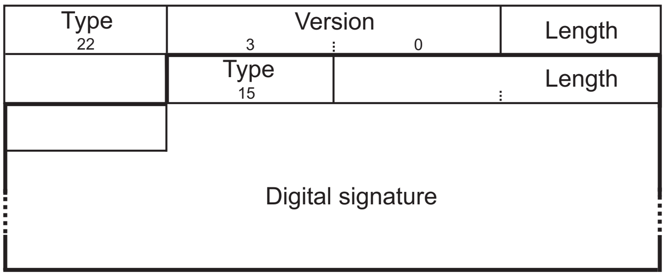

Figure 2.20 CERTIFICATEVERIFY message wrapped in an SSL record.

As illustrated in Figure 2.20, such an SSL CERTIFICATEVERIFY message starts with the usual 5-byte SSL record header, a 1-byte type field with value 0x0F or 15 (referring to a CERTIFICATEVERIFY message), and a 3-byte message length field. The body of the message then comprises a digital signature, where the exact format of the signature depends on whether the client's certificate is for RSA or DSA. If RSA is used, then the string that is signed is again the concatenation of an MD5 hash value and a SHA-1 hash value, whereas it is only a SHA-1 hash value in the case of DSA. In either case, the input to the hash functions is the same. If $handshake\_messages$ refers to the concatenation of all SSL handshake messages that have been exchanged so far, k to the master secret, and $ipad$ and $opad$ to the values introduced earlier in this chapter (repeated 48 times for MD5 and 40 times

43 This applies for all certificates from Table 2.5, except those containing fixed Diffie-Hellman parameters (i.e., `rsa_fixed_dh` and `dss_fixed_dh`).

for SHA-1), then the use of MD5 yields

$$H_{\text{MD5}} = \text{MD5}(k \parallel opad \parallel \text{MD5}(handshake\_messages \parallel k \parallel ipad))$$

and the use of SHA-1 yields

$$H_{\text{SHA-1}} = \text{SHA-1}(k \parallel opad \parallel \text{SHA-1}(handshake\_messages \parallel k \parallel ipad))$$

If RSA is used, then the digital signature is generated for $H_{\text{MD5}} \parallel H_{\text{SHA-1}}$, whereas it is generated only for $H_{\text{SHA-1}}$ in the case of DSA. In either case, the server can verify the signature with the public key that it can extract from the client certificate.

2.2.2.11 FINISHED Message

A FINISHED message is always sent immediately after a CHANGECIPHERSPEC message that is part of the SSL change cipher spec protocol (Section 2.2.3). The aim of the FINISHED message is to verify that the key exchange and authentication processes have been successfully terminated. As such, it is the first message that is protected with the newly negotiated algorithms and keys. No further acknowledgment is needed, meaning that the parties may begin sending encrypted data immediately after having sent a FINISHED message.

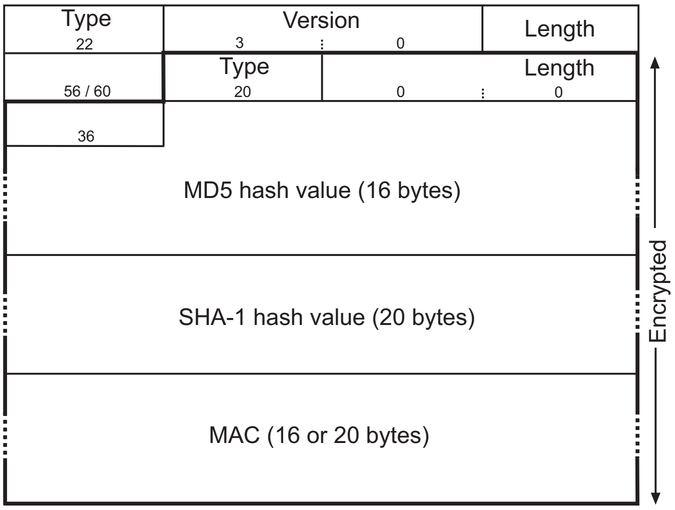

Figure 2.21 FINISHED message wrapped in an SSL record.

As illustrated in Figure 2.21, an SSL FINISHED message starts with the usual 5-byte SSL record header and is entirely protected (according to the new cipher suite). This means that a MAC is appended to the message and that the entire message—including the header—is encrypted. The message header comprises a 1-byte type field with value 0x14 or 20 (referring to a FINISHED message) and a 3-byte message length field, whereas the payload comprises a keyed 16-byte MD5 hash value and a keyed 20-byte SHA-1 hash value that are computed as follows:

$$h(k \parallel opad \parallel h(handshake\_messages \parallel sender \parallel k \parallel ipad))$$

Again, h refers to MD5 or SHA-1, k to the master secret, $ipad$ and $opad$ to the values introduced earlier in this chapter, $handshake\_messages$ to the concatenation of all SSL handshake messages that have been exchanged so far[44] (this value is different from the value used for the CERTIFICATEVERIFY message), and $sender$ to the entity that sends the FINISHED message. If the client sends the message, then this value is 0x434C4E54, whereas it is 0x53525652, if the server sends it. Note the similarity between this calculation and the hash calculation for the CERTIFICATEVERIFY message; the only differences refer to the inclusion of the sender and the different base for the construction of $handshake\_messages$. The length of the FINISHED message body is 36 bytes (16 bytes for the MD5 hash value and 20 bytes for the SHA-1 hash value). This value is written into the length field of the FINISHED message. Contrary to that, the length field of the SSL record contains the value 56 (if MD5 is used for message authentication) or 60 (if SHA-1 is used for message authentication). These values are also indicated in Figure 2.21, but keep in mind that these values refer to the cryptographic protection of the SSL record (and do not belong to the SSL record per se).

2.2.3 SSL Change Cipher Spec Protocol

As mentioned before, the SSL change cipher spec protocol is a protocol of its own that allows the communicating peers to signal to each other that they are going to switch from the current state to the newly established pending state (the FINISHED message is then the first message that is protected according to the new state). The SSL change cipher spec protocol is very simple: It basically consists of a single message (i.e., a CHANGECIPHERSPEC message) that is compressed and encrypted according to the current state. The placement of the CHANGECIPHERSPEC messages in a normal SSL handshake is illustrated in Figure 2.5. When resuming a previously established SSL session, the CHANGECIPHERSPEC message is sent immediately after the hello messages as shown in Figure 2.6.

44 Note that CHANGECIPHERSPEC messages are not SSL handshake messages, and hence they are not included in the hash computations.

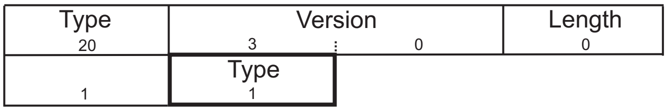

Figure 2.22 CHANGECIPHERSPEC message wrapped in an SSL record.

As illustrated in Figure 2.22, an SSL CHANGECIPHERSPEC message starts with a 5-byte SSL record header, this time referring to type 20 (standing for the SSL change cipher spec protocol). The rest of the SSL record header remains unchanged and includes a version and a length field. The length field value is actually set to one, because the CHANGECIPHERSPEC message includes only a 1-byte type field. This field is a placeholder that can currently only hold the value one.

The CHANGECIPHERSPEC message is unique in that it is not properly part of the SSL handshake but rather has its own content type and hence represents an SSL (sub)protocol of its own. Because the CHANGECIPHERSPEC message is not encrypted but the FINISHED message is, the two messages have different content types and cannot be transmitted in the same SSL record. Note, however, that the subtlety of using different content types also complicates the state machine of an SSL protocol implementation, and has therefore been a subject of controversial discussion for a long time. In the end, the change cipher spec (sub)protocol has been removed in TLS 1.3.

2.2.4 SSL Alert Protocol

The SSL alert protocol is yet another SSL (sub)protocol that allows the communicating peers to exchange alert messages. Each alert message comes with an alert level and an alert description:

- The *alert level* comprises 1 byte, where 0x01 stands for "warning" and 0x02 stands for "fatal." For all alert messages for which a particular level is not explicitly specified, the sender may determine at its discretion whether it is fatal or not. Similarly, if an alert with an alert level of warning is received, the receiver may decide at its own discretion whether to treat it as fatal error. Anyway, all alert messages that are fatal must be treated appropriately, meaning that they must result in the immediate termination of the connection.
- The *alert description* also comprises a 1-byte code that refers to a specific situation. The alert messages—with their codes and brief descriptions—are summarized in Table 2.7. For example, code 0 stands for the closure alert

Table 2.7
SSL Alert Messages

Alert	Code	Brief Description
`close_notify`	0	The sender notifies the recipient that it will not send any more messages on the connection. This alert is always a warning.
`unexpected_message`	10	The sender notifies the recipient that an inappropriate message was received. This alert is always fatal and should never be observed in communication between proper implementations.
`bad_record_mac`	20	The sender notifies the recipient that a record with an incorrect MAC was received. This alert is always fatal and should never be observed in communication between proper implementations.
`decompression_failure`	30	The sender notifies the recipient that the decompression function received improper input, meaning that it could not decompress the received data. This alert is always fatal and should never be observed in communication between proper implementations.
`handshake_failure`	40	The sender notifies the recipient that it was unable to negotiate an acceptable set of security parameters given the options available. This alert is always fatal.
`no_certificate`	41	The sender (which is always a client) notifies the recipient (which is always a server) that it has no certificate that can satisfy the server's certificate request. Note that this alert is only used in SSL (it is no longer used in TLS).
`bad_certificate`	42	The sender notifies the recipient that the certificate provided is corrupt (e.g., it contains a signature that cannot be verified correctly).
`unsupported_certificate`	43	The sender notifies the recipient that the certificate provided is of an unsupported type.
`certificate_revoked`	44	The sender notifies the recipient that the certificate provided has been revoked by the issuer.
`certificate_expired`	45	The sender notifies the recipient that the certificate provided has expired or is not currently valid.
`certificate_unknown`	46	The sender notifies the recipient that some unspecified issue arose in processing the certificate provided, rendering it unacceptable.
`illegal_parameter`	47	The sender notifies the recipient that a field in the SSL handshake message was out of range or inconsistent with some other field. This alert is always fatal.

`close_notify` that notifies the recipient that the sender will not send any more messages. Note that the sender and the server must share knowledge that a connection is ending in order to avoid a truncation attack and that either party may initiate a closure by sending a `close_notify` alert accordingly. Any data received after such an alert must be ignored. In addition to the closure alert, there are a number of other alert messages that refer to error alerts. When an error is detected, the detecting party sends a message to the other party. Upon transmission or receipt of a fatal alert message, both parties immediately close the connection and drop any information related to it.

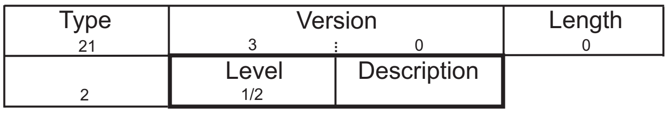

Figure 2.23 ALERT message wrapped in an SSL record.

As illustrated in Figure 2.23, an SSL ALERT message starts with a 5-byte SSL record header, this time referring to type 21 (standing for the SSL alert protocol). The rest of the SSL record header remains the same and includes a version and a length field. The length is actually set to two, because the ALERT message includes only two bytes: one byte referring to an alert level and one byte referring to an alert code. So both the SSL change cipher spec and the SSL alert protocols are very simple.

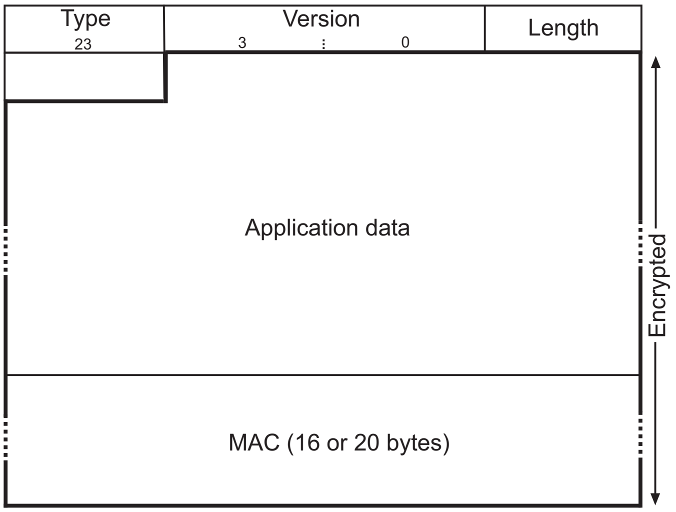

Figure 2.24 Application data wrapped in an SSL record (stream cipher).

2.2.5 SSL Application Data Protocol

As its name suggests, the SSL application data protocol allows the communicating peers to exchange data according to some application-layer protocol. More specifically, it takes application data and feeds it into the SSL record protocol for fragmentation, compression, and cryptographic protection. The resulting SSL records are sent to the recipient, where the application data is decrypted, verified, decompressed, and reassembled.

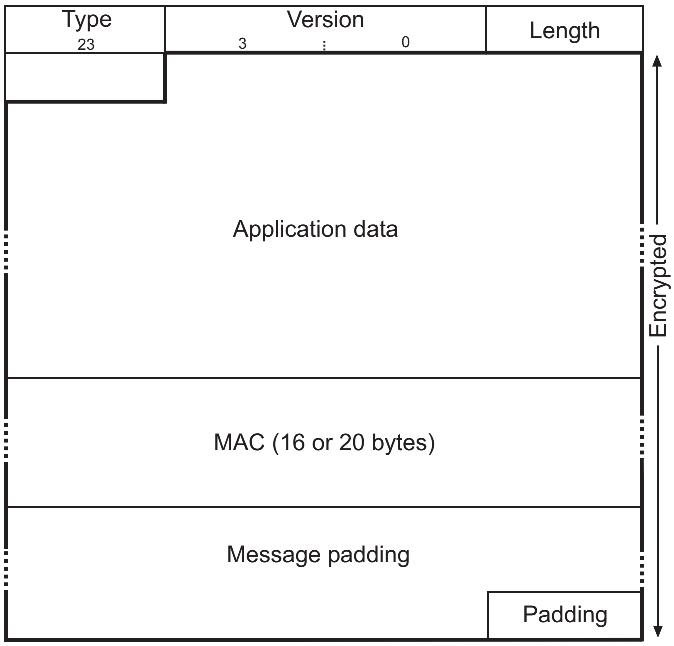

Figure 2.25 Application data wrapped in an SSL record (block cipher).

Figure 2.24 illustrates some application data encapsulated in an SSL record. As usual, the SSL record starts with a 5-byte header, including a 1-byte type field (referring to 0x17 or 23 standing for the SSL application data protocol), a 2-byte version field, and a 2-byte length field. Everything after the SSL record header is cryptographically protected according to the current state (i.e., it is authenticated using a MAC and encrypted using a cipher). Figure 2.24 illustrates the situation, if a stream cipher is used. If a block cipher is used, then some message padding may occur, and this situation is illustrated in Figure 2.25. As we will see later on (when we address padding oracle attacks), the last byte of the padding usually refers to the padding length.

2.3 PROTOCOL TRANSCRIPT

To illustrate the working principles of the SSL protocol, we consider a setting in which a client (i.e., a web browser) tries to access an SSL-enabled web server, and we use a network protocol analyzer like Wireshark to capture the SSL records that are sent back and forth. The dissection of these records shows what is going on behind the scenes (i.e., at the protocol level). Before the SSL protocol can be invoked, the client must establish a TCP connection to the server. We ignore this step and simply assume such a TCP connection to already exist between the client and server.

In our example, the client initiates the session establishment and sends a ClientHello message to the server. This message is encapsulated in an SSL record that may look as follows (in hexadecimal notation):

```
16 03 00 00 41 01 00 00   3d 03 00 48 b4 54 9e 00
6b 0f 04 dd 1f b8 a0 52   a8 ff 62 23 27 c0 16 a1
59 c0 a9 21 4a 4e 3e 61   58 ed 25 00 00 16 00 04
00 05 00 0a 00 09 00 64   00 62 00 03 00 06 00 13
00 12 00 63 01 00
```

The SSL record starts with a type field that comprises the value 0x16 (representing 22 in decimal notation, and hence standing for the SSL handshake protocol), a version field that comprises the value 0x0300 (referring to SSL 3.0), and a length field that comprises the value 0x0041 (representing 65 in decimal notation). This basically means that the fragment of the SSL record is 65 bytes long and that the subsequent 65 bytes thus represent the ClientHello message. (This refers to the entire byte sequence displayed above.) This message, in turn, starts with 0x01 standing for the SSL handshake message type 1 (referring to a ClientHello message), 0x00003d standing for a message length of 61 bytes, and 0x0300 again representing SSL 3.0. The subsequent 32 bytes—from 0x48b4 to 0xed25—represent the random value chosen by the client (remember that the first 4 bytes represent the date and time). Because there is no SSL session to resume, the session ID length is set to zero (i.e., 0x00) and no session ID is appended afterwards. Instead, the next value 0x0016 (representing 22 in decimal notation) indicates that the subsequent 22 bytes refer to the 11 cipher suites that are supported by the client. Each pair of bytes represents a cipher suite (Table 2.4 and Appendix B). The second-to-last byte 0x01 indicates that there is a single compression method supported by the client, and the last byte 0x00 refers to this compression method (which actually refers to null compression).

After having received the CLIENTHELLO message, the server is to respond with a series of SSL handshake messages. If possible, all messages are merged into a single SSL record and transmitted in a single TCP segment to the client. In our example, such an SSL record comprises a SERVERHELLO, a CERTIFICATE, and a SERVERHELLODONE message. The corresponding SSL record starts with the following sequence of five bytes:

```
16 03 00 0a 5f
```

Again, 0x16 refers to the SSL handshake protocol, 0x0300 refers to SSL version 3.0, and 0x0a5f refers to the length of the SSL record (which is actually 2,655 bytes in this example). The three abovementioned messages are then encapsulated in the rest of the SSL record.

- The SERVERHELLO message looks as follows:

  ```
  02 00 00 46 03 00 48 b4   54 9e da 94 41 94 59 a9
  64 bc d6 15 30 6c b0 08   30 8a b2 e0 6d ea 8f 7b
  6b df d5 a7 3c d4 20 48   b4 54 9e 26 8b a1 9d 26
  59 1b 5e 31 4c fe d3 2b   a7 96 26 99 55 55 41 7c
  d8 e8 44 8a 3e f9 d5 00   05 00
  ```

  The message starts with 0x02 standing for the SSL handshake protocol message type 2 (referring to a SERVERHELLO message), 0x000046 standing for a message length of 70 bytes, and 0x0300 again standing for SSL 3.0. The subsequent 32 bytes

  ```
  48 b4 54 9e da 94 41 94   59 a9 64 bc d6 15 30 6c
  b0 08 30 8a b2 e0 6d ea   8f 7b 6b df d5 a7 3c d4
  ```

  represent the random value chosen by the server (note again that the first 4 bytes represent the date and time). Afterward, 0x20 refers to a session ID length of 32 bytes, and hence the subsequent 32 bytes

  ```
  48 b4 54 9e 26 8b a1 9d   26 59 1b 5e 31 4c fe d3
  2b a7 96 26 99 55 55 41   7c d8 e8 44 8a 3e f9 d5
  ```

  represent the session ID. Remember that this ID is going to be used if the client wants to resume the SSL session at some later point in time (before the session expires). Following the session ID, 0x0005 refers to the selected cipher suite (which stands for TLS_RSA_WITH_RC4_128_SHA), and 0x00 refers to the selected compression method (which stands for null compression).

- Next, the CERTIFICATE message comprises the server's public key certificate. It is quite long and begins with the following byte sequence:

  ```
  0b 00 0a 0d 00 0a 0a
  ```

  In this byte sequence, 0x0b stands for the SSL handshake protocol message type 11 (referring to a CERTIFICATE message), 0x000a0d stands for a message length of 2,573 bytes, and 0x000a0a stands for the length of the certificate chain. Note that the length of the certificate chain must equal the message length minus 3 (that refers to the length of the length field). The remaining 2,570 bytes of the message then comprise the certificate chain required to validate the server's public key certificate (these bytes are not illustrated here).

- Last but not least, the SSL record also comprises a SERVERHELLODONE message. This message is very simple and only consists of four bytes:

  ```
  0e 00 00 00
  ```

  0x0e stands for the SSL handshake protocol message type 14 (referring to a SERVERHELLODONE message) and 0x000000 stands for a message length of zero bytes.

After having received the SERVERHELLODONE message, it is up to the client to submit a series of messages to the server. In our example, this series comprises a CLIENTKEYEXCHANGE, a CHANGECIPHERSPEC, and a FINISHED message. Each of these messages is transmitted in an SSL record of its own, but all three records can be transmitted in a single TCP segment to the server.

- The CLIENTKEYEXCHANGE message is transmitted in the first SSL record. In our example, this record looks as follows:

  ```
  16 03 00 00 84 10 00 00   80 18 4a 74 7e 92 66 72
  fa ee ac 4b f8 fb 7c c5   6f b2 55 61 47 4e 1e 4a
  ad 5f 4b f5 70 fe d1 b4   0b ef 36 52 4f 7b 33 34
  ad 23 67 f0 60 ec 67 67   35 5a cf 50 f8 d0 3d 28
  4e fb 01 88 56 06 86 3c   c7 c3 85 8c 81 2c 0d d8
  20 a6 1b 09 ee 86 c5 6c   37 e5 e8 56 96 cc 46 44
  58 ee c1 9b 73 53 ff 88   ab 90 19 53 3d f2 23 5b
  8f 57 d2 b0 74 2a bd 05   f9 9e dd 6a 50 69 50 4a
  55 8a f1 5b 9b 6d ba 6f   b0
  ```

  In the SSL record header, 0x16 stands for the SSL handshake protocol, 0x0300 refers to SSL version 3.0, and 0x0084 represents the length of the SSL record (132 bytes). After this header, the byte 0x10 stands for the SSL handshake protocol message type 16 (referring to a CLIENTKEYEXCHANGE message), and the following three bytes 0x000080 refer to the message length (128 bytes or 1,024 bits). Consequently, the remaining 128 bytes of the message represent the premaster secret (as chosen by the client) encrypted with the server's public RSA key. The RSA encryption is in line with PKCS #1.

- The CHANGECIPHERSPEC message is transmitted in the second SSL record. This record is very simple and consists only of the following 6 bytes:

  ```
  14 03 00 00 01 01
  ```

  In the SSL record header, 0x14 (20 in decimal notation) stands for the SSL change cipher spec protocol, 0x0300 refers to SSL version 3.0, and 0x0001 represents the message length of a single byte. This byte (i.e., 0x01) is the last byte of the record.

- The FINISHED message is the first message that is cryptographically protected according to the newly negotiated cipher spec. Again, it is transmitted in an SSL record of its own and may look as follows:

  ```
  16 03 00 00 3c 38 9c 10   98 a9 d3 89 30 92 c2 41
  52 59 e3 7f c7 b3 88 e6   5f 6f 33 08 59 84 20 65
  55 c2 82 cb e2 a6 1c 6f   dc c1 13 4b 1a 45 30 8c
  e5 f4 01 1a 71 08 06 eb   5c 54 be 35 66 52 21 35
  f1
  ```

  In the SSL record header, 0x16 stands for the SSL handshake protocol, 0x0300 refers to SSL version 3.0, and 0x003c represents the length of the SSL record (60 bytes). These 60 bytes are encrypted and look like gibberish to somebody not knowing the appropriate decryption key. They comprise MD5 and SHA-1 hash values of all messages that have been exchanged so far, as well as a SHA-1-based MAC. As discussed in the context of Figure 2.21, the length value to be included in such an SSL record header is 60 bytes.

After having received the CHANGECIPHERSPEC and FINISHED messages, the server must respond with the same pair of messages (not illustrated here). Afterward, application data can be exchanged in SSL records. Such a record may start as follows:

```
17 03 00 02 73
```

Here, 0x17 (23 in decimal notation) stands for the SSL application data protocol, 0x0300 for SSL version 3.0, and `0x0273` (627) for the length of the encrypted data fragment. It goes without saying that an arbitrary number of SSL records may be exchanged between the client and the server, where each record may comprise a data fragment.

2.4 SECURITY ANALYSIS

As a result of its initial success, many researchers explored the security of the SSL protocol in the 1990s. For example, soon after Netscape Communications released its first browser supporting SSL in 1996, David Wagner and Ian Goldberg showed that the method used to seed the pseudorandom generator (and hence the method to generate a premaster secret) in the browser was cryptographically weak,[45] meaning that the premaster secrets were predictable to some extent. The problem was due the fact that a particular seed was derived from a few deterministic values, such as the process ID, the ID of the parent process, and the current time, and that these values did not provide enough entropy. Note that this is not a problem of SSL per se, but rather of how the protocol was implemented by Netscape Communications. Anyway, the resulting vulnerability made the press headlines and cast a damning light on the security of the then-evolving SSL protocol. The problem could easily be solved by strengthening the pseudorandom generator, and this was quickly done by Netscape Communications.

Later in 1996, Wagner and Bruce Schneier did a (yet informal) security analysis of the SSL protocol versions 2 and 3 [17]. As a result of their analysis, they reported some weaknesses and possibilities to mount active attacks against SSL 2.0 (that was prohibited in 2011 [18]). Due to an ongoing competition between Netscape Communications and Microsoft in the early days of SSL (Section 1.2), most weaknesses of SSL 2.0 had already been corrected in SSL 3.0. So when Wagner and Schneier did their analysis, they found only a few attacks that could be mounted against SSL 3.0, such as a key exchange algorithm confusion or a version downgrade attack. They concluded that the overall security of SSL 3.0 was positive, and so they wrote that "on the whole SSL 3.0 is a valuable contribution towards practical communications security" [17].

In the aftermath of the Wagner-Schneier analysis, several researchers analyzed the security of SSL 3.0 with formal methods [19, 20].[46] Again, the results and key findings were overwhelmingly positive in the sense that no major vulnerability was found. This reaffirmed to the community that SSL 3.0 was indeed a reasonably secure protocol—at least in theory.

In practice, however, even a secure protocol can be implemented insecurely or at least provide some hook from where an attack can start. In 1998, for example,

45 http://www.drdobbs.com/windows/randomness-and-the-netscape-browser/184409807.

46 While [20] refers to the TLS protocol, most findings also apply to SSL 3.0.

Daniel Bleichenbacher discovered such a hook in the way SSL 3.0 pads the premaster secret (according to PKCS #1 version 1.5 [6]) before it is RSA-encrypted in a CLIENTKEYEXCHANGE message [21]. In fact, Bleichenbacher showed how to mount an adaptive chosen-ciphertext attack (CCA2[47]) to decrypt the premaster secret. If the adversary is successful, then he or she can derive the keying material required to break the security of the entire SSL session.

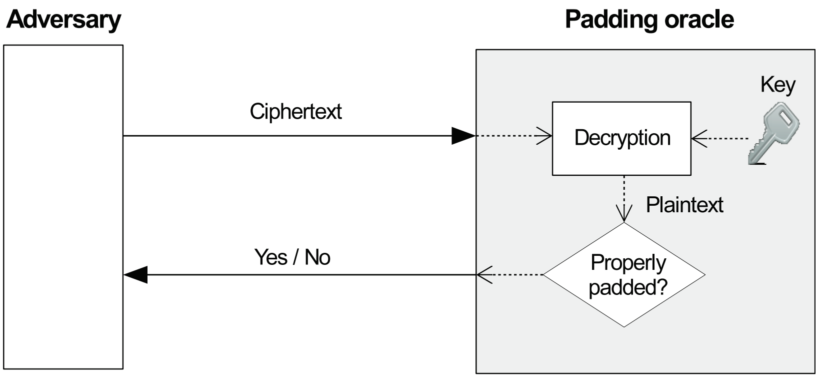

Figure 2.26 A padding oracle attack.

The Bleichenbacher attack can be seen as the first incarnation of what was later named *padding oracle attack* (i.e., a CCA2 in which the adversary has access to a padding oracle). A padding oracle, in turn, is an oracle (or function) that takes as input a ciphertext and outputs one bit of information, namely whether the plaintext that results after decryption is properly padded or not. As illustrated in Figure 2.26, the adversary (on the left side) has access to a padding oracle (on the right side). This oracle is just a black box with a specific input-output behavior. In particular, it takes as input a ciphertext and it generates as output one bit of information representing “Yes” or “No.” To produce the output, the padding oracle internally decrypts the ciphertext using the proper (decryption) key and then answers the question of whether the underlying plaintext message is properly padded (Yes) or not (No). Note that the plaintext message itself is kept only internally and that it

47 In the cryptographic literature, a chosen ciphertext attack is acronymed CCA and an adaptive chosen-ciphertext attack is acronymed CCA2.

is not returned to the adversary. It goes without saying that the notion of a padding oracle only makes sense, if the encryption in use employs some form of padding, such as, for example, in asymmetric encryption according to PKCS #1 version 1.5 or—as mentioned below—symmetric encryption with a block cipher operated in CBC mode.[48]

One problem of a padding oracle attack in general, and the Bleichenbacher attack in particular, is that the padding oracle reveals only one bit of information in every request. This means that an adversary requires a huge quantity of oracle queries. The Bleichenbacher attack, for example, requires roughly one million of oracle queries and is therefore also known as the *million message attack*. Although the attack has been improved considerably (to sometimes only a few 10,000 oracle queries), it can still be detected based on the many CLIENTKEYEXCHANGE messages that need to be sent to the server.

To mitigate the Bleichenbacher attack, one has to ensure that the server (that is under attack) does not leak any information about the correctness of the padding, including any timing information that correlates with the padding. The simplest way to achieve this is to treat mispadded messages as if they were padded properly [22]. So when a server receives a CLIENTKEYEXCHANGE message with a bad padding, it does not return an error message immediately, but rather randomly generates a premaster secret and continues the protocol with this value. The client and server then end up with cryptographic keys that are distinct, and hence the SSL session cannot be established, but no information about the padding is leaked. This seems to mitigate the attack and was therefore included in the later specified versions of the TLS protocol.

The technical details of the Bleichenbacher attack are involved and subtle (they are explained in Section A.1). Here, we only want to raise two insights that are related to the attack:

- First, the attack clearly demonstrates that CCA2 are practical and must be considered seriously. Prior to the attack, it had been thought that these attacks were purely theoretical and could not be mounted in a real-world setting. So the attack has had (and continues to have) a deep impact on cryptographic research.

- Second, the attack also demonstrated the need to update PKCS #1 version 1.5. In the same year as the attack took place, a technique known as *optimal asymmetric encryption padding* (OAEP) [23] was adopted in PKCS #1 version 2.0 [7]. Unlike ad hoc padding schemes, such as the one employed in PKCS

48 Today, some people argue that the term *padding oracle attack* only applies to the symmetric encryption version. This argument is not followed in this book. Instead, we use the term to refer to both versions—the symmetric and the asymmetric ones.

#1 version 1.5, OAEP can be proven secure against CCA2. The big advantage of OAEP is that it only modifies the way messages are padded before they are encrypted (so RSA-OAEP is the acronym of the encryption system that combines RSA and OAEP). The disadvantage of OAEP is that its security proof is only valid in the random oracle model (and not the standard model). This is disadvantageous, because the random oracle model itself is controversially discussed in the community. There are asymmetric encryption systems that are provably secure against CCA2 in the standard model (e.g., [24]), but people still prefer a simple patch for RSA, such as the one provided by OAEP (even RSA-OAEP was not immediately adopted by industry).

In the aftermath of the Bleichenbacher attack, many researchers tried to find optimizations, variants, and extensions of it. For example, in 2001, James Manger found a timing channel that can be used to mount a CCA2 against some widely deployed implementations of PKCS #1 version 2.0 [25]. Similarly, three Czech cryptologists—Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa—found another variant of the Bleichenbacher attack in 2003 [26]. The respective Klíma-Pokorný-Rosa attack exploits alert messages that are sent during the execution of the SSL/TLS protocol when resulting plaintexts are incorrectly long or contain incorrect version numbers. After 2003, some researchers have found other variants of the Bleichenbacher attack that can even be mounted remotely against network servers (e.g., [27, 28]).

After the publication of the Manger, Klíma-Pokorný-Rosa, and a few other attacks, PKCS #1 was updated to version 2.1 in 2003 [8], mainly to make these attacks more difficult to mount. In 2012, PKCS #1 was updated again to version 2.2, but this update was not motivated by security concerns. Implementing public key cryptography and protecting implementations against side-channel attacks remain timely and practically relevant topics. Unfortunately, they are not particularly simple and straightforward.

In addition to the Bleichenbacher attack and its many variants, some researchers have found other (mostly subtle) security problems in some SSL implementations. In 2002, for example, Serge Vaudenay published a paper in which he explained how CBC padding may induce a side channel that can be exploited in another padding oracle attack [29]. This publication did not come along with a feasible attack. But only one year later, in 2003, Vaudenay (and a few other researchers) published a follow-up paper in which they showed that the CBC padding problem could actually be turned into a feasible attack [30]. The attack is known as *Vaudenay attack*, and it is fully explained in Section A.2 (together with some variants, including Lucky 13). As the attack was published after the official release of TLS 1.0, SSL 3.0 had no built-in countermeasures in place. Such countermeasures were designed and retrofitted into TLS 1.1 after the publications of the Vaudenay attack. Some of them are addressed in the next chapter.

Another vulnerability that affects block ciphers operated in CBC mode was reported by Gregory Bard in 2004 [31]. It exploited the way most SSL implementations chose an IV when using a block cipher in CBC mode to encrypt the next record. Instead, of using a randomly chosen value to serve as IV, they used the last ciphertext block of the previous record. This is sometimes known as "IV chaining," and had been known to be dangerous since 1995. Bard showed how to exploit this vulnerability in a blockwise chosen-plaintext attack (CPA).[49] The severity of the attack was recognized in public in 2011, when it was shown by Thai Duong and Juliano Rizzo how to implement the attack in a devastating attack tool named *browser exploit against SSL/TLS* (BEAST). The problems of IV chaining, the attack reported by Bard, and BEAST are fully explained in Section A.3.

In 2014, things got worse when it was found that the overly simple padding scheme of SSL 3.0 enabled a very simple and devastating Vaudenay-type of padding oracle attack. The attack was named *padding oracle downgraded legacy encryption* (POODLE). Together with the successful attacks against RC4, it brought SSL to the end of its life cycle [32]. The POODLE attack is overviewed and explained in detail in Section A.4.

In the aftermath of the BEAST and POODLE attacks, it was often recommended to use a stream cipher instead of a block cipher in CBC mode. But the only stream cipher available at that time was RC4. Unfortunately, this was not an appropriate solution, because RC4 has security problems of its own [33–35].[50] For example, RC4 is known to have biases in the first 256 bytes that are generated in the key stream: The second byte is known to be biased toward zero with a probability that is twice as large as it should be (i.e., $1/128$ instead of $1/256$). The same is true for double-byte and—more generally—multiple-byte biases. To exploit these statistical weaknesses, an adversary has to obtain the same plaintext message encrypted with many different keys (we are talking about millions here). This is not simple, but still feasible, and the feasibility was reaffirmed in an attack codenamed RC4 NOMORE[51] [36]. Due to these results, most standardization bodies prohibit the use of RC4 today [16, 37].

More recently, many researchers have shown that padding oracle attacks (of the Bleichenbacher or Vaudenay type) can also be mounted against cryptographic

49 Note that a CPA is generally simpler to mount than a CCA, especially in the realm of public key cryptography, where public keys are available by default.

50 Some time ago, it was shown that RC4 as used in the *wired equivalent privacy* (WEP) protocol is insecure. This insecurity applies to WEP, but it does not automatically also apply to SSL/TLS. In fact, the flaws that had been made in applying RC4 to WEP had not been made in SSL/TLS.

51 https://www.rc4nomore.com.

devices (and their key import functions) that conform to PKCS #11 and that the respective attacks are surprisingly efficient. These results shed a dimmed light on the security of some hardware tokens that are used in the field [38, 39]. The cryptanalytical strength of padding oracle attacks is widely recognized, and designing and coming up with implementations that are resistant against such attacks has become an important goal in cryptographic research and development.

As a result of the POODLE attack and the then known statistical weaknesses of RC4 (that later led to the RC4 NOMORE attack), the IETF deprecated SSL 3.0 in 2015 [16]. This means that SSL 3.0 should no longer be used and be disabled by default. If this is not possible, then it should at least be ensured that any protocol downgrade happens with the client's blessing. This is where the signaling cipher suite value (SCSV) TLS_FALLBACK_SCSV comes into play.[52] An SCSV is some sort of a "dummy cipher suite" that is used only to signal some information from the client to the server or vice versa. In the case of the TLS_FALLBACK_SCSV, for example, the information refers to the fact that the client is knowingly repeating an SSL/TLS connection attempt over a lower protocol version than it really supports (because the last one failed for some reason). It is up to the server to compare this version with the highest version it supports. If the server supports a version higher than the one indicated by the client, then the server knows that something fishy is going on. It then aborts the connection and sends back an `inappropriate_fallback` alert message (value 86) that is fatal. This simple mechanism is specified in [40] and can mitigate some downgrade attacks. It has been used in all TLS protocol versions up to 1.2 (TLS 1.3 uses a different mechanism). More recently, all earlier TLS versions have been deprecated [41], and hence the mechanism is no longer needed—at least not in theory.

2.5 FINAL REMARKS

This chapter has introduced, overviewed, and detailed the SSL (3.0) protocol. This will help us to better understand the TLS and DTLS protocols and to shorten the respective explanations considerably. The SSL protocol is simple and straightforward—especially if RSA or Diffie-Hellman is used for key exchange. (Note that ephemeral Diffie-Hellman is the preferred key exchange mechanism today, mainly because it provides forward secrecy.) There are only a few details that

52 Note that TLS_FALLBACK_SCSV is not the only SCSV used in the field. For example, TLS_EMPTY_RENEGOTIATION_INFO_SCSV is another SCSV that allows a client to signal to a server that it supports secure renegotiation (to protect against the vulnerability mentioned in CVE-2009-3555).

can be discussed controversially, such as the use of a separate content type for CHANGECIPHERSPEC messages, and this has even changed meanwhile.

From a security perspective, simplicity and straightforwardness are advantageous properties, and hence the starting position of the SSL protocol with regard to security was very promising. During the first decade, no serious vulnerability was found. This only changed when Bleichenbacher, Vaudenay, and Bard published their results, and—even more so—when some attacks and attack tools that exploited the vulnerabilities they found were presented at security conferences worldwide. Most importantly, the POODLE and RC4 attacks brought the SSL protocol to the end of its life cycle. As mentioned before, the IETF deprecated SSL 3.0 in 2015 [16] and has since then recommended its replacement with more recent versions of the TLS protocol (i.e., TLS 1.2 and TLS 1.3). The browser vendors have followed this recommendation and disabled SSL 3.0 by default.

Like any other security technology, the SSL protocol has a few disadvantages and pitfalls (some of which also apply to TLS). For example, the use of SSL makes content screening impossible. If a data stream is encrypted using, for example, the SSL protocol with a cryptographically strong cipher, then it is no longer possible to subject the data stream to content screening and inspection. This is because the content screener only sees encrypted data in which it cannot efficiently find malicious content (even if it is using artificial intelligence, machine learning, or any other hyped technology). In order to screen content, it is necessary to temporarily decrypt the data stream and to reencrypt it just after the screening process. This calls for an SSL proxy (as further addressed in Section 5.3). Another problem that pops up when the SSL protocol is used in the field is the need for public key certificates. As mentioned above, an SSL-enabled web server always needs a certificate and must be configured in a way that it can make use of it. Additionally, a web server can also be configured in a way that it requires clients to authenticate themselves with a public key certificate. In this case, however, the clients must also be equipped with public key certificates. As there are many potential clients for a web server, the process of equipping clients with certificates is involved and has turned out to be slow—certainly slower than originally anticipated. Using client certificates over a proxy server does not make things simpler (mainly because client certificates provide end-to-end authentication but still need to be verified on the way). The original designers of the SSL protocol therefore opted to make client authentication optional in the first case. If client authentication is not optional (but mandatory), then people sometimes use the term *mutual TLS* (mTLS) to emphasize the fact that the client and server mutually authenticate each other.[53] This is certainly the preferred choice, but sometimes it is impossible due to a lack of client-side certificates. There

53 This term is sometimes also used in an SSL setting (in which the never used term mSSL should be used instead).

is much more to say about public key certificates and internet PKIs, and we allocate Chapter 6 for this topic.

References

[1] Freier, A., P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101, August 2011.

[2] Khare, R., and S. Lawrence, "Upgrading to TLS Within HTTP/1.1," RFC 2817, May 2000.

[3] Rescorla, E., "HTTP Over TLS," RFC 2818, May 2000.

[4] Hoffman, P., "SMTP Service Extension for Secure SMTP over TLS," RFC 2487, January 1999.

[5] Klensin, J., et al., "SMTP Service Extensions," RFC 1869 (STD 10), November 1995.

[6] Kaliski, B., "PKCS #1: RSA Encryption Version 1.5," RFC 2313, March 1998.

[7] Kaliski, B., and J. Staddon, "PKCS #1: RSA Cryptography Specifications Version 2.0," RFC 2437, October 1998.

[8] Jonsson, J., and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," RFC 3447, February 2003.

[9] Canetti, R., and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels," *Proceedings of EUROCRYPT '01,* Springer-Verlag, LNCS 2045, 2001, pp. 453–474.

[10] Krawczyk, H., "The Order of Encryption and Authentication for Protecting Communications (Or: How Secure is SSL?)," *Proceedings of CRYPT0 2001*, Springer-Verlag, LNCS 2139, 2001, pp. 310–331.

[11] Lepinski, M., and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards," RFC 5114, January 2008.

[12] Joux, A., "Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions," *Proceedings of CRYPT0 2004*, Springer-Verlag, LNCS 3152, 2004, pp. 316–316.

[13] Beurdouch, B., et al., "A Messy State of the Union: Taming the Composite State Machines of TLS," *Proceedings of the 36th IEEE Symposium on Security and Privacy,* San José, CA, May 2015, pp. 535–552.

[14] Adrian, D., et al., "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," *Proceedings of the ACM Conference in Computer and Communications Security,* ACM Press, New York, NY, 2015, pp. 5–17.

[15] Krawczyk, H., M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997.

[16] Popov, A., "Prohibiting RC4 Cipher Suites," RFC 7465, February 2015.

[17] Wagner, D., and B. Schneier, "Analysis of the SSL 3.0 Protocol," *Proceedings of the Second USENIX Workshop on Electronic Commerce,* USENIX Press, November 1996, pp. 29–40.

[18] Turner, S., and T. Polk, “Prohibiting Secure Sockets Layer (SSL) Version 2.0,” RFC 6176, March 2011.

[19] Mitchell, J., V. Shmatikov, and U. Stern, “Finite-State Analysis of SSL 3.0,” *Proceedings of the Seventh USENIX Security Symposium,* USENIX, 1998, pp. 201–216.

[20] Paulson, L.C., “Inductive Analysis of the Internet Protocol TLS,” *ACM Transactions on Computer and System Security,* Vol. 2, No. 3, 1999, pp. 332–351.

[21] Bleichenbacher, D., “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1,” *Proceedings of CRYPTO ’98,* Springer-Verlag, LNCS 1462, August 1998, pp. 1–12.

[22] Rescorla, E., “Preventing the Million Message Attack on Cryptographic Message Syntax,” RFC 3218, January 2002.

[23] Bellare, M., and P. Rogaway, “Optimal Asymmetric Encryption,” *Proceedings of EUROCRYPT ’94*, Springer-Verlag, LNCS 950, 1994, pp. 92–111.

[24] Cramer, R., and V. Shoup, “A Practical Public Key Cryptosystem Provably Secure Against Adaptive Chosen Ciphertext Attack,” *Proceedings of CRYPTO ’98,* Springer-Verlag, LNCS 1462, August 1998, pp. 13–25.

[25] Manger, J., “A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS#1 v2.0,” *Proceedings of CRYPTO ’01,* Springer-Verlag, August 2001, pp. 230–238.

[26] Klíma, V., O. Pokorný, and T. Rosa, “Attacking RSA-Based Sessions in SSL/TLS,” *Proceedings of Cryptographic Hardware and Embedded Systems (CHES),* Springer-Verlag, September 2003, pp. 426–440.

[27] Boneh, D., and D. Brumley, “Remote Timing Attacks are Practical,” *Proceedings of the 12th USENIX Security Symposium,* 2003, pp. 1–14.

[28] Aciiçmez, O., W. Schindler, and C.K. Koç, “Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations,” *Proceedings of the 12th ACM Conference on Computer and Communications Security,* ACM Press, New York, NY, 2005, pp. 139–146.

[29] Vaudenay, S., “Security Flaws Induced by CBC Padding—Applications to SSL, IPSEC, WTLS. . . ,” *Proceedings of EUROCRYPT ’02,* Springer-Verlag, LNCS 2332, 2002, pp. 534–545.

[30] Canvel, B., et al., “Password Interception in a SSL/TLS Channel,” *Proceedings of CRYPTO ’03,* Springer-Verlag, LNCS 2729, 2003, pp. 583–599.

[31] Bard, G.V., “Vulnerability of SSL to Chosen-Plaintext Attack,” Cryptology ePrint Archive, Report 2004/111, 2004.

[32] Barnes, R., et al., “Deprecating Secure Sockets Layer Version 3.0,” RFC 7568, June 2015.

[33] AlFardan, N., et al., “On the Security of RC4 in TLS,” *Proceedings of the 22nd USENIX Security Symposium,* USENIX, August 2013, pp. 305–320, http://www.isg.rhul.ac.uk/tls/RC4biases.pdf.

[34] Isobe, T. et al., "Full Plaintext Recovery Attack on Broadcast RC4," *Proceedings of the International Workshop on Fast Software Encryption (FSE 2013),* Springer-Verlag, LNCS 8424, 2013, pp. 179–202.

[35] Garman, C., K.G. Paterson, and T. van der Merwe, "Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS," *Proceedings of the 24th USENIX Security Symposium*, USENIX, 2015, 113–128.

[36] Vanhoef, M., and F. Piessens, "All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS," *Proceedings of the 24th USENIX Security Symposium*, USENIX, 2015, pp. 97–112.

[37] NIST Special Publication 800-52 Revision 1, "Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations," April 2014.

[38] Bortolozzo, M., et al., "Attacking and Fixing PKCS#11 Security Tokens," *Proceedings of the 17th ACM Conference on Computer and Communications Security,* ACM Press, 2010, pp. 260–269.

[39] Bardou, R., et al., "Efficient Padding Oracle Attacks on Cryptographic Hardware," *Proceedings of CRYPTO 2012,* Springer-Verlag, LNCS 7417, 2012, pp. 608–625.

[40] Moeller, B., and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks," RFC 7507, April 2015.

[41] Moriarty, K., and S. Fossati, "Deprecating TLS 1.0 and TLS 1.1," RFC 8996 (BCP 195), March 2021.

Chapter 3

TLS Protocol

In this chapter, we introduce, discuss, and put into perspective the second transport layer security protocol[1] mentioned in the title of the book (i.e., the TLS protocol). We assume the reader to be familiar with the SSL protocol as outlined in the previous chapter, and we confine ourselves to the differences between the SSL protocol and the various versions of the TLS protocol [1–4] (you may refer to Section 1.2 for a historical outline of the protocol evolution, ranging from TLS version 1.0 to 1.3). More specifically, we provide an introduction in Section 3.1, focus on the different versions of the TLS protocol in Sections 3.2 to 3.5, elaborate on HTTP strict transport security (HSTS) in Section 3.6, go through TLS protocol transcripts (for TLS 1.0 and TLS 1.2) in Section 3.7, briefly analyze the security of TLS in Section 3.8, and conclude with some final remarks in Section 3.9. This is going to be a tough ride and a long chapter, as it addresses the centerpiece of the subject matter of the book.

3.1 INTRODUCTION

The TLS protocol is structurally identical to the SSL protocol. It is a client/server protocol that is stacked on top of a reliable transport layer protocol, such as TCP in the case of the TCP/IP protocol suite, and that consists of the same two layers and (sub)protocols as the SSL protocol (with the only difference that the prefix "SSL" is replaced with "TLS"):

1 Like the SSL protocol, the TLS protocol does not, strictly speaking, operate at the transport layer. Instead, it operates at an intermediate layer between the transport layer and the application layer.

- On the lower layer, the *TLS record protocol* fragments, optionally compresses, and cryptographically protects higher-layer protocol data. The corresponding data structures are called `TLSPlaintext`, `TLSCompressed`, and `TLSCiphertext`. As with SSL (illustrated in Figure 2.4), each of these data structures represents a record and comprises a one-byte *type* field, a two-byte *version* field, another two-byte *length* field, and a *fragment* field that may be up to 2^{14} = 16,384 bytes long. The type, version, and length fields represent the TLS record header, whereas the fragment field represents the body or payload of the TLS record.
- On the higher layer, the TLS protocol comprises the following four protocols that we already know from the SSL protocol:
  - The *TLS change cipher spec protocol* (20);[2]
  - The *TLS alert protocol* (21);
  - The *TLS handshake protocol* (22);
  - The *TLS application data protocol* (23).

  Again, each protocol is identified with a unique content type value that is appended in brackets (i.e., 20, 21, 22, or 23). To allow future extensions, additional record types may be defined and supported by the TLS or DTLS record protocol. Examples include the record type 24 assigned to the Heartbeat extension[3] (Section C.2.14) and the record types 25 and 26 assigned to two DTLS specificities (Section 4.3 and Table 4.2). The content type is the value that is written into the type field of the TLS record header, so each record may comprise arbitrarily many messages of the same type and (sub)protocol.

Again, we use the term *TLS protocol* to refer to all four protocols itemized above, and we use a more specific term when we refer to a particular (sub)protocol.

Like the SSL protocol, the TLS protocol employs sessions and connections, where multiple connections may belong to the same session. Also like the SSL protocol, the TLS protocol simultaneously uses four connection states: the *current* read and write states, and the *pending* read and write states. The use of these states is identical to the SSL protocol. This means that all TLS records are processed under the current (read and write) states, whereas the security parameters and elements for the pending and next-to-be-used states are negotiated and set during the execution of

2 As outlined later in this chapter, the change cipher spec protocol is no longer used in TLS 1.3, or used only as a dummy message in some middlebox compatibility mode.

3 The Heartbeat extension was originally designed for DTLS, but it also works for TLS.

the TLS handshake protocol. This also means that the SSL state machine illustrated in Figure 2.2 equally applies to the TLS protocol.

Table 3.1
Security Parameters for a TLS Connection

`connection end`	Information about whether the entity is considered the client or the server in the connection
`bulk encryption algorithm`	Algorithm used for bulk data encryption (including its key size, how much of that key is secret, whether it is a block or stream cipher, and the block size if a block cipher is used)
`MAC algorithm`	Algorithm used for message authentication
`compression algorithm`	Algorithm used for data compression
`master secret`	48-byte secret shared between the client and the server
`client random`	32-byte value provided by the client
`server random`	32-byte value provided by the server

The state elements of a TLS session are essentially the same as the state elements of an SSL session (Table 2.2), so we don't repeat them here. At the connection level, however, the specifications of the SSL and TLS protocols are slightly different: While the TLS protocol distinguishes between the security parameters (Table 3.1) and some state elements (Table 3.2) of a connection, the SSL protocol does not make this distinction and only considers state elements (Table 2.3). However, taking the security parameters of Table 3.1 together with the state elements of Table 3.2, the differences between SSL and TLS connections are rather small.[4]

Table 3.2
TLS Connection State Elements

`compression state`	The current state of the compression algorithm
`cipher state`	The current state of the encryption algorithm (this includes all values needed to execute the algorithm, such as a key and an IV if the cipher is operated in CBC mode)
`MAC secret`	MAC secret for this connection
`sequence number`	64-bit sequence number for the records transmitted under a particular connection state (initially set to zero)

But there is a subtlety to note. Remember from Table 2.3 that there are four keys that refer to state elements of an SSL connection:

4 In addition to the security parameters summarized in Table 3.1, the PRF algorithm is yet another security parameter that was introduced in TLS 1.2 (because prior versions of the TLS protocol use a different PRF).

- The `client write MAC secret` refers to the MAC key used by the client to send (authenticated) data to the server;
- The `server write MAC secret` refers to the MAC key used by the server to send (authenticated) data to the client;
- The `client write key` refers to the key used by the client to send (encrypted) data to the server;
- The `server write key` refers to the key used by the server to send (encrypted) data to the client.

Also, if a block cipher is used in some mode of operation (e.g., CBC mode), then it may be necessary to hold some IVs. In this case, the SSL state element `initialization vectors` comprises two values:

- The `client write IV` refers to the IV used by the client to send data to the server;
- The `server write IV` refers to the IV used by the server to send data to the client.

In SSL, these values are all connection state elements that are shared between the client and the server. In contrast, TLS uses a different model, in which the client and the server both have connection state elements, but these elements are not shared. Instead, they both have a write and a read state, so that the client write state matches the server read state, and vice versa. More specifically, the client write state element `cipher state` (from Table 3.2) comprises a `client write key` and optionally a `client write IV`, whereas the client write state element `MAC secret` comprises a `client write MAC secret`. The same values appear in the server read state. So there is a total of four states for each entity and connection in TLS: A write and a read state for both the current and the pending states. This is logically the same as with SSL, but the information is stored differently. Take this as a subtle difference though.

A major difference between the SSL protocol and the TLS protocol lies in the way the keying material is generated. In Section 2.1, we saw that SSL uses an ad hoc and somehow handcrafted construction to generate the master secret and key block (from which the keying material is derived). In contrast, TLS 1.0 uses another construction that is commonly referred to as the TLS PRF.

Let us first introduce the TLS PRF, before we delve more deeply into the details of how the TLS protocol actually generates the keying material that is needed. There are some subtle differences between the PRF used in TLS versions 1.0 and 1.1

and the PRF used in TLS versions 1.2 and 1.3. We start with the former and postpone the discussion of the latter to Section 3.4.

3.1.1 TLS PRF

The TLS PRF is overviewed in Figure 3.1. The function takes as input a secret, a seed, and a label—sometimes also called an identifying label—and generates as output an arbitrarily long bit sequence that is indistinguishable from a random sequence. Similar to the SSL MAC construction (Section 2.2.1.3), the TLS PRF is based on the HMAC construction specified in RFC 2104 [5].[5]

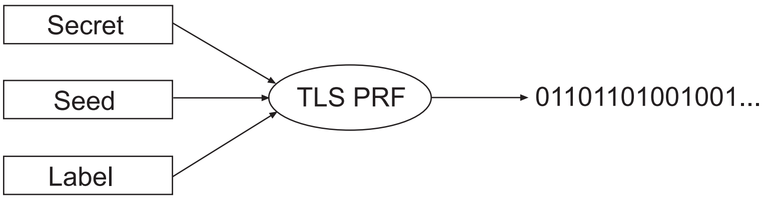

Figure 3.1 Overview of the TLS PRF.

Before we delve into the technical details of the TLS PRF, we start with a preliminary remark that is relevant to understand the design: The TLS PRF was designed in the late 1990s. At this time, people were using cryptographic hash functions that were already rumored to be less strong (i.e., less collision-resistant) than originally anticipated, namely MD5 and SHA-1. It was therefore believed that combining the two hash functions in a PRF would result in a PRF that is more secure (than if only one of the functions were used alone). This wisdom of crowds was proven wrong by Antoine Joux in 2004 [6]—five years after the TLS PRF went into the standard. This is why from today's perspective, the design of the TLS PRF is not optimal, at least not the way it is used in TLS versions 1.0 and 1.1. The combined use of MD5 and SHA-1 was later abandoned in the design of TLS versions 1.2 and 1.3 (and these versions use the supposedly stronger hash function SHA-256 instead of MD5 and SHA-1). Today, people employ highly specialized PRFs to derivate keying material, known as *key derivation functions* (KDFs). One such function, namely the *HMAC-based key derivation function* (HKDF) [7], is also based on the HMAC construction (as its name suggests) and has replaced the TLS PRF in TLS 1.3 (Section 3.5). So the TLS PRF only yields an interim solution.

5 Unlike the SSL MAC construction, the TLS PRF strictly follows the HMAC construction (and uses additions modulo 2 instead of string concatenations at the appropriate places).

More specifically, the TLS PRF is based on an auxiliary data expansion function, termed `P_hash(secret,seed)`. This function uses a single cryptographic hash function `hash` (which is, as mentioned above, MD5 or SHA-1 in TLS 1.0 and 1.1, and SHA-256 in TLS 1.2) to expand a secret and a seed into an arbitrarily long output value. Formally speaking, the data expansion function is defined as follows:

```
P_hash(secret,seed) = HMAC_hash(secret,A(1) + seed) +
                      HMAC_hash(secret,A(2) + seed) +
                      HMAC_hash(secret,A(3) + seed) +
                      ...
```

In this notation, + refers to the string concatenation operator, and `A` refers to a function that is recursively defined as

```
A(0) = seed
A(i) = HMAC_hash(secret,A(i-1))
```

for $i > 0$. The respective construction is illustrated and visualized in Figure 3.2.

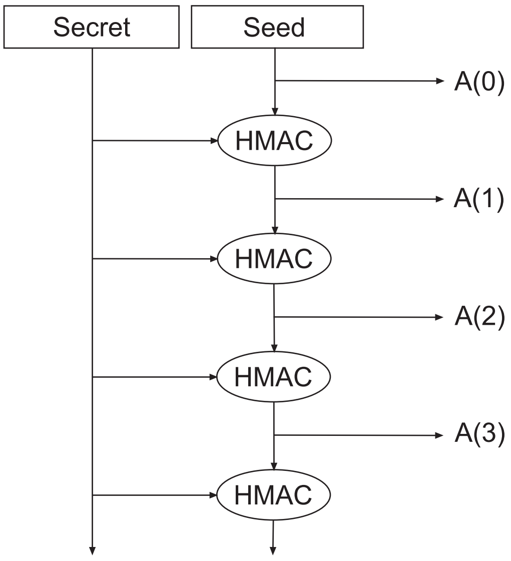

Figure 3.2 The A-function of the TLS PRF.

Depending on how many output bits are needed, the expansion function `P_hash(secret,seed)` can be applied arbitrarily many times. It is the major ingredient of the TLS PRF.

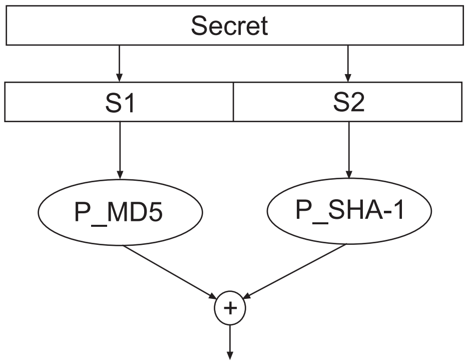

Figure 3.3 The internal structure of the TLS PRF (as used for TLS 1.0 and TLS 1.1).

Putting everything together, we are now ready to explain the TLS PRF (used in TLS 1.0 and 1.1). As illustrated in Figure 3.3, the secret is first split into two halves (i.e., S1 and S2). S1 is taken from the left half of the secret, and S2 is taken from the right half of the secret.[6] S1 and the concatenation of the label and the seed are then input to `P_MD5`, whereas S2 and the concatenation of the label and the seed are input to `P_SHA-1`. In the end, both output values are bitwise added modulo 2 (i.e., $\oplus$ or XOR). This can be formally expressed as follows:

```
PRF(secret,label,seed) =
   P_MD5(S1,label + seed) XOR P_SHA-1(S2,label + seed)
```

Note that MD5 produces an output value of 16 bytes, whereas SHA-1 produces an output value of 20 bytes. Therefore, the boundaries of the iterations of `P_MD5` and `P_SHA-1` are not aligned, and hence the expansion functions must be iterated differently many times. To generate an output of 80 bytes, for example, `P_MD5` must be iterated five times, whereas `P_SHA-1` needs to be iterated only four times (before the respective output values can be added using bitwise modulo 2 to perform the XOR operation).

As mentioned above (and further addressed in Section 3.4), the TLS PRF used in TLS 1.2 yet employs the same expansion function `P_hash(secret,seed)`, but it uses a single hash function (i.e., SHA-256). The resulting construction is simpler and more straightforward. However, either construction is efficient and performs well even if a long sequence of bits needs to be generated. Since TLS 1.3, the HKDF has replaced the TLS PRF.

6 If the secret happens to be an odd number of bytes long, then the last byte of S1 will be repeated and be the same as the first byte of S2.

3.1.2 Generation of Keying Material

The primary use of the TLS PRF is to generate the keying material needed for a TLS connection. First, the variable-length premaster secret that is the output of the key exchange algorithm (and part of the TLS session state) is used to generate a 48-byte master secret (which then represents TLS connection state according to Table 3.1). The construction is as follows:[7]

```
master_secret =
   PRF(pre_master_secret,"master secret",
       client_random + server_random)
```

From this construction, only the first 48 output bytes are used. Referring to Figure 3.1, `pre_master_secret` represents the secret, the concatenation of the two values `client_random` and `server_random` represents the seed, and the string `"master secret"` represents the label. Note that `client_random` is the same value as `client random` from Table 3.1. So the underscore character is used inconsistently in the TLS protocol specification, and we do something similar by using both terms synonymously and interchangeably. Again referring to Table 3.1, the master secret and the server and client random values are part of the TLS connection state.

Once a 48-byte master secret is generated (as described above), it is subsequently used as a source of entropy for the derivation of the various keys that are needed for the TLS connection. The keys are taken from a key block of appropriate size that is constructed as follows:

```
key_block =
   PRF(master_secret,"key expansion",
       server_random + client_random)
```

This time, `master_secret` is the secret, the concatenation of the two random values `server_random` and `client_random` is the seed (note that the two random values are written in the opposite order as compared to the construction of the master secret), and the string `"key expansion"` is the label. The key block can then be partitioned into distinct values by splitting it into blocks of appropriate sizes (for the keying material that is needed). Any additional material in the key block is silently discarded. For example, a cipher suite that uses 3DES in CBC mode and SHA-1 requires $2 \cdot 192 = 384$ bits for the two 3DES keys, $2 \cdot 64 = 128$ bits for the two IVs, and $2 \cdot 160 = 320$ bytes for the two MAC keys. This sums up to 832 bits

7 According to Section C.2.43, there is a new way of generating a master secret that protects against an attack known as the *triple handshake attack* (Section A.5). To make the distinction clear, the respective master secret is then called *extended master secret*.

(or 104 bytes, respectively). Different cipher suites may have different requirements regarding the length of the key block that needs to be generated. The resulting values are part of the respective connection state (i.e., the cipher state for the write keys and the MAC secret for the write MAC keys).

In former times (when the U.S. export controls were still in place), people were frequently using exportable ciphers with reduced key lengths. In this case, the two write keys (i.e., `client_write_key` and `server_write_key`) were used to generate two final write keys (i.e., `final_client_write_key` and `final_server_write_key`). The respective constructions were as follows (with appropriately chosen lengths):

```
final_client_write_key =
   PRF(client_write_key,"client write key",
       server_random + client_random)
final_server_write_key =
   PRF(server_write_key,"server write key",
       client_random + server_random)
```

Also, if the cipher in use happened to be an exportable block cipher, then the IVs were derived solely from the random values sent in the hello messages of the TLS handshake protocol, and hence they could be generated without taking into account any secret. Instead of using the `client_write_IV` and `server_write_IV` values mentioned above, an IV block could be generated as follows:

```
iv_block =
   PRF("","IV block",client_random + server_random)
```

In this construction, the secret is empty and the label refers to `"IV block"`. The resulting IV block was then partitioned into two appropriately sized IVs (to represent the `client_write_IV` and the `server_write_IV`).

Due to the existence of key exchange downgrade attacks like FREAK and Logjam (Section A.7), exportable ciphers should no longer be used, and hence this topic (i.e., how to generate the keying material for an exportable block cipher) is no longer relevant. We simply refer to [1] for an example of how to actually generate the keying material required for RC2 with a 40-bit key.

In many settings, MITM attacks pose a serious threat, and hence it is often necessary to mitigate such attacks. One possibility to achieve this is to cryptographically bind authentication information, such as a one-time password (OTP), to a particular TLS connection [8]. If an MITM is then able to capture the OTP and tries to submit it on the victim's behalf (i.e., to spoof the victim), then it is obvious for the recipient that the OTP is submitted on another TLS connection than it was originally bound to, and hence that something fishy is going on. This simple idea and possibility to mitigate MITM attacks was also used in a technology called *origin-bound certificates* (OBC) [9]. The technology was further developed and refined within the IETF token binding (TOKBIND) WG that was active from 2015 to 2021.[8] The result is a token binding mechanism that uses public key cryptography to cryptographically bind authentication tokens, such as HTTP cookies, to particular TLS connections (so that they cannot be submitted on other TLS connections). The token binding mechanism uses a distinct TLS extension called `token_binding` and is addressed in Section C.2.20. Here, we only note that the mechanism requires a unique value that represents a TLS connection, and that this value can be a cryptographic key (used by the connection) or something similar. In the realm of TLS, a value called *exported keying material* (EKM) serves this purpose. According to [10], an EKM value is constructed like a key block (i.e., using the same TLS PRF but with distinct labels). You may refer to the TLS protocol specification for the details. EKMs and token binding are used as an MITM protection mechanism for token-based authentication technologies, like OpenID Connect (OIDC), OAuth, and Web Authentication (WebAuthn). More generally, EKMs can be used whenever an application-layer protocol (maybe other than HTTP) needs a hook into TLS.

In addition to the TLS PRF and the generation of the keying material (including EKM), there are many other differences between the SSL protocol and the various versions of the TLS protocol. These differences are outlined, discussed, and put into perspective next. For each difference, we provide some background information and discuss the rationale that has led to the respective design. We begin with TLS version 1.0. Because TLS 1.3 is a new protocol that incorporates some fundamental changes, the respective Section 3.5 is going to be more comprehensive than the others.

3.2 TLS 1.0

It has been mentioned several times so far that TLS 1.0 is very close to and backward-compatible with SSL 3.0, and that it can therefore also be viewed as SSL version 3.1. This viewpoint is reflected in the version field value that is included in every TLS record (i.e., 0x0301). In fact, this value comprises the two bytes 0x03 and 0x01, where the former byte is standing for the major version 3 and the latter byte is standing for the minor version 1. This suggests that TLS 1.0 is conceptually similar to something like SSL 3.1.

In addition to the version number, there are a few other differences between SSL 3.0 and TLS 1.0. For example, we have just seen that both protocols employ different PRFs to determine the master secret and all the keying material that

8 https://datatracker.ietf.org/wg/tokbind/about/.

is needed. Also, the TLS protocol distinguishes between security parameters and state elements for TLS connections, whereas the SSL protocol only considers state elements. Last but not least, there are a few differences that are more subtle and require more explanation. These differences refer to the cipher suites, certificate management, alert messages, and some other points. They are addressed next.

3.2.1 Cipher Suites

As with SSL, a TLS cipher spec refers to a pair of algorithms that are used to authenticate and encrypt data, whereas a cipher suite additionally comprises a key exchange algorithm. TLS 1.0 supports the same cipher suites as SSL 3.0 (summarized in Table 2.4). However, the following three cipher suites, which employ FORTEZZA, are no longer supported and have no counterpart in TLS 1.0.[9]

- SSL_FORTEZZA_KEA_WITH_NULL_SHA
- SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA
- SSL_FORTEZZA_KEA_WITH_RC4_128_SHA

This means that there are $31 - 3 = 28$ cipher suites supported by TLS 1.0. Also, for obvious reasons, the prefixes of the cipher suites have changed from SSL_ to TLS_, so the cipher suite SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA has effectively become TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA without any substantial change with regard to its meaning. With the exception of FORTEZZA no longer being supported, the key exchange algorithms have not changed and remain the same. But there are still a few notable changes with regard to message authentication and data encryption.

3.2.1.1 Message Authentication

The MAC construction employed by the SSL protocol (Section 2.2.1.3) is conceptually similar to the standard HMAC construction, but it is not exactly the same. So for TLS 1.0, it was decided to use the HMAC construction for message authentication in a more consistent way. The input parameters are the MAC key K (that can either be the `client_write_MAC_secret` or the `server_write_MAC_secret` depending on what party is sending data) on the one hand, and the concatenation of the

9 The two-byte reference codes 0x001C (for SSL_FORTEZZA_KEA_WITH_NULL_SHA) and 0x001D (for SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA) are not reused for TLS, probably to avoid conflicts with SSL. This is not true for the reference code 0x001E (for SSL_FORTEZZA_KEA_WITH_RC4_128_SHA) that is reused to refer to the cipher suite TLS_KRB5_WITH_DES_CBC_SHA in TLS.

sequence number seq_number and the four components of the `TLSCompressed` structure, namely $type$, $version$, $length$, and $fragment$, on the other hand. The sequence number is 8 bytes long, whereas the type, version, and length fields are together 5 bytes long. This means that a total of 13 bytes is prepended to the fragment field, before the HMAC value is generated.[10] The construction can be formally expressed as follows:

$$\begin{aligned} &HMAC_K(TLSCompressed) = \\ &\quad h(K \parallel opad \parallel h(K \parallel ipad \parallel seq\_number \parallel \\ &\quad \underbrace{type \parallel version \parallel length \parallel fragment}_{TLSCompressed})) \end{aligned}$$

In this notation, h refers to the cryptographic hash function in use (as specified by the MAC algorithm parameter of the TLS connection), and $opad$ and $ipad$ refer to constants (as specified in the HMAC standard and introduced in Section 2.2.1.3). If one associates the concatenation of seq_number and the four components of the `TLSCompressed` structure with the message that is authenticated, then it is obvious that the resulting construction is in line with the standard HMAC construction. The only specialty is the incorporation of the sequence number (that is implicit and not part of the `TLSCompressed` structure). The purpose of this value is to provide some additional protection against replay attacks.

3.2.1.2 Data Encryption

SSL 3.0 was specified prior to the enactment of the current U.S. export controls. Consequently, the preferred ciphers were DES (for block ciphers) and RC4 (for stream ciphers). When TLS 1.0 was specified in 1999, the situation regarding U.S. export controls was about to change, and hence stronger ciphers were preferred. The AES was not yet standardized,[11] and hence 3DES was considered to be the strongest alternative. So TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA became the only cipher suite that was mandatory to implement in TLS 1.0. This cipher suite complements 3DES with an ephemeral Diffie-Hellman key exchange (DHE), DSA-based authentication, and SHA-1 hashing. It is still a reasonably secure cipher suite—even more than 20 years after its promotion in TLS 1.0. The only major problem is the use of CBC mode for encryption that is susceptible to padding oracle attacks (Section A.2). Furthermore, SSL 3.0 and TLS 1.0 both use IV chaining in CBC encryption,

10 The fact that 13 bytes are prepended is exploited in a padding oracle attack against the TLS protocol employing a block cipher in CBC mode that is known as *Lucky 13* (Section A.2).

11 In fact, the NIST competition to define the successor of DES and 3DES was still going on. It was only released in the following year.

meaning the IV for the encryption of a new record is the last ciphertext block of the previous record. Again, this technique has turned out to be susceptible to an attack known as BEAST (Section A.3).

If a cipher suite comprises a block cipher operated in CBC mode (such as TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA), then there is a subtle difference between SSL 3.0 and TLS 1.0: While SSL 3.0 assumes the padding (that forces the length of the plaintext that comprises the fragment field of the `TLSCompressed` structure to be a multiple of the cipher's block length) to be as short as possible, TLS 1.0 does not make this assumption. In fact, it has become possible in TLS 1.0 to add more padding bytes (up to 255) before the encryption takes place. This allows the sender of a message to hide the actual length of a message, and hence to provide some protection against traffic analysis.

Table 3.3
The Camellia-Based Cipher Suites for TLS*

Cipher Suite	Value
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	{ `0x00,0x41` }
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA	{ `0x00,0x42` }
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA	{ `0x00,0x43` }
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	{ `0x00,0x44` }
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	{ `0x00,0x45` }
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA	{ `0x00,0x46` }
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	{ `0x00,0x84` }
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA	{ `0x00,0x85` }
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA	{ `0x00,0x86` }
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	{ `0x00,0x87` }
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	{ `0x00,0x88` }
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA	{ `0x00,0x89` }
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256	{ `0x00,0xBA` }
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256	{ `0x00,0xBB` }
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256	{ `0x00,0xBC` }
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256	{ `0x00,0xBD` }
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	{ `0x00,0xBE` }
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256	{ `0x00,0xBF` }
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256	{ `0x00,0xC0` }
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256	{ `0x00,0xC1` }
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256	{ `0x00,0xC2` }
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256	{ `0x00,0xC3` }
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256	{ `0x00,0xC4` }
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256	{ `0x00,0xC5` }

* According to [11].

In addition to the move from DES to 3DES, a complementary RFC 4132 was first released in 2005[12] to propose cipher suites that comprise the Camellia block cipher.[13] In 2010, however, this RFC was updated in Standards Track RFC 5932 [11], and the list of cipher suites was expanded to also make use of SHA-256 (in addition to SHA-1). The respective cipher suites together with their hexadecimal reference code values are summarized in Table 3.3 (the cipher suites are also itemized in Appendix B). The first half of cipher suites employs SHA-1 as hash function, whereas the second half of cipher suites employs SHA-256 (all other components of the cipher suites remain the same). RFC 5932 is still valid and can be applied to all versions of the TLS protocol (it is even complemented by an informational RFC 6367 [12] that introduces new cipher suites that also comprise Camellia). While the Camellia-based cipher suites are used in Japan and partly in Europe, they are hardly ever used in the United States. In fact, from what we know today, there is no reason to replace AES-based cipher suites with Camellia-based ones.

3.2.2 Certificate Management

With regard to certificate management, there are two fundamental and far-reaching differences between SSL 3.0 and TLS 1.0:

- SSL 3.0 always requires complete certificate chains, meaning that a certificate chain must include all certificates that are required to verify the chain. In particular, this includes the certificate for the root CA. This is in contrast to TLS 1.0, where a certificate chain may only include the certificates that are needed to verify the chain up to a trusted CA (which may also be an intermediate CA). Depending on the intermediate CAs that are available, this may simplify the verification and validation of the certificates considerably.
- As outlined in Section 2.2.2.6 and Table 2.5, SSL 3.0 supports many certificate types. This is no longer true for TLS 1.0. Referring to Table 3.4, TLS 1.0 only supports the first four certificate types itemized in Table 2.5, namely `rsa_sign` (1), `dss_sign` (2), `rsa_fixed_dh` (3), and `dss_fixed_dh` (4). The numbers in brackets refer to the certificate type values. So TLS 1.0 no longer supports certificates for ephemeral Diffie-Hellman

12 This RFC was released for TLS 1.0 and we therefore mention the issue here. Note, however, that the currently valid RFC 5932 was released after the publication of TLS 1.2.

13 Camellia is a 128-bit block cipher that was jointly developed by Mitsubishi and NTT in Japan. It has similar security characteristics to AES, but unlike AES, Camellia is structurally similar to DES, and hence it also represents a Feistel network. Note that another block cipher known as ARIA was developed in Korea and added to the list of available block ciphers several years later (Section 3.9).

Table 3.4
TLS 1.0 Certificate Type Values

Value	Name	Description
1	rsa_sign	Certificate containing an RSA key
2	dss_sign	Certificate containing a DSA key
3	rsa_fixed_dh	Certificate containing a static DH key
4	dss_fixed_dh	Certificate containing a static DH key

key exchanges signed with RSA (5) or DSA (6), as well as certificates used for the FORTEZZA KEA (2). This is because a certificate that can be used to generate (RSA or DSA) signatures can also be used to sign ephemeral Diffie-Hellman keys, and because FORTEZZA-type cipher suites have been removed from TLS 1.0 entirely.

In Section 3.3.2, we will see that the certificate types that are missing in TLS 1.0 were reintroduced in TLS 1.1 as reserved values. So the question of what certificate types must be supported heavily depends on the TLS protocol version.

3.2.3 Alert Messages

TLS 1.0 uses a set of alert messages that is slightly different from SSL 3.0. The 23 alert protocol message types of TLS 1.0 are summarized in Tables 3.5 and 3.6. In Table 3.5, only the message types that are new (as compared to SSL 3.0) come with a description. All other message types have already been introduced and described in Chapter 2. In addition to the new message types, there is also one message type that has become obsolete and is now marked as reserved (i.e., `no_certificate` or `no_certificate_RESERVED` with alert code 41).[14] Due to its reserved status, it is not included in Tables 3.5 and 3.6.

3.2.4 Other Differences

The CERTIFICATEVERIFY and FINISHED messages of SSL 3.0 (Sections 2.2.2.10 and 2.2.2.11) are involved and rather difficult to construct. Consequently, they are simplified and brought more in line with the TLS PRF construction otherwise used in TLS 1.0.

- The body of the CERTIFICATEVERIFY message comprises a digital signature for the concatenated handshake messages that have been exchanged so far.

14 This is the generally used notation. When an alert message type becomes obsolete, the string _RESERVED is appended to its name.

Table 3.5
TLS 1.0 Alert Messages (Part 1)

Alert	Code	Brief Description (If New)
`close_notify`	0	
`unexpected_message`	10	
`bad_record_mac`	20	
`decryption_failed`	21	The sender notifies the recipient that a ciphertext (received in the fragment of a `TLSCiphertext` record) decrypted in an invalid way. This alert is always fatal.
`record_overflow`	22	The sender notifies the recipient that a record was too long (i.e., either a `TLSCiphertext` record was longer than $2^{14}+2{,}048$ bytes or a `TLSCompressed` record was longer than $2^{14}+1{,}024$ bytes). This alert is always fatal and should never be observed in communication between proper implementations.
`decompression_failure`	30	
`handshake_failure`	40	
`bad_certificate`	42	
`unsupported_certificate`	43	
`certificate_revoked`	44	
`certificate_expired`	45	
`certificate_unknown`	46	
`illegal_parameter`	47	
`unknown_ca`	48	The sender notifies the recipient that a valid certificate chain was received, but at least one certificate was not accepted because the CA cerificate could not be located or could not be matched with a trusted CA. This alert is always fatal.
`access_denied`	49	The sender notifies the recipient that a valid certificate was received, but when access control was applied, the sender decided not to proceed with negotiation. This alert is always fatal.

The signing key is the one that corresponds to the public key found in the client certificate.

- The body of the FINISHED message comprises 12 bytes (96 bits) of data that is constructed in a way similar to the TLS PRF. Referring to Figure 3.1, the *secret* is the master secret of the connection (i.e., `master_secret`), the *seed* is the modulo 2 sum of the MD5 hash of all concatenated handshake messages[15] and the SHA-1 hash of the same argument (i.e., `MD5(handshake_messages)+SHA-1(handshake_messages)`), and the *label* is a constant string that depends on who is actually sending the message (i.e., `finished_label`). If the client is sending the message, then the label refers to the string `"client finished."` Otherwise, if the sever is sending the message, then the label refers to the string `"server finished."` The construction of the data can thus be expressed as follows:

15 Note that any HELLOREQUEST message is excluded from the concatenation. Also note that only the handshake messages (without any record header) are included in the construction.

Table 3.6
TLS 1.0 Alert Messages (Part 2)

Alert	Code	Brief Description (If New)
`decode_error`	50	The sender notifies the recipient that a message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This alert is always fatal.
`decrypt_error`	51	The sender notifies the recipient that a handshake cryptographic operation failed, including being unable to verify a signature, decrypt a key exchange, or validate a finished message.
`export_restriction`	60	The sender notifies the recipient that a negotiation not in compliance with export restrictions was detected. This alert is always fatal.
`protocol_version`	70	The sender notifies the recipient that the protocol version the client has attempted to negotiate is recognized but not supported (for example, an older protocol version might be avoided for security reasons). This alert is always fatal.
`insufficient_security`	71	Returned instead of `handshake_failure` when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client. This alert is always fatal.
`internal_error`	80	The sender notifies the recipient that an internal error unrelated to the peer or the correctness of the protocol makes it impossible to continue. This alert is always fatal.
`user_canceled`	90	The sender notifies the recipient that this handshake is being canceled for some reason unrelated to a protocol failure. If the user cancels an operation after the handshake is complete, just closing the connection by sending a `close_notify` is more appropriate. This alert should be followed by a `close_notify`. This alert is generally a warning.
`no_renegotiation`	100	The sender notifies the recipient that a renegotiation is not appropriate. This alert is generally a warning.

```
PRF(master_secret,finished_label,
    MD5(handshake_messages) +
        SHA-1(handshake_messages))
```

As usual in TLS 1.0 (and TLS 1.1), this construction combines the two hash functions MD5 and SHA-1. The resulting 12 bytes represent a MAC that is referred to as `verify_data` in the TLS protocol specification. It goes without saying that there is client-side `verify_data` and server-side `verify_data`, and that the two values are different (because the labels and the concatenated handshake messages are different in either case).

This finishes our exposition of the differences between SSL 3.0 and TLS 1.0, and we are now ready to go one step further in the evolution of the TLS protocol (i.e., TLS 1.1).

3.3 TLS 1.1

The official TLS 1.0 protocol specification was published in 1999. Seven years later, in 2006, an updated version 1.1 of the TLS protocol was released [2]. Following the notation of SSL 3.0 and TLS 1.0, the official version number of TLS 1.1 became 3,2 (0x0302). So TLS 1.1 can also be seen as SSL 3.2.

The major differences between TLS 1.0 and TLS 1.1 are motivated by some cryptographic subtleties and vulnerabilities that had been exploited by a few attacks against block ciphers operated in CBC mode. The attacks and the respective countermeasure are explained in Appendix A, whereas the resulting protocol changes regarding cipher suites, certificate management, alert messages, and other differences are outlined next. Before that, we want to emphasize that TLS 1.1 also introduced a new way of specifying parameters and parameter values. Instead of incorporating them into the protocol specification of the respective RFC document, the IANA decided to add flexibility and therefore instantiated a number of registries for TLS parameter values, such as certificate types, cipher suites, content types, alert message types, handshake types, and many more.[16] If any of these parameters need to be added or changed, then it is no longer necessary to modify the RFC document. Instead, the parameters can be added or modified in the respective registry. This simplifies the management and update of the protocol specification considerably.

According to RFC 2434 [13] (that also yields a BCP document), there are three policies that can be used to assign values to specific parameters:

- Values assigned via *standards action* are reserved for parameters that appear in RFCs submitted to the internet standards track (and therefore approved by the IESG).
- Values assigned via *specification required* must at least be documented in an RFC or another permanent and readily available reference, in sufficient detail so that interoperability between independent implementations is possible.
- Values assigned via *private use* need not fulfill any requirement. In fact, there is no need for IANA to review such assignments and they are not generally useful for interoperability.

So a parameter value can be assigned via standards action, specification required, or private use. For some parameters, the IANA has also partitioned the range of possible values into distinct blocks. So there is a block for parameter values assigned by standards action, another block for parameter values assigned

16 https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml.

by specification required, and yet another block for parameter values assigned by private use.

3.3.1 Cipher Suites

As mentioned above, some cryptographic subtleties and respective attacks led to a number of changes in TLS 1.1. Most importantly, the Vaudenay attack (Section A.2) led to the suggestion to compute a MAC in all situations, even if the padding is incorrect and the protocol could prematurely abort. Unfortunately, this only partly mitigates the Vaudenay attack, as was later demonstrated by the Lucky 13 attack. To mitigate the attack entirely, one has to either move away from block ciphers operated in CBC mode or use authenticated encryption. Also, the susceptibility of IV chaining led to the addition of an explicit IV (that is sent along with the respective `TLSCiphertext` fragment). This successfully mitigates the BEAST attack (Section A.3).

Table 3.7
TLS 1.1 Standard Cipher Suites

Cipher Suite	Value
TLS_NULL_WITH_NULL_NULL	{ `0x00,0x00` }
TLS_RSA_WITH_NULL_MD5	{ `0x00,0x01` }
TLS_RSA_WITH_NULL_SHA	{ `0x00,0x02` }
TLS_RSA_WITH_RC4_128_MD5	{ `0x00,0x04` }
TLS_RSA_WITH_RC4_128_SHA	{ `0x00,0x05` }
TLS_RSA_WITH_IDEA_CBC_SHA	{ `0x00,0x07` }
TLS_RSA_WITH_DES_CBC_SHA	{ `0x00,0x09` }
TLS_RSA_WITH_3DES_EDE_CBC_SHA	{ `0x00,0x0A` }
TLS_DH_DSS_WITH_DES_CBC_SHA	{ `0x00,0x0C` }
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	{ `0x00,0x0D` }
TLS_DH_RSA_WITH_DES_CBC_SHA	{ `0x00,0x0F` }
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	{ `0x00,0x10` }
TLS_DHE_DSS_WITH_DES_CBC_SHA	{ `0x00,0x12` }
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	{ `0x00,0x13` }
TLS_DHE_RSA_WITH_DES_CBC_SHA	{ `0x00,0x15` }
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	{ `0x00,0x16` }
TLS_DH_anon_WITH_RC4_128_MD5	{ `0x00,0x18` }
TLS_DH_anon_WITH_DES_CBC_SHA	{ `0x00,0x1A` }
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	{ `0x00,0x1B` }

Except for these changes in the way a block cipher in CBC mode operates, there are only a few differences related to cipher suites. First of all, TLS_RSA_WITH_3DES_EDE_CBC_SHA is mandatory to implement (instead of

Table 3.8
TLS 1.1 Kerberos-Based Cipher Suites from [14]

Cipher Suite	Value
TLS_KRB5_WITH_DES_CBC_SHA	{ 0x00,0x1E }
TLS_KRB5_WITH_3DES_EDE_CBC_SHA	{ 0x00,0x1F }
TLS_KRB5_WITH_RC4_128_SHA	{ 0x00,0x20 }
TLS_KRB5_WITH_IDEA_CBC_SHA	{ 0x00,0x21 }
TLS_KRB5_WITH_DES_CBC_MD5	{ 0x00,0x22 }
TLS_KRB5_WITH_3DES_EDE_CBC_MD5	{ 0x00,0x23 }
TLS_KRB5_WITH_RC4_128_MD5	{ 0x00,0x24 }
TLS_KRB5_WITH_IDEA_CBC_MD5	{ 0x00,0x25 }

Table 3.9
TLS 1.1 AES-Based Cipher Suites from [15]

Cipher Suite	Value
TLS_RSA_WITH_AES_128_CBC_SHA	{ 0x00,0x2F }
TLS_DH_DSS_WITH_AES_128_CBC_SHA	{ 0x00,0x30 }
TLS_DH_RSA_WITH_AES_128_CBC_SHA	{ 0x00,0x31 }
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	{ 0x00,0x32 }
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	{ 0x00,0x33 }
TLS_DH_anon_WITH_AES_128_CBC_SHA	{ 0x00,0x34 }
TLS_RSA_WITH_AES_256_CBC_SHA	{ 0x00,0x35 }
TLS_DH_DSS_WITH_AES_256_CBC_SHA	{ 0x00,0x36 }
TLS_DH_RSA_WITH_AES_256_CBC_SHA	{ 0x00,0x37 }
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	{ 0x00,0x38 }
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	{ 0x00,0x39 }
TLS_DH_anon_WITH_AES_256_CBC_SHA	{ 0x00,0x3A }

TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA as with TLS 1.0). Also, all cipher suites that comprise an export-grade key exchange algorithm are discouraged, meaning that these ciphers may still be offered for backward compatibility, but they should not be negotiated actively. (In fact, the FREAK and Logjam attacks outlined in Section A.7 later showed that exportable cipher suites shouldn't be supported at all.) This applies to all cipher suites written in italics in Table 2.4 (except SSL_NULL_WITH_NULL_NULL that is still needed) and the export-grade Kerberos-based cipher suites from RFC 2712 [14]. All other Kerberos- and AES-based cipher suites proposed in [14] and RFC 3268 [15] have been incorporated into TLS 1.1 specification. The resulting cipher suites are summarized in Tables 3.7 to 3.9 (together with their respective hexadecimal code values). The Camellia-based cipher suites itemized from Table 3.3 still apply to TLS 1.1.

Table 3.10
TLS 1.1 Certificate Type Values

Value	Name
1	rsa_sign
2	dss_sign
3	rsa_fixed_dh
4	dss_fixed_dh
5	rsa_ephemeral_dh_RESERVED
6	dss_ephemeral_dh_RESERVED
20	fortezza_dms_RESERVED

3.3.2 Certificate Management

As mentioned above and summarized in Table 3.10, the certificate type values 5, 6, and 20 have been reintroduced in TLS 1.1 as reserved values (meaning that they should no longer be used). Except for that (and the fact that `fortezza_kea` is renamed to `fortezza_dms`), the certificate management of TLS 1.1 remains essentially the same as in SSL 3.0 and TLS 1.0.

3.3.3 Alert Messages

In addition to the new alert messages that have been introduced in TLS 1.0, there is also an alert message [i.e., the `no_certificate` or `no_certificate_RESERVED` message (41)] that has become obsolete in TLS 1.0. In TLS 1.1, two additional alert messages have become obsolete: `export_restriction` (60) and `decryption_failed` (21). The former has become obsolete, because export-grade encryption is no longer supported in TLS 1.1. The latter has become obsolete, because the distinction between `bad_record_mac` and `decryption_failed` alert messages enables padding oracle attacks.

3.3.4 Other Differences

There are only a few other differences between TLS 1.0 and TLS 1.1. For example, a premature closure (i.e., a closure without a mutual exchange of `close_notify` alert messages) no longer causes a TLS 1.1 session to be nonresumable. Put in other words, even if a TLS connection is closed without having the communicating peers properly exchange `close_notify` alert messages, it may still be resumable under certain conditions. This is to simplify the management of TLS connections.

3.4 TLS 1.2

TLS 1.1 was officially released in 2006. In the two subsequent years, the standardization activities continued and many people working in the field made proposals on how TLS could possibly evolve. In 2008, the next version of the TLS protocol (i.e., TLS 1.2 with version number 3,3 or 0x0303) was ready and could be released officially in RFC 5246 [3].

The biggest change in TLS 1.2 is the consolidation of the extension mechanism that has been around since TLS 1.0. It allows additional functionality to be incorporated into TLS without having to change the underlying protocol. We discuss the TLS extension mechanism in detail before we delve into the other—mostly minor—differences between TLS 1.1 and TLS 1.2. Even before that, we remind you that TLS employs a different PRF than SSL, that TLS 1.0 and 1.1 still use MD5 and SHA-1, and that this combined use of MD5 and SHA-1 is abandoned in TLS 1.2. In fact, a single—and, one hopes, more secure—cryptographic hash function is being used. The respective PRF construction is streamlined and can be expressed as follows:

```
PRF(secret,label,seed) = P_hash(secret,label+seed)
```

The cryptographic hash function `hash` is part of the cipher suite. For the typical case of using SHA-256, for example, `P_hash` refers to `P_SHA256`. Independent from the TLS PRF in use (be it the PRF for TLS 1.0 and 1.1 or the PRF for TLS 1.2), the keying material is generated in the same way (Section 3.1.2).

3.4.1 TLS Extensions

As mentioned above, TLS 1.2 comes along with an extension mechanism and a number of extensions (that can sometimes also be used in earlier versions of the SSL/TLS protocols). More specifically, the TLS 1.2 specification provides the conceptual framework and some design principles for extensions, whereas the actual definitions are provided in a complementary document (i.e., RFC 6066 [16]).[17]

TLS extensions are usually sent along with the client and server hello messages in a way that is backward-compatible, meaning that communication is possible between clients that support the extensions and servers that do not support them, and vice versa. A client can invoke a TLS extension by extending its CLIENTHELLO message. An extended CLIENTHELLO message is just a normal CLIENTHELLO message with an additional block of data that may comprise a list of extensions.

17 Note that RFC 6066 obsoletes RFC 4366 and that this RFC obsoletes RFC 3546. This, in turn, was the original RFC document that introduced the notion of a TLS extension in the first place. It dates back to 2003, when TLS 1.0 was the current protocol version.

Remember that additional information can always be appended to a CLIENTHELLO message, so an extended CLIENTHELLO message that conforms to the specification does not break a TLS server. The server can still accept such a message, even if it does not properly understand all extensions it contains.

The presence of extensions can be detected by determining whether there are bytes following the compression methods at the end of the CLIENTHELLO message. This method of detecting optional data is not in line with the usual method of having a length field, but it is used for compatibility with TLS before extensions were defined. Anyway, if the server understands the extensions, it sends back an extended SERVERHELLO message in place of a normal SERVERHELLO message. Again, the extended SERVERHELLO may comprise a list of extensions. Note that the extended SERVERHELLO message is only sent in response to an extended CLIENTHELLO message. This prevents the possibility that the extended SERVERHELLO message may break a TLS client. Also note that there is no upper bound for the length of the list of extensions. So it may happen that a client floods a server by sending a very long extensions list. If this poses a problem, then it is possible and very likely that future server implementations will limit the maximum length of extended CLIENTHELLO messages. So far, this has not been the case though.

Each extension is structurally identical and basically consists of two fields: a two-byte type field and a variable-length data field of which the first two bytes specify the length. If, for example, a client wants to signal support for the secure renegotiation extension to the server, then it appends the five bytes 0xFF, 0x01, 0x00, 0x01, and 0x00 to the end of the CLIENTHELLO message. In this encoding, the first two bytes (i.e., 0xFF and 0x01) refer to the type of the extension (that corresponds to the binary value 11111111 00000001 and the decimal value $15 \cdot 16^3 + 15 \cdot 16^2 + 0 \cdot 16^1 + 1 \cdot 16^0 = 61,440 + 3,840 + 0 + 1 = 65,281$), the next two bytes (i.e., 0x00 and 0x01) refer to the length of the data field (that corresponds to 1), and the last byte refers to the actual data (that corresponds to 0 in this case, meaning that there is no data). Other extensions are encoded similarly, and in many cases the data field is not empty.

As mentioned at the end of the previous section, the IANA maintains registries of TLS parameter values. One of these registries contains the TLS extension types that can be used. Like all other registries, this registry is a moving target and subject to change. The TLS extension types and values that were valid when this book was written are outlined and briefly explained in Appendix C. Some of these extensions are important in the field, such as the `server_name` extension used for the server name indication (SNI) or the `elliptic_curves` (later renamed to `supported_groups`) and `ec_point_formats` extensions that are used for elliptic curve cryptography (ECC). This is not only true for TLS 1.2, but also for

TLS 1.3 (Section 3.5), where most functionalities are provided by extensions, some of them being entirely new.

In summary, TLS 1.2 introduces and comes along with many extensions. For most of these extensions, it is sufficient to adapt and extend the CLIENTHELLO and SERVERHELLO messages. However, for some (newer) extensions, it is also necessary to use new TLS handshake messages. This is particularly true for the NEWSESSIONTICKET message (type 4), the CERTIFICATEURL message (type 21), the CERTIFICATESTATUS message (type 22), and the SUPPLEMENTALDATA message (type 23). These message types have no counterparts in the SSL protocol or any prior version of the TLS protocol.

Table 3.11
TLS 1.2 Cipher Suites That Require a Server-Side RSA Certificate for Key Exchange

Cipher Suite	Value
TLS_RSA_WITH_NULL_MD5	{ 0x00,0x01 }
TLS_RSA_WITH_NULL_SHA	{ 0x00,0x02 }
TLS_RSA_WITH_NULL_SHA256	{ 0x00,0x3B }
TLS_RSA_WITH_RC4_128_MD5	{ 0x00,0x04 }
TLS_RSA_WITH_RC4_128_SHA	{ 0x00,0x05 }
TLS_RSA_WITH_3DES_EDE_CBC_SHA	{ 0x00,0x0A }
TLS_RSA_WITH_AES_128_CBC_SHA	{ 0x00,0x2F }
TLS_RSA_WITH_AES_256_CBC_SHA	{ 0x00,0x35 }
TLS_RSA_WITH_AES_128_CBC_SHA256	{ 0x00,0x3C }
TLS_RSA_WITH_AES_256_CBC_SHA256	{ 0x00,0x3D }

3.4.2 Cipher Suites

The most obvious cipher suite change in TLS 1.2 is that TLS_RSA_WITH_AES_128_CBC_SHA is now mandatory to implement, and hence that TLS 1.2 strategically moves away from 3DES. Also, all cipher suites that employ DES or IDEA are no longer recommended[18] and are removed in TLS 1.2. Hence, the remaining cipher suites employ either AES or 3DES in the case of block ciphers and RC4 in the case of stream ciphers. They are itemized in Tables 3.11 to 3.13. Note that they are classified according to the key exchange mechanism they use. Also note that some cipher suites have been extended to support SHA-256 (in addition to SHA-1). Similar to all previous versions of SSL/TLS, TLS 1.2 supports the default cipher suite TLS_NULL_WITH_NULL_NULL.

18 Informational RFC 5469 specifies the use of DES and IDEA in a TLS setting for completeness, and discusses why their use is no longer recommended.

Table 3.12
TLS 1.2 Cipher Suites That Employ a Nonanonymous Diffie-Hellman Key Exchange

Cipher Suite	Value
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	{ 0x00,0x0D }
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	{ 0x00,0x10 }
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	{ 0x00,0x13 }
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	{ 0x00,0x16 }
TLS_DH_DSS_WITH_AES_128_CBC_SHA	{ 0x00,0x30 }
TLS_DH_RSA_WITH_AES_128_CBC_SHA	{ 0x00,0x31 }
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	{ 0x00,0x32 }
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	{ 0x00,0x33 }
TLS_DH_DSS_WITH_AES_256_CBC_SHA	{ 0x00,0x36 }
TLS_DH_RSA_WITH_AES_256_CBC_SHA	{ 0x00,0x37 }
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	{ 0x00,0x38 }
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	{ 0x00,0x39 }
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	{ 0x00,0x3E }
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	{ 0x00,0x3F }
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	{ 0x00,0x40 }
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	{ 0x00,0x67 }
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	{ 0x00,0x68 }
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	{ 0x00,0x69 }
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	{ 0x00,0x6A }
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	{ 0x00,0x6B }

In addition to the cipher suites itemized in the TLS 1.2 specification, there is a complementary RFC 5116 [17] that introduces algorithms for AEAD and defines a uniform interface and a registry for such algorithms. This topic is already important for TLS 1.2, but it gets even more important for TLS 1.3 (because the use of an AEAD cipher is mandatory). As its name suggests, an AEAD algorithm (or cipher) combines encryption (to provide confidentiality) with authentication (to provide integrity) in a single cryptographic transformation. The resulting combined transformation is usually more efficient and more secure than the sequential application of the respective basic transformations. AEAD can be constructed in many ways. It is, for example, possible to use a block cipher (e.g., AES) in a specific mode of operation, such as the counter with CBC-MAC mode (CCM) [18] or Galois/Counter Mode (GCM) [19]. The use of AES in CCM is specified in [20], whereas the use of AES in GCM is specified in [21, 22].[19]

AEAD ciphers are usually nonce-based, meaning that the cryptographic transformation takes a nonce as an additional and auxiliary input (in addition to the

19 The IANA maintains a registry of AEAD algorithms at https://www.iana.org/assignments/aead-parameters/aead-parameters.xhtml#aead-parameters-2.

Table 3.13
TLS 1.2 Cipher Suites That Employ an Anonymous Diffie-Hellman Key Exchange

Cipher Suite	Value
TLS_DH_anon_WITH_RC4_128_MD5	{ 0x00,0x18 }
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	{ 0x00,0x1B }
TLS_DH_anon_WITH_AES_128_CBC_SHA	{ 0x00,0x34 }
TLS_DH_anon_WITH_AES_256_CBC_SHA	{ 0x00,0x3A }
TLS_DH_anon_WITH_AES_128_CBC_SHA256	{ 0x00,0x6C }
TLS_DH_anon_WITH_AES_256_CBC_SHA256	{ 0x00,0x6D }

plaintext and the key). A nonce is just a random-looking and nonrepeating value that serves as an additional source of entropy for the (now probabilistic) encryption. Each AEAD cipher suite must specify how the nonce is constructed and how long it should be. Also, an AEAD cipher is able to authenticate associated data that is not encrypted. This is why it is called "authenticated encryption with associated data" and not only "authenticated encryption." This associated data can be some header information that cannot be encrypted by default. In the case of TLS, this data comprises the sequence number and some header fields from the `TLSPlaintext` or `TLSCompressed` structure that is subject to the AEAD cipher. Because an AEAD cipher does not comprise a separate MAC generation and verification, it does not need respective keys. It only uses encryption keys (i.e., `client_write_key` and `server_write_key`), but no MAC key. Anyway, the output of an AEAD cipher is a ciphertext that can be uniquely decrypted using the same values. So while the encryption is probabilistic, the decryption must always be deterministic.

There are situations in which public key operations are too expensive or have other management disadvantages. In these situations it may be advantageous to use symmetric keys, shared in advance among communicating parties, to establish a TLS connection. Standards Track RFC 4279 [23] specifies the following three sets of cipher suites for the TLS protocol that support authentication based on a *preshared key* (PSK):

- The first set of cipher suites (with the PSK key exchange algorithm) use only secret key algorithms and are thus especially suitable for performance-constrained environments.
- The second set of cipher suites (with DHE_PSK key exchange algorithm) use a PSK to authenticate an ephemeral Diffie-Hellman key exchange.

- The third set of cipher suites (with RSA_PSK key exchange algorithm) combine RSA-based authentication of the server with client-based authentication using a PSK.

Note that the SRP-based cipher suites can also be thought of as cipher suites belonging to the first set. The SRP protocol is computationally more expensive than a PSK-based key exchange method, but it is also cryptographically more secure (because it is designed to provide protection against dictionary attacks). Also note that RFC 5487 [24] complements RFC 4279 with regard to newer hash functions (i.e., SHA-256 and SHA-384) and AES in GCM mode, RFC 5489 [25] elaborates on ECDHE_PSK cipher suites, and RFC 4785 [26] specifies the use of PSK without encryption. All of these RFCs are not further addressed here; they can be consulted when needed.

3.4.3 Certificate Management

With regard to certificate management, a major change refers to the way certificate types are named. Until TLS 1.1, the name of the certificate type comprises the algorithm used to sign the certificate. For example, `rsa_fixed_dh` refers to a certificate for a static DH key that is signed with RSA. In TLS 1.2, this functionality is obsoleted by the `supported_signature_algorithms` parameter that can be provided by the server in a CERTIFICATEREQUEST message (in addition to the certificate type). Using this parameter, the server can now specify what hashing and signing algorithms it is willing to accept. Hence, the certificate type no longer restricts the algorithm used to sign the certificate. For example, if the server sends a `dss_fixed_dh` certificate type and accepts SHA-1 and RSA, then the client may reply with a certificate containing a static DH key that is signed with SHA-1 and RSA. For backwards compatibility, however, the names of the certificate types remain unchanged. Again, a list of currently valid certificate types is available from the IANA repository.[20]

In addition to the certificate types mentioned so far, TLS 1.2 also supports ECC and must therefore support ECC-based certificate types, such as `ecdsa_sign` (value 64) for ECDSA signing keys, as well as `rsa_fixed_ecdh` (value 65) and `ecdsa_fixed_ecdh` (value 66) for a static ECDH key exchange (with an RSA signature in the first case and an ECDSA signature in the second case). In all of these cases, the certificate must use the same elliptic curve as the server's key and a point format actually supported by the server.

20 https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-2.

Table 3.14
New TLS Alert Messages Introduced in [16]

Alert	Code	Brief Description (If New)
`unsupported_extension`	110	The sender (client) notifies the recipient (server) that it does not support an extension contained in an extended SERVERHELLO message. This alert message is always fatal.
`certificate_unobtainable`	111	The sender (server) notifies the recipient (client) that it is unable to retrieve a certificate (chain) from the URL supplied in a CERTIFICATEURL message. This alert message may be fatal.
`unrecognized_name`	112	The sender (server) notifies the recipient (client) that it does not recognize the server specified in a server name extension. This alert message may be fatal.
`bad_certificate_status_response`	113	The sender (client) notifies the recipient (server) that it has received an invalid certificate status response. This alert message is always fatal.
`bad_certificate_hash_value`	114	The sender (server) notifies the recipient (client) that a certificate hash does not match a client-provided value. This alert message is always fatal.

3.4.4 Alert Messages

In addition to alert message 41 (`no_certificate_RESERVED`) that has become obsolete in TLS 1.0 and alert messages 21 (`decryption_failed_RESERVED`) and 60 (`export_restriction_RESERVED`) that have become obsolete in TLS 1.1, there are only minor changes in the way TLS 1.2 handles alert messages. In particular, there is a new alert message 110 (`unsupported_extension`) that is fatal and goes hand in hand with the newly introduced TLS extensions mechanism. The message is sent by a client that receives an extended SERVERHELLO message containing an extension it has not put in the CLIENTHELLO message. Meanwhile, the `unsupported_extension` alert message has become part of the TLS 1.2 protocol specification (the other messages are not yet part of the TLS protocol specification but may become so in the future). The new TLS alert messages that are introduced in [16] are summarized in Table 3.14.

3.4.5 Other Differences

Compression algorithms for TLS are specified in Standards Track RFC 3749 [27]. In addition to value 0 (referring to null compression), this RFC also introduces value 1 (referring to the DEFLATE compression method and encoding format [28]). In short, DEFLATE combines Huffman encoding and LZ77:

- LZ77 is named after its inventors, Jacob Liv and Abraham Lempel, who published their compression method in 1977 [29]. In short, the method scans an input string, looks for repeated substrings, and replaces them with references that point back to their last occurrence (within the window or dictionary size).
- Huffman encoding was proposed by David A. Huffman in 1952 [30]. In short, it builds a table that maps each source symbol to a code. When more frequently occurring symbols are mapped to short codes, then an optimal encoding may be achieved.

Compression in TLS is hardly ever used in the field, and—due to the existence of compression-related attacks (Section A.6)—the general recommendation is not to use it at all. Anyway, it is no longer supported in TLS 1.3.

3.5 TLS 1.3

Roughly around 2014, the IETF TLS WG started to specify a new version of the TLS protocol. After many delays and difficulties regarding the naming[21] of the protocol and the clearing up of some compatibility issues, the work was finished in 2018, when TLS 1.3 with version number 3,4 (0x0304) was officially released and specified in RFC 8446 [4]. The designers of TLS 1.3 had two main goals in mind:

- On the one hand, they wanted to take into account all attacks that had been mounted against the previous versions of the SSL/TLS protocol (Appendix A) and come up with a new version that is as secure as possible and that possibly even improves privacy.
- On the other hand, they were influenced by some other protocol designs that were more efficient, especially in terms of RTTs.

With regard to security and privacy, the designers of TLS 1.3 deprecated and put aside all cryptographic constructions that were questionable or not in line with the state-of-the-art. In particular, TLS 1.3 no longer supports compression, uses PSK-based mechanisms instead of session resumption and renegotiation, requires

21 Note that the new version of the TLS protocol is so different from its predecessors that some people have opted for a completely new name or major version number, such as 2.0, 3.0, or even 4.0.

a PSK or a digital signature (RSA,[22] ECDSA,[23] or EdDSA[24]) for authentication and an AEAD cipher for data confidentiality and integrity, as well as a PSK and/or an ephemeral Diffie-Hellman key exchange with a subsequent key derivation based on the HKDF [7]. In terms of security, this is as far as one can go given the current knowledge in cryptography. The strongest point is certainly the fact that nonephemeral (i.e., static) RSA and DH are no longer supported. This point led to a long and controversial discussion within the IETF TLS WG.[25] Also, a strong argument in favor of the security of TLS 1.3 is the use of formal methods to verify both the protocol and some implementations thereof. Tools like *ProVerif*[26] and the *Tamarin Prover*[27] have played a crucial role here.

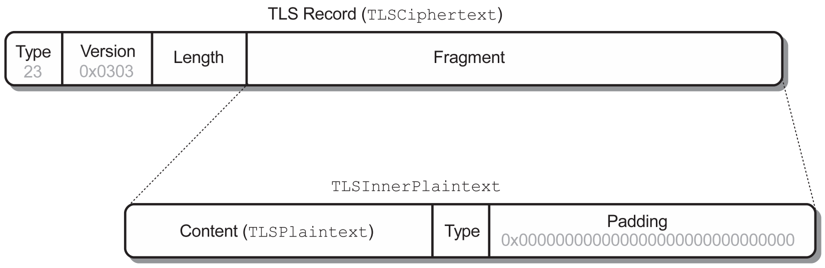

Figure 3.4 TLS 1.3 encapsulation mechanism.

22 Here, the RSA signature system can either be used natively, according to PKCS #1 version 1.5, or in a more sophisticated way, according to a RSA signature scheme with appendix (RSASSA) and a distinct padding scheme known as probabilistic signature scheme (PSS). The result (i.e., RSASSA-PSS) is known to have very good security properties. Both RSA signature systems are specified in [31].

23 ECDSA is specified in American National Standards Institute (ANSI) X9.62 [32] and FIPS PUB 186 since version 2 [33].

24 EdDSA is a variant of ECDSA that is based on twisted Edwards curves. Twisted Edwards curves, in turn, are generalized Edwards curves, and Edwards curves are just a family of elliptic curves originally studied by Harold Edwards in 2007. EdDSA is fully specified in [34, 35].

25 Fairly late in the TLS 1.3 standardization process, some representatives of the banking industry opposed to the elimination of the RSA key exchange mechanism. Their argument was the large installed base of TLS based on RSA. After a heated debate, the IETF decided against the continued use of RSA and specified how to use (elliptic curve) DHE with static (elliptic curve) Diffie-Hellman private keys instead. These keys can the be installed on various devices to allow passively monitoring intranet TLS connections. The respective architecture is specified in a document named "Data Center use of Static Diffie-Hellman in TLS 1.3" (draft-green-tls-static-dh-in-tls13-01). The dispute later culminated in a three-part standard ETSI TS 103 523-2 [36–38] and a respective middlebox security protocol (MSP). The topic is further addressed in Section 5.4.

26 https://bblanche.gitlabpages.inria.fr/proverif.

27 https://tamarin-prover.github.io.

To improve privacy, handshake messages are encrypted whenever possible. This includes public key certificates and some extensions that may be sent after the hello messages in a distinct handshake message of its own. Furthermore, TLS 1.3 supports a simple encapsulation mechanism to provide better privacy (by hiding the content type and the length of the data) and yet be backwards compatible. The respective mechanism is illustrated in Figure 3.4. The fragment of a TLS plaintext record (i.e., `TLSPlaintext`) yields the content of a new TLS inner plaintext record (i.e., `TLSInnerPlaintext`). The type and some padding (i.e., a series of zero bytes) are appended to the content field, and the result is cryptographically protected to yield the fragment of a TLS ciphertext record (i.e., `TLSCiphertext`). In the respective header, the type is set to 23 and the version to 0x0303 (that refers to `legacy_record_version`). As explained in Chapter 4, a similar but more involved encapsulation mechanism is used by the DTLS protocol.

With regard to efficiency, the designers of TLS 1.3 were strongly influenced by technologies to improve the latency and reduce the number of RTTs required to establish a secure connection. Prior to TLS 1.3, a handshake required at least two RTTs, and it was even worse when certificates had to be verified; for example, either certificate revocation lists (CRLs) had to be downloaded and checked or the online certificate status protocol (OCSP) had to be invoked (Section 6.2.2). Against this background, Google had already developed some opportunistic techniques known as *Snap Start*[28] and *False Start*[29] [39], and similar ideas had also been pursued in a key exchange protocol named OPTLS [40].[30] From a bird's eye perspective, the key idea is to have the client guess a Diffie-Hellman group supported by the server and already provide a Diffie-Hellman share in the first flight or even the first message in this flight. This may save one RTT, and the result is a 1-RTT handshake. Furthermore, if the client and server already share a PSK (e.g., from a previous session), then the client can include some early application data in the first message, resulting in a 0-RTT handshake. This is as efficient as a handshake protocol can possibly be.

As illustrated in Figure 3.5, the TLS 1.3 handshake protocol and the respective message flow are entirely new and very different from all previous SSL/TLS protocol versions. It is therefore outlined next, before more details about key derivation, certificate management, alert messages, and other differences are provided.

28 The technology is still documented in an Internet-Draft (https://tools.ietf.org/html/draft-agl-tls-snapstart-00).

29 Google implemented and deployed TLS False Start in the Chrome browser in 2010. But the consistent implementation and deployment of False Start turned out to be more difficult than originally anticipated. Due to this difficulty, the technique was not adapted in the standardization of TLS in the first place.

30 https://eprint.iacr.org/2015/978.

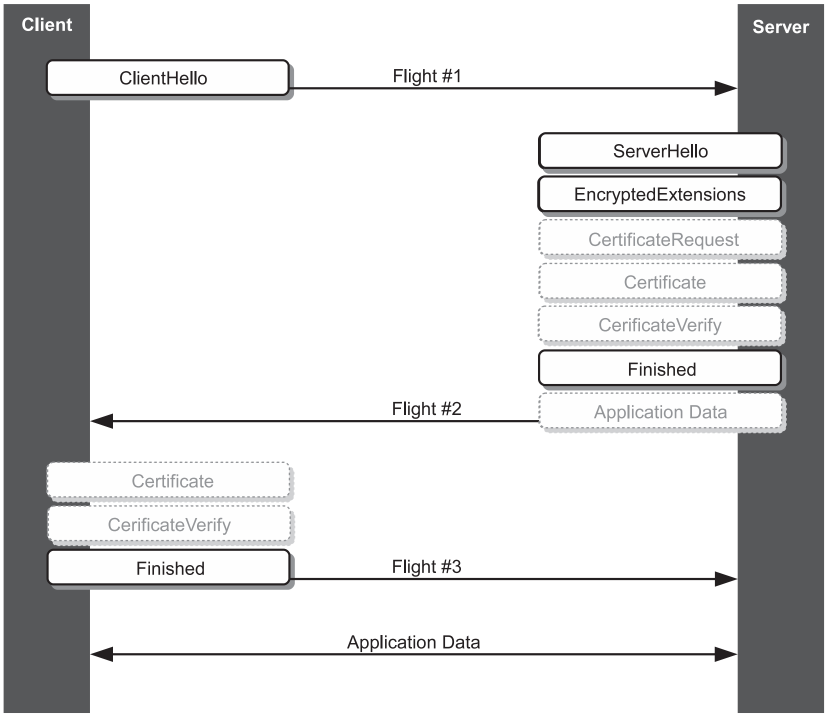

Figure 3.5 The TLS 1.3 handshake protocol message flow (overview).

3.5.1 Handshake Protocol

In the first flight of the TLS 1.3 handshake protocol, the client initiates the handshake with a CLIENTHELLO message sent to the server. As before, the message comprises a version, a random value, a session ID, a list of cipher suites and compression methods, as well as some extensions. Some of these fields are no longer used, at least not in the way they have been used before. This applies to the version, session ID, and list of compression methods. The respective field names now carry the prefix `legacy_` to bear witness to this fact.

- In all previous versions of the SSL/TLS protocol, the version field in the CLIENTHELLO message is used for version negotiation and carries the highest version number supported by the client. Because many servers do not properly implement version negotiation and reject otherwise acceptable CLIENT-HELLO messages (if the version number is higher than the one supported by the server), the version field is now replaced with a `legacy_version` field[31] set to 0x0303 (referring to TLS 1.2) and a `supported_versions` extension (Section C.2.31) that indicates the client-supported versions in preference order, with the most preferred first. The server can then select any version of the TLS protocol that is included in the `supported_versions` extension or fall back to TLS 1.2.

- As in all previous versions of the SSL/TLS protocol, the random value included in the `random` field provides entropy for the subsequent generation of the premaster secret. There is no change here.
- TLS 1.3 no longer supports session resumption, and hence the `session_id` field is no longer used actively. It is only used to resume a pre-TLS 1.3 session or to carry an arbitrary but unpredictable 32-byte random value in compatibility mode (because it is often required that this field is not empty).
- Contrary to all previous versions of the SSL/TLS protocol, a TLS 1.3 cipher suite only comprises an AEAD cipher (with a particular key length) and a cryptographic hash function for the HKDF construction (Section 3.5.2). Neither the key exchange method nor the method to authenticate certificates are defined in the cipher suite; they are now negotiated via extensions. Consequently, there are currently only five cipher suites defined for TLS 1.3 (Appendix B):
  - TLS_AES_128_GCM_SHA256 must be implemented;
  - Both TLS_AES_256_GCM_SHA384 and TLS_CHACHA20_POLY1305_SHA256 should be implemented;
  - Both TLS_AES_128_CCM_SHA256 and TLS_AES_256_CCM_8_SHA-256 can be implemented.[32]

  The cipher suites supported by the client are all included in the `cipher_suites` field. Because a single cryptographic transformation is applied (in an AEAD cipher), only the encryption keys and IVs are needed, and no MAC key needs to be generated.

31 While the value is the same, the `legacy_version` field is still named differently than the `legacy_record_version` that is the record's header version field mentioned earlier in this section.

32 Note that the former of these two cipher suites is also recommended according to the aforementioned IANA registry for cipher suites.

- Due to some compression-related attacks (Section A.6), the only supported compression method in TLS 1.3 is null (i.e., no compression). If a client sent another value in the `legacy_compression_methods` field, then the server would have to generate a fatal `illegal_parameter` alert message and immediately abort the execution of the protocol.
- Since version 1.2, the TLS protocol makes heavy use of extensions. This is even more true for TLS 1.3, and hence the CLIENTHELLO message must comprise several extensions (that are outlined and discussed in Appendix C).[33] Some of the extensions are TLS 1.3-specific, such as `pre_shared_key` (Section C.2.29), `early_data` (Section C.2.30), `supported_versions` (Section C.2.31), `cookie` (Sections 4.2.2.4[34] and C.2.32), `psk_key_exchange_modes` (Section C.2.33), and—maybe most importantly—`key_share` (Section C.2.38). These extensions are listed here in numerical order according to their value.

As usual, extensions may appear in any order.[35] They are dealt with in a request/response fashion, but there are also some extensions that are just indications without any response. A client would typically send its extension requests in the CLIENTHELLO message, whereas the server would send back its responses in SERVERHELLO, ENCRYPTEDEXTENSIONS, HELLORETRYREQUEST, or CERTIFICATE messages. But there are also situations, in which the server may send an extension request and the client may send back a response.

The TLS handshake normally begins with a CLIENTHELLO message sent from the client to the server. After reception, the server has to process the message and decide whether the parameters offered by the client (e.g., the (EC)DHE group(s) and the respective key share(s)) are supported and can be accepted here. If this is the case, then the handshake continues. If, however, the parameters are unsupported or cannot be accepted, then the server responds with a HELLORETRYREQUEST message, and the client must restart the handshake from scratch. This situation is illustrated in Figure 3.6. Mainly for the sake of backward compatibility (with middleboxes as discussed in Chapter 5), the HELLORETRYREQUEST message uses the same format and structure as the SERVERHELLO message, but it uses a fixed 32-byte random value (that basically refers to the SHA-256 hash value of the string "HelloRetryRequest"):

33 In TLS 1.3, the use of certain extensions is mandatory, as functionality has moved into extensions to preserve compatibility with previous versions of the TLS protocol. Again, servers that do not recognize certain extensions must ignore them.

34 While cookies are introduced in TLS, this reference suggests that the respective extension is mainly used in a DTLS setting.

35 The only exception here is the pre_shared_key extension that must appear as the last extension in the CLIENTHELLO message.

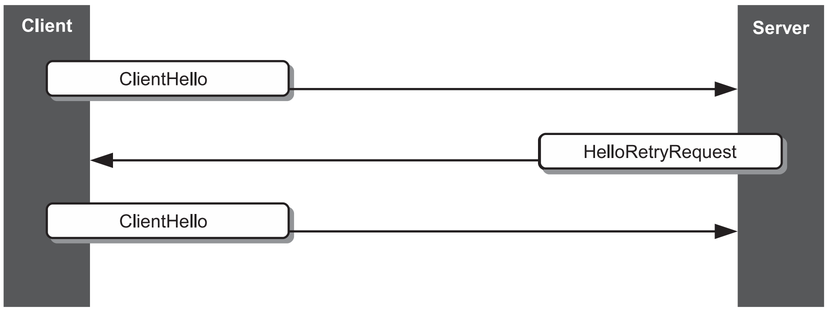

Figure 3.6 The TLS 1.3 handshake starts with mismatched parameters.

```
CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C
```

All other fields of a HELLORETRYREQUEST message are the same as the ones of a SERVERHELLO message (see below). If the server provides a `key_share` extension in the HELLORETRYREQUEST message, then the new CLIENTHELLO message must replace the originally used list of shares with a list that contains a single key share from the group indicated by the server. Otherwise (i.e., if the server doesn't provide a `key_share` extension), then the new CLIENTHELLO message's `key_share` extensions may refer to other (EC)DHE groups. In this case, the `cookie` extension must be present in the CLIENTHELLO message if one was provided in the HELLORETRYREQUEST message.

If no common parameters can be negotiated, then the server must abort the execution of the handshake protocol with an appropriate alert message. Otherwise, it sends back a SERVERHELLO message that specifies the connection parameters. More specifically, the SERVERHELLO message must contain almost the same data as the CLIENTHELLO message (i.e., a version, a random value, a session ID echo (instead of a session ID), a list of cipher suites and compression methods, as well as some extensions). Again, the version, session ID echo, and list of compression methods fields are marked with the `legacy_` prefix.

- Similar to the version indicated in the CLIENTHELLO message, the SERVER-HELLO message comprises a `legacy_version` field set to 0x0303. Furthermore, the message may include a `supported_versions` extension to indicate the server's version choice.

- The server's random value provides entropy and is carried in the SERVERHELLO message's `random` field. Remember from above that the contents of this field serves as a distinguisher between a HELLORETRYREQUEST and a true SERVERHELLO message. Furthermore, there is another subtlety to mention here: In Section 2.4, we said that the TLS_FALLBACK_SCSV can be used to mitigate some downgrade attacks. In TLS 1.3, however, the server's random value is used to provide another downgrade protection mechanism: If a TLS 1.3 server responds to a CLIENTHELLO message using TLS 1.2 or below, then it may provide a hint in the last 8 bytes of its random value.

  - If the TLS version is 1.2, then these bytes are 0x444F574E47524401 (note that the first 7 bytes refer to the ASCII-encoded string "DOWNGRD").
  - If the TLS version is 1.1 or below, then the last byte is 0x00 (instead of 0x01), and hence the 8 bytes are 0x444F574E47524400.

  In either case, the server can signal to the client that it supports TLS 1.3 but willingly downgrades to either TLS 1.2 or TLS 1.1. A TLS 1.3 client that receives a SERVERHELLO message that mandates the use of TLS 1.2 or below must check the last 8 bytes of the random value provided by the server. If they refer to either of the two byte sequences mentioned above, then the client can conclude that something fishy is going on (because the server and client would both support TLS 1.3). In this case, the client must abort the execution of the protocol and return an `illegal_parameter` alert message to the server. Otherwise (i.e., if the last 8 bytes of the server's random value don't match any of the two byte sequences), then the client knows that the server does not support TLS 1.3, and hence that the downgrade seems to be legitimate.

- As its name suggests, the `legacy_session_id_echo` field simply echoes back the contents of the CLIENTHELLO message's `legacy_session_id` field. Again, this field mainly serves the purpose of backward compatibility.

- The server-selected cipher suite and compression method refer to the selections made by the server. The respective fields are called `cipher_suite` and `legacy_compression_method`, and there is nothing special to mention here. Again, due to the attacks outlined in Section A.6, the compression method must be set to null in TLS 1.3.

- As in the case of a CLIENTHELLO message, the server can also select and package multiple nonencrypted extensions in the SERVERHELLO message. As discussed below, there may also be encrypted extensions that are packaged in a new ENCRYPTEDEXTENSIONS handshake message (type 8) that is optional and may follow immediately after the SERVERHELLO message. If a client has requested additional functionality using extensions but this functionality is not provided by the server, then the client may abort the execution of the handshake protocol at will.

After having received a CLIENTHELLO message with properly set extensions, the server must select a key exchange method; for example, PSK and/or (EC)DHE in a particular group.

- If only a PSK is used, then it must be the case that the client has sent a CLIENTHELLO message with a `pre_shared_key` extension that refers to the client's offered PSKs. In this case, the server can select one of them and return its selection in the `pre_shared_key` extension of the SERVERHELLO message.
- If an (EC)DHE is used, then the server must select a particular group (among the groups provided in the `supported_groups` extension of the CLIENTHELLO message), determine a key pair, accomplish the (EC)DHE key exchange with the private key and compute the premaster secret and the keying material to cryptographically protect the messages that are sent back and forth in the sequel. Also, the server must provide its key share in the `key_share` extension of the SERVERHELLO message. This allows the client to compute the same premaster secret and keying material.
- Last but not least, it may be the case that a PSK and an (EC)DHE are used together (by properly specifying the `psk_key_exchange_modes` extensions in either of the hello messages). In this case, the SERVERHELLO message must comprise both extensions (i.e., a `key_share` and a `pre_shared_key` extension). The goal here is to yet use the PSK, but still do an (EC)DHE to provide forward secrecy.

Unless the key exchange is based on a PSK, TLS 1.3 uses a pair of messages that authenticate the server to the client: A CERTIFICATE message (type 11) that provides a server certificate, and a subsequent CERTIFICATEVERIFY message (type 15) that conveys a signature over the entire handshake using the private key that corresponds to the public key referred to by the certificate. Note that if raw public keys (Section C.2.17) or the `cached_info` extension (Section C.2.21) are used, then the CERTIFICATE message does not contain a full-fledged certificate, but rather some other value referring to the server's public key. Also note that the CERTIFICATEVERIFY message is entirely new in TLS 1.3. In previous protocol versions, this message was only used by the client to authenticate to the server.

Now, server authentication is done explicitly, and hence the server must also send a CERTIFICATEVERIFY message to the client.

Finally, the server finishes the second flight by sending a FINISHED message to the client (without a respective CHANGECIPHERSPEC message). As usual, this message comprises a MAC over the entire handshake to authenticate it and to confirm the keys in use, and hence to bind the endpoint's identity to these keys. Also, the server may optionally send some application data to the client. Note, however, that special care must be taken here, because the client may not yet be authenticated to the server.

If the server requested a client certificate in a CERTIFICATEREQUEST (type 13) message, then the client would send a CERTIFICATE and a respective CERTIFICATEVERIFY message to the server in the third flight. Anyway, it completes the flight with a FINISHED message (that is similar to the FINISHED message mentioned above). The client and server can now exchange application data in a secure way. Note that with its three flights, a TLS 1.3 handshake is very similar to a TCP connection establishment. If combined with OCSP stapling, this allows a very fast establishment of secure connections. As mentioned above, this is a the major advantages of TLS 1.3.

Up to TLS 1.2, session IDs and session tickets are used to refer to previously established sessions and session keys. In TLS 1.3, however, session IDs and session tickets are no longer used. Instead, they are replaced with PSKs. If forward secrecy is required, then a PSK can optionally be combined with an (EC)DHE key exchange (depending on the PKS key exchange mode mentioned above).

The use of a PSK is similar to the use of a session ticket (Section C.2.26). After a normal TLS 1.3 handshake is finished, the server can establish a PSK and a respective PSK identity with a NEWSESSIONTICKET message (type 4) sent after the FINISHED messages but before any application data is sent back and forth.[36] This is illustrated in Figure 3.7. The PSK can then be used in a subsequent handshake as illustrated in Figure 3.8. In this case, the handshake does not require server authentication. The PSK is encoded in the `pre_shared_key` extension of the CLIENTHELLO message. More specifically, the client can refer to a PSK or a set of PSKs it is willing to reuse. If the server accepts one of them, then it includes the respective `pre_shared_key` extension in its SERVERHELLO message and goes through a handshake without reauthenticating itself with a certificate (i.e., no CERFTIFICATE or CERFTIFICATEVERIFY message needs to be sent). If a PSK is used this way, then the respective CLIENTHELLO and SERVERHELLO messages may still contain `key_share` extensions to do an (EC)DHE key exchange to provide forward secrecy. As already mentioned above, a PSK can be used in

36 Due to the fact that the NEWSESSIONTICKET message is sent after the main handshake, it is called a post-handshake message in [4].

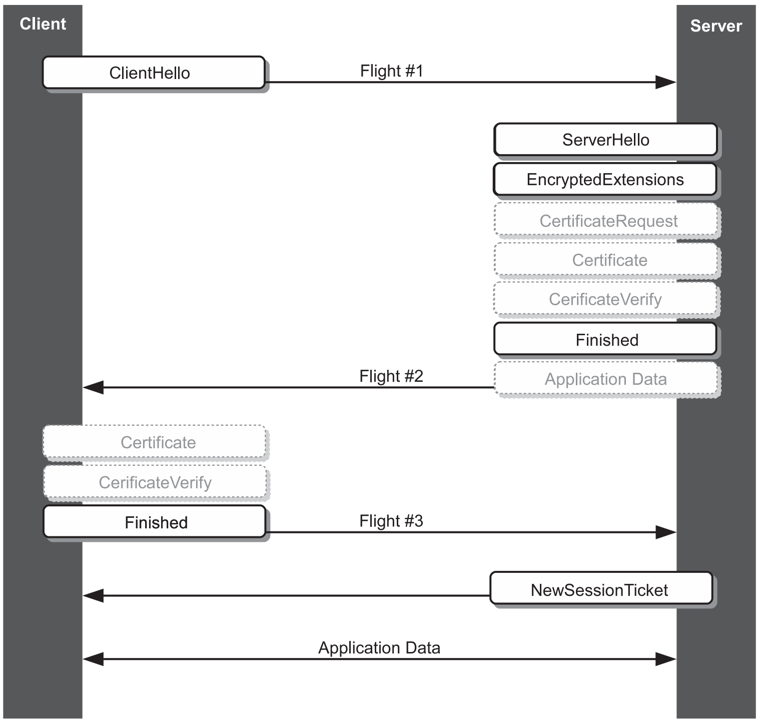

Figure 3.7 The message flow of the TLS 1.3 handshake protocol issuing a session ticket.

conjunction with (EC)DHE to provide forward secrecy, or it can be used alone (in which case forward secrecy is not provided).

If a client shares a PSK with the server, then it may also send some early application data in the first flight. The client can therefore include an `early_data` extension (Section C.2.30) in the CLIENTHELLO message. If the server supports early data, then it may also include the `early_data` extension in its ENCRYPTED-EXTENSIONS message. But the client has not yet received this message, so it can continue streaming 0-RTT data until it receives the server's FINISHED message. It then sends an ENDOFEARLYDATA message (type 5) and a FINISHED message to the server. Note that the ENDOFEARLYDATA message type is new in TLS 1.3 and does not exist in previous versions of the TLS protocol. Also note that for

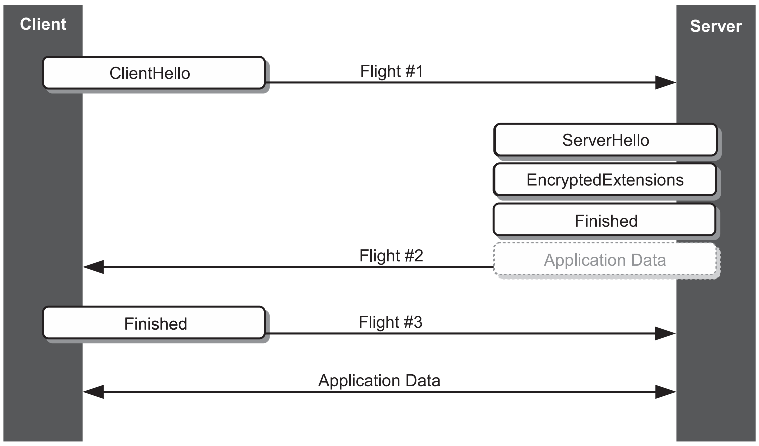

Figure 3.8 The message flow of the TLS 1.3 handshake protocol using a PSK.

the sake of efficiency the server must process the CLIENTHELLO message and then immediately send its flight of messages, rather than waiting for the client's ENDOFEARLYDATA message. If the server doesn't support early data, then the client doesn't have to send the ENDOFEARLYDATA message. The early data can be cryptographically protected (i.e., encrypted and authenticated) with the PSK, but it does not provide forward secrecy. If somebody is able to get the PSK, then he or she is also able to decrypt the early data protected with it. If forward secrecy is needed, then a complementary (EC)DHE key exchange must take place. Also, the use of a PSK to secure early data means that this data is susceptible to replay attacks, meaning that anybody can record a CLIENTHELLO message and replay it at some later point in time. 0-RTT and anti-replay mechanisms are fully addressed in Chapter 8 of [4] and not repeated here. Note, however, that any application receiving and accepting 0-RTT early data must have a mechanism in place to properly handle this issue.

3.5.2 Key Derivation

As mentioned above, TLS 1.3 differs from its predecessors in terms of key derivation. Instead of using an ad-hoc and more or less hand-crafted PRF, it uses a standardized KDF that is based on the HMAC construction. The resulting KDF is known as HKDF [7], and it follows the extract-then-expand paradigm. As such, it uses the HMAC construction for both the extraction of a uniform key from a source key and the expansion of this key into a key stream.

Let $KDF_{extract}$ be the function that uses the HMAC construction to extract a uniform key k' from a salt s and source key k.[37] It can be formally expressed as

$$HKDF_{extract}(s, k) = k' = HMAC_s(k) =$$

This basically means that the HMAC construction is used to remove any bias from the source key k, and hence to provide a key k' that is uniformly distributed. The salt serves as an additional input parameter that ensures that the same source key is mapped to different uniform keys.

Similarly, $HKDF_{expand}$ is the function that expands a uniform key k' into a key stream. The function takes two additional input parameters, namely a context string c and a byte length l for the key stream (meaning that k' must be expanded into an l-byte key stream). In the realm of TLS 1.3, the context string is the concatenation of a label and the hash value of the protocol transcript (i.e., all handshake messages exchanged so far).[38] Because these messages include the random values from the hello messages, any given transcript will also end in different keying material, even if the same key is used (as is the case when a PSK is used to establish multiple connections). Formally, the $HKDF_{expand}$ function is defined as

$$HKDF_{expand}(k', c, l) = T_1 \parallel T_2 \parallel \ldots \parallel T_n$$

where T_0 is initialized with the empty string and T_i is recursively computed as

$$T_i = HMAC_{k'}(T_{i-1} \parallel c \parallel i)$$

for $i = 1, \ldots, n$ and i is written in hexadecimal format.[39] It goes without saying that this recursive construction can be repeated as many times as required (i.e., until l bytes are generated). If l_{hash} refers to the byte length of the output of the hash function (e.g., 20 for SHA-1 or 32 for SHA-256), then $n = \lceil l/l_{hash} \rceil$.

Figure 3.9 illustrates the TLS 1.3 key derivation schedule used in TLS 1.3 (in simplified form). It is used to derive all keying material needed. This includes the

37 If no salt is provided, then the default value is an appropriately sized zero string. In HKDF terminology, the source key k also refers to input keying material (IKM).

38 The hash function for both the HMAC construction and the computation of the hash value for the protocol transcript is the one defined in the cipher suite. Furthermore, the handshake messages (that are hashed) include the handshake message type and length fields, but not the record headers.

39 In HKDF terminology, the output of the HKDF expand function refers to output keying material (OKM).

early secret to protect early data, the *handshake secret* to protect handshake data, and the *master secret* to protect normal application data. This distinction and the use of distinct keys for these purposes is a novelty in TLS 1.3.

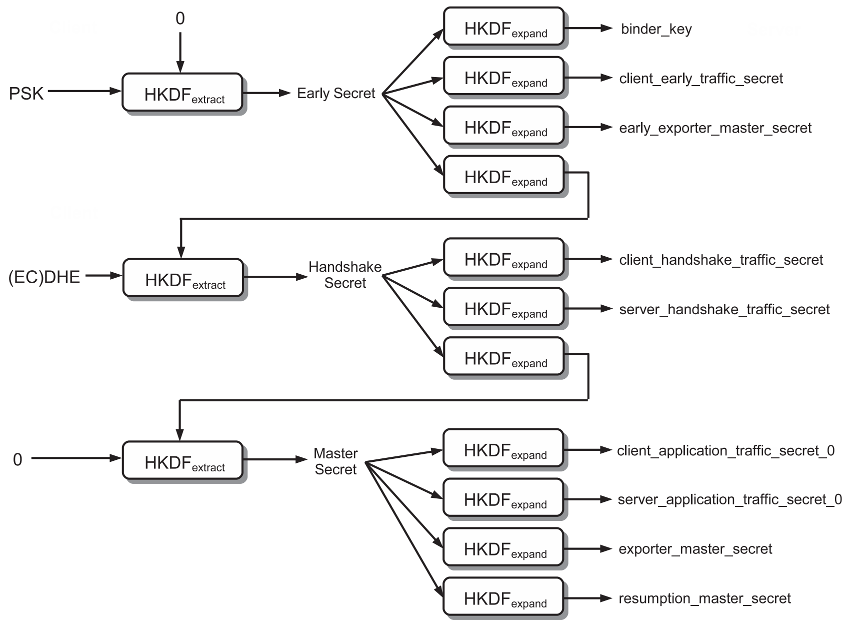
0
PSK
HKDF_extract
Early Secret
HKDF_expand
binder_key
client_early_traffic_secret
early_exporter_master_secret
(EC)DHE
Handshake Secret
client_handshake_traffic_secret
server_handshake_traffic_secret
0
Master Secret
client_application_traffic_secret_0
server_application_traffic_secret_0
exporter_master_secret
resumption_master_secret

Figure 3.9 TLS 1.3 key derivation schedule (simplified).

Starting from the top of Figure 3.9, the $KDF_{extract}$ function is used to generate an early secret. The salt is just a string of zero bytes, whereas the source key is a PSK, if one is available, and zero otherwise.[40] In either case, $KDF_{extract}$ outputs a uniform key that serves as early secret. This value acts as input key for the KDF_{expand} function that is applied four times with distinct context strings. As mentioned above, these strings consist of a label and a hash value of the protocol transcript (as fully specified in Chapter 7 of [4] and not repeated here). Anyway, the four outputs of KDF_{expand} refer to

40 The PSK may be established externally or derived from the `resumption_master_secret` value from a previous connection.

- A `binder_key` that can be used to cryptographically bind a PSK to a particular handshake;
- A `client_early_traffic_secret` that can be used to protect application data sent as early data (in the case of 0-RTT);
- An `early_exporter_master_secret` that can be used in settings where EKM (called "exporter" here) is needed for 0-RTT data;
- Another value that serves as salt for the next iteration of the $KDF_{extract}$ function.

In addition to the salt just created, a newly generated (EC)DHE key may serve as the source key for the second iteration of the $KDF_{extract}$ function. The resulting handshake secret (as mentioned above) is the uniform key that serves as input for the KDF_{expand} function. Again, the exact labels and handshake messages to compute the hash value of the protocol transcript are specified in [4]. The KDF_{expand} function is applied three times, where the outputs refer to

- A `client_handshake_traffic_secret` that can be used by the client to protect handshake traffic it sends to the server;
- A `server_handshake_traffic_secret` that can be used by the server to protect handshake traffic it sends to the client;
- Another value that serves as salt for the next iteration of the $KDF_{extract}$ function.

Finally, the $KDF_{extract}$ function is applied (with the salt just created and a key that only consists of zero bytes) a third time. The output is again a uniform key that represents the master secret and input key for the KDF_{expand} function. This function is applied four times (with distinct labels and handshake messages), where the outputs refer to

- A `client_application_traffic_secret_0` that can be used by the client to protect application sent to the server;
- A `server_application_traffic_secret_0` that can be used by the server to protect application sent to the client;
- An `exporter_master_secret` that can be used in a normal setting where EKM is needed (not 0-RTT);
- A `resumption_master_secret` that can be used to generate PSKs for subsequent handshakes.

Both the client_application_traffic_secret_0 and server_ application_traffic_secret_0 keys are normally used to protect application data. As mentioned above, the distinction between keys to protect handshake traffic and keys to protect application data is new in TLS 1.3 (and has not been made in previous versions of the SSL/TLS protocols). Also, the postfix _0 in the traffic keys' names suggests that there is a possibility to update the keys, and this is where the notion of a KEYUPDATE message (type 4) comes into play. It allows either side to update its traffic secret at any point in time (after the handshake is complete). As usual, the next-to-be-used $\star$_application_traffic_secret_N+1 can be generated with the KDF_{expand} function that is applied to $\star$_application_ traffic_secret_N and a distinct label (as well as $\star$ referring to either client or server). Note, however, that it is up to the application to invoke this recursive key update mechanism whenever needed.

3.5.3 Certificate Management

As mentioned before, TLS 1.3 still uses certificates to authenticate the server (and optionally the client). So certificates are still needed and must be managed accordingly (if no raw public keys are used). But in contrast to its predecessors, TLS 1.3 uses various extensions to specify supported algorithms and preferences (instead of certificates types).

3.5.4 Alert Messages

With regard to alert messages, TLS 1.3 deviates from all previous versions of the SSL/TLS protocols. As before, TLS 1.3 alert messages still comprise a level and a description, but the level no longer serves a purpose and can be ignored at will. In place of the level, the severity of an alert message can be derived directly from its description.

All alert messages used in TLS 1.3 are outlined in Chapter 6 and Section B.2 of [4]. Furthermore, there are a few alert messages that are no longer needed, such as decompression_failure (30) that is also renamed to decompression_ failure_RESERVED (mainly because compression is no longer supported).

3.5.5 Other Differences

Due to the fact that TLS 1.3 is so fundamentally different from all previous versions of the SSL/TLS protocols, we refrain from further elaborating on other differences here. Instead, we reemphasize the fact that the most important differences refer to the handshake messages, the cipher suites, and the way keys are used and derived in the first place.

3.6 HSTS

At the 2009 Black Hat conference, Moxie Malinspike presented "new tricks for defeating SSL in practice," and one of these tricks referred to a tool named SSLStrip that he had developed earlier. The tool acts as a MITM and attempts to remove the invocation of SSL/TLS by simply modifying unencrypted protocols that request the use of SSL/TLS. In the context of web traffic, such an SSL stripping attack can be mounted whenever a client initially accesses a server using HTTP (instead of HTTPS). This can be enforced by an adversary by simply removing the letter "s" from `https` in a requested URL. To mitigate the attack, it makes sense to strictly (i.e., always) apply HTTPS instead of HTTP to secure web traffic. This is where HSTS comes into play. HSTS is based on previous work related to ForceHTTPS[41] that was basically a prototype implementation for Firefox (implemented as a Firefox extension).

Against this background, the IETF officially specified HSTS in Standards Track RFC 6797 back in 2012 [41]. The aim was to provide a mechanism that enables websites to declare themselves to be accessible only via secure connections, where "secure" means that any version of the SSL/TLS protocols needs to be invoked before any data traffic takes place. After having received this declaration, a browser is to interact with the site only over SSL/TLS. While ForceHTTPS used an HTTP cookie to make the declaration, HSTS uses a special HTTP response header field; that is `Strict-Transport-Security`. The use of HSTS has grown tremendously (to about one-fourth of all websites[42]) in the past and has replaced ForceHTTPS entirely (it is mentioned here only for the sake of completeness and for historic reasons).

From a security perspective, the use of HSTS is strongly recommended (although it does not solve all security problems and even introduces some privacy concerns as explained below). To invoke HSTS, a web server must be configured to declare itself an HSTS host when communicating with a client (or browser, respectively). This declaration is made via the HTTP `Strict-Transport-Security` response header that must be sent over a secure connection. This means that a client that sends an HTTP request to, for example, `http://www.esecurity.ch`, must first be redirected to `https://www.esecurity.ch`. A secure connection is then established, before the server can send an HTTP `Strict-Transport-`

41 https://crypto.stanford.edu/forcehttps.

42 https://w3techs.com/technologies/details/ce-hsts.

`Security` header field to the client. The client then considers the web server to be a known HSTS host. (It creates a respective entry in its database that causes all future requests to the domain to be sent over HTTPS.) Alternatively, clients can also be configured to treat specific hosts as HSTS hosts. In this case, the HTTP `Strict-Transport-Security` response headers need not be transmitted in the first place. In fact, all browsers supporting HSTS are currently deployed and shipped with a list of known HSTS hosts (the list is sometimes called a *pre-loaded* or *preload list*).

The HSTS policy enforced by the HTTP `Strict-Transport-Security` response header can be refined with the following two directives that address policy time duration and subdomain applicability.

- The `max-age` directive is mandatory and specifies the number of seconds after the browser has received the HTTP `Strict-Transport-Security` header field, during which the browser regards the host as the known HSTS host. If a value of zero is specified, then the server signals to the browser that it should no longer consider the host to be an HSTS host.
- The `includeSubDomains` directive is optional and valueless. If present, it signals to the browser that the HSTS policy applies to the host and all subdomains (of the host's domain name). This simplifies the invocation of HSTS considerably.

The use of the directives is simple and straightforward. If, for example, a web server returns the HTTP response header

```
Strict-Transport-Security: max-age=10000000;
   InlcudeSubDomains
```

to a browser, then the browser knows that the use of SSL/TLS is required for the data exchange with this server (which then represents an HSTS host). The `max-age` directive further suggests that the HSTS policy should remain in effect for 10,000,000 seconds (which corresponds to more than 115 days or roughly four months), and the `includeSubDomains` directive suggests that the HSTS policy applies to the host and all of its subdomains.

HSTS was well received in the community. This changed a little bit when Sam Greenhalgh showed in 2015 that the feature richness of HSTS can also be turned into a sophisticated tracking technology (that found its way into the trade press under the name *HSTS supercookies*).[43] The technology employs the fact that HSTS can be invoked for each subdomain individually (instead of using the

43 The original announcement was made on https://www.radicalresearch.co.uk/lab/hstssupercookies. Unfortunately, this site does not contain the original announcement anymore.

includeSubDomains directive). This means that whether HSTS is invoked or not (for a particular subdomain) it leaks one bit of information. If used iteratively, a sequence of bits can be compiled to encode a tracking number that is unique for a particular client (or browser, respectively).

In the proof-of-concept implementation of Greenhalgh, the tracking number was 160 bits long. If, for example, the HSTS host www.esecurity.ch wanted to use supercookies to track its users, then it would put in place 160 subdomains, ranging from 000.esecurity.ch to 159.esecurity.ch, and for each bit that is equal to one in the binary representation of the tracking number, it would activate HSTS for the respective subdomain (by sending an appropriate Strict-Transport-Security HTTP response to the client). So if the binary representation of the tracking number started with 1011..., then it would activate HSTS for 000.esecurity.ch, 002.esecurity.ch, 003.esecurity.ch, ... (but not for 001.esecurity.ch). If the client then returned to www.esecurity.ch at some later point in time, the host would redirect the client to all subdomains and observe for every subdomain whether the client uses HTTP or HTTPS. If the subdomain is contacted with HTTP (HTTPS), then the respective bit position can be decoded as a zero (one). This way, the server can compile the binary representation of the tracking number. It goes without saying that this tracking technique only works if the HTTP Strict-Transport-Security response header that is initially sent from the HSTS host (in this example www.esecurity.ch) to the client does not include the includeSubDomains directive. Since this book is about security and not privacy, we don't delve deeper into this issue (but keep in mind that HSTS supercookies yield a privacy concern though).

In spite of the existence of HSTS supercookies and their potential misuse for user tracking, the IETF thinks that the advantages outweigh their disadvantages and therefore strongly encourages the use of HSTS in the field.

3.7 PROTOCOL TRANSCRIPTS

To illustrate the working principles of the TLS protocol, we provide transcripts for TLS 1.0 and TLS 1.2. In either case, the setting is the same as the one described in Section 2.3: A standard web client (i.e., Mozilla Firefox) running at 192.168.1.120 performs a TLS handshake with an Apache web server running at www.esecurity.ch or 49.12.165.202, and Wireshark is used to capture the respective TCP segments. For TLS 1.2, the log file can be downloaded from the book's website in the Wireshark's packet capture (PCAP) format. Another log file is available for TLS 1.3, but this TLS version is less interesting, because most parts are encrypted and not accessible for cleartext inspection.

3.7.1 TLS 1.0

As usual, the client initiates a TLS handshake by sending a CLIENTHELLO message to the server. Again, the message is wrapped in a TLS record that looks as follows (in hexadecimal notation):

```
16 03 01 00 41 01 00 00   3d 03 01 49 47 77 14 b9
02 5d e6 35 ff 49 d0 65   cb 89 93 7d 68 9b 55 e7
b6 49 e6 93 e9 e9 48 c0   b7 d2 13 00 00 16 00 04
00 05 00 0a 00 09 00 64   00 62 00 03 00 06 00 13
00 12 00 63 01 00
```

The TLS record starts with a type field that comprises the value `0x16` (representing 22 in decimal notation, and hence standing for the handshake protocol), a version field that comprises the value `0x0301` (referring to TLS 1.0), and a length field that comprises the value `0x0041` (representing 65 in decimal notation). This means that the fragment of the TLS record that is currently sent to the client is 65 bytes long, and hence that the following 65 bytes represent the CLIENTHELLO message. This message, in turn, starts with `0x01` standing for the TLS handshake message type 1 (referring to a CLIENTHELLO message), `0x00003d` (representing 61 in decimal notation) standing for the message length ($64 - 4 = 61$ bytes), and `0x0301` again referring to TLS 1.0. The subsequent 32 bytes—from `0x4947` to `0xd213`—represent the random value chosen by the client (remember that the first 4 bytes represent the date and time, whereas only the remaining 28 bytes are random). Because there is no TLS session to resume, the session ID length is set to zero (`0x00`), and no session ID is appended. Instead, the next value `0x0016` (representing 22 in decimal notation) indicates that the subsequent 22 bytes refer to 11 cipher suites supported by the client. Each pair of bytes represents a particular cipher suite. For example, the first two cipher suites are referenced with the values `0x0004` and `0x0005` (i.e., TLS_RSA_WITH_RC4_128_MD5 and TLS_RSA_WITH_RC4_128_SHA). The second-to-last byte `01` indicates that there is a single compression method supported by the client, and the last byte `0x00` refers to this compression method (that actually refers to null compression). No TLS extension is appended at the end of the CLIENTHELLO message.

After having received this CLIENTHELLO message, the server responds with a series of TLS handshake messages. If possible, then all messages are merged into a single TLS record and transmitted in a single TCP segment to the client. In our example, such a TLS record comprises a SERVERHELLO, a CERTIFICATE, and a

SERVERHELLODONE message. Similar to SSL, the TLS record starts with a header that comprises the following byte sequence:

```
16 03 01 0a 5f
```

Again, `0x16` refers to the TLS handshake protocol, `0x0301` refers to TLS 1.0, and `0x0a5f` refers to the length of the entire TLS record (which is actually $10 \cdot 16^2 + 5 \cdot 16 + 15 = 2,655$ bytes). The three abovementioned messages are then encapsulated in the rest of the TLS record as follows.

- The SERVERHELLO message looks as follows:

  ```
  02 00 00 46 03 01 49 47   77 14 a2 fd 8f f0 46 2e
  1b 05 43 3a 1f 6e 15 04   d3 56 1b eb 89 96 71 81
  48 d4 87 10 6d e9 20 49   47 77 14 42 53 e0 5e bd
  17 6a e9 35 31 06 f2 d2   30 28 af 46 19 d1 d2 e4
  49 0a 0c cd 90 66 20 00   05 00
  ```

  The message starts with `0x02` standing for the handshake protocol message type 2 (referring to a SERVERHELLO message), `0x000046` standing for a message length of 70 bytes, and `0x0301` standing for TLS 1.0. The subsequent 32 bytes

  ```
  49 47 77 14 a2 fd 8f f0   46 2e 1b 05 43 3a 1f 6e
  15 04 d3 56 1b eb 89 96   71 81 48 d4 87 10 6d e9
  ```

  represent the random value chosen by the server (note again that the first 4 bytes represent the date and time). Afterwards, `0x20` refers to a session ID length of 32 bytes, and hence the subsequent 32 bytes

  ```
  49 47 77 14 42 53 e0 5e   bd 17 6a e9 35 31 06 f2
  d2 30 28 af 46 19 d1 d2   e4 49 0a 0c cd 90 66 20
  ```

  represent the session ID. Remember that this value is going to be used if the client wants to resume the TLS session at some later point in time (before the session expires). Following the session ID, `0x0005` refers to the selected cipher suite (which is TLS_RSA_WITH_RC4_128_SHA in this example), and `0x00` refers to the selected compression method (which is again the null compression).

- Next, the CERTIFICATE message comprises the server's public key certificate. It is quite comprehensive and begins with the following byte sequence:

  ```
  0b 00 0a 0d 00 0a 0a
  ```

In this byte sequence, `0x0b` stands for the TLS handshake protocol message type 11 (referring to a CERTIFICATE message), `0x000a0d` stands for a message length of 2,573 bytes, and `0x000a0a` stands for the length of the certificate chain. Note that the length of the certificate chain must equal the message length minus 3 (the length of the length field). The remaining 2,570 bytes of the message then comprise the certificate chain required to validate the server's public key certificate. (Again, these bytes are not illustrated here.)

- Last but not least, the TLS record also comprises a SERVERHELLODONE message. This message is very simple and only consists of 4 bytes:

  ```
  0e 00 00 00
  ```

  `0x0e` stands for the TLS handshake protocol message type 14 (referring to a SERVERHELLODONE message), and `0x000000` stands for a message length of zero bytes.

After having received the SERVERHELLODONE message, it is up to the client to submit a series of messages to the server. In our example, this series comprises a CLIENTKEYEXCHANGE, a CHANGECIPHERSPEC, and a FINISHED message. Each of these messages is transmitted in a TLS record of its own, but all three records can be transmitted in a single TCP segment to the server. They are described as follows.

- The CLIENTKEYEXCHANGE message is sent in the first TLS record. In our example, this record looks as follows:

  ```
  16 03 01 00 86 10 00 00   82 00 80 ac 18 48 2e 50
  32 32 bb 5d 2b 35 39 f2   3d 32 cd 19 86 b4 57 e9
  c8 a5 5b ad da 29 24 22   90 bc d7 3d cd f8 94 8a
  4f 95 72 0c 13 52 52 82   e4 b0 25 f4 b8 b6 e1 7d
  2e d9 65 ce 6f 7c 33 70   12 41 63 87 b4 8b 35 71
  07 d1 0f 52 9d 3a ce 65   96 bc 42 af 2f 7b 13 78
  67 49 3e 36 6e d1 ed e2   1b b2 54 2e 35 bd cc 2c
  88 b2 2d 0c 5c bb 20 9a   d4 c3 97 e9 81 a7 a8 39
  05 1a 5d f8 06 af e4 ef   17 07 30
  ```

  In the TLS record header, `0x16` stands for the handshake protocol, `0x0301` refers to TLS 1.0, and `0x0086` represents the length of the TLS record (134 bytes). After this header, the byte `0x10` stands for the handshake protocol message type 16 (referring to a CLIENTKEYEXCHANGE message), and the following three bytes `0x000082` refer to the message length (130 bytes). Consequently, the remaining 130 bytes of the message represent the premaster secret (as chosen by the client) encrypted under the server's public RSA key. The RSA encryption is line with PKCS #1.

- The CHANGECIPHERSPEC message is transmitted in the second TLS record. This record is very simple and consists of only 6 bytes:

  ```
  14 03 01 00 01 01
  ```

  In the TLS record header, `0x14` (20 in decimal notation) stands for the change cipher spec protocol, `0x0301` refers to TLS 1.0, and `0x0001` represents the message length of one single byte. This byte (i.e., `0x01`), in turn, is the last byte included in the record.

- The FINISHED message is the first message that is cryptographically protected according to the newly negotiated cipher spec. Again, it is transmitted in a TLS record of its own. This record looks as follows:

  ```
  16 03 01 00 24 fb 94 5f   ea 62 ec 90 04 36 5a f6
  c7 c9 1e ae 5d da 70 31   cc 63 2f 81 87 97 60 46
  d0 43 fa 6e 29 94 6c cd   17
  ```

  In the TLS record header, `0x16` stands for the handshake protocol, `0x0301` refers to TLS 1.0, and `0x0024` represents the length of the TLS record (36 bytes). These 36 bytes are encrypted and look like gibberish to somebody not holding the appropriate decryption key.

After having received the CHANGECIPHERSPEC and FINISHED messages, the server must respond with the same pair of messages (not illustrated in our example). Afterward, application data can be exchanged in TLS records. Such a record may start as follows:

```
17 03 01 02 13
```

In the TLS record header, `0x17` (23 in decimal notation) stands for the application data protocol, `0x0301` stands for TLS 1.0, and `0x0213` (531) stands for the length of the encrypted data fragment (that is 531 bytes long). It goes without saying that an arbitrary number of TLS records can now be exchanged between the client and the server, and hence that the amount of application data can be arbitrarily large.

3.7.2 TLS 1.2

The transcript starts with a CLIENTHELLO message that is sent from the client to the server. This message is transmitted in a single TLS record that looks as follows:

```
16 03 01 00 be 01 00 00   ba 03 03 87 23 8f 22 bb
31 bc 06 c2 ae b7 5e b2   3f 49 68 6e e8 f3 a6 af
b4 72 ec 01 4c cd 4d 6c   06 51 13 00 00 1e c0 2b
c0 2f cc a9 cc a8 c0 2c   c0 30 c0 0a c0 09 c0 13
c0 14 00 33 00 39 00 2f   00 35 00 0a 01 00 00 73
00 00 00 15 00 13 00 00   10 77 77 77 2e 65 73 65
63 75 72 69 74 79 2e 63   68 00 17 00 00 ff 01 00
01 00 00 0a 00 08 00 06   00 17 00 18 00 19 00 0b
00 02 01 00 00 23 00 00   33 74 00 00 00 10 00 0e
00 0c 02 68 32 08 68 74   74 70 2f 31 2e 31 00 05
00 05 01 00 00 00 00 00   0d 00 18 00 16 04 01 05
01 06 01 02 01 04 03 05   03 06 03 02 03 05 02 04
02 02 02
```

As usual, the record starts with a header that comprises a type field value 0x16 (representing 22 in decimal notation), standing for a TLS record that may comprise one or several handshake messages (in this case, the record only comprises a single message), a version field value 0x0301, standing for TLS 1.0, and a length field value 0x00be (representing 190 in decimal notation), standing for the byte length of the fragment that comprises the CLIENTHELLO message. This message starts with 0x01, standing for the handshake message type 1 (referring to a CLIENTHELLO message), 0x0000ba (representing 186 in decimal notation) standing for a the length of the message, and 0x0303 standing for TLS version 1.2. So in contrast to the record header, the client signals to the server that it supports TLS 1.2, and hence that it can use TLS 1.2 to establish a session and exchange data accordingly. The following 32 bytes—ranging from 0x8723 to 0x5113—represent the random value chosen by the client (remember that the first 4 bytes represent timing information and that only the subsequent 28 bytes represent the actual random value). Because there is no TLS session to resume, the session ID length that follows is zero (i.e., 0x00) and hence no session ID is appended.

The next 2 bytes (i.e., 0x001e representing 30 in decimal notation), indicate that the subsequent 30 bytes refer to the 15 cipher suites that are supported by the client. They are summarized in Table 3.15. In the first column of the table, the code (represented by a pair of bytes) refers to a particular cipher suite as outlined in Appendix B. Because TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 is the cipher suite preferred by the client (as it appears first in the list) that is also supported by the server, this is the cipher suite that is going to be selected afterwards.

After the list of cipher suites, the byte 0x01 refers to one compression method that is supported by the client, and the byte 0x00 refers to this compression method which is null (standing for no compression). All bytes that follow refer to TLS extensions. The length of the extensions is indicated with the first two bytes; that

Table 3.15
15 Cipher Suites Supported by the Client

Code	Cipher Suite
`c0 2b`	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
`c0 2f`	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
`cc a9`	TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
`cc a8`	TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
`c0 2c`	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
`c0 30`	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
`c0 0a`	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
`c0 09`	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
`c0 13`	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
`c0 14`	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
`00 33`	TLS_DHE_RSA_WITH_AES_128_CBC_SHA
`00 39`	TLS_DHE_RSA_WITH_AES_256_CBC_SHA
`00 2f`	TLS_RSA_WITH_AES_128_CBC_SHA
`00 35`	TLS_RSA_WITH_AES_256_CBC_SHA
`00 0a`	TLS_RSA_WITH_3DES_EDE_CBC_SHA

is 0x0073 (representing 115 in decimal notation). So, the next 115 bytes refer to a total of 10 extensions that are included in the CLIENTHELLO message (and further addressed in Appendix C). We briefly outline them in the order in which they occur in the transcript (but, in principle, the order is arbitrary and does not matter).

Table 3.16
The `server_name` Extension of the CLIENTHELLO Message (25 Bytes)

Bytes	#	Description
`00 00`	2	Type of the extension (0 standing for `server_name`)
`00 15`	2	Length of the extension (21)
`00 13`	2	Length of the server name list (19)
`00`	1	Type of the server name (0 standing for `host_name`)
`00 10`	2	Length of the server name (16)
`77 ... 68`	16	ASCII encoding of the server name (i.e., `www.esecurity.ch`) 77 77 77 2e 65 73 65 63 75 72 69 74 79 2e 63 68

The first extension (i.e., `server_name`) refers to the SNI (Section C.2.1) and comprises 25 bytes as summarized in Table 3.16. In the table header, # refers to the number of bytes occupied by the field. In the description column, a number in brackets refers to the decimal value of the respective bytes (so 0x0015 refers to 21). If the value is standing for a type, then the respective type name is usually added, as well. With this information, it should be possible to read and understand the byte encoding of the `server_name` extension. In this particular case, the extension is used to tell the server that the client wants to establish a TLS session to the server `www.esecurity.ch`.

The next two extensions, `extended_master_secret` and `renegotiation_info` (Section C.2.43), provide secure renegotiation and mitigate the renegotiation and triple handshake attacks (Section A.5).

- The `extended_master_secret` extension comprises 4 bytes that are summarized in Table 3.17. The client uses this extension to signal to the server that it supports the protection mechanism.

Table 3.17
The `extended_master_secret` Extension of the CLIENTHELLO Message (4 Bytes)

Bytes	#	Description
`00 17`	2	Type of the extension (23 standing for `extended_master_secret`)
`00 00`	2	Length of the extension (0)

- The `renegotiation_info` extension comprises 5 bytes that are summarized in Table 3.18. Again, the client uses this extension to signal to the server that it supports the protection mechanism.

Table 3.18
The `renegotiation_info` Extension of the CLIENTHELLO Message (5 Bytes)

Bytes	#	Description
`ff 01`	2	Type of the extension (65281 standing for `renegotiation_info`)
`00 01`	2	Length of the extension (1)
`00`	1	Length of the renegotiation info extension (0)

Both extensions are empty and do not include any data. In either case, the server is going to tell the client in the SERVERHELLO message whether it also supports the mechanism; that is, if it supports the mechanism, then the respective extension is also included in the SERVERHELLO message (see below).

Using the following pair of related extensions (i.e., `elliptic_curves` and `ec_point_formats`), the client signals to the server that it supports ECC (Section C.2.10):

- The `elliptic_curves` extension comprises 12 bytes that are summarized in Table 3.19. Using this extension, the client tells the server that its supports the three elliptic curves secp256r1, secp384r1, and secp521r1. According to Table 3.19, each curve is referenced with a 2-byte code.

Table 3.19
The `elliptic_curves` Extension of the ClientHello Message (12 Bytes)

Bytes	#	Description
`00 0a`	2	Type of the extension (10 standing for `elliptic_curves`)
`00 08`	2	Length of the extension (8)
`00 06`	2	Length of the elliptic curve codes (6)
`00 17`	2	Code of the elliptic curve secp256r1 (23)
`00 18`	2	Code of the elliptic curve secp384r1 (24)
`00 19`	2	Code of the elliptic curve secp521r1 (25)

- In addition, the `ec_point_formats` extension comprises 6 bytes that are summarized in Table 3.20. Using this extension, the client informs the server that it does not compress any data related to ECC, and hence that it uses an uncompressed format.[44] This is the usual behavior of most clients in use today.

Table 3.20
The `ec_point_formats` Extension of the ClientHello Message (6 Bytes)

Bytes	#	Description
`00 0b`	2	Type of the extension (11 standing for `ec_point_formats`)
`00 02`	2	Length of the extension (2)
`01`	1	Length of the point formats (1)
`00`	1	Uncompressed format (0)

The next extension (i.e., `session_ticket`) comprises 4 bytes that are summarized in Table 3.21. It is used to signal to the server that the client supports session tickets (Section C.2.26). If the server also supports session tickets and such tickets can hence be used, then they are transmitted in subsequent handshake messages (see below).

As discussed in Section C.2.15, NPN was a predecessor extension of ALPN (mainly used for SPDY). Support for NPN (type 13172) is signaled by the

44 ECC works with points on an elliptic curve over a finite field. Each point that fulfills the equation of the curve is a group element. Because it is possible to compute the y-coordinate from the x-coordinate on the fly (using the curve equation), one only needs to store the x-coordinates of the points. On the other hand, in uncompressed format one always stores the x- and y-coordinate of a given point.

Table 3.21
The `session_ticket` Extension of the CLIENTHELLO Message (4 Bytes)

Bytes	#	Description
`00 23`	2	Type of the extension (35 standing for `session_ticket`)
`00 00`	2	Length of the extension (0)

`next_protocol_negotiation` extension that comprises 4 bytes as summarized in Table 3.22, and support for ALPN (type 16) is signaled by the `application_layer_protocol_negotiation` extension that comprises 18 bytes as summarized in Table 3.23. In this table, the last 3 lines refer to the supported protocols 0x6832 standing for h2 in ASCII (and hence HTTP/2 over TLS) and 0x687474702f312e31 standing for http/1.1 (and hence HTTP/1.1 over TLS). The IANA provides a comprehensive register of all ALPN protocol IDs in use today.[45]

Table 3.22
The `next_protocol_negotiation` Extension of the CLIENTHELLO Message (4 Bytes)

Bytes	#	Description
`33 74`	2	Type of the extension (13172 standing for `next_protocol_negotiation`)
`00 00`	2	Length of the extension (0)

Table 3.23
The `application_layer_protocol_negotiation` Extension of the CLIENTHELLO Message (18 Bytes)

Bytes	#	Description
`00 10`	2	Type of the extension (16)
`00 0e`	2	Length of the extension (14)
`00 0c`	2	Length of the ALPN extension (12)
`02`	1	Length of the ALPN string (2)
`68 32`	2	ALPN next protocol (h2 in ASCII)
`08`	1	Length of the ALPN string (8)
`68 74 74 70 2f 31 2e 31`	8	ALPN next protocol (http/1.1 in ASCII)

The `status_request` extension that comes next comprises 9 bytes that are summarized in Table 3.24. As explained in Section C.2.6, this extension asks

45 http://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids.

for OCSP stapling supported by the server. In this example, the client does not ask for OCSP responses for multiple servers; otherwise, the `status_request_v2` extension (value 17) would have to be invoked.

Table 3.24
The `status_request` Extension of the CLIENTHELLO Message (9 Bytes)

Bytes	#	Description
`00 05`	2	Type of the extension (5 standing for `status_request`)
`00 05`	2	Length of the extension (5)
`01`	1	Certificate status type (standing for OCSP)
`00 00`	1	Length of the responder ID list (0)
`00 00`	2	Length of the request extensions

Table 3.25
The `signature_algorithms` Extension of the CLIENTHELLO Message (28 Bytes)

Bytes	#	Description
`00 0d`	2	Type of the extension (13 standing for `signature_algorithms`)
`00 18`	2	Length of the extension (24)
`00 16`	2	Length of the signature hash algorithms (22)
`04 01`	2	SHA256 (4) and RSA (1)
`05 01`	2	SHA384 (5) and RSA (1)
`06 01`	2	SHA512 (6) and RSA (1)
`02 01`	2	SHA-1 (2) and RSA (1)
`04 03`	2	SHA256 (4) and ECDSA (3)
`05 03`	2	SHA384 (5) and ECDSA (3)
`06 03`	2	SHA512 (6) and ECDSA (3)
`02 03`	2	SHA-1 (2) and ECDSA (3)
`05 02`	2	SHA384 (5) and DSA (2)
`04 02`	2	SHA256 (4) and DSA (2)
`02 02`	2	SHA-1 (2) and DSA (2)

Finally, the `signature_algorithms` extension (Section C.2.12) comprises 28 bytes that are summarized in Table 3.25. The client uses this extension to tell to the server what hash and signature algorithms it supports. This includes SHA-1, SHA256, SHA384, and SHA512 for hashing, and RSA, DSA, and ECDSA for signing. Note, however, that the combination of SHA512 and DSA (i.e., 0x0602) is missing in this list.

After having received the CLIENTHELLO message, the server responds with a series of TLS handshake messages. The first of these messages is a SERVERHELLO message that is packaged in a TLS record of its own. It looks as follows:

```
16 03 03 00 41 02 00 00   3d 03 03 f8 7a 56 de 47
c4 fb 13 64 a6 a0 8f 18   91 46 38 5e a8 8d 68 0c
a9 c6 23 a5 58 5f 6b 53   99 98 22 00 c0 30 00 00
15 00 00 00 00 ff 01 00   01 00 00 0b 00 04 03 00
01 02 00 23 00 00
```

Again, the first 5 bytes refer to the TLS record header that comprises a 1-byte field referring to the content type (0x16 standing for a handshake message), a 2-byte field referring to the TLS version (0x0303 standing for TLS 1.2),[46] and a 2-byte field referring to the length of the record (0x0041 standing for 65 bytes). So the subsequent 65 bytes comprise the TLS record fragment that includes the SERVERHELLO message.

Table 3.26
The server_name Extension of the SERVERHELLO Message (4 Bytes)

Bytes	#	Description
00 00	2	Type of the extension (0 standing for server_name)
00 00	2	Length of the extension (0)

Table 3.27
The renegotiation_info Extension of the SERVERHELLO Message (5 Bytes)

Bytes	#	Description
ff 01	2	Type of the extension (65281 standing for renegotiation_info)
00 01	2	Length of the extension (1)
00	1	Length of the renegotiation info extension (0)

The actual SERVERHELLO message starts with a byte that refers to the type of the handshake message (0x02 standing for SERVERHELLO), 3 bytes that refer to the byte length of the message (0x00003d standing for 61), and 2 bytes that refer to the TLS version (0x0303 standing for TLS 1.2). The following 32 bytes (i.e., from 0xf87a to 0x9822) refer to the random value chosen by the server (note again that the first 4 bytes represent timing information). Afterwards, 0x00 refers to the length of the session ID (meaning that no session ID is transferred), 0xc030 refers to the cipher suite selected by the server (i.e., TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384), 0x00 refers to the compression method null, and 0x0015 refers to the byte length of the four TLS extensions

46 Here, the server signals to the client that it supports TLS 1.2 (instead of only TLS 1.0).

Table 3.28
The `ec_point_format` Extension of the SERVERHELLO Message (8 Bytes)

Bytes	#	Description
`00 0b`	2	Type of the extension (11 standing for `ec_point_formats`)
`00 02`	2	Length of the extension (2)
`03`	1	Length of the point formats (3)
`00`	1	Uncompressed point format (0)
`01`	1	ANSI X9.62 compressed prime point format (1)
`02`	1	ANSI X9.62 compressed char2 point format (2)

Table 3.29
The `session_ticket` Extension of the SERVERHELLO Message (4 Bytes)

Bytes	#	Description
`00 23`	2	Type of the extension (35 standing for `session_ticket`)
`00 00`	2	Length of the extension (0)

that follow. They are summarized in Tables 3.26 to 3.29. They basically inform the client that the server also supports the `server_name`, `renegotiation_info`, `ec_point_formats`, and `session_ticket` extensions, and hence that these extensions can be used in the sequel. The other extensions may not be supported by the server, such as, for example, the `extended_master_secret` extension, or may directly be used without any further notification, such as the `elliptic_curves` extension.

According to the TLS handshake protocol, the next messages that are sent from the server to the client are the CERTIFICATE, the SERVERKEYEXCHANGE,[47] and the SERVERHELLODONE message.

The CERTIFICATE message is very large and not fully depicted here. It starts with the following 32 bytes (and comprises 2,443 additional bytes):

```
16 03 03 09 ab 0b 00 09   a7 00 09 a4 00 05 08 30
82 05 04 30 82 03 ec a0   03 02 01 02 02 12 03 99
...
```

Again, the TLS record header comprises 0x16 referring to the handshake message type, 0x0303 referring to TLS 1.2, and 0x09ab referring to a 2,475 bytes long record. The CERTIFICATE message then starts with 0x0b referring to the message

47 This message is required, because the server has selected the cipher suite TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 that comprises a Diffie-Hellman key exchange.

type (11 standing for CERTIFICATE), 0x0009a7 referring to the message length (2,471 bytes), 0x0009a4 referring to the byte length of all certificates in the certificate chain (2,468 bytes), and 0x000508 referring to the byte length of the first certificate in the chain (1,288 bytes). This certificate starts with the byte sequence 0x308205. If it is decoded, then it can be recognized that the certificate is issued for `www.esecurity.ch`. The second half of the CERTIFICATE message comprises the certificate for the issuer of the certificate for `www.esecurity.ch`. This certificate is 1,174 bytes long and has been issued by the Let's Encrypt Authority X3. This CA, in turn, has been certfied by the DST Root CA X3 that serves as the root CA for Let's Encrypt.

After the CERTIFICATE message, the server sends the SERVERKEYEXCHANGE message that is packaged in a TLS record of its own. It looks as follows:

```
16 03 03 01 4d 0c 00 01   49 03 00 17 41 04 12 4c
f0 6c a9 99 5a 2c 9b c7   8c 5a d3 03 02 ef 12 f6
57 d0 63 d8 b6 d6 0d 54   4e 24 61 2a 38 1f 04 51
5c c1 a1 fb 92 1a 43 ba   4e 9a 11 21 2c f7 e7 60
b7 9f 50 3c c4 37 c3 b3   e1 9a 64 26 11 69 04 01
01 00 08 78 50 6b c4 a3   63 58 32 62 aa 06 a3 22
34 1c 56 f9 02 f1 f2 b9   64 ea 54 c7 2d a3 5e 9d
45 dc 77 25 e7 b8 69 b5   96 59 ce d2 e8 fa 75 86
b0 49 d2 00 78 91 99 35   38 10 05 f8 9b da f7 21
8e c2 10 bd e7 62 23 ab   42 b2 8c d5 f6 bd fe 39
d0 6e 6a 40 5a bd 34 32   2f c9 2d 9e d5 86 05 bc
71 dd a3 a9 f2 64 1e cb   e9 16 6e 07 7f 2a ac 74
cc 0e ce 49 96 06 e6 ab   54 46 f1 28 23 c5 82 fd
a8 fb 72 0c c3 9e f5 54   57 b7 1d 69 83 8b 1e 78
16 e8 a3 e3 ab 96 7f 26   db 40 d1 86 76 f3 ed 5c
9a 72 65 e7 ab d1 81 55   2b 9a 62 15 6e 10 10 9b
08 9e 6c eb 3f 27 e5 4f   f7 0a ce ab 4e 7e 34 42
10 80 b1 80 86 86 4f b3   ed f8 8c 82 8a 68 7b 8a
d9 8d 83 3d 58 55 82 09   50 a8 5a 83 63 27 d7 e0
59 ab 87 79 1f 27 e2 79   df cd 61 ee 0f d5 ab e5
6d a3 95 15 48 8f 7d 12   22 6c 3a 22 9e 56 14 28
36 6b
```

The first 5 bytes 0x160303014d refer to the record header, whereas the subsequent 4 bytes 0x0c000149 refer to the header of the SERVERKEYEXCHANGE message that is packaged in the record. The record length is 0x014d (standing for 333 in decimal notation) bytes, whereas the message length is 0x000149 (standing for $333 - 4 = 329$ in decimal notation) bytes. The first byte of the message (i.e., 0x0c) refers to the message type (12) that announces a SERVERKEYEXCHANGE message. The cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384, so an ECC-based ephemeral Diffie-Hellman key exchange needs to be performed. This means that the SERVERKEYEXCHANGE message is used to convey the server's Diffie-Hellman parameters. The first byte after the message header (i.e., 0x03) refers to the curve type `named_curve`, and the subsequent two bytes (i.e., 0x0017) refer to the first named curve secp256r1 that was originally proposed by the client. In the following byte 0x41 (65 in decimal notation), the server indicates the byte length of its public Diffie-Hellman parameter. So the subsequent 65 bytes, ranging from 0x0412 to 0x1169, refer to this parameter. In the following two pairs of bytes, the server specifies the signature hash algorithm (0x0401 standing for SHA256 and RSA) and the length of the signature (0x0100 standing for 256). Consequently, the final 256 bytes, ranging from 0x0878 to 0x366b, refer to an RSA signature for the client's and server's random numbers, as well as the server's public Diffie-Hellman parameter that is included in the SERVERKEYEXCHANGE message. The client can extract and verify the RSA public key from the CERTIFICATE message, and use this key to verify the signature for the server's public Diffie-Hellman parameter accordingly.

Afterwards, the server finishes its flight by sending a SERVERHELLODONE message to the client. The respective TLS record only comprises the following 9 bytes:

```
16 03 03 00 04 0e 00 00 00
```

As usual, the first 5 bytes refer to the record header. The last two bytes of this header (i.e., 0x0004) actually refer to the length of the fragment. This fragment, in turn, only comprises a message header. The first byte of this header (i.e., 0x0e) refers to the message type (14 standing for SERVERHELLODONE), and the subsequent 3 zero bytes refer to the length of the message. Because this value is zero, the message actually comprises no data.

It is now up to the client to send a CLIENTKEYEXCHANGE message to the server to provide its public Diffie-Hellman parameter. This is done in a TLS record of its own, and this record looks as follows:

```
16 03 03 00 46 10 00 00   42 41 04 f5 2f 62 a6 ed
cb dd 00 4c b5 18 19 39   40 8f 28 40 44 d0 13 28
45 3a 44 79 cc 70 a3 38   62 1d f3 14 4a eb 25 b5
80 53 07 ee ed d4 8b 00   38 3b 56 b5 fb fd 43 6f
ee 4a 78 e9 85 04 67 45   bf 7e e7
```

In this record header, 0x0046 refers to the length of the fragment (70 bytes) that comprises the CLIENTKEYEXCHANGE message. The message starts with the byte 0x10 that refers to the type of the message (16 standing for CLIENTKEYEXCHANGE) and the 3 bytes 0x000042 that refer to the remaining length of the message (66 bytes). The byte 0x41 then refers to the length of the client's public Diffie-Hellman parameter that follows, and hence the remaining 65 bytes from 0x04f5 to 0x7ee7 refer to this parameter. Using the Diffie-Hellman parameters, the ECDHE key exchange can now be performed on either side of the channel. Finally, the client finishes the handshake by sending a CHANGECIPHERSPEC message to the server. The respective TLS record looks as follows:

```
14 03 03 00 01 01
```

The record consists of a TLS header and a fragment with only one byte. In the header, 0x14 refers to the content type of the record (20 standing for CHANGECIPHERSPEC), 0x0303 refers to TLS 1.2, and 0x0001 refers to the length of the message (1). So the TLS message comprises a single byte, namely 0x01 referring to a CHANGECIPHERSPEC message.

Immediately following a CHANGECIPHERSPEC message, the client sends a FINISHED message to the server. Packaged in a TLS record, this message looks as follows:

```
16 03 03 00 28 00 00 00   00 00 00 00 00 28 0c c7
48 ac 1f 1e 5d 87 9e 67   3f 6d 05 24 bb 22 ea 59
7c e2 de 9e 8a 57 f8 ba   8e b5 77 a3 75
```

The first 5 bytes refer to the record header, of which 0x0028 refers to the length of the fragment (40 bytes). This fragment comprises the FINISHED message. Because it is the first message protected under the newly established keys, it is somehow more difficult to explain. In essence, the 40 bytes consist of the following three components:

- An 8-byte nonce that is not encrypted. Normally, AES-GCM encryption uses a 12-byte IV that consists of this nonce (that is explicit and transmitted in the clear) and a 4-byte salt (that is implicit and not transmitted). In most cases, the nonce is implemented by a counter that can either start with zero or a random number. In this particular case, a counter that starts with zero is used, and hence the nonce consists of 8 zero bytes.

- A 16-byte ciphertext from 0x280c to 24bb that refers to the FINISHED message in encrypted form. The message consists of a 4-byte header and a 12-byte data field. As usual, the header includes a message type (0x14 or 20 standing for FINISHED) and a length field (0x00000c standing for 12 bytes). The data field includes the `verify_data` field that is computed as the result of the TLS PRF over the master secret, a label,[48] and the hash value for the handshake messages exchanged so far. The entire message represents a single AES block.

- A 16-byte MAC from 0x22ea to 0xa375 that authenticates the FINISHED message.

The FINISHED message concludes the flight in which the client may send handshake messages to the server. Afterwards, it is again the server's turn. Because the client and server both provide support for the `session_ticket` extension, the server now sends a NEWSESSIONTICKET message to the client. This message is packeted in a TLS record that looks as follows:

```
16 03 03 00 da 04 00 00   d6 00 00 01 2c 00 d0 e9
83 1f 96 68 12 d0 75 eb   ba 57 75 93 ec b6 41 48
dc 14 fc 56 3a 34 8a 75   2c 39 9b 56 c8 00 5e 8b
23 fc e0 83 18 ff 0a 35   0c b1 48 75 34 d4 e3 62
d2 ec 20 a1 50 f6 e2 cc   09 dc 4b 8b 38 69 40 61
fc e1 c4 0b 36 2d 59 d1   0c 1a 8a 6b ef 78 62 1c
e1 15 04 3b 8a 18 63 11   6f b9 03 45 37 67 02 c7
95 42 0d 0b 92 f8 c5 9f   75 e9 60 8c 70 e7 c7 d3
9d c9 1a 1d 48 a8 7b cd   5b 90 51 79 36 85 59 c1
d7 40 9a f9 b0 2e 58 e2   67 f5 45 36 12 19 d3 f7
16 4a 85 58 62 03 f2 cf   09 44 02 d0 4a 89 b6 62
53 4c 08 81 90 21 57 e8   cb 76 ba 1c 54 f6 c8 b4
8d da 11 04 5b 1c b7 dc   36 df 7e 02 88 38 89 62
b7 ed 3e ef 2b aa 81 9c   5b d9 75 b2 71 7a b2
```

The TLS record header specifies a fragment length of 0x00da (218) bytes. This fragment comprises the NEWSESSIONTICKET message that starts with a byte 0x04 referring to the handshake type (4 standing for NEWSESSIONTICKET) and a 3-byte length 0x0000d6 (standing for 214 bytes). The session ticket itself has 3 fields:

- The first 4 bytes 0x0000012c refer to a session ticket lifetime of 300 (300 seconds).
- The second 2 bytes 0x00d0 refer to the length of the session ticket (208 bytes).
- The remaining 208 bytes from 0xe983 to 0x7ab2 refer to the session ticket. In our example, this is by far the largest part of the NEWSESSIONTICKET message. Note that these 208 bytes are encrypted with a key specifically generated for the encryption of session tickets.

48 If the client sends the FINISHED message, then the label is "client finished." Otherwise (i.e., if the server sends the FINISHED message), then the label is "server finished."

It is not outlined here, but the next time the client tried to resume a session, it would send this session ticket in its CLIENTHELLO message.

Finally, the server has to send a CHANGECIPHERSPEC and a subsequent FINISHED message to the client to finish the handshake. The server's CHANGECIPHERSPEC message is identical to the client's one. It looks as follows:

```
14 03 03 00 01 01
```

The FINISHED message, in turn, is very similar to the FINISHED message the client sent to the server. It looks as follows:

```
16 03 03 00 28 fd ca 2c   10 a4 7f 0a 7c 95 2f ef
79 02 f5 70 44 42 96 0a   d4 40 41 d1 6b 93 f8 2a
24 3a 35 13 42 94 0a 10   34 04 02 9d 27
```

Again, the first 5 bytes refer to the record header, of which 0x0028 refers to the fragment length (40 bytes). This fragment comprises the FINISHED message that is again protected under the newly established keys. The meaning of the fields are the same as before. So the first 8 bytes refer to the explicit nonce in the clear, the next 16 bytes refer to the encrypted message, and the remaining 16 bytes refer to a MAC for this message. Note that the nonce used by the server is different from the nonce used by the client. While the client used a counter that starts with zero, the server uses a counter that starts with a completely random number. In general, it is up to the implementation to decide whether it uses a counter and at what number it starts counting.

After the client and server have exchanged their FINISHED messages, the TLS 1.2 session is established and application data can now be transmitted in cryptographically protected form. The first 32 bytes of a TLS record that comprises application data and is sent from the client to the server looks as follows:

```
17 03 03 01 6e 00 00 00   00 00 00 00 01 98 36 6d
e3 c0 93 91 61 26 69 d3   27 e4 5b e4 6d e5 b3 40
...
```

Here, the byte 0x17 refers to the content type of the record (23 standing for application data), 0x0303 refers to TLS 1.2, 0x016e refers to the length of the message that is packeted in the record. This message is encrypted and authenticated. It starts with an 8-byte nonce that is incremented by 1 from to the nonce used previously. The last 16 bytes refer to a 16-byte MAC, and the bytes in between refer to application data encrypted with AES-GCM.

3.8 SECURITY ANALYSIS

Due to the fact that the SSL and TLS protocols have a long history, they have also been subject to many security analyses in the past (see Section 2.4 for the SSL protocol). Other results are reported in [42–45], and an informal security analysis of TLS 1.2 is provided in Section F of [3].

The most important attacks against the SSL and TLS protocols are outlined, discussed, and put into perspective in Appendix A. Some of these attacks have already been mentioned in the realm of SSL, such as the padding oracle attacks that are due to Bleichenbacher and Vaudenay, BEAST, POODLE, and RC4 NOMORE. Other attacks are more specific to TLS, such as Lucky 13 (that is a refined version of the Vaudenay attack), renegotiation attacks, including the triple handshake or 3SHAKE attack, compression-related attacks (e.g., CRIME, TIME, and BREACH), and key exchange downgrade attacks (e.g., FREAK and Logjam). The lessons learned from all of these attacks have been incorporated in the design of TLS 1.3, meaning that this version of TLS is believed to be resistant against all known attacks. This also includes some MITM attacks that target bearer tokens, and that can sometimes be mitigated by the use of token binding (Section C.2.20). The good security properties of TLS 1.3 are also due to formal security analyses that were done as part of the design process for TLS 1.3 [46–50]. But there are also two caveats to be mentioned here:

- First, a secure protocol does not exclude the existence of implementation and configuration flaws that may be exploited in future attacks. Remember the Heartbleed case that was a devastating attack (caused by a simple implementation flaw).
- Second, even a highly secure protocol version like TLS 1.3 cannot be used exclusively, meaning that in a real-world setting there are going to be multiple TLS versions used simultaneously. Unfortunately, this gives room to sometimes rather sophisticated cross-protocol attacks. A prominent example here is the DROWN attack (Section A.1.1).

Another cross-protocol attack was reported in [51]. If a server supports TLS 1.3 and TLS 1.2, then an adversary may mount a MITM attack against a client even if the client only supports TLS 1.3. The trick is to mount a Bleichenbacher attack against the server to create a valid signature, and to use this signature to authenticate to the client. This illustrates the fact that a client may be subject to attack, even though it is not even supporting a vulnerable TLS version.

Another cross-protocol attack—or rather set of attacks—that has made a lot of press headline is known as ALPACA, an acronym standing for “application layer protocols allowing cross-protocol attacks" [52].[49] The general idea is that a client sends some data within an intended protocol (e.g., an HTTP request) and that the adversary then enforces this request to be redirected to a different service (e.g., SMTP). This can be done by a MITM attack, by a CSRF, or by some other means. If the HTTP and SMTP servers are protected with TLS and share the same certificate (e.g., a wildcard certificate issued for a target domain), then it may be possible to confuse them and mount an attack. Such attacks are usually not devastating, but if no mitigation is put in place, then they still pose some risk. Mitigation is provided by some TLS extensions, like SNI (Section C.2.1) and—even more importantly—ALPN (Section C.2.15).

Similar cross-protocol attacks are likely to be found and published in the future, hence the claim that TLS 1.3 is immune to all possible attacks is certainly illusive. Needless to say that this statement applies to all protocols and is not restricted to TLS 1.3.

3.9 FINAL REMARKS

In this chapter, we have overviewed, discussed, and put into perspective all versions of the TLS protocol; that is TLS 1.0, 1.1, 1.2, and 1.3. While TLS 1.0 started as a simple protocol, TLS 1.1 and TLS 1.2 added some complexity and turned TLS into a protocol that supports many features and is therefore quite involved. This helps in making the protocol more flexible and useful in many (maybe nonstandard) situations and use cases. This includes, for example, situations in which the use of SSL/TLS is optional [i.e., the client decides on an ad hoc basis (i.e., opportunistically) whether to use SSL/TLS with a particular (nonauthenticated) server or to connect in the clear]. This practice is sometimes referred to as *opportunistic security*. According to [53], "protocol designs based on opportunistic security use encryption even when authentication is not available, and use authentication when possible, thereby removing barriers to the widespread use of encryption on the internet." If the only available alternative is no security, then opportunistic security may be a good choice. However, keep in mind that opportunistic security may also be dangerous: If people know that security is in place, then they may ignore the fact that the security is only opportunistic and behave as if the data is fully protected. This, in turn, may lead to a wrong user behavior. Opportunistic security can therefore also be seen as a dual-edged sword.

TLS 1.2 is the typical result of a standardization process: It supports almost all technologies and techniques the cryptographic community has come up with. This

49 https://alpaca-attack.com.

includes, for example, to the AES (in possibly new modes of operation, like GCM[50]), ECC, HMAC, and SHA-2. Whenever a new cryptographic technology or technique is proposed, there is strong incentive to write an RFC document that specifies how to incorporate this technology into the TLS ecosystem. (The respective RFC document can be experimental, informational, or even submitted to the internet standards track.) Examples refer to the use of the SRP protocol in a TLS extension (Section C.2.11) or the use of the Japanese and Korean block ciphers Camellia [11, 12] and ARIA [54, 55] in respective cipher suites. There is even an informational RFC [56] that argues about how to use TLS (since version 1.2) in compliance with Suite B cryptography of the NSA.[51] Due to the fact that NSA involvement is regarded as being suspicious today, this RFC is neither widely known nor used in practice. Last but not least, there have even been some proposals to add cipher suites supporting quantum cryptography to TLS. (This is not something that is recommended, but it illustrates the point that any cryptographic technology can be proposed to be used in the TLS ecosystem.)

The downside of TLS 1.2's flexibility and feature richness is that interoperability becomes an issue. In fact, the more flexible and feature-rich a protocol specification is, the more difficult it usually is to create interoperable implementations. This general rule of thumb applies to any (security) protocol, but it particularly also applies to TLS. When people specified TLS 1.3, care was therefore taken that the respective protocol is reduced to its core, and that only basic and undisputed functionalities are incorporated. The result is something that is considered to be highly secure, but also as simple as possible to enable interoperability between various implementations. Unfortunately, the existence of middleboxes acting as interception proxies has turned out to be a major obstacle for interoperability and end-to-end deployment of TLS 1.3. We therefore revisit this topic in Chapter 5.

References

[1] Dierks, T., and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, January 1999.

[2] Dierks, T., and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, April 2006.

[3] Dierks, T., and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, August 2008.

50 Note that there are also some security issues to consider when using GCM. For example, as reported in https://eprint.iacr.org/2016/475, nonce reuse poses a considerable threat for TLS, and there are some mitigation techniques that can be taken into account here.

51 Note that the Commercial National Security Algorithm (CNSA) Suite replaced Suite B in 2018, so Suite B is officially withdrawn (but [56] still prevails).

[4] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, August 2018.

[5] Krawczyk, H., M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, February 1997.

[6] Joux, A., "Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions," *Proceedings of CRYPTO 2004,* Springer-Verlag, LNCS 3152, 2004, pp. 306–316.

[7] Krawczyk, H., and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC 5869, May 2010.

[8] Oppliger, R., R. Hauser, and D. Basin, "SSL/TLS Session-Aware User Authentication," *IEEE Computer*, Vol. 41, No. 3, 2008, pp. 59–65.

[9] Dietz, M., et al., "Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web," *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12),* USENIX Association, Bellevue, WA, 2012, pp. 317–331.

[10] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)," RFC 5705, March 2010.

[11] Kato, A., M. Kanda, and S. Kanno, "Camellia Cipher Suites for TLS," RFC 5932, June 2010.

[12] Kato, A., and M. Kanda, "Addition of the Camellia Cipher Suites to Transport Layer Security (TLS)," RFC 6367, September 2011.

[13] Narten, T., and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs," RFC 2434 (BCP 26), October 1998.

[14] Medvinsky, A., and M. Hur, "Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)," RFC 2712, October 1999.

[15] Chown, P., "Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)," RFC 3268, June 2002.

[16] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions," RFC 6066, January 2011.

[17] McGrew, D., "An Interface and Algorithms for Authenticated Encryption," RFC 5116, January 2008.

[18] Dworkin, M., *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality*, NIST Special Publication 800-38C, May 2004.

[19] Dworkin, M., *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC,* NIST Special Publication 800-38D, November 2007.

[20] McGrew, D., and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)," RFC 6655, July 2012.

[21] Salowey, J., A. Choudhury, and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS," RFC 5288, August 2008.

[22] Rescorla, E., “TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM),” RFC 5289, August 2008.

[23] Eronen, P., and H. Tschofenig (eds.), “Pre-Shared Key Ciphersuites for Transport Layer Security (TLS),” RFC 4279, December 2005.

[24] Badra, M., and I. Hajjeh, “ECDHE_PSK Cipher Suites for Transport Layer Security (TLS),” RFC 5487, March 2009.

[25] Badra, M., “Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode,” RFC 5489, March 2009.

[26] Blumenthal, U., and P. Goel, “Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS),” RFC 4785, January 2007.

[27] Hollenbeck, S., “Transport Layer Security Protocol Compression Methods,” RFC 3749, May 2004.

[28] Deutsch, P., “DEFLATE Compressed Data Format Specification version 1.3,” RFC 1951, May 1996.

[29] Ziv, J., and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, IEEE, Vol. 23, No. 3, May 1977, pp. 337–343.

[30] Huffman, D.A., “A Method for the Construction of Minimum Redundancy Codes,” *Proceedings of the Institute of Radio Engineers*, Vol. 40, No. 9, September 1952, pp. 1098–1101.

[31] Moriarty, K. (Ed.), et al., “PKCS #1: RSA Cryptography Specifications Version 2.2,” RFC 8017, November 2016.

[32] ANSI X9.62, “Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA),” November 2005.

[33] FIPS PUB 186-2, “Digital Signature Standard (DSS),” 2000.

[34] Bernstein, D.J., et al., “High-Speed High-Security Signatures,” *Journal of Cryptographic Engineering*, Vol. 2, August 2012, pp. 77–89.

[35] Josefsson, S., and I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA),” RFC 8032, January 2017.

[36] ETSI TS 103 523-1, “CYBER; Middlebox Security Protocol; Part 1: MSP Framework and Template Requirements,” Version 1.1.1, December 2020.

[37] ETSI TS 103 523-2, “CYBER; Middlebox Security Protocol; Part 2: Transport Layer MSP, Profile for Fine Grained Access Control,” Version 1.1.1, February 2021.

[38] ETSI TS 103 523-3, “CYBER; Middlebox Security Protocol; Part 3: Enterprise Transport Security,” Version 1.3.1, August 2019.

[39] Langley, A., N. Modadugu, and B. Moeller, “Transport Layer Security (TLS) False Start,” RFC 7918, August 2016.

[40] Krawczyk, H., and H. Wee, “The OPTLS Protocol and TLS 1.3,” *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*, IEEE, 2016, pp. 81–96.

[41] Hodges, J., C. Jackson, and A. Barth, "HTTP Strict Transport Security," RFC 6797, November 2012.

[42] Krawczyk, H., K.G. Paterson, and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis," *Proceedings of CRYPTO 2013*, Springer-Verlag, LNCS 8042, 2013, pp. 429–448.

[43] Morrissey, P., N.P. Smart, and B. Warinschi, "The TLS Handshake Protocol: A Modular Analysis," *Journal of Cryptology*, Vol. 23, No. 2, April 2010, pp. 187–223.

[44] Sheffer, Y., R. Holz, and P. Saint-Andre, "Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)," RFC 7457, February 2015.

[45] Bhargavan, K., et al., "Verified Cryptographic Implementations for TLS," *ACM Transactions on Information and System Security (TISSEC)*, Vol. 15, No. 1, March 2012, pp. 1–32.

[46] Dowling, B., et al., "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates," *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS 2015)*, ACM, New York, NY, October 2015, pp. 1197–1210.

[47] Dowling, B., et al., "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol," *Journal of Cryptology*, Vol. 34, Springer-Verlag, 2021, pp. 1–69.

[48] Cremers, C., et al., "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication," *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE, 2016, pp. 470–485.

[49] Paterson, K.G., and T. van der Merwe, "Reactive and Proactive Standardisation of TLS," *Proceedings of Security Standardisation Research (SSR 2016)*, Springer-Verlag, LNCS 10074, 2016, pp. 160–186.

[50] Cremers, C., et al., "A Comprehensive Symbolic Analysis of TLS 1.3," *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, ACM, New York, NY, October 2017, pp. 1773–1788.

[51] Jager, D., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption," *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS 2015),* ACM Press, New York, NY, October 2015, pp. 1185–1196.

[52] Brinkmann, M., et al., "ALPACA: Application Layer Protocol Confusion - Analyzing and Mitigating Cracks in TLS Authentication," *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21),* USENIX Association, August 2021, pp. 4293–4310.

[53] Dukhovni, V., "Opportunistic Security: Some Protection Most of the Time," RFC 7435, December 2014.

[54] Lee, J., et al., "A Description of the ARIA Encryption Algorithm," RFC 5794, March 2010.

[55] Kim, W., et al., "Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)," RFC 6209, April 2011.

[56] Salter, M., and R. Housley, "Suite B Profile for Transport Layer Security (TLS)," RFC 6460, January 2012.

Chapter 4

DTLS Protocol

In this chapter, we focus on the DTLS protocol that is the UDP variant of the TLS protocol, meaning that it can be used to secure UDP-based applications and respective protocols. More specifically, we introduce the topic in Section 4.1, outline DTLS versions 1.0, 1.2, and 1.3 in Sections 4.2 to 4.4,[1] analyze the security of (the various versions of) the DTLS protocol in Section 4.5, and conclude with some final remarks in Section 4.6. Instead of explaining the DTLS protocol from scratch, we assume the reader to be familiar with the SSL/TLS protocols as outlined in Chapters 2 and 3, and we mainly address the key differences between the DTLS and the SSL/TLS protocols. This also means that this chapter does not stand for itself, but can only be understood after having read the two previous chapters. I think that this is appropriate, mainly because hardly anybody not familiar with SSL/TLS is going to study DTLS in the first place (since DTLS is still far less widely deployed in the field than SSL/TLS).

4.1 INTRODUCTION

As mentioned several times so far, the SSL/TLS protocols are stacked on top of a connection-oriented and reliable transport layer protocol, such as TCP in the case of IP networks, and hence they can be used to secure TCP-based applications only. But there is an increasingly large body of applications and respective protocols that are not based on TCP, but rather use UDP as a transport layer protocol. Examples include low-latency media streaming, real-time communications (e.g., IP telephony and videoconferencing),[2] multicast communications, online gaming, and—maybe

1 There is no DTLS version 1.1.

2 For example, UDP is heavily used in Web Real-Time Communication (WebRTC) that provides a framework for real-time communications over the internet.

most importantly—application protocols used for the Internet of Things (IoT), such as the constrained application protocol (CoAP).

In contrast to TCP, UDP only provides a best-effort datagram delivery service that is connectionless and unreliable in nature. So neither the SSL protocol nor any of the TLS protocol versions can be used to secure UDP-based applications and respective protocols (because these protocols require reliable connections to exist in the first place). The same is true for other connectionless transport layer protocols occasionally used in the field, such as the Datagram Congestion Control Protocol (DCCP) [1] or the Stream Control Transmission Protocol (SCTP) [2]. What both protocols have in common with UDP is that an application protocol layered on top of them cannot natively invoke SSL/TLS. This is a problem, and the designers and developers of such a protocol usually have three options for action:

- They can change the application protocol to make sure that it is layered on top of TCP (instead of UDP, DCCP, or SCTP). Unfortunately, this is not always possible, and there are many application protocols that perform poorly over TCP. For example, application protocols that have stringent latency and jitter requirements cannot make use of TCP's loss and congestion correction mechanisms and respective algorithms (as they introduce too much latency and jitter).
- They can use an internet layer security protocol, such as IPsec/IKE, to make sure that it is transparently invoked for all application protocols (independent from the transport layer protocol in use). Unfortunately, internet layer security protocols in general, and IPsec/IKE in particular, have several disadvantages and are often difficult to deploy and use in the field (some reasons are discussed in [3]).
- They can design the security features directly into the (new) application protocol. In this case, it no longer matters on what transport layer protocol the application is based on. Unfortunately, the design, implementation, and deployment of a new protocol is often difficult and error-prone. So it is not very likely that such an endeavor is going to be successful, at least not in the long term.

The bottom line is that all three options have severe disadvantages, and that they are not particularly useful in practice. Instead, the most desirable way to secure an application protocol would be to use SSL/TLS or a similar technology that is equally simple and also runs in application space, without requiring kernel modifications. In the past, there have been a few such proposals, like Microsoft's STLP (mentioned in Section 1.2) or the WAP Forum's wireless TLS (WTLS) protocol. Both proposals have not been successful in the field and have silently sunk into oblivion.

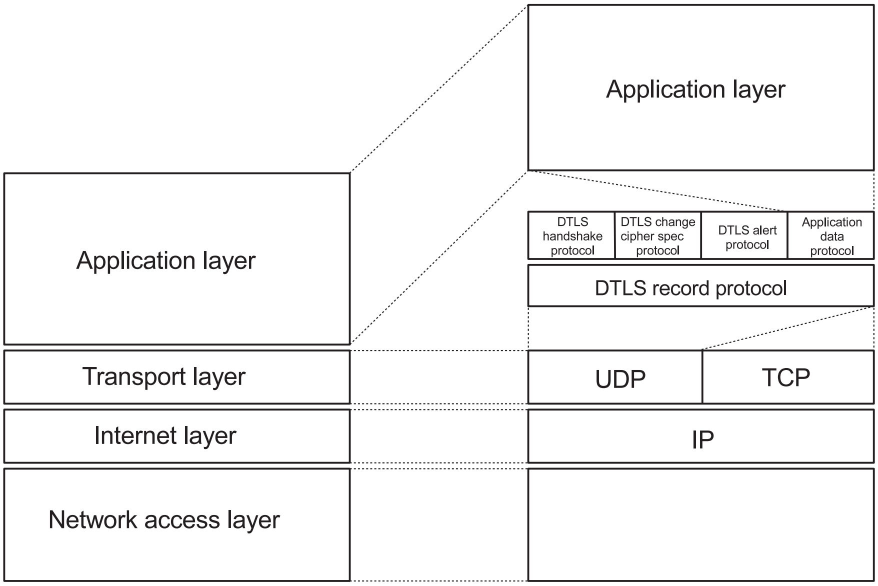

Figure 4.1 The placement of the DTLS protocol in the TCP/IP protocol stack.

When the IETF TLS WG became aware of the problem in the early 2000s, it started an activity to adapt the TLS protocol to secure UDP-based applications (i.e., to construct something like "TLS over datagram transport"). The term that was originally coined in a 2004 paper [4] for the yet-to-be-defined protocol was *datagram TLS*, and this term was finally adopted by the WG. The goal was to define a protocol that derives as little as possible from TLS but is layered on top of UDP (instead of TCP). The fact that DTLS derives as little as possible from TLS means that the placement of the DTLS protocol in the TCP/IP protocol stack is similar to the one of TLS and that the protocol is structurally identical. This is illustrated in Figure 4.1 (as compared to Figure 2.1). The only differences refer to the names and the fact that DTLS is layered on top of UDP. This also means that the DTLS protocol must be able to deal with datagrams that are not reliably transmitted, meaning that they can be delayed, reordered, lost, or even replayed (also as a consequence of routing errors). Note that the normal TLS protocol has no internal facilities to handle this type of unreliability, and hence TLS implementations would usually break when layered on top of UDP. This should be different with DTLS, and yet DTLS should be deliberately designed to be as similar to TLS as possible, both to minimize new security invention and to maximize the amount of code and infrastructure reuse.

The resulting DTLS protocol provides a solution for the security requirements of UDP-based applications, as well as DCCP- and SCTP-based applications [5, 6]. So, one can reasonably expect that many applications will make use of DTLS in the future. Either it is used alone to satisfy the security requirements, or it is used in conjunction with some complementary protocols, such as the Secure Real-time Transport Protocol (SRTP) and the Secure Real-time Transport Control Protocol (SRTCP) in the case of IP telephony.

There is no well-known UDP port for DTLS, because—similar to SSL/TLS—the port number depends on the application protocol in use. For example, Radius over DTLS typically employs UDP port number 2083, session traversal utilities for NAT (STUN) over DTLS typically employs UDP port number 5349 [7], and CoAP over DTLS typically employs UDP port number 5684. Again, a comprehensive summary is available from the IANA.[3]

So far, there have been three official versions of the DTLS protocol: DTLS version 1.0 was released in 2006 and specified in RFC 4347 [8]. It was defined as a series of deltas from TLS 1.1 (not TLS 1.0). Similarly, DTLS version 1.2 released in 2012 and (as specified in RFC 6347 [9]) was defined as a series of deltas from TLS 1.2, and DTLS version 1.3 released in 2022 and (as specified in RFC 9147 [10]) was defined as a series of deltas from TLS 1.3. Again, note that there is no version 1.1 of DTLS, and that the release of version 1.2 (immediately after version 1.0) was to bring DTLS standardization in line (and hence synchronized) with the numbering of TLS.

Independent from the facts that DTLS is layered on top of UDP and UDP only provides a connectionless transport layer service, DTLS must still use some shared state between the communicating entities. As with SSL/TLS, this state defines a security context and is established in a handshake. In the realm of IPSec/IKE, such a security context is also called *security association*, or *association* in short. As with SSL/TLS, the preferred name in DTLS is a *connection*. So the terms DTLS connection and (security) association are used synonymously and interchangeably here.

Until DTLS 1.2, a DTLS connection has always been referenced by the IP addresses and port numbers of the entities involved. In Standards Track RFC 9146 [11], however, the notion of a *connection identifier* (CID) is introduced for DTLS 1.2. In short, a CID is a logical identifier that refers to some security context (that

3 https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

is otherwise referenced by IP addresses and port numbers). To improve the privacy properties, there can be multiple CIDs to identify a single security context, and it is recommended to use a new and fresh CID whenever an entity changes its IP address or port number. There is much more to say about CIDs and their proper use in DTLS (1.2 and 1.3), and we postpone this topic to later parts of this chapter.

Most things that have been said for SSL/TLS also apply to DTLS (this is why the DTLS protocols can be defined as a series of deltas as mentioned before). Among many other things, this also includes the cipher suites registered by the IANA. There are only a few cipher suites that cannot be used for DTLS, such as the family of cipher suites that comprise the stream cipher RC4 (for the reasons discussed below). In Appendix B, the cipher suites that can be used in a DTLS setting are marked with an X.

Instead of explaining the DTLS protocol in full detail, we now take advantage of the fact that we are already familiar with the SSL/TLS protocols, and we can therefore restrict our explanations to the peculiarities of the (various versions) of the DTLS protocol and the major differences from the SSL/TLS protocols. Again, this is in line with the official DTLS protocol specifications.

From a bird's eye perspective, there are two major problem areas that are caused by the use of UDP and must therefore be resolved by DTLS in one way or another.

- First, UDP provides a connectionless best-effort datagram delivery service that operates at the transport layer. This means that—similar to IP packets—each UDP datagram is transmitted and processed independently from other datagrams, and hence that it must be possible to encrypt and decrypt each DTLS record (that is transmitted in such a UDP datagram) independently from all other records that have been encrypted and decrypted so far. This severely limits the statefulness of the cryptographic operations that can be used. Note that TLS records are usually not processed independently and that there are at least two types of interrecord dependencies.
  - In some cipher suites, a cryptographic state must be maintained between subsequent records. If, for example, a stream cipher like RC4 is used, then the key stream needs to be synchronized between the sender and the recipient, and hence the key stream index yields interrecord dependency. Similarly, if a block cipher in CBC mode and IV chaining[4] was used, then there would be an obvious interrecord dependency.

4 Remember from SSL 3.0 and TLS 1.0 that IV chaining means that the last ciphertext block of a record serves as the IV for the encryption of the next record.

  - As addressed in Section 3.2, the TLS protocol provides protection against replay and message reordering attacks by using a MAC that comprises a sequence number (that is implicit to the record). It goes without saying that the sequence number is incremented for each TLS record, so the sequence number yields another interrecord dependency.

  If DTLS is to process each record independently from all other records, then any type of interrecord dependency must be removed. This can be done by ignoring some cryptographic mechanisms or by adding explicit state to the records. If, for example, stream ciphers are not used in DTLS, then the stream cipher key state need not be synchronized in the first place. This is why RC4-based cipher suites are not considered as viable options in the DTLS protocol specifications.[5] Similarly, TLS 1.1 has explicit CBC state added to a TLS record (Section 3.3). DTLS uses the same mechanism, and an explicit sequence number is added to each record. As outlined below, this automatically leads to a new record format used in DTLS.

- Second, the unreliability of UDP also means that DTLS messages, such as handshake messages, may be delayed, reordered, lost, or replayed. It also means that such messages may need to be fragmented and reassembled for transit. The SSL/TLS protocols don't have to deal with these issues, mainly because these protocols are layered on top of TCP, and TCP provides this type of reliability. But if UDP is used (instead of TCP), then some reliability features (of TCP) must be retrofitted into DTLS. This means that message sequence numbers (in addition to per-record sequence numbers) must be added to enable fragmented messages to be reassembled and in-order delivery in case datagrams are reordered. Furthermore, due to the lack of real connections, UDP is also more susceptible to denial of service (DoS) attacks. It may therefore be useful to incorporate some techniques into DTLS that can mitigate DoS attacks, or at least make such attacks more difficult to mount.

Both problem areas have led to changes in the DTLS record protocol (for the first problem area) and the DTLS handshake protocol (for the second problem area). We first look at these changes in DTLS 1.0, before we outline a few additional changes in DTLS 1.2 and DTLS 1.3.

5 Theoretically, it would be possible to use RC4 with a per-record seed. But this is fairly inefficient, especially considering the fact that the first 512 bytes of an RC4 keystream should be discarded (because they have bad, or cryptographically weak, properties).

4.2 DTLS 1.0

While the DTLS record protocol is changed to handle the first problem area and the DTLS handshake protocol is changed to handle the second problem area, the cipher spec and application data protocols remain largely unchanged. Also, the DTLS alert protocol only slightly differs from its TLS counterpart: While MAC errors in TLS must result in connection termination, this need not be the case in DTLS. Instead, the receiving DTLS implementation should silently discard the offending record and continue with the transmission of the other DTLS records. This is possible because DTLS records are independent from each other. Only if a DTLS implementation chooses to generate an alert when it receives a message with an invalid MAC must it also generate a `bad_record_mac` alert (code 20). As outlined in Section 4.5, this subtlety has been exploited by some specific attacks. We focus on the DTLS 1.0 record and handshake protocols next.

4.2.1 Record Protocol

To deal with the first problem area mentioned above, each DTLS record must stand for itself, be cryptographically protected (so record manipulation can be detected), and be numbered in one way or another (so reordered or lost records can also be detected). To achieve this, each DTLS record comprises an explicit 8-byte sequence number (in addition to the type, version, length, and fragment fields that are also present in an SSL/TLS record). To make sure that each sequence number is distinct, it consists of two fields:

- A 2-byte *epoch* field that comprises a counter value that is incremented on every cipher state change.
- A 6-byte *sequence number* field that comprises an explicit sequence number that is incremented for every DTLS record (sent in the same epoch).

The epoch field is initially set to zero and is incremented each time after a CHANGECIPHERSPEC message is sent (so the CHANGECIPHERSPEC message is the last message sent in an epoch). The sequence number field, in turn, always refers to a specific epoch and is incremented on a per-record basis. It always begins with zero in each epoch. The resulting DTLS record numbering scheme is outlined in Figure 4.2. It starts with a CHANGECIPHERSPEC message that is wrapped in a DTLS record with an epoch value of i and a sequence number value of j. After the record is sent, a new epoch with value $i + 1$ begins. All DTLS records that are sent in this epoch are incrementally numbered starting with zero. The last record sent in epoch $i + 1$ is another CHANGECIPHERSPEC message (with a sequence number

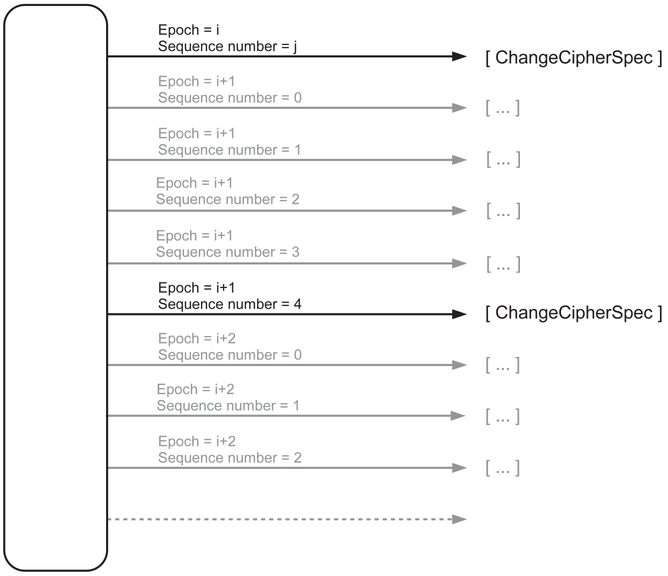

Figure 4.2 The DTLS record numbering scheme.

value of 4 in this example). This causes again the beginning of a new epoch. So the next DTLS record that is sent has an epoch value of $i + 2$ and a sequence number value of zero. The sequence number value is again incrementally numbered for all subsequent DTLS records in epoch $i + 2$. This numbering scheme continues until the end of the connection.

Taking into account the epoch and sequence number fields, the DTLS 1.0 record format is illustrated in Figure 4.3. Comparing it to Figure 2.4 reveals the fact that the two new fields are the only difference between SSL/TLS and DTLS records—at least in their native form. As explained later, this is going to change when CIDs come into play (in DTLS 1.2 and even more so in DTLS 1.3).

Remember from Section 3.2.1.1 that the HMAC construction used in TLS 1.0 employs an implicit sequence number seq_number that is not transmitted with the record. In DTLS, however, the sequence number is explicit (as explained above) and is again 8 bytes (64 bits) long. If we replace seq_number with $epoch \parallel sequence\ number$ in the formula of Section 3.2.1.1, then we get the (same) formula

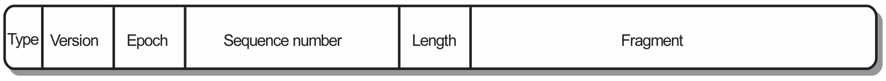

Figure 4.3 The outline of a DTLS 1.0 record.

to compute a MAC in DTLS:

$$\begin{aligned}&HMAC_K(DTLSCompressed) = \\&\quad h(K \parallel opad \parallel h(K \parallel ipad \parallel epoch \parallel sequence\,number \parallel \\&\quad type \parallel version \parallel length \parallel fragment))\end{aligned}$$

Last but not least, we note that there is a subtle difference between the TLS and DTLS record formats related to the version field: In a DTLS record this field always comprises the 1's complement of the DTLS version in use. If, for example, the DTLS version is 1.0, then the 1's complement is 254,255 (or 0xFEFF in hexadecimal notation). These bytes are included in a DTLS 1.0 record's version field. For DTLS 1.2, the 1's complement is 254,253 (or 0xFEFD in hexadecimal notation), and for DTLS 1.3, the 1's complement is 254,252 (or 0xFEFC in hexadecimal notation). So future version numbers are decreasing, whereas in fact the true version number is increasing. This is counterintuitive, and the maximal difference between TLS and DTLS version values is only to ensure that records from the two protocols can be easily separated and told apart.

4.2.2 Handshake Protocol

To deal with the second problem area mentioned above, the DTLS handshake protocol also needs to be changed a little bit (to overcome the unreliability of UDP). More specifically, the changes refer to the possibility of message fragmentation and reassembly, message retransmission, replay detection, and protection against some specific DoS attacks.

4.2.2.1 Message Fragmentation and Reassembly

Remember that the header of an SSL/TLS handshake message comprises a 3-byte length field, meaning that such a message can be up to $2^{24} - 1$ bytes long (that is then transmitted in multiple records of at most 2^{14} bytes each). Normally, handshake messages are much shorter and can be transmitted in a single record (or even multiple handshake messages in a single record). This is also true for DTLS, and hence DTLS records are usually kept smaller than the maximum transmission unit (MTU) or path MTU (PMTU)—mainly to avoid IP fragmentation. In the case of an Ethernet segment, for example, the standard MTU size is 1,500 bytes, and a larger handshake message must be transmitted in multiple DTLS records. This is where message fragmentation and reassembly come into play.

To enable message fragmentation and reassembly, each DTLS handshake message header comprises three new fields—in addition to the type and length fields that are part of an SSL/TLS handshake message header and are also used in DTLS. The new fields sum up to 8 bytes and are as follows:

- A 2-byte *message sequence* field comprises a sequence number for the handshake message that is sent. The first message each side transmits in a handshake has a value of zero, whereas every subsequent message has a message sequence value that is incremented by one. When a message is retransmitted, the same message sequence value is used. Note that this message sequence number is distinct and has nothing to do with the sequence number that is used in the DTLS record header (see above). So if a handshake message is retransmitted (in a new DTLS record), then only the sequence number of the DTLS record has a new value and the message sequence value of the message remains the same. Also, when an end system receives a handshake message, it can use the sequence number to determine whether it is the next message it expects. If it is, then it can be processed immediately; otherwise, it can be queued for later processing (i.e., once all previous messages have been received).
- A 3-byte *fragment offset* field contains a value that refers to the number of bytes (from the handshake message) that are contained in previously sent records.
- A 3-byte *fragment length* field contains a value that refers to the number of bytes (from the handshake message) that are contained in the current record.

Let us assume a handshake message i that needs to be fragmented into four DTLS records (i.e., R_1, R_2, R_3, and R_4), each of them carrying a certain number of bytes (n_j refers to the number of bytes from message i that are carried in R_j for $1 \leq j \leq 4$, so the length of the handshake message i is $n_1 + n_2 + n_3 + n_4$). Each record is sent in a distinct handshake message, where the header of each message comprises the fields itemized above (among others). So the (long) handshake message is actually sent in four (usually much shorter) messages. Table 4.1 summarizes the values of the three fields for the four messages that are sent.

Note that an unfragmented message is a degenerated case with a fragment offset of zero and a fragment length equal to the message length. Also note that the

Table 4.1
Exemplary Header Fields Used for Message Fragmentation and Reassembly

Header Field	R_1	R_2	R_3	R_4
Message sequence	i	i	i	i
Fragment offset	0	n_1	$n_1 + n_2$	$n_1 + n_2 + n_3$
Fragment length	n_1	n_2	n_3	n_4

message sequence, fragment offset, and fragment length values are included in the computation of a handshake message transcript; this is going to be different in the case of DTLS 1.3.

4.2.2.2 Message Retransmission

Due to the fact that DTLS is layered on top of UDP, UDP datagrams and respective DTLS records and handshake messages may get lost. On the record layer, sequence numbers can deal with this issue. But on the handshake protocol layer, there is no mechanism in TLS to handle packet loss (since TLS assumes a reliable connection where packet loss does not occur).[6] Consequently, DTLS must deal with this issue. The standard way to do so is to start a retransmission timer when a message is sent, and to resend the message if an acknowledgment is not received before the timer expires and a respective timeout occurs.

If, for example, a client sends a CLIENTHELLO message to the server to initiate a handshake, then the client also starts a retransmission timer. The server is to respond with either a SERVERHELLO or a HELLOVERIFYREQUEST message before the timer expires.[7] If it does not, then the client knows that something abnormal is going on (i.e., either the CLIENTHELLO message or the server response has been lost). In this case, the client resends the CLIENTHELLO message. When the server receives the retransmission, it also resends its SERVERHELLO or HELLOVERIFYREQUEST message (even if it has sent it before).

On the other side, the server also maintains a retransmission timer for the messages it sends, and it resends the latest message whenever that timer expires. Note, however, that this does not apply to the HELLOVERIFYREQUEST message (to avoid the need to create state on the server side).

6 Once the handshake has taken place and a DTLS connection is established, it is up to the application to handle packet loss.

7 Note that the HELLOVERIFYREQUEST message type is new and DTLS-specific, and that it has not been used so far. In contrast to other messages, it is not included in the MAC computation for the CERTIFICATEVERIFY and FINISHED messages. Also note that the message has been renamed to HELLORETRYREQUEST in TLS 1.3 (Section 4.4.2).

The fact that a series of subsequent handshake messages may be grouped in a flight suggests that the retransmission granularity is also at the level of a flight. As we will see in Section 4.4.2, this granularity level is refined in DTLS 1.3, where explicit ACK messages can be used to acknowledge the receipt of individual DTLS records. This means that the retransmission granularity is at the level of individual records in DTLS 1.3.

4.2.2.3 Replay Protection

In addition to the reordering and loss of DTLS records or messages, one may also be worried about records or even messages that are replayed (either because of a routing error or an attack). For this purpose, DTLS optionally provides support for replay detection. The technique used is borrowed from IPsec/IKE, where a bitmap window of received packets is maintained (for each epoch). Packets that are too old to fit in the window and packets that have previously been received are silently discarded. DTLS supports a similar replay protection technique for DTLS records, and applications can invoke it at will.

4.2.2.4 Protection Against DoS Attacks

Because the DTLS handshake protocol takes place over a datagram delivery service, it is susceptible to at least two DoS attacks (which are sometimes also known as resource clogging attacks).

- The first attack is obvious and refers to a standard resource clogging attack: The adversary initiates a handshake, and this handshake clogs some computational and communicational resources on the victim side.
- The second attack is less obvious and refers to an amplification attack: The adversary sends a CLIENTHELLO message apparently sourced by the victim to the server. The server then sends a potentially much longer CERTIFICATE message to the victim that must process the message.

To mitigate such attacks, the DTLS protocol uses a *cookie exchange* mechanism that is borrowed from another network security protocol, namely Photuris [12] that is a predecessor of the IKE protocol (version 1 and 2 [13]). Before the proper handshake begins, the server must provide a stateless cookie in a special HELLOVERIFYREQUEST message, and the client must replay this cookie in the CLIENTHELLO message to demonstrate that it is capable of receiving packets sent to the claimed IP address. A cookie should be generated in such a way that it can be verified without retaining per-client state on the server. Ideally, it is a keyed one hash value of some client-specific parameters, such as the client IP address. The key in use needs to be known only to the server, so it is not a cryptographic key that must be established in some complicated and secure way. While DTLS 1.0 supports cookies up to a size of 32 bytes only, this maximum cookie size is enlarged in DTLS 1.2 to 255 bytes.

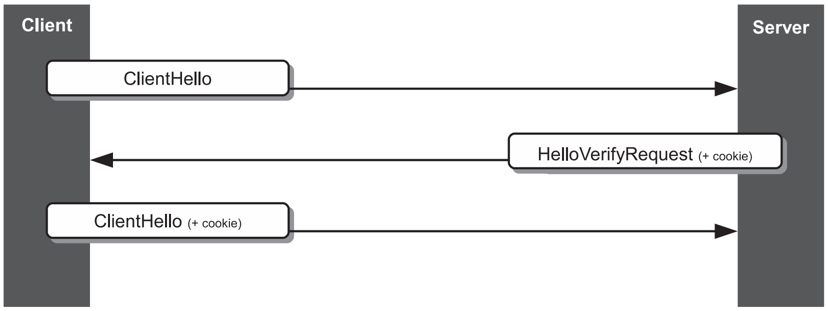

Figure 4.4 The cookie exchange mechanism used by the DTLS handshake protocol.

The cookie exchange mechanism used by the DTLS handshake protocol (up to version 1.2) is illustrated in Figure 4.4. First, the client sends a CLIENTHELLO message without a cookie to the server. The server then generates a cookie for this particular client and sends it back in the HELLOVERIFYREQUEST message. Finally, the client resends the CLIENTHELLO message, but this time the message contains the cookie just received from the server. This cookie can be verified by the server as follows:

- If the cookie is valid, then the DTLS handshake protocol can start (similar to a TLS handshake). This means that the protocol continues with a SERVER-HELLO message and all subsequent messages remain the same.
- If the cookie is invalid, then the server should treat the CLIENTHELLO message as if it did not contain a cookie in the first place.

The aim of the cookie exchange is to force a client to use an IP address, under which it can receive data (it is therefore also referred to as a *return-routability check*). Only if it receives a valid cookie (under this address) can it start a handshake. This should make DoS attacks with spoofed IP addresses difficult to mount. It does not protect against DoS attacks that are mounted from legitimate IP addresses though. So an adversary can still mount a DoS attack from a legitimate IP address, or use spoofed IP addresses for which he or she knows valid cookies. So the protection is not foolproof; it only provides a first line of defense against DoS attacks that are mounted from spoofed IP addresses [14].[8]

According to the DTLS protocol specifications, the cookie exchange is optional. It is suggested that servers be configured to perform a cookie exchange whenever a new handshake is being performed (they may choose not to do a cookie exchange when a session is resumed though), and that clients must be prepared to do a cookie exchange with every new DTLS handshake.

4.3 DTLS 1.2

DTLS 1.2 is very similar to DTLS 1.0. In particular, there is no (noticeable) change in the handshake protocol. But a major change refers to the record protocol and the possibility of using CIDs [11][9] to effectively improve the overall performance of DTLS.

To understand the rationale behind a CID, it is important to remember that DTLS is layered on top of UDP (that provides a connectionless datagram delivery service), and hence that there are no connections in place over which DTLS records can be transmitted. If an entity receives a DTLS record, then it must decide to what security context this record belongs to. This is not trivial, because the entity may have many simultaneous security contexts in place, maybe even with different peers. In the normal case, this decision is based on IP addresses and port numbers. But if the source IP address and/or port number changes during the lifetime of a security context, then the receiving entity may be unable to identify the correct security context. This is a problem, and since the use of dynamically changing IP addresses and port numbers has increased over time—mainly due to the use of network address translation (NAT)—CIDs (that remain stable and do not change over time) provide a solution and simplify the situation here. Note that a CID is conceptually similar to an SPI in the realm of IPsec/IKE.

The notion of a CID comes along with a new record format that also provides content type encryption and record-layer padding (similar to the TLS 1.3 encapsulation mechanism outline in Section 3.5). The new record format is illustrated

8 If one wants to improve protection, then one can change the key to generate cookies more frequently. After a key change, all previously issued cookies then become invalid; that is the cookies now have a defined and restricted lifetime. Note, however, that this increases key management overhead on the server side. The respective trade-off is further addressed in the IKE protocol specification [13] and not repeated here. Another possibility to improve protection (briefly mentioned in [10]) is to store timestamps in the cookie and reject cookies that are generated outside some well-defined interval of time.

9 Note that [11] was published together with the DTLS 1.3 specification in 2022.

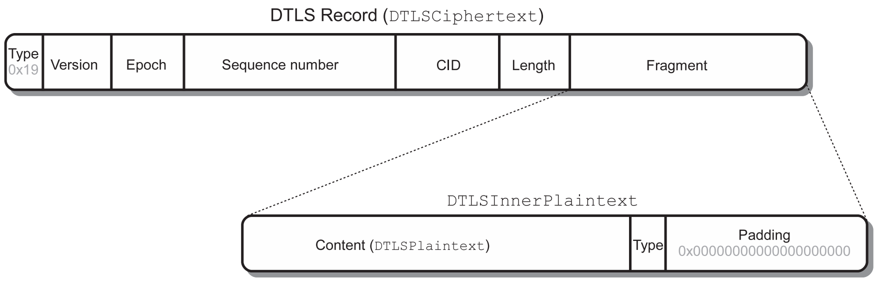

Figure 4.5 The outline of the DTLS 1.2 CID-enhanced ciphertext record format.

in Figure 4.5 (note that this format is different from the format shown in Figure 4.3). Again, the fragment of a DTLS plaintext record (i.e., `DTLSPlaintext`) is embedded in the content field of a DTLS inner plaintext record; for example, `DTLSInnerPlaintext`, that also comprises a content type and possibly some padding (that consists of a series of zero bytes). This DTLS inner plaintext record is then subject to encryption and MAC computation (according to the cipher suite in use). Ideally, an AEAD cipher is used here, but if this is not possible, then at least the `encrypt_then_mac` extension (Section C.2.19) may be invoked. In either case, the result is embedded as the fragment of a DTLS ciphertext record; that is, `DTLSCiphertext` (that employs the new format). The header of such a record comprises the usual fields, but its type is set to a new value, called `tls12_cid`,[10] that refers to 25 or 0x19 in hexadecimal notation (Table 4.2). According to this value, the recipient of the DTLS record recognizes the fact that a CID is used and that the plaintext is thus encapsulated according to the new record format. Similar to TLS 1.3, the new type value and padding that occur in the `DTLSInnerPlaintext` structure improve the privacy properties of DTLS 1.2.

The use of a CID for encrypted records can be negotiated via the DTLS-only extension `connection_id` (Section C.2.39). If this extension is present, then the respective data field must contain the CID value the sender expects the recipient to use when transmitting encrypted DTLS records. Also, a zero-length CID value indicates that the sender is prepared to transmit encrypted DTLS records using a CID but does not want the recipient to use one when sending. This is independent from whether the sender is a client or a server (i.e., it is equally true in either case).

10 The IANA has registered this content type value in its “TLS ContentType” registry. Note, however, that it is only applicable to DTLS 1.2.

4.4 DTLS 1.3

Due to the alignment with TLS 1.3, there are some fundamental changes in both the DTLS 1.3 record and handshake protocols. They are addressed next.

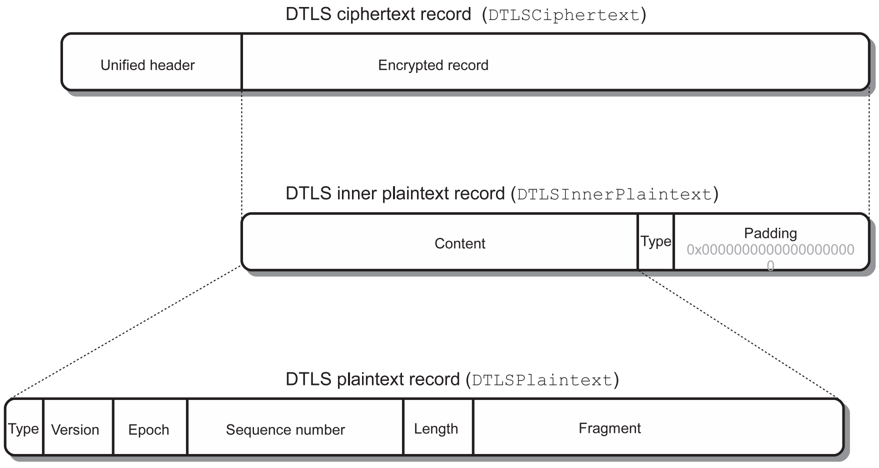
DTLS ciphertext record (DTLSCiphertext)
Unified header
Encrypted record
DTLS inner plaintext record (DTLSInnerPlaintext)
Content
Type
Padding
0x00000000000000000000
DTLS plaintext record (DTLSPlaintext)
Type
Version
Epoch
Sequence number
Length
Fragment

Figure 4.6 The outline of the DTLS 1.3 ciphertext record format.

4.4.1 Record Protocol

The DTLS 1.3 record format is optimized with a variable length encoding of the header. If a record is not encrypted or does not make use of the CID mechanism, then the normal record format (Figure 4.3) still applies to DTLS 1.3. In all other cases, a new ciphertext record format (with a slightly different CID mechanism than the one from DTLS 1.2) is used in DTLS 1.3. This format is illustrated in Figure 4.6. Here, a DTLS ciphertext record (i.e., `DTLSCiphertext`) consists of a variable-length header, called a *unified header*, and an encrypted part, called an *encrypted record*. As its name suggests, the encrypted record comprises a DTLS inner plaintext structure (i.e., `DTLSInnerPlaintext`) that is cryptographically protected according to the cipher suite (this construction provides content type encryption and record-layer padding in a way that is similar to DTLS 1.2). Finally, the content field of the DTLS inner plaintext structure embodies the DTLS plaintext structure (i.e., `DTLSPlaintext`) including the application-layer data in the fragment field.

In contrast to DTLS 1.0 and 1.2, DTLS 1.3 uses epoch and sequence number fields that are 8 bytes long each. This means that the concatenation of the two fields is 16 bytes or 128 bits long. On the one hand, the resulting 128-bit value can be used in an ACK message to identify a particular DTLS record (as discussed below). On the other hand, it is also input to an AEAD cipher (in a rather involved way that is not further addressed here). As discussed below, DTLS 1.3 assigns distinct epoch values to the messages of the handshake protocol. Also, DTLS records must not be sent if the sequence number exceeds $2^{48} - 1$, and hence the two most significant bytes of the sequence number must always be set to 0x0000.

In the new DTLS 1.3 ciphertext record format, the CID is part of the unified header (if present), but most other header fields appear in the header of the plaintext record (that is embedded in the inner plaintext record and encrypted in the ciphertext record). According to Figure 4.6, these fields include:

- A 1-byte type field that identifies the content type of the DTLS record (Table 4.2).
- A 2-byte version field that is set to `legacy_record_version` (standing for 254,253 and hence DTLS 1.2) for all records other than the one that comprises the initial CLIENTHELLO message. For compatibility reasons, this record's version field is set to 254,255 (standing for DTLS 1.0). As with TLS 1.3, the actual version negotiation takes place through the `supported_versions` extension, so the `legacy_record_version` value in the version field can safely be ignored by the implementation.
- A 2-byte epoch field that refers to the two least significant bytes of the full (8-byte) epoch value of the DTLS record.
- A 6-byte sequence number field that refers to the six least significant bytes of the full (8-byte) sequence number value of the DTLS record.
- A 2-byte length field that specifies the length of the fragment.

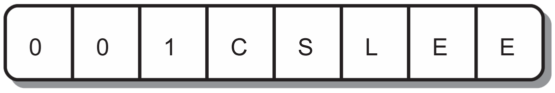

Figure 4.7 The bit mask of the unified header's first byte.

The unified header that is (often) used in DTLS 1.3 has a variable length and uses a distinct format: It starts with a byte that represents a bit mask and is illustrated in Figure 4.7. From left to right, it comprises a fixed 3-bit sequence 001, a C-bit that is set only if a CID is used, an S-bit that indicates the length of the sequence number (where 0 is standing for 8 bits and 1 is standing for 16 bits), an L-bit that is set only if the length is present in the unified header, and two E-bits that comprise the two least significant bits of the epoch value.

After the first byte (that represents the bit mask), the unified header comprises one or several fields, depending on the bits set in the mask. If the C-bit is set, then the header comprises a variable-length CID. Note that the CID is optional, meaning that the encapsulation of a DTLS record and a unified header may be used even if the CID mechanism is not used at all. After the CID field (that may also be empty), the unified header comprises the low-order 8 or 16 bits from the sequence number of the record. The value is 16 bits long if the S-bit is set (to 1), whereas it is only 8 bits long if the S-bit is not set (or set to 0, respectively). Finally, the unified header may also comprise a 16-bit length field. Again, the existence of this field is triggered by the L-bit, and this means the length field can be omitted by clearing the L-bit (or setting it to 0, respectively). In this case, the record consumes the entire rest of the UDP datagram (in which the record is transported),[11] and it is thus not possible to have multiple DTLS records without length fields in the same UDP datagram.[12] Following this line of argumentation, there are two border cases:

- If the bit mask is 001000EE (with EE referring to the low-order two bits of the epoch), then there is only an 8-bit sequence number, before the encrypted record is appended. This refers to a minimum-length header.
- If the bit mask is 001111EE, then there is variable-length CID, a 16-bit sequence number, and a 16-bit length before the encrypted record is appended. This refers to a maximum-length header.

In DTLS 1.0 and 1.2, the content type of a record always appears in the first byte of the header. This makes it easy to interpret and process an incoming record. In DTLS 1.3, the first byte still determines how a record must be interpreted and processed, but the rules to do so are more involved. Note that a record may not be encrypted, in which case the normal record format according to Figure 4.3 applies. In this case, the content type is provided in the first byte of the header and may refer to one of the values itemized in Table 4.2.

- Value 20 suggests that the record comprises a CHANGECIPHERSPEC message that only occurs in DTLS 1.0 or 1.2.

11 The same argument applies if another transport layer protocol than UDP is used.

12 If multiple DTLS records are packaged in the same UDP datagram, then omitting the length field must only be used for the last record.

Table 4.2
Content Type Values for DTLS Records

Value	Meaning
20	CHANGECIPHERSPEC message (DTLS 1.0 and 1.2)
21	Alert message (plaintext)
22	Handshake message (plaintext)
23	Application data (DTLS 1.0 and 1.2)
24	Heartbeat message (DTLS 1.0 and 1.2)
25	DTLS ciphertext with CID (DTLS 1.2)
26	ACK message (plaintext, DTLS 1.3)

- Values 21, 22, or 26 suggest that the record comprises an alert, handshake, or ACK message that is not encrypted (otherwise the message would be sent in a record that uses the new format). Note that an ACK message can only occur in DTLS 1.3.
- Values 23 or 24 suggest that the record comprises application data or a heartbeat message in DTLS 1.0 or 1.2.
- Value 25 suggests that the record comprises a DTLS 1.2 CID-enhanced ciphertext record according to Figure 4.5 in Section 4.3.

If the new record format applies, then the first byte of the unified header of the DTLS ciphertext record begins with 001.[13] After decryption, the content type suggests whether the record refers to an alert message (21), a handshake message (22), application data (23), a heartbeat message (24), or an ACK message (26). If none of these values occur, then an error must result. If neither the first byte matches a value from Table 4.2 nor does the first byte begin with 001, then the record must be rejected.

In DTLS 1.3, the message transcript is computed over all (complete and reassembled if necessary) handshake messages. This requires removing the message sequence, fragment offset, and fragment length fields from the message headers. Note that this is different from the previous versions of DTLS, where all fields are considered when computing the message transcript. Also note that DTLS 1.3 yet uses the same fields to compute the message transcript as TLS 1.3, but that the actual transcripts are disjoint (because they use different version numbers and labels in the key derivation processes).

13 To avoid any ambiguity with DTLS content types, the range between 32 (00100000 in binary notation) and 63 (00111111) must be excluded from future allocations by the IANA.

4.4.2 Handshake Protocol

In contrast to its predecessors, the DTLS 1.3 handshake protocol is aligned with the TLS 1.3 handshake protocol and reuses the same message formats, flows, and logic (Section 3.5.1).[14] This applies to the following messages (with type values provided in brackets):

- CLIENTHELLO (1)
- SERVERHELLO (2)
- NEWSESSIONTICKET* (4)
- ENDOFEARLYDATA (5)[15]
- ENCRYPTEDEXTENSIONS (8)
- CERTIFICATE (11)
- CERTIFICATEREQUEST (13)
- CERTIFICATEVERIFY (15)
- FINISHED (20)
- KEYUPDATE* (24)

Furthermore, a few messages have been removed in DTLS 1.3 (as compared to DTLS 1.2); for example, the SERVERHELLODONE, CHANGECIPHERSPEC,[16] SERVERKEYEXCHANGE, and CLIENTKEYEXCHANGE messages. While the SERVERHELLODONE and CHANGECIPHERSPEC messages have been removed without any replacement, the contents of the SERVERKEYEXCHANGE and CLIENTKEYEXCHANGE messages have moved to extensions.

Last but not least, there are also two new messages related to CIDs that have been introduced in DTLS 1.3:

- REQUESTCONNECTIONID* (9)
- NEWCONNECTIONID* (10)

14 There are few exceptions here. For example, DTLS 1.3 implementations do not use the TLS 1.3 compatibility mode, session IDs are not supported anymore (and hence the respective field is named `legacy_session_id`), and CHANGECIPHERSPEC messages must not be sent at all.

15 Because DTLS records have epochs, ENDOFEARLYDATA messages are not really required to determine the end of early data. This means that ENDOFEARLYDATA messages are somehow obsolete in a DTLS setting.

16 Note that this message is not a handshake message, but is rather a single message from the subprotocol of the same name.

If a client and server have negotiated the use of CIDs with the `connection_id` extension (Section C.2.39), then either side can request a CID (using the REQUESTCONNECTIONID* message) or indicate a set of CIDs the sender wishes its peer to use (using the NEWCONNECTIONID* message). So if an entity received a REQUESTCONNECTIONID* message, then it would immediately respond with a NEWCONNECTIONID* message. Note that all messages marked with a superscript star (*) refer to post-handshake messages, meaning that these messages are sent and received after the handshake is complete.

As mentioned before, DTLS 1.3 assigns dedicated epoch values to messages in the handshake protocol to identify the correct cipher state (refer to Section 3.5.2 for the various keys that are used in the key schedule).

- Epoch value 0 is used for all unencrypted messages. This applies to the CLIENTHELLO, SERVERHELLO, and HELLORETRYREQUEST messages.
- Epoch value 1 is used for all messages that are protected with keys derived from `client_early_traffic_secret`. Note that this epoch value is skipped if the client doesn't send early data.
- Epoch value 2 is used for all messages that are protected with keys derived from `client_handshake_traffic_secret` (if the sender is the client) or `server_handshake_traffic_secret` (if the sender is the server). This applies to many handshake messages, including the ENCRYPTEDEXTENSIONS, CERTIFICATEREQUEST, CERTIFICATE, CERTIFICATEVERIFY, and FINISHED messages. It does not apply to post-handshake messages that are protected under the appropriate application traffic key.
- Epoch value 3 is used for payloads that are protected with keys derived from `client_application_traffic_secret_0` (if the sender is the client) or `server_application_traffic_secret_0` (if the sender is the server). This may include post-handshake messages, such as a NEWSESSIONTICKET message.
- Epoch values 4 to $2^{64} - 1$ are used for all payloads that are protected with keys derived from `client_application_traffic_secret_N` (if the sender is the client) or `server_application_traffic_secret_N` (if the sender is the server) for all `N` > 0.

Using these reserved epoch values, it is clear for a receiving entity what cipher state has been used to cryptographically protect a message. The same state must be used to deprotect it. Note in this context that epoch values never wrap (i.e., instead of wrapping, an entity must always terminate the connection and restart from scratch).

As explained in Section 4.2.2.2, DTLS 1.0 (as well as DTLS 1.2) uses a timer to detect lost messages and start a message retransmission (if the timer expires). All messages in a flight are treated equally, meaning that either none or all messages of a flight are retransmitted. This is different in DTLS 1.3, where a new content type for ACK messages is introduced (Table 4.2). An ACK message allows an entity to acknowledge individual records. It therefore comprises a list of record numbers (typically from the same flight) that the entity has received and either processed or buffered, in numerically increasing order. If a handshake protocol flight has an expected response, such as the flight from epoch 0/1 or 2 (as explained above), then no explicit acknowledgement is needed, because the response implicitly acknowledges the receipt. But if the flight doesn't have an expected response, then an ACK message may be used. Examples include the client's final handshake flight and some post-handshake messages, such as a NEWSESSIONTICKET message provided by the server. In either case, an ACK message is sent by the peer entity to signal proper receipt. Note that an ACK message itself is not a handshake message, but rather a message with a separate content type 26 (instead of 22 as for handshake messages). This avoids having ACK messages being added to the handshake protocol transcript. Also note that ACK messages can still be transmitted in the same UDP datagram as handshake messages though.

Finally, we note that the DTLS 1.3 protocol reuses the HELLORETRYREQUEST message from TLS 1.3 (Section 3.5.1) to replace the HELLOVERIFYREQUEST message from DTLS 1.0 and 1.2. Remember that the HELLORETRYREQUEST has the same format as a SERVERHELLO message, but for convenience the term HELLORETRYREQUEST message is still used (in [10] and here) as if it were a distinct message. Also, it is used for message retransmission (Section 4.2.2.2) and the cookie exchange (Section 4.2.2.4) in lieu of a HELLOVERIFYREQUEST message. Figure 4.4 therefore still applies with HELLOVERIFYREQUEST (+ cookie) being replaced with HELLORETRYREQUEST (+ cookie). But while a HELLOVERIFYREQUEST message is a special message type that can carry a cookie in one of its fields, the HELLORETRYREQUEST message doesn't have such a field. Here, the cookie must be carried in a so-called `cookie` extension (Section C.2.32) originally defined for TLS 1.3 but mainly used for DTLS 1.3. Referring to Figure 4.4, the client first sends a CLIENTHELLO message without a `cookie` extension to the server. The server then generates a cookie for this particular client and sends it back to the client in the HELLOVERIFYREQUEST message. Because this message is essentially a SERVERHELLO message, it can make use of extensions and include the generated cookie in its `cookie` extension. Finally, the client resends the CLIENTHELLO message, but this time the message also contains the cookie just received from the server in the respective extension. This way, the cookie exchange mechanism can be retrofitted into the DTLS 1.3 handshake protocol, even though HELLOVERIFYREQUEST messages are no longer in use.

4.5 SECURITY ANALYSIS

Given the fact that the DTLS protocol was designed to be as similar as possible to the TLS protocol, one can reasonably expect that the security analyses of TLS, at least in principle, still apply to DTLS, and hence that—at least as far as version 1.3 is concerned—"the security guarantees for DTLS 1.3 are the same as TLS 1.3" [10]. Also, taking into account that TLS/DTLS 1.3 only employs state of the art cryptography, such as AEAD ciphers and no static key exchange, people generally have a comfortable gut feeling about the security of DTLS 1.3. It is, however, just a gut feeling, and fairly little is known about the real security of the protocol.

From all attacks that can be mounted against the SSL/TLS protocols (Appendix A), there are some attacks that work against DTLS, and there are some attacks that don't work. For example, compression-related attacks and padding oracle attacks generally work against DTLS, whereas attacks against RC4 don't work—simply because RC4 is not a supported encryption algorithm for DTLS.

With regard to padding oracle attacks, the protection mechanisms that have been built into TLS are also present in DTLS, and hence one would expect DTLS to be resistant against such attacks, too. Unfortunately, this is not always the case, and there are some padding oracle attacks that can still be used to recover plaintext. For example, in 2012, some padding oracle attacks against two DTLS 1.2 implementations that are in widespread use (i.e., OpenSSL and GnuTLS) were demonstrated [15].[17] To understand the attacks it is important to remember that a DTLS record with invalid padding is silently discarded (without MAC verification) and that no alert message is generated and sent back to the sender. This means that an adversary can exploit neither the type of alert message nor its timing behavior, and this, in turn, suggests that the padding oracle attacks that have been mounted against SSL/TLS do not work against DTLS. But in [15], it was shown that the lack of alert messages can be compensated with Heartbeat messages. As argued in Section C.2.14, Heartbeat is a TLS (and DTLS) extension that makes a lot of sense in a DTLS setting. So instead of sending attack messages and waiting for the respective alert message (to evaluate its type or timing), the adversary can send a Heartbeat request message immediately following the attack message. The time it takes until he or she receives the response message leaks some information about the amount of computation that has been done intermediately on the server side.

17 The attacks can decrypt arbitrary amounts of ciphertext in the case of OpenSSL and the four most significant bits of the last byte in every block in the case of GnuTLS.

If the padding is invalid, then the server has comparably little to do (in particular, because MAC verification is not performed in this case). But if the padding is valid, then the record is decrypted and the MAC is verified. This takes some time that is noticeable in at least some cases. The timing difference can even be amplified by sending multiple identical records in a sequence (instead of a single message).[18] In this case, the amount of work that has to be done by the server is multiplied by the number of identical records. This makes the attack more feasible. Furthermore, the fact that an invalid padding does not terminate a connection in the case of DTLS further simplifies the attack. Remember that in the SSL/TLS case, the adversary has to work with a huge quantity of simultaneous connections that can be used only once. This is not true for DTLS. Here, a single connection can be used to send as many DTLS records with invalid padding as needed. This simplifies padding oracle attacks considerably. Luckily, such attacks are no longer an issue in DTLS 1.3, because the use of an AEAD cipher is strictly enforced (so no block cipher in CBC mode is supported anymore).

Maybe the biggest worry with regard to the security of the DTLS protocol—and the one that deserves further study—is related to the fact that DTLS is stacked on top of UDP instead of TCP. There may be entirely new attacks that take advantage of this fact. In network security, it is well known that UDP-based applications are more difficult to secure than TCP-based ones, so it might be the case that similar arguments also apply to DTLS. With the future deployment of DTLS, it is possible and likely that research will more thoroughly address this question, and it will be interesting to learn the results. In the meantime, we have to live with the comfortable gut feeling (mentioned above) and assume that the DTLS protocol provides a reasonable level of security.

4.6 FINAL REMARKS

This chapter elaborated on the DTLS protocol that represents the UDP variant of the SSL/TLS protocols. The differences are minor and are mainly due to the properties of UDP being a connectionless best-effort datagram delivery protocol that operates at the transport layer. This requires some changes in the DTLS record and handshake protocols. The changes are so minor that it is often argued that the security of DTLS is comparable to the security of SSL/TLS. However, according to Section 4.5, this argument should be taken with a grain of salt. It may be the case that some entirely new weaknesses or vulnerabilities will be found and revealed in the future. This is always possible, but in the case of the DTLS protocol it is more likely, simply

18 In [15], the term *train of packets* is used to refer to multiple records.

because the security of the DTLS protocol has not been sufficiently scrutinized so far.

Compared to SSL/TLS, DTLS is still a relatively new protocol that is not yet widely deployed. There is an increasingly large number of DTLS implementations available on the market, but these implementations are also relatively new and have not been thoroughly analyzed and tested so far. There is a comprehensive analysis of some DTLS implementations using protocol state fuzzing that has found a few serious vulnerabilities [16], but more analyses are needed.

The lack of implementation experience goes hand in hand with the fact that there are only a few studies about the optimal deployment of DTLS. As DTLS allows finer control of timers and record sizes, it is worthwhile doing additional analyses here, for example, to determine the optimal values and backoff strategies. This is another research area that deserves further study (in addition to security). The same is true for the firewall traversal of the DTLS protocol. As we will see in the following chapter, many firewall technologies are well-suited for TCP-based applications and application-layer protocols, but they are less well-suited for UDP-based applications and application-layer protocols. Consequently, the secure firewall traversal of DTLS is yet another topic researchers may care about and look into.

References

[1] Kohler, E., M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," RFC 4340, March 2006.

[2] Stewart, R. (ed.), "Stream Control Transmission Protocol," RFC 4960, September 2007.

[3] Bellovin, S., "Guidelines for Specifying the Use of IPsec Version 2," RFC 5406 (BCP 146), February 2009.

[4] Modadugu, N., and E. Rescorla, "The Design and Implementation of Datagram TLS," *Proceedings of the Network and Distributed System Security Symposium (NDSS),* Internet Society, 2004.

[5] Phelan, T., "Datagram Transport Layer Security (DTLS) over the Datagram Congestion Control Protocol (DCCP)," RFC 5238, May 2008.

[6] Tuexen, M., R. Seggelmann, and E. Rescorla, "Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)," RFC 6083, January 2011.

[7] Petit-Huguenin, M., and G. Salgueiro, "Datagram Transport Layer Security (DTLS) as Transport for Session Traversal Utilities for NAT (STUN)," RFC 7350, August 2014.

[8] Rescorla, E., and N. Modadugu, "Datagram Transport Layer Security," RFC 4347, April 2006.

[9] Rescorla, E., and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347, January 2012.

[10] Rescorla, E., H. Tschofenig, and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3," RFC 9147, April 2022.

[11] Rescorla, E., et al., “Connection Identifier for DTLS 1.2,” RFC 9146, March 2022.

[12] Karn, P., and W. Simpson, “Photuris: Session-Key Management Protocol,” RFC 2522, March 1999.

[13] Kaufman, C., et al., “Internet Key Exchange Protocol Version 2 (IKEv2),” RFC 7296, October 2014.

[14] Oppliger, R., “Protecting Key Exchange and Management Protocols Against Resource Clogging Attacks,” *Proceedings of the IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security,* Kluwer Academic Publishers, 1999, pp. 163–175.

[15] AlFardan, N.J., and K.G. Paterson, “Plaintext-Recovery Attacks against Datagram TLS,” *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012),* February 2012.

[16] Fiterau-Brostean, P, et al., “Analysis of DTLS Implementations Using Protocol State Fuzzing,” *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020),* USENIX Association, 2020, pp. 2523–2540

Chapter 5

Firewall Traversal

In spite of the fact that firewalls are omnipresent today, their use and interplay with the SSL/TLS and DTLS protocols remains tricky and somehow contradictory. On the one hand, the SSL/TLS protocols are used to provide end-to-end security services and hence to enable end-to-end connectivity. On the other hand, firewalls are used to restrict or at least better control this end-to-end connectivity. It is, therefore, not obvious if and how the SSL/TLS protocols can effectively traverse firewalls. Taking HTTP over SSL/TLS (i.e., HTTPS) as an example, this is what this chapter is all about. It provides an introduction in Section 5.1, elaborates on SSL/TLS tunneling and proxying in Sections 5.2 and 5.3, discusses middlebox mitigation in Section 5.4, and concludes with some final remarks in Section 5.5. Note that a lot has been done to leverage the SSL/TLS protocols in proxy-based firewalls, not only for HTTP but also for many other protocols. So this chapter is not meant to be comprehensive and only explores the tip of the iceberg. There are so many use cases and deployment scenarios that it is literally impossible to address them all.

5.1 INTRODUCTION

There are many possibilities to define the term *internet firewall*, or *firewall* in short. According to RFC 4949 [1], for example, a firewall refers to “an internetwork gateway that restricts data communication traffic to and from one of the connected networks (the one said to be ‘inside’ the firewall) and thus protects that network’s system resources against threats from the other network (the one that is said to be ‘outside’ the firewall).” This definition is fairly broad and not precise in mathematical terms.

In the early days of the firewall technology, William R. Cheswick and Steven M. Bellovin defined a firewall (system) as a collection of components placed between two networks that collectively have the following three properties [2]:

1. All traffic from inside to outside, and vice versa, must pass through the firewall.
2. Only authorized traffic, as defined by the local security policy, is allowed to pass through.
3. The firewall itself is immune to penetration.

Note that these properties are design goals, meaning that a failure in one aspect does not necessarily mean that the collection is not a firewall, simply that it is not a good one. Consequently, there are different levels of security a firewall may achieve. As indicated in property 2 (with the notion of "authorized traffic"), there must be a security policy in place that specifies what traffic is authorized for the firewall, and this policy must be strictly enforced. In fact, it turns out that the specification of a security policy is key to the successful deployment of a firewall, or, alternatively speaking, any firewall without an explicitly specified and strictly enforced security policy is pointless in the field (because it gets more and more porous over time).

There are many technologies that can be used (and combined) to implement a firewall. They range from *static* and *dynamic*[1] *packet filtering* to *proxies*—or *gateways*—that operate at the transport or application layer. In some literature, the former are called *circuit-level gateways*, whereas the latter are called *application-level gateways* [2]. Also, there are many possibilities to combine these technologies in real-world firewall configurations, and to operate them in some centralized or decentralized way. In fact, there are increasingly many firewalls—so-called *personal firewalls*—that are operated in a decentralized way, typically at the desktop level. Most modern desktop operating systems, including Microsoft Windows and Linux, provide some built-in (personal) firewall functionality.

For the purpose of this book, we don't delve into the design and deployment of a typical firewall configuration. There are many books that elaborate on these issues (e.g., [3–5]). Instead, we assume a firewall to exist, and we further assume that this firewall at least comprises an HTTP proxy server. If a firewall did not comprise an HTTP proxy server, then it would be condemned to use only packet filters and/or circuit-level gateways to mediate HTTP data traffic. Such a firewall is

1 Dynamic packet filtering is also known as *stateful inspection*.

not very effective with regard to the security it is able to provide (at least not with regard to HTTP security).[2]

If an HTTP proxy server is in place and a client (or browser) wants to use HTTP to connect to an origin web server, then the corresponding HTTP request is delivered to the HTTP proxy server and forwarded from there. The HTTP proxy server acts as a mediator for the HTTP connection, meaning that the client and server talk to the proxy server, while they both think that they are talking directly to each other. Hence, the HTTP proxy server represents (and can be seen as) a legitimate MITM. In some situations, this is acceptable (or even desirable), but in some other situations it is not. It is particularly worrisome if the user is not aware of the proxy server's existence—a situation that frequently occurs today.

In general, different application protocols may have different requirements with regard to proxy servers, and on a high level of abstraction an application protocol can either be proxied or tunneled through a proxy server:

- When we say that an application protocol is *proxied*, we mean that the corresponding proxy server is aware of the specifics of the protocol and can understand what is going on at the protocol level. This allows such things as protocol-level filtering (including, for example, protocol header anomaly detection), access control, accounting, and logging. Examples of protocols that are usually proxied include Telnet, FTP, SMTP, and—maybe most importantly here—HTTP.
- Contrary to that, we say that an application protocol is *tunneled* when we mean that the corresponding proxy server (which basically acts as a circuit-level gateway) is not aware of the specifics of the protocol and cannot understand what is going on at the protocol level. It is simply relaying—or tunneling—data traffic between the client and server, and it does not necessarily understand the protocol in use. Consequently, it cannot perform such things as protocol-level filtering, access control, accounting, and logging to the same extent as it is possible for a full-fledged proxy server. Examples of protocols that are usually tunneled include proprietary protocols, protocols for which a proxy server is not available, and protocols that provide support for end-to-end encryption (E2EE).

With regard to the SSL/TLS protocols, the two possibilities itemized above are illustrated in Figure 5.1. While (a) shows SSL/TLS tunneling, (b) shows SSL/TLS

2 For the sake of completeness, we note that sometimes people use the term *firewall* to refer to a packet filter and that they then use the terms *proxy* and *reverse proxy* to refer to our notion of a firewall. The terminology we use in this book is inherently more general, and we use the term *firewall* in a largely technology-independent way.

proxying. In either case, there is a proxy server in the middle that tunnels or proxies the SSL/TLS connection. Note that in a practical setting, there would be two proxy servers (i.e., a client-side proxy server and a server-side proxy server). The client-side proxy server would be operated by the organization of the client, whereas the server-side proxy server would be operated by the organization of the server. Each proxy server runs independently and can proxy or tunnel the SSL/TLS protocol at will.

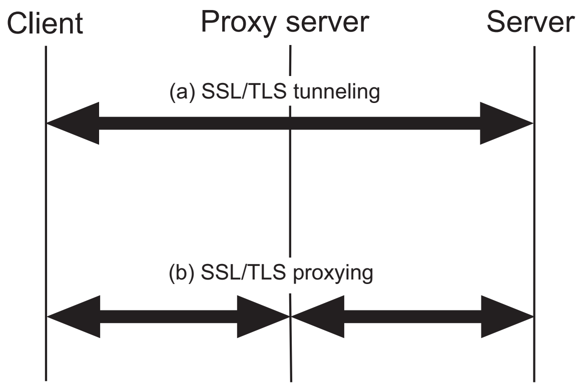

Figure 5.1 Two possibilities of a proxy server.

The important difference is that there is only one SSL/TLS connection from the client to the (origin) server in the case of SSL/TLS tunneling, whereas there are two SSL/TLS connections[3]—one from the client to the proxy server and another from the proxy server to the origin server—in the case of SSL/TLS proxying. The proxy server then pretends to be the client while communicating with the origin server, and in turn impersonates the origin server in its communication with the client. In the case of multiple proxy servers, they form a sequence between the client and the origin server, with each proxy server playing both the role of a client and a server during each round trip. In SSL/TLS tunneling, the proxy server is passive in the sense that it yet provides connectivity, but it does not interfere with the data transmission. Contrary to that, in SSL/TLS proxying the proxy server is active and able to fully control and interfere with the data transmission. Needless to say, this has some severe security and privacy implications.

In the past, it has been common practice for companies and organizations to tunnel outbound SSL/TLS connections and proxy inbound SSL/TLS connections.

3 In some literature, the two SSL/TLS connections that derive from the SSL/TLS connection that is split by the proxy server are called *segments*. This term, however, is not used in this book.

This practice, however, is about to change as the deployment settings are getting more involved, and content screening and data loss prevention (DLP) are often required to be implemented, as well. Let us now more thoroughly address SSL/TLS tunneling and SSL/TLS proxying for HTTPS.

5.2 SSL/TLS TUNNELING

In an early attempt to address the problem of having SSL or HTTPS traffic traverse a proxy-based firewall, Ari Luotonen from Netscape Communications proposed a simple mechanism that allowed an HTTP proxy server to act as a tunnel for SSL-enhanced protocols [6]. The mechanism was named *SSL tunneling*, and it was effectively specified in a series of Internet-Drafts. Today, the mechanism—or rather the HTTP CONNECT method for establishing end-to-end tunnels across HTTP proxy servers—is part of the HTTP/1.1 specification [7]. An additional specification is therefore no longer required (and the respective Internet-Drafts are now obsolete).

In short, SSL/TLS tunneling allows a client to open a secure tunnel through an HTTP proxy server that resides on a firewall.[4] When tunneling SSL/TLS, the HTTP proxy server must not have access to the data being transferred in either direction. Instead, the HTTP proxy server only needs access to the source and destination IP addresses and port numbers to set up an appropriate SSL/TLS connection that represents a tunnel. Consequently, there is a handshake between the client and the HTTP proxy server to establish the connection between the client and the origin server through the intermediate proxy server. To make SSL/TLS tunneling be backward-compatible, the handshake must be in the same format as normal HTTP requests, so that proxy servers without support for this feature can still determine the request as impossible to serve and to provide an error notification. As such, SSL/TLS tunneling is not really SSL/TLS-specific. Instead, it is a general mechanism to have an intermediary establish a connection between two endpoints, after which bytes are simply copied back and forth by the intermediary.

In the case of HTTP, SSL/TLS tunneling uses the HTTP CONNECT method to have the HTTP proxy server connect to the origin server. To invoke the method, the client must specify the host name and port number of the origin server (separated with a colon), followed by a space, a string specifying the HTTP version number (e.g., HTTP/1.0), and a line terminator. Afterward, a series of zero or more HTTP request header lines and an empty line may follow. Hence, the first line of a fictitious HTTP CONNECT request message may look as follows:

```
CONNECT www.esecurity.ch:443 HTTP/1.0
```

4 In some literature, such an HTTP proxy server is also called non-TLS-terminating. Note, however, that it does not even have to be aware and knowledgeable about TLS.

This example requires an SSL/TLS-enabled web server running at port 443 of `www.esecurity.ch`. The message is sent by the client, and it is received by the HTTP proxy server. The proxy server, in turn, tries to establish a TCP connection to port 443 of `www.esecurity.ch`. If the server accepts the TCP connection, then the HTTP proxy server starts acting as a relay between the client and the server. This means that it copies back and forth data sent through the connection. It is then up to the client and the server to perform an SSL/TLS handshake to establish a secure connection between them. This handshake is opaque to the HTTP proxy server, meaning that the proxy server need not be aware of the fact that the client and the server perform a handshake.

SSL/TLS tunneling can also be combined with the normal authentication and authorization mechanisms employed by an HTTP proxy server [8]. For example, if a client invokes the HTTP CONNECT method but the proxy server is configured to require user authentication and authorization, then the proxy server does not immediately set up a tunnel to the origin server. Instead, the proxy server responds with a 407 status code and a `Proxy-Authenticate` response header to ask for user credentials. The corresponding HTTP response message may begin with the following two lines:

```
HTTP/1.1 407 Proxy authentication required
Proxy-Authenticate: ...
```

In the first line, the proxy server informs the client that it has not been able to serve the request, because it requires client (or user) authentication. In the second line, the proxy server challenges the client with a `Proxy-Authenticate` response header and a challenge that refers to the authentication scheme and the parameters applicable to the proxy for this request (not displayed). It is then up to the client to send the requested authentication information to the proxy server. Hence, the next HTTP request that it sends to the proxy server must comprise the credentials (representing the authentication information). The corresponding HTTP request message may begin with the following two lines:

```
CONNECT www.esecurity.ch:443 HTTP/1.1
Proxy-Authorization: ...
```

In the first line, the client repeats the request header to connect to port 443 at `www.esecurity.ch`. This is the same request as before. In the second line, however, the client provides a `Proxy-authorization` request header that comprises the credentials as requested by the proxy server. If these credentials are correct, then the proxy server connects to the origin server and hot-wires the client with it.

Note that the CONNECT method provides a lower level function than many other HTTP methods. You may think of it as some kind of escape mechanism for saying that the proxy server should not interfere with the transaction but merely serve as a circuit-level gateway and forward the data stream accordingly. In fact, the proxy server should not need to know the entire URL that is being requested—only the information that is needed to serve the request, such as the host name and port number of the origin web server. Consequently, the HTTP proxy server cannot verify that the protocol being spoken is really SSL/TLS, and it should therefore limit allowed (tunneled) connections to well-known SSL/TLS ports, such as 443 for HTTPS (or another port number assigned by the IANA for another application protocol).

SSL/TLS tunneling is enabled by all HTTP clients and proxy servers that support HTTP and the respective CONNECT method. As further addressed in [9], SSL/TLS tunneling also has a few practical disadvantages ranging from difficulties in network management, monitoring, troubleshooting, performance optimization, and caching, to security and privacy issues related to intrusion and malware detection, DLP, and fraud monitoring. Many things that are possible with unencrypted data become impossible if data is end-to-end encrypted. The proxy server cannot even ensure that a particular application protocol (e.g., HTTP) is used on top of SSL/TLS. It can verify the port number in use, but this number does not reliably tell what application protocol the client and origin server are using (if encryption is invoked). If, for example, the client and the origin server have agreed to use port 443 for some proprietary protocol, then the client can have the proxy server establish an SSL/TLS tunnel to this port and use the tunnel to transmit any application data of its choice. It may be HTTP (or HTTPS, respectively), but it may also be any other protocol. The bottom line is that the HTTP proxy server can control neither the protocol in use nor the data that is being transmitted. In some application settings, this level of ignorance is dangerous and cannot be accepted.

Against this background, most companies and organizations support SSL/TLS tunneling only for outgoing connections and enforce SSL/TLS proxying for all incoming connections. If SSL/TLS tunneling were used for inbound connections, then a proxy server would have to relay the connection to the internal origin server, and this server would then have to implement the SSL/TLS protocols. Unfortunately, this is not always the case, and some internal web servers do not currently support the SSL/TLS protocols (and hence they do not represent HTTPS servers). In this situation, one may think about using a special software, such as stunnel,[5] that may act as an SSL/TLS wrapper and does the SSL/TLS processing on the server's behalf. The more servers natively support SSL/TLS, the fewer use cases are for this type of software though.

5 https://www.stunnel.org.

5.3 SSL/TLS PROXYING

As its name suggests, SSL/TLS proxying requires a server that proxies SSL/TLS (instead of establishing a tunnel). This implies that the proxy server understands the SSL/TLS protocols and terminates each connection that passes through. More specifically, SSL/TLS proxying mandates the following four-step procedure:

- First, the user has his or her client establish a first SSL/TLS connection to the proxy server.
- Second, the proxy server may authenticate and authorize the client (if required).
- Third, the proxy server establishes a second SSL/TLS connection to the origin server. Note that the use of SSL/TLS is not required here and that it may be sufficient to establish a TCP connection (over which data is then transmitted).
- Fourth, the proxy server mediates data between the two SSL/TLS connections, optionally doing content screening and caching. This can be done because the data is decrypted and optionally reencrypted by the proxy server.

The distinguishing feature of an SSL/TLS proxy server is that it terminates all SSL/TLS connections and hence that no SSL/TLS tunneling occurs. This makes everything transparent to the proxy server, but it also means that end-to-end security cannot be achieved and that the proxy server yields a legitimate MITM. Due to its ability to intercept data traffic, an SSL/TLS proxy server is often called *interception proxy* or *middlebox*.

There is a subtlety regarding SSL/TLS proxying: When the client establishes a first SSL/TLS connection to the proxy server (in step one of the four-step procedure mentioned above), then the proxy server must authenticate itself to the client. This means that it typically sends a CERTIFICATE message to the client as part of the handshake. As its name suggests, this message comprises a server certificate that can afterwards be verified by the client. There are two possibilities here: Either the proxy server shares the origin server's private key and certificate, or it must be able to issue a suitable certificate on the fly.

- According to the first possibility, the proxy server can be equipped with the origin server's certificate. This is not the problem. The problem only occurs if the proxy server has to authenticate to the client on the origin server's behalf by using this server's private key to generate a digital signature. To enable this, either a copy of the private key must be locally stored on the proxy server, or the proxy server must interact with the origin server to generate such a signature when needed. Either case has disadvantages, but the second possibility seems to be less disadvantageous here; so people often go for it.

- According to the second possibility, the proxy server can issue a suitable certificate for the origin server on the fly. More specifically, it generates an ephemeral public key pair and a respective certificate that looks as if it belongs to the origin server. This means that the proxy server acts as a CA and that the CA's public key must be trusted by the client, meaning that it must be part of the client's trust store. Otherwise, the client doesn't accept the certificate by default, but rather informs the user that the server is trying to authenticate itself with an untrusted certificate. It is then up to the user to decide whether the connection should be established or not (given the fact that the certificate is not trustworthy).

In practice, the second possibility is most often used. This is particularly true in a corporate setting, where the clients' certificate and trust stores are fully managed.[6] Things get more involved if the proxy server is not operated by the organization whom the clients belong to (we address this exceptional but practically relevant case in the following section) or some proprietary applications or IoT devices are used that rely on their own trust stores with no or little room for customization.

Another subtlety regarding SSL/TLS proxying refers to the key exchange methods that can be used. If the key exchange is based on static keys, such as in the case of RSA and static (i.e., nonephemeral) Diffie-Hellman, then SSL/TLS proxying is simple and straightforward, using any of the two possibilities mentioned above. If, however, the key exchange is ephemeral (i.e., (EC)DHE), then SSL/TLS proxying does not work per se. In such a setting, SSL/TLS proxying only works if the proxy and origin servers share the keying material. Again, this is seldom the case, especially if the servers are operated by different organizations and there may even be multiple proxy servers in place.

As mentioned above, SSL/TLS tunneling is primarily used for outbound connections, whereas SSL/TLS proxying is primarily used for inbound connections. In this case, the proxy server represents an inbound proxy[7] for the SSL/TLS

6 Note, for example, that the same technology of installing a root certificate in a client's trust store (to afterwards issue arbitrary certificates on the fly) was used in 2015 in at least two documented cases: The company Superfish used it to control the advertisements inserted into an HTTPS data stream to Lenovo laptops (https://us-cert.cisa.gov/ncas/alerts/TA15-051A), whereas Comodo shipped a tool named PrivDog that used it to stop illegitimate and unwanted advertisements. Both cases were extensively discussed in the media.

7 In the literature, inbound proxies are often called *reverse proxies*. In this book, however, we use the term *inbound proxy*, as there is no reverse functionality involved. In fact, a reverse proxy is doing nothing differently than a normal proxy server. The only difference is that it primarily serves inbound connections (instead of outbound connections).

connections. To the best of our knowledge, the first inbound proxy was developed by a group of researchers at the DEC Systems Research Center in 1998. They basically used a combination of SSL client authentication and URL rewriting techniques in a technology called *secure web tunneling*[8] [10]. A similar technology to access internal web servers was independently developed and combined with a one-time password system by a distinguished group of researchers at AT&T Laboratories [11].[9]

Since these early days, SSL/TLS proxy servers, interception proxies, and—most importantly—all kinds of middleboxes have become established and widely deployed in the field, sometimes even more widely than originally anticipated (e.g., [12, 13]). They enable a company or organization to connect to the internet, while still supporting features like content screening and DLP. On the flip side, however, a middlebox is often stealthy for its users (if it is able to issue certificates for origin servers on the fly) and thus breaks end-to-end security.[10] In [12], for example, it is reported that more than 5% of all middleboxes inject unwanted or even malicious content into web pages. This is worrisome, and it may even have got worse since then.

Backed by a 2015 Computer Emergency Response Team (CERT) Coordination Center (CC) post[11] and the somehow disillusioning results reported in [12], the U.S. Cybersecurity & Infrastructure Security Agency (CISA[12]) issued a security alert about HTTPS interception weakening TLS security already in 2017.[13] Most importantly, the alert warns about the use of HTTPS inspection products that do not correctly perform certificate validation, and this, in turn, directly weakens the end-to-end security that HTTPS originally aims to provide. In Section 5.4, we delve more deeply into the timely and practically relevant topic of HTTPS interception and middlebox mitigation.

8 Note that, in spite of its name, the technology refers to SSL/TLS proxying and not tunneling.

9 Note that both the DEC Systems Research Center and AT&T Laboratories do no longer exist to do research in this form.

10 Based on the heuristics and suggestions provided in [12], Cloudflare has developed two tools to detect middleboxes: MITMEngine is an open-source library for HTTPS interception detection that is based on heuristics about the normal look and feel of observed CLIENTHELLO messages, and MALCOLM—an acronym standing for "measuring active listeners, connection observers, and legitimate monitors"—is a complementary dashboard displaying metrics about HTTPS interception observed by Cloudflare with MITMEngine on its own network infrastructure. Further information about the tools is available at https://blog.cloudflare.com/monsters-in-the-middleboxes. The statistics collected with the tools even suggest that the level of HTTPS interception has increased since the publication of [12].

11 https://insights.sei.cmu.edu/cert/2015/03/the-risks-of-ssl-inspection.html.

12 The CISA is part of the Department of Homeland Security.

13 https://www.cisa.gov/uscert/ncas/alerts/TA17-075A.

5.4 MIDDLEBOX MITIGATION

We have seen that SSL/TLS tunneling is not always possible, due to security reasons that may prohibit its use, and that SSL/TLS proxying is often required in the field. Strictly speaking, this means that SSL/TLS is not really an end-to-end security technology. Rather, it is an end-to-middlebox or even end-to-first-middlebox security technology, and what is going on behind the (first) middlebox is not visible from the end.

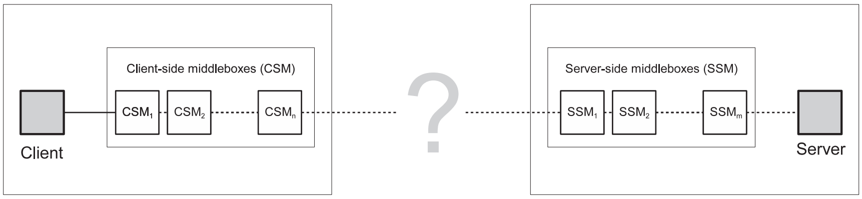

Figure 5.2 An SSL/TLS deployment setting with multiple middleboxes.

Figure 5.2 illustrates an SSL/TLS deployment setting with multiple middleboxes. On left side, there is a client with a sequence of $n \geq 0$ client-side middleboxes (CSM); such as CSM$_1$, CSM$_2, \ldots$, CSM$_n$. The client seeks to access the origin server on the right side. This server can also be located behind a sequence of $m \geq 0$ server-side middleboxes (SSM); such as SSM$_1$, SSM$_2, \ldots$, SSM$_m$, whereas between the client and the server side everything is possible and there may be other middleboxes that don't even belong to the client or server organization (symbolically illustrated with a large question mark).

If the client communicates with the server, then the data passes all intermediate middleboxes (CSM$_1$, CSM$_2, \ldots$, CSM$_n$, ..., SSM$_1$, SSM$_2, \ldots$, SSM$_m$). All middleboxes except the first one (i.e., CSM$_1$) are outside the scope of the client, meaning that there is no possibility for the client to control their working principles. Only the client organization is able to control the CSMs. But again, the existence and working principles of the SSMs are outside the scope of the client organization. If, for example, the server shares its private key(s) with SSM$_1$, then neither the client nor the client organization is able to detect this fact. All SSMs are then stealthy from the client perspective (similarly, all CSMs can be made stealthy from the server perspective by the client organization). As mentioned above, this means that the security provided by SSL/TLS is not really end-to-end, but rather end-to-(first)-middlebox.

If we have a closer look at the problem, then we may recognize that many end-to-end security technologies face a similar problem, and that a user can hardly ever be sure that he or she is connected to the origin server or only an SSM. If they share the keying material, then the origin server and the SSM (and all other SSMs) may be seen as one single entity, and in this case this entity represents the other end. But it may still be the case that some CSMs are located between the client and the other end. If the client is administered by the client organization, then again there is almost no possibility to detect the existence of a CSM. A possibility that may seem to work is to pin the certificate of the origin server, but again, if the organization administers the client, then it can also control certificate pinning. To make things worse, certificate pinning has several practical disadvantages that limit its usefulness in the field (especially in such a nontrivial setting as discussed here). The bottom line is that certificate pinning does not provide a practical and complete solution for the middleware mitigation problem.

Another possibility one may think of is to import keying material exported from TLS [14] into an application (Section 3.1.2), and thus make sure that the client and server are using the same TLS connection. This defeats any middlebox and is advantageous from a security perspective. But it is also a disadvantage in practice: If there is a middlebox that needs to be put in place (for some legitimate reason), then there is hardly any possibility to bypass the mechanism. This makes the mechanism difficult to deploy in the field.

Besides certificate pinning and exported keying material, the question of how to effectively mitigate middleboxes is still an important and timely research topic, and many people have proposed (partial) solutions. From a bird's eye perspective, the solutions can be grouped into three main categories that are based on (1) TLS extensions and specifically crafted certificates, (2) encryption, and (3) confidential computing (using trusted execution environments (TEEs) and secure enclaves).

1. The first category of solutions is based on TLS extensions and specifically crafted certificates—so called *proxy certificates*[14] [15]—or short-lived *delegated credentials* (according to the terminology introduced by an Internet-Draft[15]). In either case, the goal is to make the use of middleboxes transparent to the endpoints of a TLS connection (i.e., the client and the server) or to control the middleboxes' privileges and access rights in a visible and auditable fashion. There are many proposals specified in Internet-Drafts,[16] but there are

14 Such certificates have a `proxyAuthentication` value in the `ExtendedKeyUsage` field. They were originally introduced for grid computing, but they can also be used in the realm of TLS.

15 https://datatracker.ietf.org/doc/html/draft-ietf-tls-subcerts-15.

16 Two examples (that expired a long time ago) include https://tools.ietf.org/html/draft-mcgrew-tls-proxy-server-01 and https://tools.ietf.org/html/draft-rhrd-tls-tls13-visibility-01.

many more proposals published in the academic literature, such as end-to-end fine-grained HTTP (EFGH) [16], multicontext TLS (mcTLS[17]) [17],[18] and middlebox-aware TLS (maTLS[19]) [18]. The proposals differ in some details,[20] and it is too early to tell what proposal(s) will prevail in the long term.

2. The second category of solutions is based on special encryption technologies, such as searchable encryption or order-preserving encryption, that allow certain operations to be performed on encrypted data. This means that the middlebox does not have to decrypt data before it can process it, and this, in turn, means that the middlebox need not be trusted by its users. The current state of the art poses several technical challenges and limitations, and the functionalities of such technologies are currently still limited to pattern-matching and range-filtering. There are a few prototype implementations, including BlindBox [19], Embark [20], and PrivDPI [21]; none of them are ready for prime time though.
3. Last but not least, the third category of solutions is based on confidential computing and respective technologies, such as TEEs and secure enclaves (as, for example, provided by Intel's software guard extensions (SGX) or trusted domain extensions (TDX), as well as ARM's TrustZone). This means that these solutions require hardware support. Again, there a few prototype implementations, such as SGX-Box [22], STYX [23], middlebox TLS (mbTLS) [24], Safebricks [25], and Phoenix [26]. Again, this is still a research topic, and none of these technologies have been widely deployed so far.

The second and third categories are not backward-compatible, meaning that there is no migration path to incrementally deploy the respective solutions (instead, all currently deployed middleboxes must be replaced with ones that support proper encryption or use hardware support). This makes the solutions from the first category the simplest ones to deploy, and hence also the most promising ones. But these solutions still have their caveats when deployed in the field. Most importantly, the TLS extensions and proxy certificates are forwarded from middlebox to middlebox, and there is no possibility to supervise the legitimate behavior of all boxes in a sequence.

Due to the trend of enforcing forward secure key exchange methods only (i.e., (EC)DHE), the use of middleboxes has even become more challenging. This

17 https://mctls.org.

18 Based on mcTLS, the European Telecommunications Standards Institute (ETSI) is about to draft a *Transport Layer Middlebox Security Protocol* (TLMSP). The MSP and the controversial role the ETSI plays in the field are further addressed below.

19 https://middlebox-aware-tls.github.io.

20 https://arxiv.org/abs/2010.16388.

became very obvious in the standardization process of TLS 1.3. Many organizations (mainly from the banking industry) opposed to the banishment of static keys. The main argument was that the operation of middleboxes would become much more difficult. As mentioned in Section 3.5, the opponents finally managed to have a version of TLS that still supports static keys standardized by the ETSI [27–29]. The protocol was first named enterprise TLS (eTLS). After a legal dispute with the IETF, it was renamed to Enterprise Transport Security (ETS) and is now part of a so-called MSP series—sometimes also called TLMSP. Due to its inability to provide forward secrecy, eTLS/ETS/MSP/TLMSP is controversially discussed in the community and it is unlikely that it will prevail in the long term.[21] There is even a CVE (i.e., CVE-2019-9191) stating that the protocol (however it is named) yields a vulnerability because it does not provide forward secrecy.

An interesting situation occurs in the realm of *content delivery networks* (CDNs), such as the ones provided by companies like Akamai[22] and CloudFlare[23]: If an origin server is operated inside a CDN, then it makes a lot of sense to enforce SSL/TLS proxying by the edge servers of the CDN. This means that any inbound HTTPS request must be terminated at the edge and that the respective edge server must represent an SSL/TLS proxy server. If the edge server supported SSL/TLS tunneling, then all origin servers operated inside the CDN would be susceptible to attacks. This defeats one of the original purposes of a CDN, namely to protect all inside servers from the outside world (i.e., attacks from the internet). So a CDN should better enforce SSL/TLS proxying, hence the respective edge servers must terminate all inbound SSL/TLS connections. This, in turn, means that the edge servers must have access to the private keys of the origin servers. This is a problem because the origin server is operated by a different company or organization than the one that operates the CDN. There are basically two possibilities to resolve this issue: Either the keys are deposited and securely stored on the edge servers, or the keys are made otherwise available and accessible to these servers. For obvious reasons, the first possibility is simpler and more straightforward to deploy. However, it puts the origin servers' private keys at higher risk. This is why most CDNs provide a feature by which private keys can be stored and managed by the owners of the origin servers themselves. More specifically, such a key is stored and managed on a key server that is operated by the owner of the origin server. Instead of delivering the key to the edge server, the key server then performs all cryptographic operations that employ this key. Most importantly, if the edge server has to decrypt a CLIENTKEYEXCHANGE message to extract a premaster secret, then it sends the respective CLIENTKEYEXCHANGE message to the key server and the key server

21 https://www.eff.org/deeplinks/2019/02/ets-isnt-tls-and-you-shouldnt-use-it.
22 https://www.akamai.com.
23 https://www.cloudflare.com.

extracts the premaster secret on the edge server's behalf. This way, the edge server does not see or otherwise learn the private key. In the case of CloudFlare, for example, such a feature is available as *Keyless SSL*.[24] Other CDN providers use other terms to refer to the same or a similar functionality.[25] With the wide deployment of SSL/TLS in the field, features like Keyless SSL are getting more and more important.

There are, however, a few caveats to mention when it comes to deploying Keyless SSL and similar technologies: If an edge server is compromised, then it can have the key server decrypt any premaster secret of its choice. Hence, the edge servers must be trusted not to misbehave, and this level of trust may be prohibitive in some environments. Also, the key server must be highly available, partially defeating the purpose of using a CDN, and an additional round trip is required between the edge server and the key server per TLS handshake. Hence, Keyless SSL is certainly something to consider, but it is not a silver bullet for all security requirements.

5.5 FINAL REMARKS

This chapter has addressed the practically relevant problem of how the SSL/TLS protocols can (securely) traverse a firewall. There are basically two possibilities: SSL/TLS tunneling and SSL/TLS proxying. From a security perspective, SSL/TLS proxying is the preferred choice, since the firewall can then fully control the data traffic that is sent back and forth. In fact, there are many *web application firewalls* (WAFs) that represent SSL/TLS proxy servers with some complementary features, such as content inspection and DLP. The design and implementation of such WAFs that are resistant to various types of attacks is a timely and fastly evolving topic.

Today, many companies and organizations still use SSL/TLS tunneling for outbound connections and SSL/TLS proxying for inbound connections. But due to content-driven attacks (e.g., malware), this practice is about to change, and many security professionals opt for proxying outbound SSL/TLS connections, as well. Depending on the application setting, this can be simple or difficult to deploy and put in place. If the SSL/TLS proxy server is operated by the same organization, then SSL/TLS proxying is rather simple. But if it is operated by another organization or multiple SSL/TLS proxy servers are operated in sequence by different organizations, then SSL/TLS proxying may be difficult. This is particularly true if users must still

24 https://www.cloudflare.com/ssl/keyless-ssl.

25 For example, limited usage of remote key (LURK) is a general protocol for context-dependent interactions with remote keying material. Contexts are described in LURK extensions (e.g., for use in TLS handshakes). There are several Internet-Drafts related to LURK, such as https://tools.ietf.org/html/draft-mglt-lurk-tls-use-cases-02, https://tools.ietf.org/html/draft-mglt-lurk-lurk-01, and https://tools.ietf.org/html/draft-mglt-lurk-tls12-05.

have end-to-end connectivity to private applications, such as internet banking or e-voting. As there are only a few applications and servers of this type, it may make sense to allowlist them and tunnel SSL/TLS through.

Due to the connectionless and best-effort nature of UDP, making the DTLS protocol traverse a firewall is conceptually even more challenging than SSL/TLS. In particular, proxy-based firewalls do not natively work for UDP, and hence it is not obvious how to effectively implement a DTLS proxy server. Dynamic packet filtering and stateful inspection techniques can be used instead. If the DTLS protocol is successful (that is reasonable to expect), then these techniques are likely to become more important in the future to provide viable solutions here.

References

[1] Shirey, R., "Internet Security Glossary, Version 2," RFC 4949 (FYI 36), August 2007.

[2] Cheswick, W.R., and S.M. Bellovin, "Network Firewalls," *IEEE Communications Magazine,* September 1994, pp. 50–57.

[3] Zwicky, E.D., S. Cooper, and D.B. Chapman, *Building Internet Firewalls*, 2nd edition, O'Reilly, Sebastopol, CA, 2000.

[4] Cheswick, W.R., S.M. Bellovin, and A.D. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*, 2nd edition, Addison-Wesley, Reading, MA, 2003.

[5] Stewart, J.M., and D. Kinsey, *Network Security, Firewalls, and VPNs*, 3rd edition, Jones & Bartlett Learning, Burlington, MA, 2020.

[6] Luotonen, A., and K. Altis, "World-Wide Web Proxies," *Computer Networks and ISDN Systems,* Vol. 27, No. 2, 1994, pp. 147–154.

[7] Fielding, R., et al., "Hypertext Transfer Protocol—HTTP/1.1," RFC 2816, June 1999.

[8] Franks, J., et al., "HTTP Authentication: Basic and Digest Access Authentication," RFC 2817, June 1999.

[9] Moriarty, K., and A. Morton (eds.), "Effects of Pervasive Encryption on Operators," RFC 8404, July 2018.

[10] Abadi, M., et al., "Secure Web Tunneling," *Proceedings of 7th International World Wide Web Conference,* Elsevier Science Publishers B.V., Amsterdam, Netherlands, 1998, pp. 531–539.

[11] Gilmore, C., D. Kormann, and A.D. Rubin, "Secure Remote Access to an Internal Web Server," *Proceedings of ISOC Symposium on Network and Distributed System Security,* February 1999.

[12] Durumeric, Z., et al., "The Security Impact of HTTPS Interception," *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS 2017),* The Internet Society, February 2017.

[13] Tsirantonakis, G., et al., "A Large-scale Analysis of Content Modification by Open HTTP Proxies," *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018),* The Internet Society, February 2018.

[14] Rescorla, E., “Keying Material Exporters for Transport Layer Security (TLS),” RFC 5705, March 2010.

[15] Tuecke, S., et al., “Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile,” RFC 3820, June 2004.

[16] Fossati, T., V.K. Gurbani, and V. Kolesnikov, “Love All, Trust Few: on Trusting Intermediaries in HTTP,” *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox),* August 2015, pp. 1–6.

[17] Naylor, D., et al., “Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS,” *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM 2015),* August 2015, pp. 199–212.

[18] Lee, H., et al., “maTLS: How to Make TLS middlebox-aware?,” *Proceedings of the Network and Distributed System Security Symposium (NDSS 2019),* The Internet Society, February 2019.

[19] Sherry, J., et al., “BlindBox: Deep Packet Inspection over Encrypted Traffic,” *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM 2015),* August 2015, pp. 213–226.

[20] Lan, C., et al., “Embark: Securely Outsourcing Middleboxes to the Cloud,” *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2016),* USENIX Association, 2016, pp. 255–273.

[21] Ning, J., et al., “PrivDPI: Privacy-Preserving Encrypted Traffic Inspection with Reusable Obfuscated Rules,” *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 2019),* ACM Press, 2019, pp. 1657–1670

[22] Han, J., et al., “SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module,” *Proceedings of the First Asia-Pacific Workshop on Networking (APNet 2017),* 2017, pp. 99–105.

[23] Wei, C., et al., “STYX: A Trusted and Accelerated Hierarchical SSL Key Management and Distribution System for Cloud Based CDN Application,” *Proceedings of the 2017 Symposium on Cloud Computing (SoCC 2017),* ACM Press, 2017, pp. 201–213.

[24] Naylor, D., et al., “And Then There Were More: Secure Communication for More Than Two Parties,” *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2017),* ACM Press, November 2017, pp. 88–100.

[25] Poddar, R., et al., “Safebricks: Shielding network functions in the cloud,” *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2018),* USENIX Association, 2018, pp. 201–216.

[26] Herwig, S., C. Garman, and D. Levin, “Achieving Keyless CDNs with Conclaves,” *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020),* USENIX Association, 2020, pp. 735–751.

[27] ETSI TS 103 523-1, “CYBER; Middlebox Security Protocol; Part 1: MSP Framework and Template Requirements,” Version 1.1.1, December 2020.

[28] ETSI TS 103 523-2, “CYBER; Middlebox Security Protocol; Part 2: Transport Layer MSP, Profile for Fine Grained Access Control,” Version 1.1.1, February 2021.

[29] ETSI TS 103 523-3, “CYBER; Middlebox Security Protocol; Part 3: Enterprise Transport Security,” Version 1.3.1, August 2019.

Chapter 6

Public Key Certificates and Internet PKI

Previous chapters have emphasized that the SSL/TLS and DTLS protocols yet require public key certificates but that the management of these certificates is not addressed in the protocol specifications. This basically means that the management of these certificates must be handled outside the scope of SSL/TLS, and that a PKI for the internet is needed. This is what this chapter is all about. In particular, we introduce the topic in Section 6.1, elaborate on X.509 certificates in Section 6.2, address server and client certificates in Sections 6.3 and 6.4, overview a few problems and pitfalls in Section 6.5, discuss certificate legitimation in Section 6.6, and conclude with some final remarks in Section 6.7. We already mentioned in the preface that this topic is important and that its respective chapter is going to be comprehensive and long. This promise is delivered now. There are even more things to say, and readers who want to get more information are referred to the many books that are available on the topic (e.g., [1–3]).[1]

6.1 INTRODUCTION

According to RFC 4949 [4], the term *certificate* refers to "a document that attests to the truth of something or the ownership of something." Historically, the term certificate was coined and first used by Loren M. Kohnfelder to refer to a digitally signed record holding a name and a public key [5]. As such, the certificate attests to the legitimate ownership of a public key and attributes a public key to a principal, such as a person, a hardware device, or any other entity. The resulting certificates

1 Other references and sources of further reading include the NIST Special Publications 800-15, 800-25, and 800-32 that were officially withdrawn in 2021. At the same time, however, a revision of these documents was announced, meaning that they will be published in revised form sometime in the future.

are called *public key certificates*. They are used by many cryptographic security protocols, including SSL/TLS and DTLS. Again referring to RFC 4949 [4], a public key certificate is special type, namely one "that binds a system entity's identifier to a public key value, and possibly to additional, secondary data items." As such, it is a digitally signed data structure that attests to the ownership of a public key; that is, the public key belongs to a particular entity.

More generally (but still in line with RFC 4949), a certificate cannot only be used to attest to the legitimate ownership of a public key as in the case of a public key certificate but also to attest to the truth of any property attributable to the certificate owner. Such a certificate is commonly referred to as an *attribute certificate* (AC). In the realm of SSL/TLS, ACs may be used in the authorization extensions; for example, `client_authz` and `server_authz` (Section C.2.8). According to RFC 4949 [4], an AC is a "digital certificate that binds a set of descriptive data items, other than a public key, either directly to a subject name or to the identifier of another certificate that is a public-key certificate." Hence, the major difference between a public key certificate and an AC is that the former includes a public key (i.e., the public key that is certified), whereas the latter includes a list of attributes (i.e., the attributes that are certified). In either case, the certificates are issued (and possibly revoked) by authorities that are trusted by a community of users.

- In the case of public key certificates, such authorities are called CAs[2] or—more related to digital signature legislation—*certification service providers* (CSPs). The governance and operational procedures of a CA (or CSP) have to be specified and documented in a pair of documents: a certificate policy (CP) and a certificate practice statement (CPS). The CP and CPS documents are key when it comes to assessing the trustworthiness of a CA.
- In the case of attribute certificates, such authorities are called *attribute authorities* (AAs).

It goes without saying that a CA and an AA may be operated by the same organization. As soon as ACs start to take off, it is possible and very likely that CAs will also try to establish themselves as AAs. It also goes without saying that a CA can have one or several *registration authorities* (RAs)—sometimes also called *local registration authorities* or *local registration agents* (LRAs). The functions an RA carries out vary from case to case, but they typically include the registration and authentication of principals that are to become certificate owners. In addition, the RA may also be involved in tasks like token distribution, certificate revocation reporting, key generation, and key archival. In fact, a CA can delegate some of its

2 In the past, CAs were sometimes also called TTPs. This is particularly true for CAs that are operated by government bodies.

authorities (apart from certificate signing) to RAs. Consequently, RAs are optional components that are transparent to the users. Also, the certificates that are generated by the CAs may be made available by online directories and certificate repositories, such as provided by LDAP servers.

In short, a PKI consists of one (or several) CA(s). According to RFC 4949 [4], a PKI is "a system of CAs (and, optionally, RAs and other supporting servers and agents) that perform some set of certificate management, archive management, key management, and token management functions for a community of users in an application of asymmetric cryptography." Another way to look at a PKI is as an infrastructure that can be used to issue, validate, and revoke public keys and public key certificates. As such, it comprises a set of agreed-upon standards, CAs, structures among multiple CAs, methods to discover and validate certification paths, operational and management protocols, interoperable tools, and supporting legislation. A PKI and the operation thereof are therefore quite involved socio-technical items.

In the past, PKIs have experienced a hype and many companies and organizations have announced that they want to provide certification services on a commercial basis. As discussed toward the end of this chapter, many of these service providers have not been commercially successful and have gone out of business. Most providers that have prevailed make their living from other businesses and subsidize their PKI activities accordingly.

Many standardization bodies are working in the field of public key certificates and PKIs. Most importantly, the Telecommunication Standardization Sector of the ITU has released (and is periodically updating) a recommendation that is commonly referred to as ITU-T X.509 [6], or X.509 for short (Section 6.2). ITU-T X.509 has also been adopted by many other standardization bodies, including, for example, the ISO/IEC JTC1 [7]. Furthermore, a few other standardization bodies also work in the field of profiling ITU-T X.509 for specific application environments.[3]

In 1995, the IETF recognized the importance of public key certificates, and chartered a now concluded IETF Public-Key Infrastructure X.509 (PKIX[4]) WG with the intent of developing internet standards needed to support an X.509-based PKI for the internet community. (This PKI is sometimes called *internet PKI*.) The PKIX WG has initiated and stimulated a lot of standardization and profiling activities within the IETF (that was aligned with the activities within the ITU-T). In spite of the practical importance of the specifications of the IETF PKIX WG (especially

3 To "profile" ITU-T X.509—or any general standard or recommendation—basically means to fix the details with regard to a specific application environment. The result is a profile that elaborates on how to use and deploy ITU-T X.509 in the environment.

4 http://datatracker.ietf.org/wg/pkix/charter/.

regarding CP/CPS), we do not delve into the details in this book—also because the work is well documented in a series of RFC documents.

Following the original idea of Kohnfelder, a public key certificate comprises at least the following three pieces of information:

- A public key;
- Some naming information;
- One or more digital signatures.

The *public key* is the raison d'être for the public key certificate, meaning that the certificate only exists to certify a public key in the first place. The public key, in turn, can be from any public key cryptosystem, including RSA, Elgamal, Diffie-Hellman, or (EC)DSA, as long as the key is static (ephemeral keys normally don't come along with a certificate). The format (and hence also the size) of the public key depends on the cryptosystem in use.

The *naming information* is used to identify the owner of the public key (and hence the public key certificate). If the owner is a user, then the naming information typically consists of at least the user's first name and surname (also known as family name). In the past, there have been some discussions about the namespace that can be used here. For example, the ITU-T recommendation X.500 introduced the notion of a DN that can be used to identify entities, such as public key certificate owners, in a globally unique namespace. However, since then, X.500 DNs have not really taken off, at least not in the realm of naming persons. In this realm, the availability and appropriateness of globally unique namespaces have been challenged in the research community [8]. In fact, the simple distributed security infrastructure (SDSI) initiative and architecture [9] originated from the argument that a globally unique namespace is not appropriate for the global internet and that logically linked local namespaces are simpler and therefore more likely to be deployed (this point is further explored in [10]). As such, work on SDSI inspired the establishment of a simple public key infrastructure (SPKI) WG within the IETF security area. The WG was chartered in January 1997 to produce a certificate infrastructure and operating procedure to meet the needs of the internet community for trust management in a way that was as easy, simple, and extensible as possible. This was partly in contrast (and in competition) to the IETF PKIX WG. The IETF SPKI WG published a pair of experimental RFCs [11, 12] before its activities were finally abandoned in 2001.[5] Consequently, the SDSI and SPKI initiatives have turned out to be dead ends for the internet as a whole; hence, they are not further addressed in this book either. They hardly play a role in today's discussions about the management of public key

5 The WG was formally concluded on February 10, 2001, only four years after it was chartered.

certificates. However, the underlying argument that globally unique namespaces are not easily available on the internet somehow remains valid.

Last but not least, the *digital signature(s)* is (are) used to attest to the fact that the other two pieces of information (i.e., the public key and the naming information) belong together. The digital signature(s) turn(s) the public key certificate into a data structure that is useful in practice, mainly because it can be verified by anybody who knows the signatory's (i.e., CA's) public key. These keys are normally distributed with particular software—be it at the operating system or application software level.

Today, there are two practically relevant types of public key certificates: *OpenPGP certificates* and *X.509 certificates*. As further addressed in [13], the two types use slightly different certificate formats and trust models, where a trust model refers to the set of rules a system or application uses to decide whether a certificate is valid. In the direct trust model, for example, a user trusts a public key certificate because he or she knows where it came from and considers this entity as trustworthy.

- With regard to the certificate formats, a distinguishing feature of an X.509 certificate is that there is one single piece of naming information and that there is one single signature that vouches for this binding. This is different in the case of an OpenPGP certificate. In such a certificate, there can be multiple pieces of naming information bound to a public key, and there can even be multiple signatures that vouch for these bindings.
- With regard to the trust models, OpenPGP employs a cumulative trust model (further addressed in [13]), whereas ITU-T X.509 employs a hierarchical trust model as addressed below.

In practice, X.509 certificates clearly dominate the field, especially in the realm of SSL/TLS. This is why we only focus on X.509 certificates here. People who want to use and play around with OpenPGP certificates in an SSL/TLS setting can use the `cert_type` extension to do so (Section C.2.9).

6.2 X.509 CERTIFICATES

As mentioned above (and suggested by their name), X.509 certificates conform to the ITU-T recommendation X.509 [6] that was first published in 1988 as part of the X.500 directory series of recommendations (that also introduced the notion of a DN). It specifies both a certificate format and a certificate distribution scheme (using ASN.1). The original X.509 certificate format has gone through two major revisions:

- In 1993, the X.509 version 1 (X.509v1) format was extended to incorporate two new fields, resulting in the X.509 version 2 (X.509v2) format.

- In 1996, the X.509v2 format was revised to allow for additional extension fields. This was in response to the attempt to deploy certificates on the global scale. The resulting X.509 version 3 (X.509v3) specification has since been reaffirmed every couple of years.

When people nowadays refer to X.509 certificates, they essentially refer to X.509v3 certificates (and the version denominator is often left aside in the acronym). Let us now have a closer look at the X.509 certificate format and hierarchical trust model it is based on.

6.2.1 Certificate Format

With regard to the use of X.509 certificates on the internet, the profiling activities within the IETF PKIX WG are particularly important. Among the many RFC documents produced by this WG, RFC 5280 [14] is the most relevant and important one. Without delving into the details of the respective ASN.1 specification for X.509 certificates, we note that an X.509 certificate is a data structure that basically consists of the following fields (remember that arbitrary extension fields are possible, as well):[6]

- *Version:* This field is used to specify the X.509 version in use (i.e., 1, 2, or 3)
- *Serial number:* This field is used to specify a serial number for the certificate (i.e., a unique integer value assigned by the (certificate) issuer). The pair consisting of the issuer and the serial number must be unique. (Otherwise, it would not be possible to uniquely identify an X.509 certificate.)
- *Algorithm ID:* This field is used to specify the OID of the algorithm that is used to digitally sign the certificate. For example, 1.2.840.113549.1.1.5 is the OID that refers to `sha1RSA` (i.e., the combined use of SHA-1 and RSA).
- *Issuer:* This field is used to name the issuer. As such, it comprises the DN of the CA that issues (and digitally signs) the certificate.
- *Validity:* This field is used to specify a validity period for the certificate. The period, in turn, is defined by two dates, namely a start date (i.e., not before) and an expiration date (i.e., not after).

6 From an educational viewpoint, it is best to compare the field descriptions with the contents of real certificates. If you run a Windows operating system, then you may look at some certificates by running the certificate snap-in for the management console (just enter “certmgr” on a command line interpreter). The window that pops up summarizes all certificates that are available at the operating system level.

- *Subject:* This field is used to name the subject (i.e., the owner of the certificate), typically using a DN.
- *Subject Public Key Info:* This field is used to specify the public key (together with the algorithm) that is certified.
- *Issuer Unique Identifier:* This field can be used to specify some optional information related to the issuer of the certificate (only in X.509 versions 2 and 3).
- *Subject Unique Identifier:* This field can be used to specify some optional information related to the subject (only in X.509 versions 2 and 3). This field typically comprises some alternative naming information, such as an email address or a DNS entry.
- *Extensions:* Since X.509 version 3, this field can be used to specify some optional extensions that may be critical or not. While critical extensions need to be considered by all applications that employ the certificate, noncritical extensions are truly optional and can be considered at will. Among the most important extensions are *key usage* and *basic constraints*.
  - The key usage extension uses a bit mask to define the purpose of the certificate; that is whether it is used for normal digital signatures (0), legally binding signatures providing nonrepudiation (1), key encryption (2), data encryption (3), key agreement (4), digital signatures for certificates (5) or CRLs addressed below (6), encryption only (7), or decryption only (8). The numbers in parentheses refer to the respective bit positions in the mask.
  - The basic constraints extension identifies whether the subject of the certificate is a CA and the maximum depth of valid certification paths that include this certificate. This extension is not needed in a certificate for a root CA and should not appear in a leaf (or end entity) certificate. However, in all other cases, it is required to recognize certificates for intermediate CAs.[7]

7 For the sake of completeness, we note that the *#OprahSSL* vulnerability that hit the world in July 2015 refers to a buggy implementation of the basic constraints extension in OpenSSL. It is documented in CVE-2015-1793. In short, the vulnerability allows the verification of the basic constraints extension to be bypassed, and hence a leaf certificate can be used as if it were a certificate for an intermediate CA. Consequently, the holder of the certificate can issue certificates for any web server of his or her choice. This, in turn, can be used to mount a MITM attack. Fortunately, exploiting the #OprahSSL vulnerability is not as simple as it may look like.

Furthermore, there is an *extended key usage* extension that can be used to indicate one or more purposes for which the certified public key may be used (in addition to or in place of the purposes indicated in the key usage extension field), and a *subject alternative name* (SAN) extension that is frequently used to indicate a DNS name for the subject.

The last three fields make X.509v3 certificates very flexible, but also very difficult to deploy in an interoperable way. Anyway, the certificate must come along with a digital signature that conforms to the digital signature algorithm specified in the algorithm ID field.

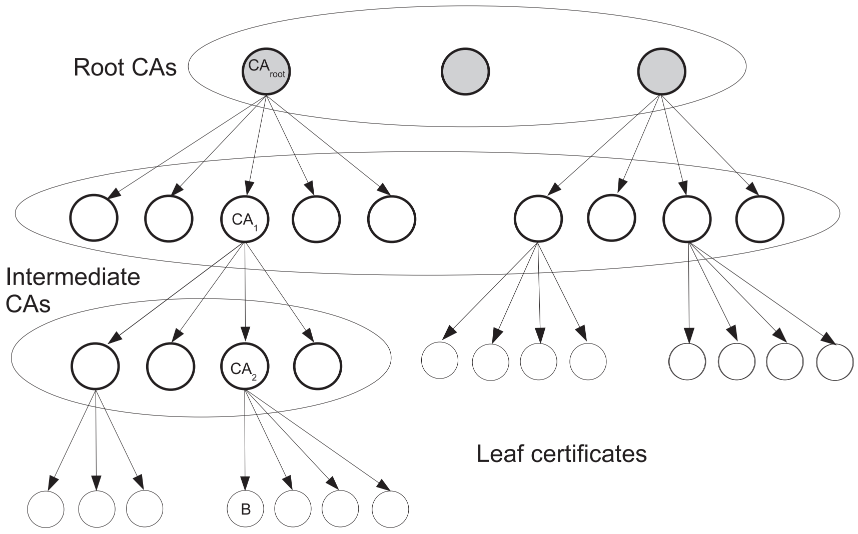

Figure 6.1 A hierarchy of trusted root and intermediate CAs that issue leaf certificates.

6.2.2 Hierarchical Trust Model

X.509 certificates are based on the hierarchical trust model that—as its name suggests—is built on a hierarchy of commonly trusted CAs. As illustrated in Figure 6.1, such a hierarchy is structured as a tree, meaning that there is one or several *root CAs* at the top level of the hierarchy. A root CA certificate is self-signed, meaning that the issuer and subject fields are the same and must be trusted by default. From a theoretical perspective, a self-signed certificate is not particularly useful. Anybody can claim something and issue a (self-signed) certificate for this claim. In the case of a public key certificate, it basically says: “Here is my public key, trust me.” There is no argument that speaks in favor of this claim. However, to bootstrap hierarchical trust, one or several root CAs with self-signed certificates are unavoidable (because the hierarchy must have a starting point from where trust can expand). In Figure 6.1, the set of root CAs consists of three CAs (i.e., the three shadowed CAs illustrated at the top). In reality, we are talking about dozens or even a few hundred CAs that are preconfigured in a trust store. For example, all major vendors of operating systems have programs to officially include a CA in their trust store (Section 6.3). The same is true for application software vendors, such as Adobe. The respective software can then be configured to use the operating system’s trust store, its own list of root CAs, or even both. So there is a lot of flexibility. Unfortunately, the flip side of this flexibility is that it sometimes overcharges the users.

In the hierarchy of trusted CAs, each root CA may issue certificates for other CAs that are then called *subordinate* or *intermediate CAs*. A subordinate CA is also an intermediate CA, but its distinguishing fact is that it is operated by the same organization as the root CA (or—in the recursive case—the certificate-issuing subordinate CA). Anyway, all intermediate CAs may form multiple layers in the hierarchy. (This is why we call the entire construction a hierarchy in the first place.) At the bottom of the hierarchy, the intermediate CAs may issue certificates for entities, such as users and web servers. These certificates are sometimes called *leaf certificates*. They have a parameter setting (with regard to the basic constraints extension) that ensures that they cannot be used to issue other certificates.

Equipped with one or several root CAs and respective root certificates, a user who receives a leaf certificate may try to find a *certification path* or *chain*; that is a sequence of certificates that leads from a trusted certificate (of a root or intermediate CA) to the leaf certificate. Each certificate certifies the public key of its successor. Finally, the leaf certificate is typically issued for an entity, such as a web server or a user. Let us assume that CA_{root} is a root certificate and B is an entity for which a certificate must be verified. In this case, a certification path or chain with n intermediate CAs (i.e., CA_1, CA_2, ..., CA_n) may look as follows:

$$\begin{array}{l} CA_{root} \ll CA_1 \gg \\ CA_1 \ll CA_2 \gg \\ \ldots \\ CA_{n-1} \ll CA_n \gg \\ CA_n \ll B \gg \end{array}$$

In this notation, $X \ll Y \gg$ refers to a certificate issued by X for the public key of Y. There are many subtleties omitted here, so the focus is entirely on the chain. Figure 6.1 illustrates such a chain with two intermediate CAs. It consists of $CA_{root} \ll CA_1 \gg$, $CA_1 \ll CA_2 \gg$, and $CA_2 \ll B \gg$. If intermediate CAs are supported, as is the case in TLS 1.0 (Section 3.2.2), then it may be sufficient to find a chain that leads from an intermediate CA's certificate to a leaf certificate. This may shorten certification chains considerably.

The simplest model one may think of is a certification hierarchy representing a tree with a single root CA. In practice, however, more general structures are possible, using multiple root CAs, intermediate CAs, and other CAs that may even issue cross certificates (i.e., certificates that are issued by CAs for other CAs).[8] In such a general structure (or mesh), a certification path is not unique and multiple certification paths may coexist. In this situation, it is required to have metrics in place that allow one to handle multiple certification paths. The design and analysis of such metrics is a research topic that is not further addressed in this book.

As mentioned above, each X.509 certificate has a validity period, meaning that it is well defined until when the certificate is supposedly valid. However, in spite of this information, it may still be possible that a certificate needs to be revoked ahead of time. For example, it may be the case that a user's private key gets compromised or a CA goes out of business. For situations like these, it is necessary to address certificate revocation in one way or another. The simplest way is to have the CA periodically issue a CRL [14] (i.e., a blocklist that enumerates all certificates by their serial numbers that have been revoked so far). Note that a certificate that has already expired does not need to appear in the CRL. The CRL only enumerates not-yet-expired but revoked certificates. Because a CRL may get very large, people have come up with the notion of a delta CRL. As its name suggests, a delta CRL restricts itself to the newly revoked certificates (i.e., certificates that have been revoked since the release of the latest CRL). In either case, CRLs tend to be impractical to handle, and hence the trend goes in the direction of retrieving online status information about the validity of a certificate. The protocol of choice is the OCSP [15]. OCSP is conceptually very simple: An entity using a certificate issues a request to the certificate-issuing entity (or OCSP responder, respectively) to find out whether the certificate is still valid. In the positive case, the responder sends back a response that

8 In spite of the fact that we call the trust model employed by ITU-T X.509 hierarchical, it is not so in a strict sense. The possibility to define cross certificates enables the construction of a mesh (rather than a hierarchy). This means that something similar to a web of trust can also be established using X.509. The misunderstanding partly occurs because the X.509 trust model is mapped to the directory information tree (DIT), which is hierarchical in nature (each DN represents a leaf in the DIT). Hence, the hierarchical structure is a result of the naming scheme rather than the certificate format. This should be kept in mind when arguing about trust models. Having this point in mind, it may be more appropriate to talk about centralized trust models (instead of hierarchical ones).

basically says "yes." In the negative case, however, things are more involved, and there are a few security problems that may pop up (Section 6.5). So OCSP works fine in theory, but it has a few problems and subtleties to consider. Some (but not all) of them can be addressed with OCSP stapling and the TLS `status_request` and `status_request_v2` extensions (Section C.2.6).

For the sake of completeness we mention here that the IETF has also standardized a *server-based certificate validation protocol* (SCVP) that can be used by a client that wants to delegate certificate path construction and validation to a dedicated server [16]. This may make sense if the client has only a few computational resources at hand to handle these tasks. In spite of the fact that the standard has been around for almost a decade, it has not seen widespread adoption and use in the field so far. This is partly because OCSP stapling can be used in these cases. SCVP has therefore silently sunk into oblivion.

6.3 SERVER CERTIFICATES

All nonanonymous key exchange methods of SSL/TLS require the server to provide a public key certificate—or rather a certificate chain—in a CERTIFICATE message. The certificate type must be in line with the key exchange method in use. Typically, this is an X.509 certificate that conforms to the profiles specified by the IETF PKIX WG. If the server provides such a chain, then it must be verified by the client. This means that the client must verify that each certificate in the chain is valid and that the chain leads from a trusted root or intermediate CA certificate to a leaf certificate for the server. If the verification succeeds, then the server certificate is immediately accepted by the client. Otherwise (i.e., if the verification fails), then the server certificate must either be refused by the client or the client must ask the user whether he or she wants to accept the certificate nonetheless (forever or for a single connection). The GUI for this dialog is browser-specific, but the general idea is always the same. Unfortunately, all empirical investigations reveal the embarrassing fact that most users simply click through such a dialog, meaning that they almost certainly click "OK" when they are asked to confirm acceptance of the certificate in question. Due to this fact, some modern browsers make it very difficult for the users to simply click through.

As mentioned above, all major browser vendors have programs for CA inclusion in place that allow root or intermediate CAs to have their certificates be included in the browsers' trust stores. These programs are yet similar in spirit but different in detail. They all require that a CA undergoes independent audits specifically designed for CAs. These audits typically take into account standards and best practices provided by standardization bodies (e.g., [17–19]) and professional organizations, such as the Canadian Chartered Professional Accountants and its WebTrust seal.[9] As is usually the case, a CA that is accepted by one program will usually also be accepted by other programs. There is even a *common CA database*[10] (CCADB) jointly managed by Mozilla, Microsoft, Google, and a few other companies. Note that Apple does not participate here. This is because Apple's trust store has always been relatively closed.

The market for server certificates has been dominated by a few internationally operating CSPs. Except for Let's Encrypt which is addressed below, these are DigiCert[11] (with its many subsidiaries, like GeoTrust[12] and QuoVadis[13]), ComodoCA,[14] and Sectigo.[15] Other CSPs have a relatively small market share. Typically, the validity period of a server certificate is a few years (e.g., one to five years), and its costs are in the range of nothing up to a few hundred U.S. dollars per year. The costs of a server certificate largely depend on its validation mechanism. There are three such mechanisms:

- *Domain validation* (DV) means that it is validated whether the entity that requests a certificate also controls the respective domain. In practice, this means that a confirmation mail is sent to one of the mail addresses that is affiliated with the domain. If the recipient approves the request (e.g., by following a link in the mail), then a DV certificate is issued automatically. If confirmation via mail is not possible, then any other means of communication and demonstration of domain control is possible and can be used accordingly.
- *Organization validation* (OV) means that the identity and authenticity of the entity that requests a particular certificate is properly verified. As the word "properly" is not well defined, there are usually many ways to do the verification, and hence there is a lot of inconsistency in how OV certificates are issued and how the relevant information is encoded in the certificate.
- *Extended validation* (EV) certificates were introduced to address the lack of consistency in OV certificates. This means that the validation procedures are thoroughly defined and documented by the CA/Browser Forum.[16] As its name suggests, the CA/Browser Forum consists of commercial CA operators,

9 https://www.cpacanada.ca/en/business-and-accounting-resources/audit-and-assurance/overview-of-webtrust-services.
10 https://www.ccadb.org.
11 https://www.digicert.com.
12 https://www.geotrust.com.
13 https://www.quovadisglobal.com.
14 https://www.comodoca.com.
15 https://sectigo.com.
16 http://www.cabforum.org.

browser manufacturers, and WebTrust assessors. As EV certificates try to achieve a high level of security and assurance, they are sometimes also called *high-assurance* (HA) certificates. Such certificates tend to be a little bit more expensive than the DV and OV counterparts, but otherwise there is hardly any disadvantage for the service providers that want to show their high security standards to their customers. They have been widely deployed in the field, and the major browsers have had specific indicators in their GUIs to make sure that users are aware that EV certificates are being used.

It goes without saying that the issuance of DV certificates can be fully automated and can therefore be very fast. In fact, the duration of the issuance process is bounded by the speed in which the confirmation mail is answered and the respective fees are paid. On the other end of the spectrum, the issuance of an EV certificate can take several days or weeks. If a web site needs to be established as fast as possible, then this amount of time is certainly prohibitive. To make things worse, EV certificates are no longer differentiated in the GUIs of the major browsers.

Complementary to the notions of DV, OV, and EV certificates, there are two other types of certificates that need to be mentioned here (for the sake of completeness):

- As mentioned in Section 2.2.2.4, Netscape and Microsoft introduced International Step-Up and SGC certificates in the 1990s. These certificates allowed an international browser to invoke and make use of strong cryptography. (Otherwise it was restricted to export-grade cryptography.) Only a few CSPs were authorized and approved by the U.S. government to issue such certificates (in particular, VeriSign, which was later acquired by Symantec). The legal situation today is completely different, and browser manufacturers are generally allowed to ship their products incorporating strong cryptography overseas. Against this background, International Step-Up and SGC certificates no longer make sense. This is particularly true for International Step-Up certificates, because hardly anybody is using the Netscape Navigator anymore.
- To be able to invoke the SSL/TLS protocols, a server is typically configured with a certificate that is issued for a particular fully qualified domain name (FQDN). If such a certificate has, for example, been issued for `secure.esecurity.ch`, then it cannot be used for another subdomain of `esecurity.ch`, such as `www.esecurity.ch`. In theory, this is perfectly fine and represents the use case the certificate is intended to serve. In practice, however, this is problematic and requires a domain owner to procure many certificates (one certificate for each FQDN). In addition to higher costs, this means that many certificates must be managed simultaneously. To simplify

the situation, some CSPs offer *wildcard certificates*. Such a certificate has a wildcard in its domain name, meaning that it can be used to secure multiple FQDNs at the expense of a generally higher price. If, for example, there is a wildcard certificate issued for `*.esecurity.ch`, then this certificate can be used for all subdomains mentioned above (as well as any other subdomain of `esecurity.ch`). This simplifies certificate management considerably and is particularly well suited for server farms that support load balancing. The security implications, however, cannot be ignored, because it may be confusing for a user not to know exactly to which server he or she is connecting to.

While International Step-Up and SGC certificates are no longer used in the field, wildcard certificates are still in action. This is because their major advantage (i.e., simplicity) overrides their major disadvantages (i.e., higher price and user inability to distinguish the servers within a domain).

To further simplify things and bring down the prices for DV certificates, the Electronic Frontier Foundation (EFF) joined Mozilla, Cisco, Akamai, IdenTrust, and a group of researchers from the University of Michigan to form the Internet Security Research Group (ISRG[17]). In 2015, the ISRG launched an initiative[18] prominently named "Let's Encrypt." The aim of the initiative was to enable a more straightforward transition from HTTP to HTTPS, and hence to provide an automatic certificate management environment (ACME) that allows a web service provider to receive and automatically install a DV certificate for free. Meanwhile, ACME was further developed and promoted by an IETF WG of the same name, and this WG has specified ACME in Standards Track RFC 8555 [20] and several complementary RFC documents and Internet-Drafts.[19] The results are so overwhelmingly successful that Let's Encrypt certificates clearly dominate the field. In fact, there are millions of certificates that have been automatically issued and are actively managed with ACME.

6.4 CLIENT CERTIFICATES

From a theoretical perspective, there is no big difference between a server certificate and a client certificate. While the former is issued to a FQDN or domain (in the case of a wildcard certificate), the latter is typically issued to a user. The user can then

17 The ISRG is a California public benefit corporation whose application for recognition of tax-exempt status under Section 501(c)(3) of the Internal Revenue Code is currently pending with the IRS. ISRG's mission is to reduce financial, technological, and education barriers to secure communication over the internet.

18 https://letsencrypt.org.

19 https://datatracker.ietf.org/wg/acme/documents.

employ the client certificate when he or she uses SSL/TLS to connect to a server that requires certificate-based client authentication. (Remember that client authentication has always been optional in SSL/TLS and that it is sometimes used to differentiate mTLS from TLS.)

The major difference between server and client certificates is the way they are usually put on the market: While the former are available from Let's Encrypt and some commercially operating CSPs (as discussed above), the latter are more difficult to procure. This is because many CSPs have left this business segment and only sell server certificates. The CSPs that still sell client certificates mainly focus on other use cases and applications, such as document signing and secure messaging (e.g., S/MIME), and only make the point that the same certificates can also be used for client authentication in SSL/TLS. This use case is not assumed to move the masses though.

But client certificates are successful in areas, where they can be automatically generated, rolled out, and managed. For example, this is the case in organizations that run Windows PKI (or similar) services. Such services allow users to be provisioned with automatically generated certificates that can afterwards be used for multiple purposes, including client authentication in SSL/TLS. If such services are available, then the use of client certificates is simple and straightforward and the respective solutions scale well. Otherwise, mainly due to the lack of automation, client certificates can only be used for relatively small user communities and populations. In such environments, client certificates are typically neither supported nor used.

6.5 PROBLEMS AND PITFALLS

The use of public key certificates and PKIs is simple in theory but difficult in practice. In fact, there are many problems and pitfalls that may be exploited in specific attacks. For example, since the early days of PKIs, the use of notoriously insecure hash functions like MD5 and SHA-1 has troubled the community. In 2009, Moxie Marlinspike demonstrated two serious attacks that exploit the ambiguity of ASN.1 encoding in X.509 and a distinct vulnerability in OCSP. Five years later, a group of researchers played around with artificially crafted certificates (e.g., by randomly mutating parts of real certificates and including some unusual combinations of extensions and constraints) that are able to disturb the certificate validation process in many unexpected ways [21]. They coined the term *frankencerts* to refer to such certificates, and this type of automated adversarial testing has turned out to be successful.

Since 2011, several commercially operating CAs have come under attack [22]. In fact, people have successfully broken into (trusted) root CAs and misused the respective machinery to fraudulently issue certificates for widely used domains and servers. Two particular incidents shocked the community and have been widely discussed in the media:

- In March 2011, at least two Italian Comodo resellers[20] were compromised and nine server certificates were fraudulently issued.
- In July 2011, DigiNotar was compromised and at least 513 server certificates were fraudulently issued. DigiNotar was a Vasco company that also operated a PKI for the Dutch government (named PKIoverheid). The compromise came as a surprise, because DigiNotar had been audited and certified by an accredited company, and because it even affected EV and wildcard certificates. This makes the attacks that are feasible with the respective certificates exceptionally powerful. Only a few days after the compromise was made public, DigiNotar was liquidated by Vasco.

After having been attacked, both victimized companies claimed that they had been subject to an advanced persistent threat (APT). Unfortunately, the acronym APT is often used as a synonym for something that is inherently inevitable. However, this is not always the case and is clearly not the case here—as was demonstrated by an interim report[21] and an independent final report.[22]

Due to an analysis of the log files of the OCSP servers of the affected CAs, one knows today that the fraudulently issued certificates were mainly used in the Middle East. In fact, it is rumored that the certificates had been used to implement politically motivated MITM attacks against the users of popular web sites and social media in Iran. The feasibility of such attacks had been discussed in academia prior to the incident [23], but there was only little evidence that such attacks really occurred in practice. This changed fundamentally, and, more recently, at least one DLP provider has received a CA certificate from Trustwave that allows it to issue valid certificates for any server of its choice.[23] To make things worse, the same attack vectors that work for politically motivated MITM attacks also work for nonpolitically motivated attacks against e-commerce applications, like internet banking or remote internet voting. Hence, protecting e-commerce applications against MITM attacks is something to consider more seriously. Technologies like mTLS (using client certificates), SSL/TLS session-aware user authentication [24], and token binding (Section C.2.20) have an important role to play here. Fraudulently issued certificates

20 GlobalTrust (https://www.globaltrust.it) and InstantSSL (http://www.instantssl.it). Note, however, that InstantSSL nowadays is a subsidiary of Sectigo.

21 http://cryptome.org/0005/diginotar-insec.pdf.

22 https://www.tweedekamer.nl/downloads/document?id=2012D40224.

23 Trustwave has since revoked the certificate and vowed to refrain from issuing such certificates in the future.

are dangerous and can be used for many other attacks, such as replacing system kernel software with malware by using a valid-looking code-signing certificate. Such a certificate has, for example, played a key role in the deployment of the Stuxnet worm [25] that has initialized or revitalized interest in supervisory control and data acquisition (SCADA) system security. The set of attacks that become feasible with fraudulently issued certificates is huge.

In what follows, we don't address the specifics of these attacks. Instead, we use a probability-theoretic observation or argument to make the point that similar attacks will occur again and again (unless one changes the underlying paradigms) and that new countermeasures—as discussed in the following section—are urgently needed.

Let us assume a list of n commonly trusted root CAs (i.e., $CA_1, CA_2, \ldots,$ CA_n). In reality, each software vendor has its own list of trusted root CAs (i.e., trust store), but—for the sake of simplicity—we assume that a common list exists. Each CA_i ($1 \leq i \leq n$) is compromised with a probability $0 \leq p_i \leq 1$ within a given time interval (e.g., 1 year). This can be formalized as follows:

$$\Pr[CA_i \text{ is compromised}] = p_i$$

For the purpose of this book, we ignore the difficulty of determining values for p_i. Instead, we assume that these values are known or can be determined in one way or another. We then note that $1 - p_i$ refers to the probability of CA_i not being compromised,

$$\Pr[CA_i \text{ is not compromised}] = 1 - p_i,$$

and we compute the probability that no CA is compromised as the product of the probabilities of all n trusted root CAs not being compromised:

$$\begin{aligned}
\Pr[\text{no CA is compromised}] &= \Pr[CA_1 \text{ is not compromised}] \times \\
&\quad\ \Pr[CA_2 \text{ is not compromised}] \times \\
&\quad\ \ldots \times \\
&\quad\ \Pr[CA_n \text{ is not compromised}] \\
&= (1-p_1)(1-p_2)\ldots(1-p_n) \\
&= \prod_{i=1}^{n}(1-p_i)
\end{aligned}$$

Hence, the probability that at least one CA is compromised is the complement of this expression:

$$\begin{aligned}\text{Pr[at least one CA is compromised]} &= 1 - \text{Pr[no CA is compromised]} \\ &= 1 - \prod_{i=1}^{n}(1-p_i)\end{aligned}$$

This formula yields the probability that we face a problem (similar to the Comodo or DigiNotar incidents) for any n and $p_1, \ldots, p_n$ within the given time interval. If we assume an equal probability p for all trusted root CAs, then $1 - \prod_{i=1}^{n}(1-p_i)$ equals $1-(1-p)^n$. If, for example, we assume $p = 0.01$ (meaning that an adversary can attack a CA with a success probability of 1%), then the probability that one faces a problem with $n = 10$ CAs is roughly 0.1 or 10%. This seems to be tolerable. However, as we increase the value of n, the success probability for the adversary asymptotically approaches 1. For $n = 100$, for example, the probability is about 0.64, and for $n = 200$, the probability is about 0.87. This is the order of magnitude for the size of currently used lists of trusted root CAs. Because n is likely to grow in future software releases, it is almost certain that we will face problems again and again. (This statement holds for all possible values of p, because smaller values of p only require larger values of n.)

The bottom line is that if any of the trusted root CAs of the internet PKI gets compromised, then forged certificates can be issued for any domain or server. To make things worse, this even applies to subordinate or intermediate (trusted) CAs. The overall situation is dangerous and worrisome. Consider the following two points:

- First, all currently used certificate revocation mechanisms, such as CRLs or OCSP, are capable of addressing the problem of a fraudulently issued certificate once a fraud is discovered. They implement a blocklist approach, meaning that they aim at identifying certificates that have been put on a blocklist. However, a certificate that is fraudulently issued does not automatically appear on such a list. Instead, it looks perfectly fine and valid, and hence any question of whether the certificate has been revoked needs to be answered in the negative. What is needed in addition to conventional certificate revocation mechanisms is a way to handle the case in which a CA is compromised and certificates have been fraudulently issued. This is something that has not been properly addressed so far, and this omission needs be repaired in the future.
- Second, in the internet PKI, all trusted CAs are equally valuable and appropriate attack targets. From the adversary's viewpoint, it does not really matter what CA he or she compromises as long as he or she is able to compromise

at least one. In some sense, all trusted root and subordinate CAs are sitting in the same boat, meaning that the compromise of a single CA is enough to compromise the security of the entire system (because the adversary will take advantage of a certificate that is fraudulently issued by any compromised CA of his or her choice). This is unfortunate and sometimes neglected when certification service providers argue that they have better security in place than their competitors. With such a statement they only claim that the attack will not exploit one of their certificates but a certificate issued by a competitor. This does not improve the overall situation and is not a good security design. It is better to go for a design in which a CA that is compromised is not free to issue certificates for any entity of its choice. So the fundamental question is who is authorized to issue certificates or declare them as valid for a given entity. This question is related to certificate authorization and is different from what people normally discuss when they talk about trust models.

So, we have to deal with two problem areas: certificate revocation and certificate authorization. We note that the two problem areas are not independent, meaning that any approach to handle certificate authorization must also address certificate revocation in one way or another. Hence, both areas can be subsumed under the new term *certificate legitimation* [22]. It is about who is legitimately authorized to issue and revoke certificates for a given entity (e.g., website or domain) or—alternatively speaking—whether a given certificate is legitimate or not. This includes many other topics, like certificate (path) validation.

There is some prior work related to certificate legitimation. For example, the name constraints extension for X.509 certificates can be used to restrict the range of identities for which a CA is authorized to issue certificates. It has been around since 1999, but it has never been used in the field (because some operating systems ignore and do not process the extension, and because it is not in line with some business models). However, in a world in which trusted CAs may get compromised and fraudulently issued certificates are likely to appear in the field, certificate legitimation is getting more and more important and key to the security of the internet PKI and the applications that may depend on it.

6.6 CERTIFICATE LEGITIMATION

There are several approaches that have been developed and are further refined to support certificate legitimation in one way or another. We address public key pinning, DNS resource records, distributed notaries, and—most importantly—certificate transparency.

6.6.1 Public Key Pinning

Probably the simplest approach to address the certificate authorization problem mentioned above is to fix the public keys that are authorized to be used by a particular piece of software, and to pin (i.e., hard-encode) these keys in the software. More specifically, this means that the software holds an allowlist of hash values from the public keys that are accepted as authoritative, and whenever it has to decide about the legitimacy of a public key it has to make sure that its hash value is included in the list. Needless to say that it is also possible to pin a certificate instead of a public key (although this may be a little bit more involved).

Historically, Google was the first company that used this technology (i.e., in the Chrome browser since version 13). The term that was coined in these early days was *public key pinning*—sometimes also called *certificate pinning*. It was used by Google to specify a set of public keys that were authorized to authenticate Google sites. After this initial use, public key pinning was generalized to be applicable to other (non-Google) sites, as well. In fact, Google crafted an RFC document that was later submitted to the internet standards track [26]. The resulting technology is known as HPKP, an acronym standing for *HTTP public key pinning*. The basic idea is that whenever a browser connects to a site using SSL/TLS, the site can use a new HTTP response header (i.e., `Public-Key-Pins`) to pin one or several public keys to that particular site. The public keys are referenced using cryptographic hash values and may be valid for a restricted amount of time (specified in seconds using a `max-age` directive). During this lifetime, no other public key than the pinned ones can actually be used.

By its very nature, HPKP is a trust-on-first-use (TOFU) mechanism that has advantages and disadvantages. The advantages are related to simplicity and ease of use, whereas the disadvantages are more related to reliability, security, and—maybe most importantly—scalability. For example, it is not immediately clear what happens if a site loses control of its private key. In this case, the site cannot properly authenticate itself during the execution of the SSL/TLS handshake protocol anymore. To overcome this problem, public key pinning requires a site to always specify a backup public key (in addition to the regular key). So in case of an emergency, the browsers equipped with the backup key can still continue to authenticate the site. More worrisome, public key pinning also has a bootstrapping problem: If an adversary manages to pin a wrong public key to a given site, then the legitimate owner of that site is permanently locked out (unless he or she can still use the backup key). Above all, public key pinning hardly scales with the number of keys. If the list of keys is large and dynamically changing, then managing the list and providing timely updates to all entities that make use of it is difficult and error-prone. Public key pinning therefore works best for software vendors that authorize only a few keys, such as operating system vendors that want to make sure that all updates and patches are coming from an authentic source. All big vendors use this technology in one way or another, some of them only to distribute urgent patches and updates. At the application level, however, HPKP is no longer considered to be the way to go (and the other approaches, in particular, certificate transparency, seem to be more seminal here).

Last but not least, note that sometimes another term pops up in the context of public key pinning; such as TACK, an acronym standing for *trust assurances for certificate keys*. It refers to a public key pinning variant that was originally proposed by Trevor Perrin and Moxie Marlinspike in 2012. There are basically two differences between TACK and normal public key pinning.

1. In TACK, the public key pairs are generated directly by the servers that want to pin a public key (instead of a CA). This means that the public (and respective private) keys are server-provided (and not CA-provided). The claimed advantage is that TACK is thus independent from any CA or internet PKI.
2. In TACK, it is envisioned that the pinning occurs at the SSL/TLS level, using the TLS extension mechanism. This means that public key pinning is independent from HTTP and can be used by any application protocol protected by SSL/TLS. This is in contrast to HPKP, which is strictly bound to HTTPS.

After the authors of the TACK proposal provided some proof-of-concept implementations for a few popular platforms, the initiative was not further pursued and has silently sunk into oblivion in the past decade. Note that the originally proposed TLS extension is not even included in the official list of TLS extensions (Appendix C) anymore. So it is mainly interesting for historical reasons.

6.6.2 DNS Resource Records

If one wanted to make public key pinning more general and flexible, then it would make sense to replace the static allowlist of authorized public keys (or respective certificates) with a more dynamic one. This new list can then be changed at will. In 2011, for example, the EFF pursued this idea in a project named *Sovereign Keys*.[24] The goal of the project was to enable each domain owner to use a public key pair, of which the public key is recorded in some publicly verifiable form, such as a DNS entry, and the respective private key—representing the sovereign key for that particular domain—can be used to digitally sign and hence authorize certificates.

24 https://www.eff.org/sovereign-keys.

Unfortunately, the Sovereign Keys project has not progressed past its early stage, and there are many problems that remain unsolved (e.g., what happens if the sovereign key gets lost or compromised). The lack of recovery mechanism makes the approach very difficult to deploy and use in the field.

A conceptually similar but more mature approach to link certificates to the DNS is pursued by the IETF. It basically consists of two complementary DNS resource records:

- On the one hand, the certification authority authorization (CAA) DNS resource record specified in Standards Track RFC 6844 [27] allows a domain owner to specify one or more CAs that are authorized to issue certificates for that domain.
- On the other hand, the TLS authentication (TLSA) record defined by the DNS-based authentication of named entities (DANE[25]) WG and specified in Standards Track RFC 6698 [28] allows a domain owner to specify what public keys or respective certificates can be used for certificate validation. DANE is conceptually related to the X.509 name constraints extension. The only difference is that DANE (in contrast to the X.509 name constraints extension) operates in a DNS context, whereas the X.509 name constraints extension does not. The major disadvantage of DANE is that it introduces a new dependency, namely the one on DNS security (DNSSEC). While DANE can, in principle, be operated without DNSSEC, it makes a lot of sense from a security perspective to combine the two technologies and to deploy DANE only in environments that have already implemented DNSSEC. As DNSSEC is being implemented and deployed, this dependency is going to be a less severe problem in the future. However, as DNSSEC is critical for DANE, it is also possible and likely that the DNS (including DNSSEC) is going to be a more attractive target for attack in the future.

The distinction between CAA and TLSA resource records is that the former specify an authorization control to be performed by a certificate issuer, whereas the latter specify a verification control to be performed by a relying party after the certificate is issued. It goes without saying that the two resource records are not mutually exclusive, and that they complement each other nicely. Hence, technologies like Sovereign Keys and—even more importantly—CAA and TLSA resource records improve security and strengthen resistance against MITM attacks. It is therefore reasonable to expect support to grow in the future.

25 http://datatracker.ietf.org/wg/dane/.

6.6.3 Distributed Notaries

If one wants to go one step further (than public key pinning and DNS usage) and even change the trust model, then one can design and come up with entirely new approaches. Such an approach was, for example, originally proposed in the *Perspectives* project[26] [29]. Here, a relying party can verify the legitimacy of a public key or certificate it aims to use for a particular server by asking a couple of distributed notary servers—called notaries. If the notaries vouch for the same public key or certificate, then it seems likely that the public key or certificate in question is correct. This is because a spoofing attack typically occurs locally, and hence it is very unlikely that multiple notaries are compromised simultaneously. The approach was originally prototyped in a Firefox add-on (with the same name) that could be used to verify SSH keys. Since then, the add-on has been extended to also address SSL/TLS certificates. The Perspectives project was originally launched in 2008, but it is still up and running.

Based on the work that had been done in the Perspectives project, Moxie Marlinspike implemented another Firefox add-on called *Convergence* that was publicly announced at the 2011 Black Hat conference. Convergence is best seen as a conceptual fork of Perspectives with some aspects of the implementation improved. To improve privacy, for example, requests to notaries are routinely proxied through several servers so that the notary that knows the identity of the client does not know the contents of the request. Also, Convergence caches site certificates for extended periods of time to improve performance. Convergence had some momentum when it was initially launched in 2011, but it has not seen a lot of activity since then and has silently sunk into oblivion.

Distributed notaries (like the ones used in the realms of Perspectives and Convergence) are useful when making collective decisions about the trustworthiness of a particular certificate. They still work if only a few CAs (or notaries) are compromised; they do not work if many CAs (or notaries) are compromised simultaneously. Fortunately, this is seldom the case, and hence distributed approaches seem to be useful and may help improve the situation. However, these approaches also have a few caveats. By depending on multiple external systems for trust, they make decision-making difficult. They also introduce problems related to performance, availability, and—maybe most importantly—running costs. Also, large websites often deploy many certificates (all with the same name). If these certificates are verified with different notaries, then it is very likely that many false positives are generated. This is because a view from one notary might not be the only correct one. Due to all of these disadvantages, distributed approaches and notaries still have a shadowy existence.

26 https://perspectives-project.org.

6.6.4 Certificate Transparency

As mentioned above, it is more appropriate to use an allowlist to address the legitimacy of a certificate than it is to use a blocklist. Any certificate that has been properly ordered and delivered and for which the appropriate certification fees have been paid is included in such a list of legitimate certificates. If somebody compromised a CA and fraudulently issued a certificate, then he or she would also have to fake the record for this certificate. This is feasible (in particular, for an insider), but it tends to be more involved and hence more expensive.

There are many situations in which an allowlist works better and provides better security than a blocklist. In the real world, for example, one can implement border control with either approach (or even a combination of the two):

- Using a blocklist, the agency in charge of border control has a list of persons who are not allowed to pass the border and enter the country, respectively. Everybody else is allowed to pass.
- Using an allowlist, the agency has a list of persons who are allowed to pass the border. Everybody else is not allowed to pass.

It goes without saying that a blocklist is more comfortable for traveling persons but less secure for the country (or the agency in charge of border control, respectively), and vice versa for an allowlist. In a real-world setting, the two lists are typically combined: a blocklist for known terrorists and criminals and an allowlist for people who originate from specific countries or hold a valid visa (or—more precisely in the case of the United States, have a valid Electronic System for Travel Authorization registration). When applying this analogy to the PKI realm, one has to be careful and take it with a grain of salt. While the primary focus of border control is authorization, the primary focus of PKI is authentication. Hence, there are subtle differences to consider. For example, many visas stand for themselves, and their revocation status need not be checked. This is not the case with certificates that almost always have to be checked for their status (i.e., whether they have been revoked).

Similarly, in the realm of firewall design, a blocklist can be used to implement a default-permit stance (i.e., everything is allowed unless it is explicitly forbidden), whereas an allowlist refers to a default-block stance (i.e., everything is forbidden unless it is explicitly allowed). Here, it is undisputed and commonly agreed that a default-block stance is more secure, but again the analogy has to be taken with a grain of salt: Possession of a certificate that can be validated is already an affirmation of an identity with some level of assurance, whereas an IP source address in the realm of a firewall or packet filter does not come along with any assurance.

In spite of these analogies and the advantages of allowlists in several settings, the PKI community has traditionally believed that the disadvantages of allowlists outweigh their advantages, and hence that a blocklist approach is more appropriate. This belief is, for example, evident in the discussions on the IETF PKIX WG mailing list (that are available in respective archives). In the aftermath of the Comodo and DigiNotar attacks, however, this design decision was revisited, and some people started to see it differently. Most importantly, Google launched a project that has become highly successful meanwhile. It is named *Certificate Transparency*[27] (CT) and the rationale behind its design is documented in [30]. Also, it is specified in RFC documents [31] (version 1.0) and [32] (version 2.0) that resulted from the now concluded IETF public notary transparency (TRANS) WG[28] that was active in the security area. Note that both RFC documents are yet classified as experimental, but that the success of CT merits more than an experimental technology. In fact, CT has been adopted by many key players in the field, maybe also because Google has used its market power to enforce the deployment and use of CT. The figures speak for themselves: As of this writing, almost 8 billion certificates have been logged since CT started in 2013. Google Chrome (since 2015), Apple Safari (since 2018), and Microsoft Edge mandate the use of CT, whereas Mozilla Firefox doesn't.[29]

From a bird's eye perspective, CT is a framework that allows one to implement an allowlist (i.e., a list of legitimately issued certificates). It basically consists of the following three components:

- *Logs* are append-only, tamper-resistant (by the means of cryptography), and publicly auditable data structures that—similar to many blockchains—are implemented as binary hash trees, sometimes also called Merkle trees [33]. When a CA issues a certificate, it is required that it is appended (as a leaf) to at least one log (that may be operated by the certificate-issuing CA or a third party). Each log must have a public HTTPS interface for querying.
- *Monitors* keep logs under permanent surveillance, meaning that they regularly query the logs to verify whether all logged certificates also appear in the logs and no illegitimately issued or suspicious certificate is found.
- Finally, *auditors* occasionally verify the consistency of logs (i.e., whether they behave correctly and do not contain certain certificates).

27 https://certificate.transparency.dev.

28 https://datatracker.ietf.org/wg/trans/about.

29 One possibility to check whether the currently used browser always requires CT is to connect the browser to `https://no-sct.badssl.com`. Only if the error message `NET::ERR_CERTIFICATE_TRANSPARENCY_REQUIRED` is displayed by the browser does it require CT. Otherwise, a red alert page indicates that it doesn't always require CT.

As CT is a framework, multiple setups are possible. For example, a log can be operated by a certificate-issuing CA or some other trusted party, a monitor can be operated by a CA, and an auditor may be embedded in a browser. This way, the browser's auditor can check a server-provided certificate on demand (i.e., verify that the server domain's certificate is contained in a properly functioning log). It is important to note that anybody can operate a log; since it is publicly verifiable (using monitors and/or auditors), its operator does not need to be trusted. Besides Google, there are a few companies that operate a log, including Let's Encrypt, DigiCert, and Sectigo (that are otherwise providing server certificates), as well as the CDN provider Cloudflare.

Logs, monitors, and auditors interact with each other in some defined (and specified) way. For example, the HTTPS interface allows one to query the log for certain certificates. Also, it allows one (i.e., typically a CA) to append a certificate to the log. After having told the log to append a given certificate, the log must respond with a *signed certificate timestamp* (SCT). The SCT, in turn, is a promise that the certificate will be appended to the log within a time period called the *maximum merge delay* (MMD). The MMD is usually set to 24 hours by default. The SCT is returned immediately to the certificate-issuing CA, whereas the append operation may be executed at some later point in time (within the MMD). Also, the HTTPS interface can be used to fetch Merkle audit or consistency proofs. A Merkle audit proof assures that a given certificate has been correctly appended to the log, whereas a Merkle consistency proof—as its name suggests—assures the log's consistency in the sense that all later versions of the log include everything from the original version in the same order, and that all new certificates come after the certificates from the original version. If a log is consistent (in this sense), then one can be sure that no certificate was removed or later inserted. Auditors typically make use of Merkle consistency proofs.

As an example, let us consider the case in which the domain owner of `esecurity.ch` wants to make use of CT for `www.esecurity.ch`. First, he or she requests a certificate from a CT-supporting CA. The CA, in turn, checks that the domain owner has the right to request such a certificate, and—if he or she does—creates a dummy certificate known as precertificate. The precertificate yet contains all the information a normal certificate does, but it also includes a poison extension so that CT-aware browsers don't accept it. Precertificates help to break a deadlock: Before a CA can log a certificate, the certificate needs an SCT; but for the certificate to get an SCT, it needs to have been submitted to a log. The CA can now submit the precertificate to a log, and the log responds with an SCT. This is the log's promise to append the certificate in due time (i.e., within the MMD). Note that there may be multiple SCTs—one from each log the CA has sent the precertificate to.

The CA can now either include the SCT(s) into the certificate (using X.509v3 extensions) or treat them as separate data structures and forward them to the domain owner. In either case, the SCT(s) accompany the certificate throughout its lifetime, and `www.esecurity.ch` can use them to authenticate itself in an SSL/TLS handshake. If they are included in the certificate, then they are also included in the server's CERTIFICATE message. Otherwise, the SCT(s) must be included in the SERVERHELLO message, typically as TLS extensions. In the preferred case, either the `signed_certificate_timestamp` extension or the `transparency_info` extension is used (Section C.2.16). Alternatively, however, the OCSP-related `status_request` extension can be used for this purpose (Section C.2.6). Anyway, both the number of logs, and the selection of logs a CA may choose, is determined by the browser (and its policy). As of this writing, for example, Apple Safari and Google Chrome both require at least two SCTs, also depending on the certificate lifetime.

More recently, a group of researchers has proposed a complementary technology called *CRLite* that can be used to combine CT data and internet scan results with cascading Bloom filters to compress revocation information from 300 megabytes of revocation data to 1 megabyte [34]. Prior to CRLite, Google used a technology called CRLSets[30] and Mozilla used a similar technology called OneCRL[31] to gather certificate revocation information centrally and push it out to clients. CRLite is a unification technology that comes along with massive performance improvements. The bottom line is that CT and CRLite may provide a viable alternative to certificate revocation based on CRLs or OCSP.

6.7 FINAL REMARKS

This chapter addressed public key certificates and internet PKI as far as they are relevant for the SSL/TLS and DTLS protocols. The standard of choice in this area is still ITU-T X.509, meaning that most server and client certificates in use today are X.509 certificates. Hence, there are many CSPs that provide commercial certification services to the public. Most of them make a living from selling server certificates to website operators (at least this was true until Let's Encrypt hit the market). The marketing of client certificates has turned out to be more involved than originally anticipated. In fact, most CSPs that have originally focused on client certificates have gone out of business (for reasons discussed in [35]) or have changed their business model fundamentally. The bottom line is that we are still far away from having a full-fledged PKI that can be used for SSL/TLS support on the client side and that

30 https://www.chromium.org/Home/chromium-security/crlsets/.

31 https://wiki.mozilla.org/CA/Revocation_Checking_in_Firefox#OneCRL.

web application providers therefore have to use other client or user authentication technologies. Ideally, these technologies are part of the TLS protocol specification and implemented accordingly. Unfortunately, this is not always the case, and client or user authentication is still done at the application layer (i.e., on top of SSL/TLS). This works fine but introduces the problem of making MITM attacks feasible.

When it comes to using public key certificates, trust is a major issue. Each browser comes along with a preconfigured set of trusted CAs—either root CAs or intermediate CAs. If a web server provides a certificate issued by such a CA, then the browser accepts the certificate (after proper validation) without user interaction. This is convenient and certainly the preferred choice from a usability perspective. From a security perspective, however, the preferred choice is to empty the set of trusted CAs and selectively include only the ones that are considered to be trustworthy. Unfortunately, this is seldom done, and there are only a few companies and organizations that distribute browser software with customized sets of trusted CAs.

The internet PKI is highly distributed and consists of many physically, geographically, and/or organizationally separated CAs that need to be trusted to some extent. Some of the CAs can be controlled by hostile organizations or totalitarian regimes, empowering them to mount large-scale MITM attacks. Also, the recent attacks against trusted CAs have put the security and usefulness of an internet PKI at stake. As it is possible and likely that similar attacks will occur again and again, it makes a lot of sense to design and implement appropriate countermeasures. Certificate legitimation is important, and many of the approaches discussed in the previous section look promising. This is particularly true for the DNS resource records that should be used on a large scale, ideally combined with DNSSEC, and CT (optionally improved with CRLite) that is almost mandatory today. Such approaches do not solve all security problems, but they make the resulting system more resilient against specific attacks. Interestingly, the approaches are not mutually exclusive and can be combined at will. We expect more proposals (and proposals on how to combine the existing approaches) to be developed and published in the future. The main issues are industry adoption and large-scale deployment.

References

[1] Adams, C., and S. Lloyd, *Understanding PKI: Concepts, Standards, and Deployment Considerations*, 2nd edition, Pearson Education, London, UK, 2010.

[2] Buchmann, J.A., E. Karatsiolis, and A. Wiesmaier, *Introduction to Public Key Infrastructures*, Springer-Verlag, Berlin, 2016.

[3] Ashbourn , J., *PKI Implementation and Infrastructures*, CRC Press, Boca Raton, FL, 2023

[4] Shirey, R., "Internet Security Glossary, Version 2," RFC 4949, August 2007.

[5] Kohnfelder, L.M., “Towards a Practical Public-Key Cryptosystem,” Bachelor’s thesis, Massachusetts Institute of Technology (MIT), Cambridge, MA, May 1978, http://groups.csail.mit.edu/cis/theses/kohnfelder-bs.pdf.

[6] ITU-T, *Recommendation X.509: Information Technology—Open Systems Interconnection—The Directory: Public-key and Attribute Certificate Frameworks*, 2012.

[7] ISO/IEC 9594-8, *Information Technology—Open Systems Interconnection—The Directory: Public-key and Attribute Certificate Frameworks,* 2014.

[8] Ellison, C., “Establishing Identity Without Certification Authorities,” *Proceedings of the 6th USENIX Security Symposium,* 1996, pp. 67–76.

[9] Rivest, R.L., and B. Lampson, “SDSI—A Simple Distributed Security Infrastructure,” September 1996, http://people.csail.mit.edu/rivest/sdsi10.html.

[10] Abadi, M., “On SDSI’s Linked Local Name Spaces,” *Journal of Computer Security,* Vol. 6, No. 1–2, September 1998, pp. 3–21.

[11] Ellison, C., “SPKI Requirements,” RFC 2692, September 1999.

[12] Ellison, C., et al., “SPKI Certificate Theory,” RFC 2693, September 1999.

[13] Oppliger, R., *End-to-End Encrypted Messaging*, Artech House, Norwood, MA, 2020.

[14] Cooper, D., et al., “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” RFC 5280, May 2008.

[15] Santesson, S., et al., “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol—OCSP,” Track RFC 6960, June 2013.

[16] Freeman, T., et al., “Server-Based Certificate Validation Protocol (SCVP),” RFC 5055, December 2007.

[17] ETSI TS 101 456, *Electronic Signatures and Infrastructures (ESI); Policy Requirements for Certification Authorities Issuing Qualified Certificates,* Version 1.4.3, May 2007.

[18] ETSI TS 102 042, *Electronic Signatures and Infrastructures (ESI); Policy Requirements for Certification Authorities Issuing Public Key Certificates,* Version 2.4.1, February 2013.

[19] ISO 21188, *Public Key Infrastructure for Financial Services—Practices and Policy Framework,* April 2018.

[20] Barnes, R., et al., “Automatic Certificate Management Environment (ACME),” RFC 8555, March 2019.

[21] Brubaker, C., et al., “Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations,” *Proceedings of the 2014 IEEE Symposium on Security and Privacy,* IEEE Computer Society, 2014, pp. 114–129.

[22] Oppliger, R., “Certification Authorities Under Attack: A Plea for Certificate Legitimation,” *IEEE Internet Computing,* Vol. 18, No. 1, January/February 2014, pp. 40–47.

[23] Soghoian, C., and S. Stamm, "Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL (Short Paper)," *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, Springer-Verlag, 2011, pp. 250–259.

[24] Oppliger, R., R. Hauser, and D. Basin, "SSL/TLS Session-Aware User Authentication," *IEEE Computer,* Vol. 41, No. 3, March 2008, pp. 59–65.

[25] Langner, R., "Stuxnet: Dissecting a Cyberwarfare Weapon," *IEEE Security & Privacy,* Vol. 9, No. 3, 2011, pp. 49–51.

[26] Evans, C., C. Palmer, and R. Sleevi, "Public Key Pinning Extension for HTTP," RFC 7469, April 2015.

[27] Hallam-Baker, P., R. Stradling, and J. Hoffman-Andrews, "DNS Certification Authority Authorization (CAA) Resource Record," RFC 8659, November 2019.

[28] Hoffman, P., and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA," RFC 6698, August 2012.

[29] Wendtandt, D., D.G. Andersen, and A. Perrig, "Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing," *Proceedings of the USENIX 2008 Annual Technical Conference (ATC 2008),* USENIX Association, Berkeley, CA, 2008, pp. 321–334.

[30] Laurie, B., and C. Doctorow, "Secure the Internet," *Nature,* Vol. 491, November 2012, pp. 325–326.

[31] Laurie, B., A. Langley, and E. Kasper, "Certificate Transparency," RFC 6962, June 2013.

[32] Laurie, B., E. Messeri, and R. Stradling, "Certificate Transparency Version 2.0," RFC 9162, December 2021.

[33] Merkle, R., "Secrecy, Authentication, and Public Key Systems," PhD Thesis, Stanford University, 1979.

[34] Larisch, J., et al., "CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers," *Proceedings of the IEEE Symposium on Security and Privacy,* IEEE, 2017, pp. 539–556.

[35] Lopez, J., R. Oppliger, and G. Pernul, "Why Have Public Key Infrastructures Failed So Far?" *Internet Research,* Vol. 15, No. 5, 2005, pp. 544–556.

Chapter 7

Concluding Remarks

After having thoroughly discussed the SSL/TLS and DTLS protocols (and a few complementary topics), we now finish the book with four concluding remarks.

- First, the book does not provide statistics about the use of the SSL/TLS and DTLS protocols in the field. Note that statistics are valid only for a relatively short period of time, and hence books seem to be the wrong media to provide them. Online resources and trade press articles are more appropriate here. Most importantly, the SSL Pulse[1] maintained by Qualys SSL Labs provides a lot of useful statistical material.[2] Complementary statistics are provided by security companies, like F5 with its TLS Telemetry Report (for the year 2021),[3] and numerous research papers (not even referenced here). Most statistics bear witness to the facts (a) that the use of the SSL/TLS protocols is steadily increasing and (b) that SSL/TLS is by far the predominant security technology on the internet. It is even used in areas where people would have argued a few years ago that the computational power is not sufficient, such as mobile computing and IoT. Consequently, SSL/TLS is embedded in most applications used in daily life.
- Second, an immediate consequence of SSL/TLS's triumphant advance is that its security is subject to a lot of public scrutiny. Many researchers have looked and continue to look into the security of the SSL/TLS and DTLS protocols and their implementations. Many problems have been found both at the protocol and implementation levels. Throughout this book, we have

1 https://www.ssllabs.com/ssl-pulse/.

2 In the past, the ICSI Certificate Notary (https://www.eff.org/de/observatory) and the EFF SSL Observatory (https://www.eff.org/de/observatory) served similar purposes. But these services and respective websites have shut down meanwhile.

3 https://www.f5.com/labs/articles/threat-intelligence/the-2021-tls-telemetry-report.

come across many vulnerabilities and respective attacks (Appendix A). Some of these vulnerabilities and attacks are not particularly worrisome and can be mitigated easily (by patching the respective implementations or reconfiguring them when used in the field). However, some vulnerabilities and attacks are devastating and indeed frightening. Still the most important example here is Heartbleed, which has had (and continues to have) a deep impact on the open-source software community. It has clearly demonstrated that security is a tricky and overly involved topic and that the entire security of a system can fail if only a tiny detail is implemented incorrectly (this fact of life has recently been demonstrated to be true again by the Log4j vulnerability or software bug). Against this background, any initiative that is aimed at coming up with implementations that have formally verified security is useful and appreciated. One such example is miTLS that is written in F#[4] and typechecked with F7.[5] More research activities in this area are needed and encouraged.

- Third, there are several documents that elaborate on how to configure a system that implements the SSL/TLS and DTLS protocols in a secure way—be it a client or a server. Most importantly, there is an IETF document [1] that not only makes recommendations from a theoretical and security-centric viewpoint, but also takes into account what is realistic and can actually be deployed in the field.[6] Most recommendations made in [1] have already been mentioned in this book. For example, according to Section 2.1, the IETF has changed its preference in favor of a separate port strategy and now recommends the use of a separate port for HTTPS (instead of using an upgrade mechanism in HTTP to dynamically invoke TLS). Also, according to Section 3.6, the IETF strongly encourages the use of HSTS in the field. TLS 1.3 supports 0-RTT, but the use of this feature requires caution (Section 3.5). The cipher suites that are recommended in [1] for TLS 1.2 combine ECDHE with RSA- or ECDSA-based authentication (to provide forward secrecy), use authenticated encryption with 128-bit or 256-bit AES operated in GCM, and employ two hash functions from the SHA-2 family (i.e., SHA-256 and SHA-384). This results in the following four cipher suites:

4 https://fsharp.org.
5 https://raweb.inria.fr/rapportsactivite/RA2010/moscova/uid19.html.
6 The original version of this document (RFC 7525) was published in 2015, when the industry was transitioning to TLS 1.2. Nowadays, this transition is largely complete, and the transition to TLS 1.3 is going on. This has made it necessary to update the recommendations that now comprise advice for both TLS 1.2 and TLS 1.3. Also, the resulting BCP 195 comes along with RFC 8996 [2] that deprecates TLS 1.0, TLS 1.1, and DTLS 1.0, meaning that BCP 195 refers to [1] and [2]. Needless to say that [1] also has as an explicit goal to encourage migration of most uses of TLS 1.2 to TLS 1.3.

  - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256;
  - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384;
  - TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256;
  - TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384.

  A cipher suite that operates in CBC mode should not be used, unless the `encrypt_then_mac` (Section C.2.19) extension is negotiated. For TLS 1.3, the recommended cipher suites are provided in the specification (Section 3.5.1). There are similar recommendations from the Mozilla project regarding the proper configuration of server side TLS,[7] minimal recommendations from the German Federal Office for Information Security (BSI) [3], SSL/TLS deployment best practices from Qualys,[8] proper guidance from the Open Web Application Security Project (OWASP),[9] a revised guide from the U.S. NIST,[10] as well as security controls guidelines for SSL/TLS management from the SANS Institute (not even referenced here). All of these resources are highly recommended reading for anybody working in the field. Last but not least, there are tools that allow for automatic validation of TLS configurations and thus help the developers to find attacks and the server administrators to find misconfigurations. Some tools are available online, such as Qualys SSL Labs' SSL Server Test,[11] whereas other tools can be downloaded and executed locally, such as `testssl.sh`[12] or the TLS-Scanner.[13]

- Fourth, it is important to emphasize that TLS 1.3 is a true security milestone in the evolution of the SSL/TLS protocols. While SSL 3.0, TLS 1.0, TLS 1.1, and TLS 1.2 have steadily increased their functionality and feature richness, TLS 1.3 is the first protocol version that is severely restricted to using only strong cryptography (e.g., AEAD ciphers and key exchange methods that always provide forward secrecy). This is uncompromising and somehow breaks with the tradition of protocol design in standardization committees, but it is going to have a strong impact on the security of TLS. All known attacks against the predecessors (Appendix A) no longer work against TLS 1.3. In practice, however, deployed systems are only going to profit from this

7 https://wiki.mozilla.org/Security/Server_Side_TLS.
8 https://www.ssllabs.com/projects/best-practices.
9 https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html.
10 https://www.nist.gov/news-events/news/2014/04/nist-revises-guide-use-transport-layer-security-tls-networks.
11 https://www.ssllabs.com/ssltest.
12 https://testssl.sh.
13 https://github.com/tls-attacker/TLS-Scanner.

advanced level of security if they are not configured in a backward-compatible way (otherwise downgrade attacks are likely to be mounted). Unfortunately, this is difficult to achieve and has led to many tweaks in the design of TLS 1.3 to achieve compatibility, interoperability, and extensibility; for example, the generate random extensions and sustain extensibility (GREASE) mechanism introduced in [4]. The bottom line is that the security of the SSL/TLS and DTLS protocols is still going to bother us in the future, and hence that this book has neither become obsolete nor does it have a happy ending. The story continues.

With these concluding remarks, we finish our explanation report about the SSL/TLS and DTLS protocols. We hope that we have provided enough background information to not leave you stranded, and we wish you good luck for all of your future endeavors in this area.

References

[1] Sheffer, Y., P. Saint-Andre, and T. Fossati, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 9325 (BCP 195), November 2022.

[2] Moriarty, K., and S. Fossati, "Deprecating TLS 1.0 and TLS 1.1," RFC 8996 (BCP 195), March 2021.

[3] German Federal Office for Information Security (BSI), "Mindeststandard des BSI zur Verwendung von Transport Layer Security (TLS)," Version 2.1, April 9, 2020.

[4] Benjamin, D., "Applying Generate Random Extensions and Sustain Extensibility (GREASE) to TLS Extensibility," RFC 8701, January 2020.

Appendix A

Attacks Against SSL/TLS

This appendix explores the attack landscape and hence addresses the attacks that have been mounted against the SSL/TLS protocols so far. It starts with two padding oracle attacks, which are the Bleichenbacher and Vaudenay attacks (including many variants), and then moves on to other attacks, like BEAST and POODLE, as well as renegotiation, compression-related, and key exchange downgrade attacks. Each attack is outlined, explained in some detail, and put into perspective, also addressing countermeasures and counterattacks. The aim is to provide a comprehensive picture, so that the appendix is self-contained and provides a summary of the most relevant attacks that have targeted the SSL/TLS protocols and the respective ecosystem. Note, however, that there are several attacks not further addressed here, such as the "double goto fail" bug and Heartbleed mentioned in the preface, as well as attacks against the collision resistance property of cryptographic hash functions like MD5 and SHA-1 (e.g., SLOTH[1]), attacks against stream ciphers (e.g., RC4 NOMORE[2] mentioned in Chapter 2) and 64-bit block ciphers (e.g., Sweet32[3]), attacks against cryptographic primitives and respective implementations (e.g., Racoon[4]), as well as attacks against public key certificates and their proper use. The last topic is partly addressed in Chapter 6. Note that SSL/TLS has a huge attack surface, and that there are plenty of attack vectors that can be exploited in the field. More worrisome, some attack vectors may not even be known today and will be discovered and published after the release of this book. So neither this appendix nor this book is complete in terms of possible attacks.

1 https://www.mitls.org/pages/attacks/SLOTH.
2 https://www.rc4nomore.com.
3 https://sweet32.info.
4 https://raccoon-attack.com.

A.1 BLEICHENBACHER ATTACK

As mentioned in Section 2.4, Bleichenbacher published a CCA2 against PKCS #1 version 1.5 [1] in 1998 [2].[5] This version of PKCS #1 was then heavily used in RSA-based key exchanges of the SSL/TLS handshake protocol; for example, to pad the 48-byte premaster secret to the length of the RSA modulus n before it is encrypted and sent in a CLIENTKEYEXCHANGE message to the server. Consequently, the *Bleichenbacher attack* affected many versions of the SSL/TLS protocols until it was finally mitigated in TLS 1.3.

From a bird's eye perspective, the Bleichenbacher attack is based on two well-known facts about RSA public key encryption.

- On the one hand, it is known that RSA public key encryption (at least when used in textbook form) is susceptible to a simple and straightforward CCA [3]: If an adversary has access to a decryption oracle and is tasked to decrypt a given ciphertext c (i.e., to compute $m = (c^d) \bmod n$ without directly feeding c into the oracle), then he or she can choose a random value r and query the oracle with the now innocent-looking ciphertext $c' = (r^e c) \bmod n$. If the oracle computes $m' = (c'^d) \bmod n$ and returns this value to the adversary, then he or she can recover the plaintext message m by computing $m = (m'r^{-1}) \bmod n$. This is true, because

$$\begin{aligned} m'r^{-1} &\equiv (c'^d)r^{-1} \pmod n \\ &\equiv (r^e c)^d r^{-1} \pmod n \\ &\equiv r^{ed} c^d r^{-1} \pmod n \\ &\equiv r m r^{-1} \pmod n \\ &\equiv r r^{-1} m \pmod n \\ &\equiv m \pmod n \end{aligned}$$

  Mathematically speaking, this CCA exploits the fact that RSA encryption is multiplicatively homomorphic.

5 Historically speaking, the Bleichenbacher attack was the first padding oracle attack that was ever published (refer to Figure 2.26 for an illustration of a padding oracle attack). Before Bleichenbacher published his attack, it was assumed that CCAs were only theoretically relevant and could not be mounted in practice (because decryption oracles were not known to exist). This changed fundamentally when Bleichenbacher published his results. It suddenly became clear that decryption oracles exist, at least in some limited form, and that they pose a real threat to many implementations. As such, the Bleichenbacher attack really changed the way people think about CCAs and the necessity to make encryption systems resistant against CCAs in general and padding oracle attacks in particular.

- On the other hand, it is also known that RSA encryption has the bit security property, meaning that all bits of a plaintext message are equally well protected. This suggests that somebody who is able to decrypt a particular plaintext message bit from a given ciphertext, such as the least significant bit (LSB) [4] or the most significant bit (MSB) [5], can—at least in principle—decrypt the entire ciphertext.

The bit security property of RSA is a dual-edged sword; it is good, because it ensures that all bits of a plaintext message are equally well protected. But it is also bad, because it means that an implementation that leaks a single plaintext message bit can potentially be (mis-)used as an oracle to decrypt the entire ciphertext. In fact, the bit security proof provided in [5] comes with a respective algorithm, and the Bleichenbacher attack can also be seen as an alternative for this algorithm (that basically replaces the leaky implementation with a more powerful padding oracle). This, in turn, means that the adversary can send adaptively chosen ciphertexts to the oracle, and that the oracle returns one bit of information for every such ciphertext, namely whether the plaintext message that results from decrypting it conforms to PKCS #1 block type 2. In the positive case, the ciphertext is said to be *PKCS #1 conforming*, or *PKCS conforming* in short. In what follows, we sometimes ignore the version 1.5 for PKCS #1.

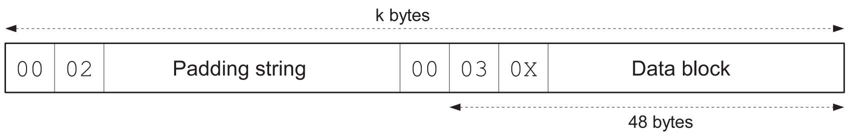

Figure A.1 PKCS #1 version 1.5 block format for RSA encryption (block type 2).

Figure A.1 illustrates the block format of a plaintext message prepared for RSA encryption that is in line with PKCS #1 version 1.5 (block type 2). Its length must be equal to the size of the RSA modulus n (we use k to refer to the respective byte length[6]). Starting from the left side, the block begins with a pair of 0x00 and 0x02 bytes (referring to block type 2), a variable length[7] padding string, another zero byte 0x00, and a 48-byte data block. In many implementations, the data block comprises two bytes referring to the version of the protocol in use and the premaster

6 For 2,048-bit RSA, k is equal to $2048/8 = 256$ and the padding string is thus $256 - 48 - 3 = 205$ bytes long.

7 The padding string must be at least 8 bytes long. Also, it must not include a zero byte, otherwise the PKCS #1 encoding cannot be decoded unambiguously.

secret. The protocol version is included in the data block to mitigate some version downgrade attacks. The first byte of the version field is always set to 0x03, whereas the second byte is set to 0x0X, with X is equal to 0 in SSL 3.0, 1 in TLS 1.0, 2 in TLS 1.1, and 3 in TLS 1.2 (note that TLS 1.3 no longer supports RSA-based key exchange, and hence this type of message has become obsolete meanwhile).

The Bleichenbacher attack exploits the fact that a PKCS conforming ciphertext must decrypt to a plaintext message that is in line with PKCS #1. Most importantly, it must begin with the two bytes 0x00 and 0x02. If an adversary can have many ciphertexts be decrypted, and each PKCS conforming ciphertext yields some information (about the underlying plaintext message), then he or she may perform one operation with the private key d that is unknown to the adversary. This operation can either be the decryption of a given ciphertext or the generation of a digital signature for a message of the adversary's choice. Most importantly, the adversary can decrypt a CLIENTKEYEXCHANGE handshake message in an RSA-based key exchange that he or she has recorded previously. Again, this message includes the premaster secret that is encrypted with the server's public key. If the adversary is able to mount a Bleichenbacher attack against this message using the server as a padding oracle, then he or she may succeed in decrypting it and retrieving the respective premaster secret. Equipped with this secret, the adversary can then decrypt all messages sent and received in the entire session (if a recording of the session is available in the first place).

Table A.1
Possible Checks for PKCS #1 Block Format Type 2 Compliance

No.	Description	Success Probability
1	The first (leftmost) two bytes of m are 0x00 and 0x02	$1/2^{16}$
2	At least 8 bytes after the third byte are not equal to 0x00	$(255/256)^8$
3	m comprises a 0x00 byte after the 11th byte	$1 - (255/256)^{k-10}$
4	The $(k-48)$th byte is equal to 0x00	$1/2^8$
5	The $(k-47)$th and $(k-46)$th bytes refer to a valid version	$5/2^{16}$

To better understand the Bleichenbacher attack, let us see what happens if the adversary chooses a ciphertext c and submits it (in a CLIENTKEYEXCHANGE handshake message) to the server that acts as a padding oracle. The server decrypts c (i.e., computes $m = (c^d) \bmod n$) and checks whether the resulting plaintext message m conforms to PKCS #1 block format type 2. If this is the case, then c is PKCS conforming; otherwise, it is not. Note, however, that this check is not unique, and that it may comprise multiple parts. In the simplest case, the check only verifies the first two bytes (i.e., whether they are equal to 0x00 and 0x02). In more involved cases, it may also verify that the padding string is at least 8 bytes long and does not comprise a 0x00 byte. Finally, it may even check whether the $(k-48)$th byte is equal to 0x00, and whether the $(k-47)$th and $(k-46)$th bytes refer to a valid SSL/TLS protocol version (as mentioned above). There are at least five possible checks summarized in Table A.1—together with their respective success probabilities. Different implementations come along with different checks, and for the outline of the Bleichenbacher attack we restrict ourselves to the first check only. It has a success probability of

$$\frac{1}{2^8} \cdot \frac{1}{2^8} = \frac{1}{2^8 \cdot 2^8} = \frac{1}{2^{8+8}} = \frac{1}{2^{16}}$$

This suggests that in one out of 2^{16} ciphertexts sent to the padding oracle, the oracle will confirm PKCS #1 block format type 2 compliance. If more checks are invoked, then the overall success probability decreases, and this means that the oracle must be queried more often to get back something useful, and hence that the oracle becomes less efficient.

In either case (and due to its function as a padding oracle), the server communicates the result of the check to the adversary, without revealing any other information about the plaintext message m. This communication can occur directly, for example, through alert messages, but it can also occur indirectly, for example, through side-channels. If, for example, c is PKCS conforming, meaning that m conforms to the PKCS #1 block format type 2, then the protocol continues until it recognizes that m is flawed and aborts. Otherwise, it aborts immediately (after having received and decrypted c), and this means that there is a timing difference between the situation that m conforms to PKCS #1 and the situation that it does not. This timing difference, in turn, yields a side-channel that can be exploited in an attack. So one way to mitigate the Bleichenbacher attack is to destroy the timing difference and the respective side-channel (as discussed below).

If the adversary learns that c is PKCS conforming (using only the first check from Table A.1), then he or she knows that the first two bytes of m are 0x00 and 0x02. If it is interpreted as an integer, this means that

$$2B \leq m < 3B$$

for $B = 2^{8(k-2)}$ and k referring to the byte length of the RSA modulus n (i.e., $2^{8(k-1)} \leq n < 2^{8k}$). This also means that m is an integer in the interval $[2B, 3B)$ or $[2B, 3B-1]$. This is the information needed to mount the Bleichenbacher attack. Due to the fact that RSA encryption is multiplicatively homomorphic, the adversary can construct many related ciphertexts that are subject to the same line of argumentation. Each ciphertext c_i ($i \geq 1$) is constructed as $c_i = (s_i^e c) \bmod n$, where s_i is a random (but adaptively chosen) integer and e is the server's public key (together with n). If the padding oracle responds in the affirmative, then the adversary knows that c_i is PKCS conforming, meaning that the two leading bytes of

$$c_i^d \equiv (s_i^e c)^d \equiv (s_i^e)^d (c)^d \equiv (s_i^{ed})(c)^d \equiv s_i m \pmod{n}$$

are 0x00 and 0x02, and hence that this value must again be in the same interval $[2B, 3B)$. The fact that both m and $s_i m$ (for some s_i) are in the same interval allows the adversary to narrow the interval, and the iteration of this procedure leads to nested intervals for m.

Let us consider a mind experiment to illustrate the idea. Assume $x = 100$ and somebody is able to find the following three relations about x:

$$\begin{aligned} 180 &\leq 2x < 220 \\ 240 &\leq 3x < 309 \\ 396 &\leq 4x < 440 \end{aligned}$$

From the first relation, we know that $90 \leq x < 110$, and hence that x must lay in the interval $[90, 110)$; for example, $x \in [90, 110)$. Similarly, we know that $80 \leq x < 103$ and $x \in [80, 103)$ from the second relation, and $99 \leq x < 110$ and $x \in [99, 110)$ from the third relation. Combining all three relations leads to the insight that $99 \leq x < 103$ and $x \in [99, 103)$. If somebody is able to continue the series of relations, then he or she may be able to uniquely determine the value of x.

In a more complex setting, this is actually the working principle of the Bleichenbacher attack. It narrows the interval to a single value, and retrieves m from there. Typically (and according to the analysis provided in [2]), 2^{20}—which is slightly more than one million—related ciphertexts are needed. The Bleichenbacher attack is therefore also known as the *million message attack*. Since the number varies with some implementation details and is subject to optimization, this (alternative) attack name is not descriptive, and hence it is not used in this book.

The algorithm originally proposed by Bleichenbacher to implement the attack is summarized in Algorithm A.1. It takes as input a public RSA key (e, n) and a ciphertext c that is to be decrypted, and it generates as output a plaintext message m that is the result of the decryption operation applied to c. As mentioned above, the attack can also be used to yield a signing operation with the private key d that corresponds to (e, n), but this is not further addressed here.

Algorithm A.1 basically consists of the following four steps that are iterated in some specific way:

- In step 1, c is blinded and turned into a ciphertext c_0 that is PKCS conforming. This is achieved by searching an integer s_0 such that $c_0 = c(s_0)^e \bmod n$ is PKCS conforming. If c is already PKCS conforming (what is usually the

Algorithm A.1 Algorithm That Implements the Bleichenbacher Attack

```
$(e, n), c$
-----------------------------------------------------------------
Step 1: Blinding $c$
    Search integer $s_0$ such that $c(s_0)^e \bmod n$ is PKCS conforming
    $c_0 = c(s_0)^e \bmod n$
    $M_0 = \{[2B, 3B-1]\}$
    $i = 1$
Step 2: Searching for PKCS-conforming messages
    case $(i = 1)$: Search smallest integer $s_1 \geq n/(3B)$ such that $c_0(s_1)^e \bmod n$ is
               PKCS conforming (step 2.a)
        $(i > 1) \wedge (|M_{i-1}| > 1)$: Search smallest integer $s_i > s_{i-1}$ such that $c_0(s_i)^e \bmod n$ is
               PKCS conforming (step 2.b)
        $(i > 1) \wedge (|M_{i-1}| = 1)$: Choose small integers $r_i$ and $s_i$ such that $r_i \geq 2\frac{bs_{i-1}-2B}{n}$
               and $\frac{2B+r_in}{b} \leq s_i \leq \frac{3B+r_in}{a}$, until $c_0(s_i)^e \bmod n$ is
               PKCS conforming (step 2.c)
Step 3: Narrowing the set of solutions
    $M_i \leftarrow \bigcup_{(a,b,r)} \left\{ \left[ \max\left(a, \left\lceil \frac{2B+rn}{s_i} \right\rceil\right), \min\left(b, \left\lfloor \frac{3B-1+rn}{s_i} \right\rfloor\right) \right] \right\}$
    for all $[a, b] \in M_{i-1}$ and $\frac{as_i-3B+1}{n} \leq r \leq \frac{bs_i-2B}{n}$
Step 4: Computing the solution
    If $M_i$ contains only one interval of length 1 (i.e., $M_i = \{[a, a]\}$)
               then $m = a(s_0)^{-1} \bmod n$
               else $i = i + 1$ and go to step 2
-----------------------------------------------------------------
$m$
```

case if c is a CLIENTKEYEXCHANGE message), then s_0 is simply set to one. According to what has been said above, $c_0^d = c^d(s_0)^{ed} = ms_0 = m_0 \pmod n$ is going to be an integer in the interval $[2B, 3B-1]$. This interval is set to M_0 (that yields the starting set of solutions), and the index i is initialized and set to 1. In the subsequent steps, the algorithm tries to find nested intervals for m_0 until an interval with a single value prevails. The algorithm then uses step 4 to retrieve m from m_0.

- In step 2, the algorithm searches for PKCS-conforming messages that may be used for the nested intervals (to distill m_0). Depending on whether $i = 1$ or $i > 1$, and in the second case whether $|M_{i-1}| = 1$ or $|M_{i-1}| > 1$, there are three cases that have to be distinguished and need to be treated differently (steps 2.a – 2.c):

  – If $i = 1$ (meaning that the algorithm is in the first iteration), then the algorithm searches the smallest integer $s_1 \geq n/(3B)$, such that

$c_0(s_1)^e \bmod n$ is PKCS conforming.[8] In Algorithm A.1, this case is referred to step 2.a.

- If $i > 1$ and $|M_{i-1}| > 1$ (meaning that the algorithm is no longer in the first iteration and there are at least two intervals left), then the algorithm searches the smallest positive integer $s_i > s_{i-1}$, such that $c_0(s_i)^e \bmod n$ is PKCS conforming. In Algorithm A.1, this case is referred to step 2.b.
- If $i > 1$ and $|M_{i-1}| = 1$ (meaning that the algorithm is no longer in the first iteration and there is only one interval $M_{i-1} = \{[a, b]\}$ left), then the algorithm chooses small integers r_i and s_i, such that

$$r_i \geq 2\frac{bs_{i-1} - 2B}{n}$$

and

$$\frac{2B + r_i n}{b} \leq s_i \leq \frac{3B + r_i n}{a}$$

until $c_0(s_i)^e \bmod n$ is PKCS conforming. The first condition (regarding r_i) is to make sure that the remaining interval is divided roughly in half. The second condition (regarding s_i) is derived from the fact that $m_0 s_i$ modulo n must be in $[2B, 3B - 1]$, and hence that

$$2B \leq m_0 s_i - r_i n < 3B$$

for some integer r_i.[9] This is equivalent to

$$2B + r_i n \leq m_0 s_i < 3B + r_i n$$

and

$$\frac{2B + r_i n}{m_0} \leq s_i < \frac{3B + r_i n}{m_0}$$

Considering that $a \leq m_0 \leq b$, this can be written as above. In Algorithm A.1, this case is referred to step 2.c.

8 The threshold $n/(3B)$ applies and can be used to optimize the algorithm, because there are no smaller integers that can satisfy the requirement.

9 Note that the middle term could also be written as $m_0 s_i + r_i n$ instead of $m_0 s_i - r_i n$.

- In step 3, the set of solutions is narrowed according to s_i that has been found in step 2. More specifically, the algorithm derives M_i from M_{i-1}, where M_i refers to the set of intervals for m_0 in step i (i.e., m_0 must be in any of these intervals). The derivation starts from the fact that $m_i = m_0 s_i - rn$ for some integer r. This means that $m_0 = (m_i + rn)/s_i$. Because $2B \leq m_i \leq 3B - 1$, a new interval for m_0 can be computed as follows:

$$\frac{2B + rn}{s_i} \leq m_0 \leq \frac{3B - 1 + rn}{s_i} \qquad \text{(A.1)}$$

  This formula yields new constraints for m_0, but it also depends on r that is still unknown. This value refers to the number of times one has to subtract n from the product $m_0 s_i$ to go back in the interval $[0, n-1]$ again. This value cannot be determined precisely (because m_0 and m_i are unknown), but its possible values can be limited based on the fact that m_0 and m_i are both in $[2B, 3B-1]$. From $m_i = m_0 s_i - rn$, it follows that $r = (m_0 s_i - m_i)/n$, and hence that

$$\frac{as_i - 3B + 1}{n} \leq r \leq \frac{bs_i - 2B}{n}$$

  This yields a few possible values for r, and these values can be used to narrow the set of solutions, and to construct M_i. To achieve this, all $[a, b] \in M_{i-1}$ are intersected with the intervals for m_0 that can be derived from r according to relation (A.1). If two intervals are disjunct, then there is no intersection and the respective interval is not going to be part of M_i. Formally, M_i can be constructed as follows:

$$M_i = \bigcup_{(a,b,r)} \left\{ \left[\max\left(a, \left\lceil \frac{2B + rn}{s_i} \right\rceil \right), \min\left(b, \left\lfloor \frac{3B - 1 + rn}{s_i} \right\rfloor \right) \right] \right\}$$

- In step 4, it is checked whether M_i contains only one interval of length 1 (i.e., $M_i = \{[a, a]\}$). If this is the case, then the solution $m = c^d \bmod n$ can be computed from a and s_0 (from step 1) according to $m = a(s_0)^{-1} \bmod n$. Otherwise, i is incremented by 1 and the algorithm returns to step 2.

When Bleichenbacher first implemented the attack, he used alert messages to decide whether a chosen ciphertext was PKCS conforming or not. But he also pointed out that timing differences could be used instead. So when the TLS designers tried to mitigate the attack (in the TLS 1.0 specification that was released in the subsequent year), they made the alert messages unique and gave the following hints about how to disable timing differences:[10]

10 This text first appeared at the bottom of page 44 of the TLS 1.0 specification.

> "The best way to avoid vulnerability to this attack is to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks. Thus, when it receives an incorrectly formatted RSA block, a server should generate a random 48-byte value and proceed using it as the premaster secret. Thus, the server will act identically whether the received RSA block is correctly encoded or not."

These hints were refined in TLS 1.1, but the respective specification still recommends to generate a random 48-byte value and proceed using it as the premaster secret only when the server receives an incorrectly formatted RSA block. But there is a subtlety to consider here: If the server generates a random number, then this takes some time, and this time delay may create a side-channel. If the response is delayed, then the server is more likely to have received an incorrectly formatted RSA block. To mitigate this (arguably small) timing side-channel, it is necessary to always generate a random number—independent from whether the received RSA block is correctly formatted or not. This implementation hint (of always generating a random value, even if the decrypted block is correctly formatted) was included in the TLS 1.2 specification (in 2008), making people believe that the Bleichenbacher attack was successfully mitigated.

Unfortunately, this was not the end of the story and there are quite a few variants of the Bleichenbacher attack that affect the SSL/TLS ecosystem and have had an impact on the TLS design. Let us briefly explain some of them to complete the picture.

A.1.1 DROWN, ROBOT, and CATs

In 2008, the TLS 1.2 specification (with the abovementioned implementation hint) was released. Only a couple of years later, in 2016, it was shown that there were many TLS 1.2 servers that yet implemented the hint, but were still susceptible to a subtle variant of the Bleichenbacher attack named DROWN[11]—an acronym derived from *decrypting RSA with obsolete and weakened encryption* [6]. If a TLS 1.2 server—for the sake of backwards-compatibility—still supported SSL 2.0 and used the same RSA key for key exchange, then the resulting implementation could be susceptible to a sophisticated cross-protocol attack.[12]

The DROWN attack starts from the abovementioned implementation hint. Remember that the hint suggests to use a random number (instead of the premaster secret) if and only if the received value refers to an incorrectly formatted RSA block.

11 https://drownattack.com.

12 In a more realistic setting, it may also be the case that the server supporting SSL 2.0 is distinct from the target server that supports TLS 1.2. The only requirement is that they share the same RSA key for key exchange.

So if an adversary can submit the same ciphertext c twice and learn in each case whether the server thereafter uses the same master key, then he or she can distinguish the legitimate case (where c is a correctly formatted RSA block and the master key is the same) and the illegitimate case (where c is an incorrectly formatted RSA block and the master keys are different). This reintroduces the vulnerability and reenables the Bleichenbacher attack.

So the key question for the adversary is how to distinguish the two cases. In TLS 1.2 (and its predecessors), there is no possibility; but in SSL 2.0, there is. The SSL 2.0 handshake protocol is slightly different from the SSL 3.0 and all TLS 1.0 to 1.2 handshake protocols. Most importantly, after having received c in a CLIENTMASTERKEY message (that is the SSL 2.0 equivalent of the CLIENTKEYEXCHANGE message), the server computes the encryption key and immediately sends back a SERVERVERIFY message. This message comprises the client nonce in encrypted form. If the adversary can enforce the use of an export cipher (that is restricted to 40 bits), then he or she may brute-force the encryption key from the SERVERVERIFY message. (In SSL 3.0 and all TLS handshake protocols this is not possible, because there is no such SERVERVERIFY message.) If this is done for either of the c values, then the adversary can distinguish the two cases, tell whether c is correctly formatted, and hence mount the Bleichenbacher attack. This is the general plan, but there are still a few subtleties that must be taken into account. For example, the formats of a CLIENTMASTERKEY message (in SSL 2.0) and a CLIENTKEYEXCHANGE message (in TLS 1.2) are distinct, and hence it is not obvious how a c (taken from a TLS 1.2 handshake) can be fed into the SSL 2.0 oracle. There are mathematical tricks that can be played here. Also, there are some statistical limitations that lead to the situation in which the attack only works for 1 out of 1,000 ciphertexts. Last but not least, the DROWN attack also exploited some implementation flaws in the OpenSSL libraries of that time. These flaws have mostly been remedied and are not further addressed here.

The bottom line is that DROWN has shown that the patch to mitigate the Bleichenbacher attack (as provided in the implementation hint) is not fool-proof, and that there may still be ways to mount the Bleichenbacher attack or some variants thereof. Finding such attacks has become a fairly popular research area.

- In 2017, for example, a group of researchers identified some vulnerable implementations from at least seven vendors, including F5, Citrix, and Cisco, in an attack named *Return of Bleichenbacher's Oracle Threat* (ROBOT[13]) [7];
- In 2018, another group of researchers identified several other vulnerable implementations in so-called *cache-like attacks* (CATs) [8].

13 https://robotattack.org.

The story continues until today, and from time to time new research groups find new attacks in other application domains. The story thus continues.

A.1.2 Klíma-Pokorný-Rosa Attack

So far, we have only talked about the first check from Table A.1. But five years after Bleichenbacher's original publication, in 2003, a group of three Czech cryptologists—Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa—published an attack that additionally takes into account and exploits the fifth check [9]. According to the names of the researchers, this attack is known as the *Klíma-Pokorný-Rosa attack*.

The attack starts from the observation that some SSL/TLS implementations check the protocol version bytes (i.e., the $(k-47)$th and $(k-46)$th bytes) if and only if the first check succeeds. A random value is generated anyway (to serve as the premaster secret in case the first check fails). But depending whether the first check succeeds, the fifth check is additionally performed. This introduces a timing difference between the case that the first check succeeds (and the fifth check is performed) and the case that it fails (and hence the fifth check is not performed). This timing difference yields a side-channel that can be exploited to mount the Bleichenbacher attack again (this is in spite of the implementation hint).

To mitigate the Klíma-Pokorný-Rosa attack, it must be ensured that incorrectly formatted message blocks and/or mismatched version numbers are treated in a manner indistinguishable from correctly formatted blocks. There are several ways to achieve this, and two possibilities are outlined on pages 58 and 59 of the TLS 1.2 protocol specification; they are not repeated here.

A.1.3 Manger Attack

All attacks addressed so far exploit some characteristics of PKCS #1 version 1.5. Consequently, a proper strategy to mitigate the attacks is to change the respective padding format in a way that makes RSA encryption secure against CCA2. This was done around the same time when Bleichenbacher published his results. In 1998, PKCS #1 version 1.5 was updated to version 2.0 [10] and a new padding scheme, named optimal asymmetric encryption padding (OAEP) was adapted [11]. In spite of the fact that OAEP had been shown to be provably secure against CCA2 in the random oracle model, the industry did not rush to implement this new standard. Instead, most implementations stayed with PKCS #1 version 1.5 and tried to get along with the Bleichenbacher attack as described so far.

To make things worse (from the perspective of the new version of PKCS #1), it was shown soon after the release of PKCS #1 version 2.0 that some widely deployed implementations were still susceptible to some variants of the Bleichenbacher attack. For example, the Manger attack [12] exploits the fact that—according to PKCS #1 version 2.0—a properly padded plaintext message must begin with a zero byte 0x00. This means that the message interpreted as an integer must not be larger than a certain threshold.[14] What Manger now observed was that some implementations prematurely aborted when the message (if interpreted as an integer) was too big. This means that there was a timing difference between the case in which the message is smaller than the threshold and the case in which it is larger, and this timing difference yields another oracle that can be used to mount a highly efficient Bleichenbacher-like attack.

At first sight, the Manger attack seems to contradict the provable security of RSA-OAEP used in PKCS #1 version 2.0. But the security claim is not that all possible implementations are secure, and hence a cryptographic system that is secure in theory can still be implemented in a way that is susceptible to specific (side-channel) attacks. This a common theme in applied cryptography, and to mitigate the Manger attack, it is important to avoid any information leakage about the fact that a message is larger than the threshold. This is difficult to achieve in general and must be considered for each implementation individually.

As already mentioned in Section 2.4, due to the Klíma-Pokorný-Rosa, Manger, and a few other attacks, PKCS #1 was updated in 2003 to version 2.1 [13], and in 2012 to version 2.2 [14]—this time not for security reasons. This is where we stand today, and anybody using RSA for encryption should employ the padding scheme specified in PKCS #1 version 2.1 or 2.2, and still make sure that the implementation is free of timing and other side-channels. This is challenging (to say the least) and to some extent even open-ended. Hence, the story is likely to continue and Bleichenbacher attacks will remain an issue as long as RSA-based key exchanges prevail. Again, this is no longer the case in TLS 1.3, and hence at least this type of attack is no longer an issue (as long as a version downgrade is impossible to perform).

A.2 VAUDENAY ATTACK

The Bleichenbacher attack affects PKCS #1 as used in asymmetric encryption. As such, it is relevant for asymmetric encryption only and is not directly applicable to symmetric encryption. However, Serge Vaudenay published a paper on padding oracle attacks against CBC padding in 2002 [15]—four years after Bleichenbacher published his work. The resulting Vaudenay attack is best seen as the symmetric analog of the Bleichenbacher attack. It had only been theoretically interesting,

14 The threshold here is $B = 2^{8(k-1)}$, where k refers to the byte length of the RSA modulus n.

until it was shown one year later that it can actually be mounted against real-world applications using SSL/TLS [16]. Since then, the feasibility of the Vaudenay attack and some variants has been demonstrated in many other application settings where CBC mode is used (e.g., [17, 18]). Such attacks pose a real threat and implementations are better shown to be resistant against them. Luckily, the general trend goes away from using CBC mode in favor of modes of operation that support authenticated encryption, such as GCM and CCM.

If a block cipher is operated in CBC mode, then some padding may be required to make sure that the plaintext message length is a multiple of the block length. In the case of DES and 3DES, for example, the block length is 64 bits or 8 bytes. In the more likely case of AES, the block length is 128 bits or 16 bytes. In either case, the last block of the plaintext must be padded to comprise this number of bytes. In theory, there are many padding schemes that can be used for this purpose. In practice, however, PKCS #7 provides the most widely deployed padding scheme used in the field:[15] The last block of a plaintext message is then aligned with the block length by repeatedly filling it with a byte that refers to the length of the padding. To make the padding unambiguous, it is necessary to add a full block of padding, even if the plaintext message is already aligned with the block length. Otherwise, it would be impossible to distinguish a message that is aligned with the block length and that contains bytes that look like valid padding from a message that is properly padded. If, for example, a message had a final block that ended with 0x01, then it would be unclear whether 0x01 is part of the message or represents the padding byte.

The need to add a dummy padding block if the length of a plaintext message is a multiple of the block length also applies to SSL/TLS. There is, however, a subtle difference: While PKCS #7 requires that the padding byte that is repeatedly written refers to the actual length of the padding, SSL/TLS padding requires that this byte refers to the padding length minus one. This is a minor issue that does not really matter, and it is therefore often neglected when people talk about SSL/TLS padding.[16] If, for example, the padding were one byte long, then the SSL/TLS padding byte would be 0x00 (instead of 0x01 as with original PKCS #7). If the padding were two bytes, then the padding byte would be 0x01, and this byte would

15 PKCS #5 employs the same padding scheme but is restricted to a block length of 8 bytes (whereas PKCS #7 supports variable block lengths up to 255 bytes).

16 There are different explanations to justify this deviation from PKCS #7. The most obvious one is that the SSL protocol was originally designed in a time where conformance to PKCS in general, and PKCS #7 in particular, was not so important. So the designers of the SSL protocol argued that the last byte in the padding refers to the padding length, while the actual padding refers to the bytes between the message and this length indicator. Following this line of argumentation leads to a situation in which the padding length is equal to the actual padding length minus one. From a theoretical perspective, it does not really matter whether SSL/TLS or PKCS #7 padding is used, and so the two possibilities are equivalent for all practical purposes.

be written twice. This continues until the case in which an entire block is appended. In our example (using AES with a block length of 16 bytes), the byte that is repeated 16 times would be 0x0F. If the block length was bigger than 16 bytes, then bigger values would also be possible. Because the maximal value of a byte is 256 (referring to 0xFF), 256 bytes is the biggest possible block length that is supported. This is equally true for PKCS #7 and SSL/TLS padding.

So CBC padding in SSL/TLS requires the last plaintext block to end with one of the following 16 possible byte sequences:

$$\begin{array}{c}
\text{0x00} \\
\text{0x01 0x01} \\
\text{0x02 0x02 0x02} \\
\text{0x03 0x03 0x03 0x03} \\
\ldots \\
\underbrace{\text{0x0F 0x0F} \ \ldots \ \text{0x0F 0x0F 0x0F}}_{16}
\end{array}$$

This fact is exploited by the Vaudenay attack. Note, however, that a similar attack could be mounted against any padding scheme (which makes it possible to distinguish a valid padding from an invalid one). If one randomly chooses a ciphertext block, then it is very unlikely that the decryption operation yields a plaintext block that is properly padded. In the case of CBC padding, for example, properly padded means that it must end with any of the byte sequences itemized above.

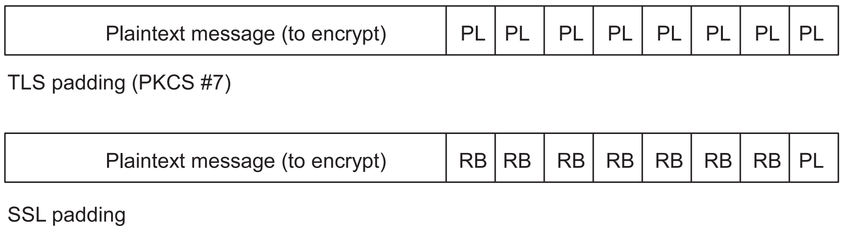

Figure A.2 TLS and SSL padding.

For the sake of completeness, we mention here that SSL uses a slightly different padding format than TLS. While in TLS padding all padding bytes refer to the padding length minus one, in SSL padding this is true only for the last byte of the padding. All other padding bytes can be randomly chosen and comprise arbitrary values. This is illustrated in Figure A.2 (where PL stands for the padding length and RB stands for a random byte). As further explored in Section A.4, this simplifies padding oracle attacks considerably and has even been exploited in a devastating attack known as POODLE.

What Vaudenay showed in his work is that an adversary who has access to a PKCS #7 padding oracle can use it to efficiently decrypt CBC encrypted messages [15]. At first sight, this seems surprising, because the amount of information the oracle provides is very small (i.e., one bit). Also, the resulting attack is purely theoretical, because there are many requirements that need to be fulfilled so that a padding oracle can actually be exploited in the field. However, Vaudeney and a few other researchers showed how to construct a padding oracle that can work in practice—at least under certain circumstances [16]. So whether an adversary really has access to a padding oracle mainly depends on the implementation under attack, and there are many details to consider.

In what follows, we just introduce the basic idea of the Vaudenay attack, and we don't delve into the (sometimes subtle) details. We assume an adversary who has a k-byte CBC-encrypted ciphertext block C_i that he or she wants to decrypt (without knowing the proper decryption key). So k represents the block length of the block cipher in number of bytes (e.g., $k = 16$ for AES). Maybe the adversary knows that C_i comprises some secret information, such as a user password, a bearer token, or something similar. If the adversary does not know that a particular block comprises the secret information, then it may still be possible to repeat the attack multiple times. Anyway, we can either refer to the entire block C_i or to a specific byte in this block. In the second case, we use an index j with $1 \leq j \leq k$ in square brackets to refer to a particular byte. So $C_i[j]$ refers to the jth byte in block C_i, where j is an integer between 1 and k. This also means that C_i can be written as follows:

$$C_i = C_i[1] \parallel C_i[2] \parallel C_i[3] \parallel \ldots \parallel C_i[k-1] \parallel C_i[k]$$

The same notation applies to the underlying plaintext block P_i (which, by the way, is the target of the attack):

$$P_i = P_i[1] \parallel P_i[2] \parallel P_i[3] \parallel \ldots \parallel P_i[k-1] \parallel P_i[k]$$

Referring to the explanation of the Bleichenbacher attack, a padding oracle attack is a CCA2 in which an adversary sends arbitrary ciphertexts to the padding oracle, and for each of these ciphertexts the oracle returns one bit of information, namely whether the underlying plaintext is properly padded or not. In the case of a Vaudenay attack, the target ciphertext block (i.e., the ciphertext block the adversary wants to decrypt) is CBC-encrypted with a k-byte block cipher. If a block cipher is operated in CBC mode, then the recursive formula to decrypt a ciphertext block C_i ($i \geq 1$) looks as follows:

$$P_i = D_K(C_i) \oplus C_{i-1} \tag{A.2}$$

The decryption starts with $i = k$ and ends with $i = 1$ (using $C_0 = IV$, where IV refers to an initialization vector). Equation (A.2) basically means that C_i is decrypted by first decrypting it (using the decryption function $D(\cdot)$ with the key K) and then adding the result bitwise modulo 2 to the predecessor ciphertext block C_{i-1}. This is illustrated in Figure A.3. This figure should be kept in mind when going through the attack step by step.

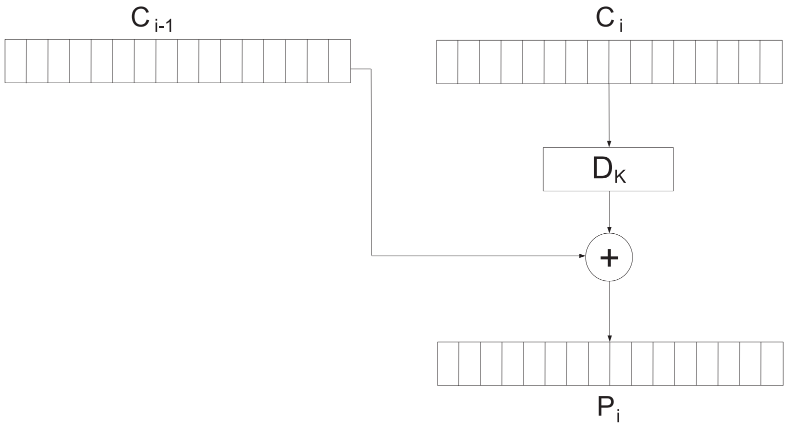

Figure A.3 CBC decryption of C_i.

To mount a Vaudenay attack and decrypt the ciphertext block C_i, the adversary can sequentially go through each byte of C_i (in reverse order) and process it individually (i.e., $C_i[k], C_i[k-1], \ldots, C_i[1]$). This simplifies the attack considerably and brings the complexity from 2^{128} (if an entire block needs to be attacked simultaneously) down to $16 \cdot 2^8 = 2^4 \cdot 2^8 = 2^{12}$ (if the $k = 16$ bytes of the block can be attacked individually). In contrast to 2^{128}, 2^{12} is a very moderate complexity that can be handled, and hence the respective attack is feasible and real.

In each step of the attack, the adversary combines the to-be-decrypted ciphertext block C_i with an artificially crafted predecessor ciphertext block C' to form a two-block message $C' \parallel C_i$. This message is then sent to the padding oracle, where it is decrypted according to (A.2). The result of this decryption step is a two-block plaintext message $P_1' \parallel P_2'$. As is usually the case in a padding oracle attack, this message is not revealed to the adversary. Instead, the oracle only informs the adversary whether P_2' is properly padded or not. P_1' is not revealed to the adversary, and it is only used to enable the attack.

The adversary knows two pieces of information: On the one hand, he or she knows from the CBC decryption equation (A.2) that

$$P_2' \quad = \quad D_K(C_i) \oplus C'$$

On the other hand, he or she also knows from CBC encryption that

$$C_i = E_K(P_i \oplus C_{i-1})$$

Combining these two pieces of information (i.e., replacing C_i in the first equation with the right side of the second equation), it follows that

$$\begin{aligned} P_2' &= D_K(C_i) \oplus C' \\ &= D_K(E_K(P_i \oplus C_{i-1})) \oplus C' \\ &= P_i \oplus C_{i-1} \oplus C' \end{aligned}$$

In this equation, the adversary knows C_{i-1} and can control C', but he or she does not know anything about P_i or P_2'. So there is one equation with two unknown values, and this means that neither P_2' nor P_i can be determined. It is now important to note that $P_2' = P_i \oplus C_{i-1} \oplus C'$ must hold at the block level and that it must also hold at the byte level (because the only operation that is used is the bitwise addition modulo 2). This means that

$$P_2'[j] = P_i[j] \oplus C_{i-1}[j] \oplus C'[j] \tag{A.3}$$

must hold for every byte $1 \leq j \leq k$ (where j represents the byte position in the target ciphertext block). At the byte level, the adversary is in a much better position because he or she may know something about the padding used in $P_2'[j]$. This knowledge can be turned into an efficient algorithm to find $P_i[j]$, and hence to decrypt $C_i[j]$. This idea applies to all k bytes of C_i, and hence the bytes can be attacked individually in sequence.

Let us now delve more deeply into the attack and the way it actually works. We assume an adversary who faces a ciphertext block C_i that is CBC-encrypted with a 16-byte block cipher (i.e., $k = 16$). Referring to what we said before, the adversary starts with the rightmost byte $C_i[k]$ (i.e., $C_i[16]$). The respective attack scenario is illustrated in Figure A.4. Note that the adversary also knows C_{i-1} because this ciphertext block has been transmitted together with C_i; but this block is not illustrated in Figure A.4. The adversary can now craft various two-ciphertext-block messages $C' \parallel C_i$ that all look similar: The second block is always the target ciphertext block, whereas the first block is different for every message. In fact, these blocks differ only in their rightmost byte (i.e., $C'[16]$). All other bytes of C' are set to 0x00. So C' looks as follows:

$$C' = \underbrace{00 \parallel 00 \parallel \ldots \parallel 00 \parallel 00 \parallel 00}_{15} \parallel C'[16]$$

The adversary can send as many such two-block ciphertext messages $C' \parallel C_i$ as he or she likes to the padding oracle (one after the other). Normally, the oracle decrypts $C' \parallel C_i$ to $P_1' \parallel P_2'$ and then decides whether the plaintext is properly padded or not. It is properly padded if $P_2'[16]$ is equal to 0x00. There are other possibilities, but they are less likely to occur. This means that the adversary can do an exhaustive search for $C'[16]$ until the padding oracle yields an affirmative response. In the worst case, the adversary has to try out all 256 possible values for $C'[16]$. However, on average, 128 tries are going to be sufficient.

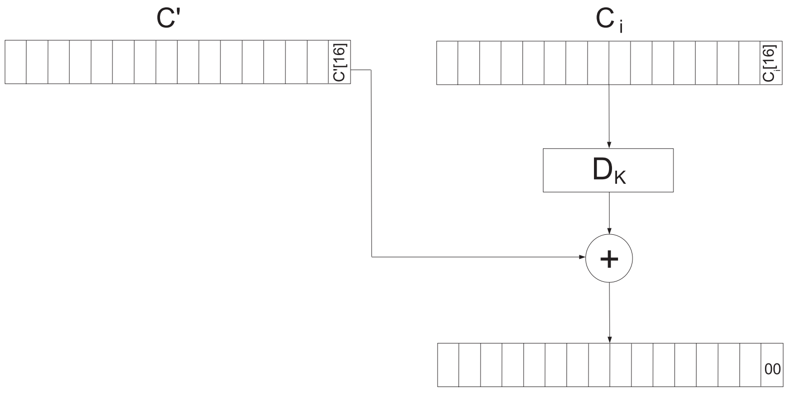

Figure A.4 CBC padding attack against $C_i[16]$.

According to (A.3), the adversary now knows that

$$P_2'[16] = 00 = P_i[16] \oplus C_{i-1}[16] \oplus C'[16]$$

and this means that

$$\begin{aligned} P_i[16] &= 00 \oplus C_{i-1}[16] \oplus C'[16] \\ &= C_{i-1}[16] \oplus C'[16] \end{aligned}$$

As the adversary knows $C_{i-1}[16]$ and has just found $C'[16]$ (i.e., the byte value that yields a valid padding), he or she is now able to determine $P_i[16]$ and thus decrypt $C_i[16]$ without knowing the decryption key. This means that he or she has just decrypted one byte of the target ciphertext block C_i and that the attack can be continued to decrypt the other bytes of C_i in exactly the same way.

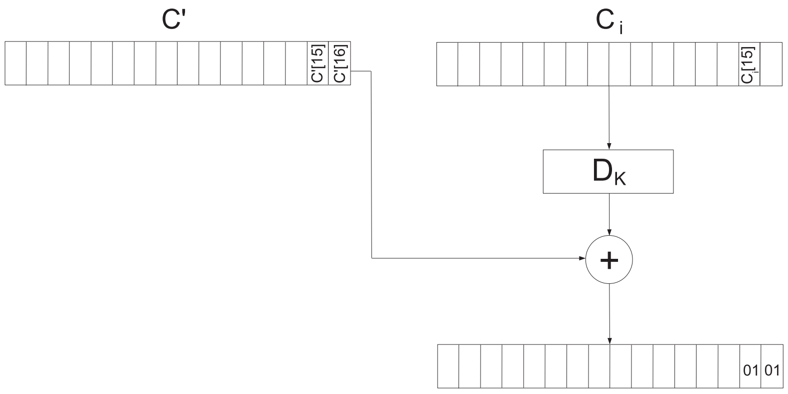

Figure A.5 CBC padding attack against $C_i[15]$.

In the next step, the adversary targets $C_i[15]$. The respective situation is illustrated in Figure A.5. Again, the adversary can craft various two-ciphertext-block messages $C' \parallel C_i$ that differ in the last two bytes of C' (i.e., $C'[15]$ and $C'[16]$) and send them to the padding oracle. Because the last but one byte is the target of this step, a different padding is most likely to occur; for example, 0x01 0x01 instead of 0x00. This also means that $C'[16]$ needs to be adapted a little bit. From $P_2'[16] = 01 = P_i[16] \oplus C_{i-1}[16] \oplus C'[16]$ it follows that $C'[16] = 01 \oplus P_i[16] \oplus C_{i-1}[16]$.

This adaption is not illustrated in Figure A.5. Once it is done, the adversary can again mount an exhaustive search for a ciphertext block C' that yields a valid padding after decryption. C' comprises 14 zero bytes, one byte $C'[15]$ that is incremented, and the now adapted (and new) value of $C'[16]$:

$$C' = \underbrace{00 \parallel 00 \parallel \ldots \parallel 00 \parallel 00 \parallel 00}_{14} \parallel C'[15] \parallel C'[16]$$

Again, the adversary has to try out all 256 possible values for $C'[15]$ in the worst case and 128 values in the average case. After having found a value for $C'[15]$ that causes the decryption of $C' \parallel C_i$ to be properly padded, the adversary knows that

$$P_2'[15] = 01 = P_i[15] \oplus C_{i-1}[15] \oplus C'[15]$$

This, in turn, means that $P_i[15] = 01 \oplus C_{i-1}[15] \oplus C'[15]$. As the adversary knows $C_{i-1}[15]$ and has just found $C'[15]$, he or she can now determine $P_i[15]$, and hence decrypt $C_i[15]$ without knowing the proper decryption key. This, in turn, means that he or she has now already decrypted two bytes of C_i, namely $C_i[16]$ and $C_i[15]$.

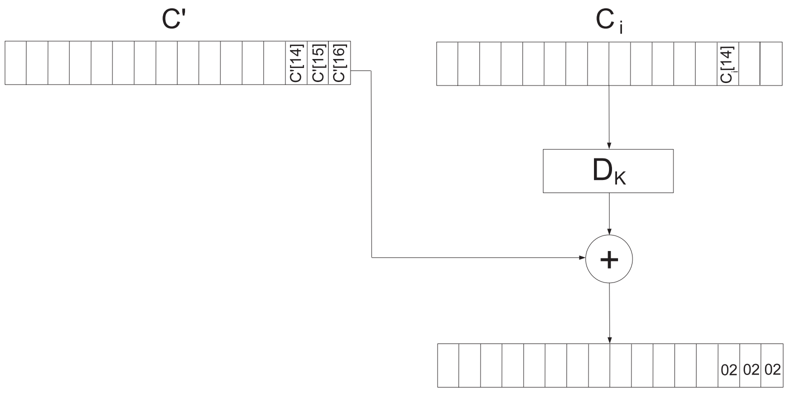

Figure A.6 CBC padding attack against $C_i[14]$.

To continue the attack, the adversary goes for $C_i[14]$. This is illustrated in Figure A.6. This time, the padding is valid if the three last bytes of P_2' (i.e., $P_2'[14]$, $P_2'[15]$, and $P_2'[16]$) are all equal to 0x02. As before, the adversary needs to adapt the values of $C'[15]$ and $C'[16]$ before he or she can mount an exhaustive search for

$C'[14]$:

$$C'[15] = 02 \oplus P_i[15] \oplus C_{i-1}[15]$$
$$C'[16] = 02 \oplus P_i[16] \oplus C_{i-1}[16]$$

So C' comprises 13 zero bytes, one byte $C'[14]$ that is incremented, and the adapted (and new) values for $C'[15]$ and $C'[16]$:

$$C' = \underbrace{00 \parallel 00 \parallel \ldots \parallel 00 \parallel 00 \parallel 00}_{13} \parallel C'[14] \parallel C'[15] \parallel C'[16]$$

Using the ciphertext block C', the adversary can mount an exhaustive search for $C'[14]$. The padding oracle yields an affirmative response if the decryption of $C' \parallel C_i$ is properly padded. The adversary then knows that

$$P'_2[14] = 02 = P_i[14] \oplus C_{i-1}[14] \oplus C'[14]$$

This, in turn, means that $P_i[14] = 02 \oplus C_{i-1}[14] \oplus C'[14]$. As the adversary knows $C_{i-1}[14]$ and has just found $C'[14]$, he or she can determine $P_i[14]$. This, in turn, means that he or she has decrypted $C_i[14]$.

The attack continues for $C_i[13], C_i[12], \ldots, C_i[1]$, until the entire ciphertext block C_i is decrypted. This is not explicated here, but can be done as an exercise. Also, there are several websites that provide numerical examples for the Vaudeney attack and how it proceeds.[17]

The bottom line is that the adversary can decrypt the entire ciphertext block C_i without knowing the proper decryption key. All the adversary must have access to is a padding oracle. In theory, there are several possibilities to instantiate such an oracle. In TLS 1.0, for example, there are two distinct alert messages for a decryption failure (i.e., `decryption_failed` with code value 21) and a MAC verification failure (i.e., `bad_record_mac` with code value 20).[18] This allows an adversary to distinguish the case that the decryption failed due to an invalid padding and the case that the decryption was successful (and hence the padding was valid) but the MAC verification was not. Being able to make this distinction is at the core of Vaudenay's attack. The adversary can send encrypted records to the victim, and for each of these records the victim responds with whether the decryption (due to an incorrect padding) or the MAC verification fails.

17 Such an example is provided, for example, at https://blog.skullsecurity.org/2013/a-padding-oracle-example.

18 As outlined in Table 2.7, there is only the bad_record_mac in SSL 3.0. Also, the decryption_failed alert message that was introduced in TLS 1.0 was abandoned in TLS 1.1 (mainly to make Vaudeney attacks more difficult to mount).

This works in theory. But in practice, it is not clear whether such an attack can actually be mounted. Even if TLS 1.0 is used, the output of the padding oracle is encrypted and not directly accessible to the adversary. Also, a connection is usually aborted prematurely once an error condition has occurred and a respective fatal alert message has been sent. This severely limits the feasibility of the attack. But there are still situations in which a Vaudenay attack is feasible. In a 2003 follow-up paper, Vaudenay and his research colleagues showed how such an attack can be mounted against a password transmitted over an SSL/TLS connection [16]. Such a situation occurs, for example, if an IMAP client uses SSL/TLS to securely connect to an IMAP server. The attack cleverly combines three ideas:

- First, the fact that a successful MAC verification requires significantly more processing time than an early abortion of the protocol due to invalid padding yields a timing channel that can be exploited (even without seeing the respective alert messages in the clear). To optimize the timing channel, the message size can be increased to its maximum value.
- Second, an attack can employ multiple SSL/TLS sessions simultaneously, if the secret that needs to be decrypted (e.g., an IMAP password) always appears at the same location within a session.
- Third, knowledge about the distribution of the plaintext messages may be turned into a dictionary attack that may help to more efficiently determine low-entropy secrets.

Contrary to the first paper (that was theoretically important), the second paper was practically relevant and made press headline. In fact, the resulting pressure originating from the media became so strong that the designers of the TLS protocol had to take precautions to remedy the situation and to mitigate the respective risks. In fact, they had to update TLS 1.0 in a way that better protects the protocol against Vaudenay-type attacks.

An obvious protection mechanism is to make it impossible for an adversary to distinguish the two abovementioned alert messages. This was proposed by Bodo Möller in a 2003 posting to the OpenSSL mailing list.[19] In fact, Möller recommended neglecting the distinction between the two alert messages and sending a `bad_record_mac` alert message in either case (so a `bad_record_mac` alert message must be returned if a `TLSCiphertext` fragment decrypts in an invalid way—either because its length is not an even multiple of the block length or its padding, if checked, is not correct). Instead of only suppressing

19 http://www.openssl.org/∼bodo/tls-cbc.txt.

`decryption_failed` alert messages, Vaudenay suggested making alert messages time-invariant by simulating a MAC verification even if a padding error occurs and to add random noise to the time delay [16]. In the TLS 1.1 specification both recommendations—the one of Möller and the one of Vaudenay et al.—were taken into account. In fact, the `decryption_failed` alert message was made obsolete, and the TLS 1.1 specification in RFC 4346 further requires that

> "Implementations must ensure that record processing time is essentially the same whether or not the padding is correct. In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet. For instance, if the pad appears to be incorrect, the implementation might assume a zero-length pad and then compute the MAC. This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

To avoid a timing channel, the specification requires a MAC to be computed, even if the padding is incorrect. This sounds simple and straightforward, but it is not: If the padding is incorrect, then it is not defined on what data the MAC should be computed. More specifically, the padding length is undefined, and hence the padding can either be short or long. The TLS 1.1 and 1.2 specifications make this uncertainty explicit and recommend computing a MAC as if there were a zero-length pad. But as quoted above from RFC 4346, "this leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment." Furthermore, it is argued that this timing channel "is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal."

This line of argumentation was commonly accepted in the community until Nadhem J. AlFardan and Kenny Paterson demonstrated the opposite in 2013 [19]. In fact, they showed empirically that padding oracle attacks still exist—even if the defense strategy of the TLS specification is put in place. This is particularly true for DTLS, but—in some limited forms—it is also true for TLS. For the reasons discussed below, the attacks are referred to as *Lucky Thirteen*, or *Lucky 13* in short.[20]

To understand the working principles of the Lucky 13 attacks, we have to go back to the HMAC construction (Section 3.2.1.1). As its name suggests, an HMAC provides message authentication; it is computed and appended to the message prior to encryption (following the AtE approach). The message that is authenticated comprises the fragment field of the `TLSCompressed` structure and 13 bytes of

20 The attacks are also documented in CVE-2013-0169.

additional data that are prepended. More specifically, the 13 bytes comprise an 8-byte sequence number field and the usual 5-byte header of the `TLSCompressed` structure (comprising type, version, and length fields). So the point to take away is that 13 bytes are prepended to the message before it is authenticated (and a respective HMAC value is computed and appended). In the analysis of the attacks it was shown that this number of bytes is advantageous and hence a lucky choice for the adversary, and hence the attack was named Lucky 13—maybe also in lack of a better and more descriptive name.

The length of the HMAC value depends on the cryptographic hash function in use. If MD5 is used, then the length is 16 bytes (128 bits). If SHA-1 is used, then the length is 20 bytes (160 bits). The use of a function from the SHA-2 family would lead to an even longer HMAC value. In spite of their different lengths, almost all cryptographic hash functions in use today (except the recently standardized functions from the SHA-3 family) follow a construction that was independently proposed by Ralph C. Merkle and Ivan B. Damgård in the late 1980s [20, 21].[21] The respective Merkle-Damgård construction uses a compression function that is iteratively applied to 64-byte chunks of the message (where each output of the compression function is chained into the next step). The construction also employs an encoding step known as Merkle-Damgård strengthening. It ensures that the data for which a hash value is computed is padded so that its length is aligned with a 64-byte boundary. The padding consists of an 8-byte length field and at least one byte of padding. This means that the minimal padding length in the Merkle-Damgård construction is 9 bytes.

If an HMAC value is computed with an iterative hash function (using the Merkle-Damgård construction), then the respective implementation is going to have a distinctive timing profile: Messages of length up to $64 - 9 = 55$ bytes can be encoded into a single 64-byte block, meaning that the respective HMAC value can be computed with only 4 evaluations of the compression function. If a message contains between 56 and $55 + 64 = 119$ bytes, then the respective HMAC value can be computed with 5 evaluations of the compression function. In general, an additional evaluation of the compression function is needed for each additional 64-byte block of data. Generally speaking, if n refers to the byte-length of a message, then $\lceil \frac{n-55}{64} \rceil + 4$ evaluations of the compression function are required. Depending on the implementation, the running time required to compute an HMAC value may leak some information about the length of the (unpadded) message, perhaps up to a resolution of 64 bytes. This is the small timing channel referred to earlier that is exploited in the Lucky 13 attacks. The attacks therefore represent timing attacks in which an adversary—acting as a MITM—can read and inject self-crafted ciphertext messages. In cryptanalysis, this means that Lucky 13 is actually a CCA.

21 Both papers were presented at CRYPTO ’89.

The first attack is conceptually simple and represents a *distinguishing attack* against CBC encryption. In such an attack, the adversary chooses a pair of equally long plaintext messages (P_0, P_1) and has one of these messages encrypted according to the rules of TLS record processing (meaning that the message is authenticated, padded, and CBC encrypted). The resulting ciphertext C_d is handed over to the adversary, and the adversary's task is to decide the bit value of d (i.e., whether C_d is the encryption of P_0 or P_1) with a success probability that is significantly greater than one half.[22] An adversary can mount such an attack, for example, by choosing the following two plaintext messages:

- P_0 that consists of 32 arbitrary bytes (AB) followed by 256 copies of 0xFF;
- P_1 that consists of 287 arbitrary bytes followed by 0x00.

Visually speaking, these constructions can be represented as follows:

$$\begin{aligned} P_0 &= \underbrace{\text{AB} \parallel \text{AB} \parallel \ldots \parallel \text{AB}}_{32} \parallel \underbrace{\text{0xFF} \parallel \text{0xFF} \parallel \ldots\ldots\ldots \parallel \text{0xFF}}_{256} \\ P_1 &= \underbrace{\text{AB} \parallel \text{AB} \parallel \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \parallel \text{AB}}_{287} \parallel \text{0x00} \end{aligned}$$

Both messages are equally long (i.e., 288 bytes), and hence fit into 18 plaintext blocks (for a 128-bit or 16-byte block cipher like AES). The adversary submits (P_0, P_1) for encryption and gets back a TLS record that consists of a header and a ciphertext C_d, where C_d stands for a CBC encrypted version of P_d (that can be either P_0 or P_1), and the encoding step adds a MAC and some padding to the message. Because the end of P_d aligns with a block boundary, the additional MAC and padding bytes are encrypted in separate blocks from P_d. This means that the adversary can form a new ciphertext block C'_d with the same header but without the additional MAC and padding bytes. (This basically means that C_d is truncated to 288 bytes.) This new ciphertext block C'_d is then submitted for decryption. There are two cases to consider:

- If $d = 0$, then C'_d refers to P_0, and this, in turn, means that the underlying plaintext message has a long padding (i.e., 256 0xFF bytes) and that the actual message is short. If, for example, we assume the use of MD5 (that generates 16 bytes hash values), then the message is only $288 - 256 - 16 = 16$ bytes long.

22 Note that the adversary can always randomly guess whether d is 0 or 1 with a success probability of one half. He or she is only successful if the attack is better than a random guess.

- If $d = 1$, then C'_d refers to P_1, and this means that the padding is short and the message is long. Again using MD5 (16 bytes), the respective message is as long as $288 - 1 - 16 = 271$ bytes.

In either case, the MAC needs to be computed and verified. With an overwhelmingly large probability that MAC is not going to be valid, and this, in turn, means that an error message is returned. If the error message appears after a relatively short period of time, then it is likely that the message is short, and this speaks in favor of the first case (i.e., P_0). If, however, the error message appears late, then the second case (i.e., P_1) is more probable. The bottom line is that the timing behavior of an implementation may yield some information about the underlying plaintext (i.e., the longer the delay, the shorter the padding), and this information can be turned into a distinguishing attack that is feasible in some situations.

Distinguishing attacks are theoretically interesting but not practically relevant. So the more important question is whether a distinguishing attack can be turned into a plaintext recovery attack. In [19] it was shown that partial and even full plaintext recovery attacks are possible, and hence that the question must be answered in the affirmative. The attacks exploit the fact that the processing time for a ciphertext representing a TLS record (and hence the appearance time of error messages) depends on the amount of padding that the receiver interprets the plaintext as containing. By placing a target ciphertext block at the end of an encrypted record, an adversary can arrange that the respective plaintext block is interpreted as padding, and hence the processing time depends on the plaintext bytes and may leak some information about them. Because this information leakage is comparably small, large amounts of padding are usually needed to create a significant timing difference. So in the general case, plaintext recovery attacks are involved and expensive to mount.

Let C^* be a ciphertext block whose corresponding plaintext block P^* the adversary wants to recover. C' denotes the ciphertext block that immediately precedes C^* (so $C' \parallel C^*$ is part of the ciphertext). According to CBC decryption, it is known that

$$P^* = D_K(C^*) \oplus C'$$

and this is what the adversary is heading for. As usual, it is assumed that the adversary is capable of eavesdropping on the communications and of injecting messages of his or her choice into the network. For the sake of simplicity, we assume the use of a block cipher with an explicit IV (as introduced in TLS 1.1) and a block length of 16 bytes (e.g., AES). Furthermore, we assume a cryptographic hash function that generates hash values of 20 bytes, and hence the MAC construction

HMAC-SHA-1. Other parameters require slightly different attack versions and are not addressed here.

To mount a plaintext recovery attack against P^* (using C^* and C'), the adversary compiles a series of TLS records $\overline{C(\Delta)}$ that are sent to the victim for decryption. Each record depends on a randomly chosen (and hence distinct) 16-byte block Δ, but is otherwise built the same way:

$$\overline{C(\Delta)} = \text{HDR} \parallel C_0 \parallel C_1 \parallel C_2 \parallel C' \oplus \Delta \parallel C^*$$

HDR refers to the TLS record header, whereas C_0, C_1, C_2, $C' \oplus \Delta$, and C^* are five 16-byte blocks that represent the fragment. C_0 is an IV, whereas all other blocks are non-IV normal ciphertext blocks. C_0, C_1, and C_2 are arbitrary and can be chosen at will, whereas the two other blocks depend on C', Δ, and C^*. If such a TLS record $\overline{C(\Delta)}$ is sent to the victim, then the TLS record header is discarded, and the non-IV ciphertext blocks are decrypted. The resulting 64-byte plaintext message P consists of four blocks:

$$P = P_1 \parallel P_2 \parallel P_3 \parallel P_4$$

Among these blocks, the adversary is only interested in P_4. According to CBC decryption,

$$\begin{aligned} P_4 &= D_K(C^*) \oplus (C' \oplus \Delta) \\ &= P^* \oplus \Delta \end{aligned}$$

This means that P_4 is related to the target plaintext block P^* in some Δ-dependable way. This applies at the block level, but it also applies at the byte level, such as,

$$P_4[i] = P^*[i] \oplus \Delta[i] \tag{A.4}$$

for every byte $i = 0, \ldots, 15$. Because Δ randomizes the block, the padding of P_4 is incorrect in most cases. But in some cases, the padding can still be correct. Anyway, depending on the exact amount of padding, HMAC verification may take a different amount of time, and this is what enables the attack in the first place.

- If the padding is incorrect (what occurs in most cases), then the rightmost 20 bytes are interpreted as HMAC value, and the remaining $64 - 20 = 44$ bytes together with the 13 bytes of sequence number and record header form the basis for the HMAC verification. Hence, the HMAC is computed over $44 + 13 = 57$ bytes. According to what has been said above, this requires 5 evaluations of the compression function.

- If the padding is correct (what occurs only in a few cases), then there are still different possibilities: 0x00, 0x0101, 0x020202, 0x03030303, ...
  - If the padding is 0x00, then the rightmost byte is removed and the next 20 bytes are interpreted as HMAC value. The remaining $64-1-20 = 43$ bytes and the 13 bytes of sequence number and record header form the basis for the HMAC verification. Hence, the HMAC is computed over $43+13 = 56$ bytes. Again, this requires 5 evaluations of the compression function.
  - If the padding is 0x0101, then the rightmost two bytes are removed and the next 20 bytes are interpreted as HMAC value. The remaining $64-2-20 = 42$ bytes and the 13 bytes of sequence number and record header form the basis for the HMAC verification. Hence, the HMAC is computed over $42+13 = 55$ bytes. This now requires only 4 evaluations of the compression function.
  - If the padding is 0x020202, 0x03030303, ..., then the situation remains unchanged, and the HMAC computation requires only 4 evaluations of the compression function.

  Among the paddings that require only 4 evaluations of the compression function, 0x0101 is the most likely to occur one. So this is the one an adversary may go for.

The bottom line is that there is a timing difference between P_4 ending with 0x0101 and P_4 ending with any other two bytes, and hence a value for the two last bytes of $\overline{C(\Delta)}$ can be detected (in which the underlying message block P_4 ends with 0x0101). This means that after 2^{16} trials (in the worst case), the adversary can find a value for $\overline{C(\Delta)}$ that triggers this case. According to (A.4), the last two bytes of P^* can be recovered as follows:

$$\begin{aligned} P^*[15] &= P_4[15] \oplus \Delta[15] \\ P^*[14] &= P_4[14] \oplus \Delta[14] \end{aligned}$$

Once this is done, the adversary can recover the remaining bytes of P^* in a way that is similar to the original Vaudenay attack. For example, to recover the third-to-last byte $P^*[13]$, the adversary can use his or her new knowledge about the last two bytes of P^* to set $\Delta[15]$ and $\Delta[14]$ so that P_4 ends with two 0x02 bytes. He or she then generates candidates $\overline{C(\Delta)}$ as before, but modifying $\Delta[13]$ only. After at most $2^8 = 256$ trials, he or she finds a value for $\Delta[13]$ that satisfies $P_4[13] = P^*[13] \oplus \Delta[13] =$ 0x02, and hence $P^*[13]$ can be recovered as $P^*[13] = \Delta[13] \oplus$ 0x02. Recovery of each subsequent byte of P^* requires at most $2^8 = 256$ trials. In total, the attack requires $2^{16}+14\cdot 2^8$ trials in the worst case. Note, however, that the attack complexity can be reduced significantly by combining it with techniques borrowed from the BEAST attack (Section A.3) to position target plaintexts so that only one unknown byte at a time resides in the target block. This significantly reduces the number of queries needed to carry out the attack. The attack complexity can be further reduced assuming that the plaintext language can be modeled using a finite-length Markov chain. This is a perfectly fair assumption for natural languages, but it is not a valid assumption for randomly generated messages.

The attack works in theory. In practice, however, there are at least two severe complications and problems to overcome:

- First, the TLS session is destroyed as soon as the adversary submits his or her first ciphertext (and keep in mind that he or she must be able to submit as many as $2^{16} + 14 \cdot 2^8$ ciphertexts);
- Second, the timing differences are very small and may even be hidden by network jitter.

The first problem can be overcome by mounting a multisession attack, in which the same plaintext is repeated in the same position in many simultaneous sessions. The second problem can be overcome in the same multisession setting by iterating the attack many times for each Δ value and then performing statistical processing of the recorded times to estimate which value of Δ is the most likely one. The overall feasibility of the attacks depends on many implementation details. Also, due to DTLS' tolerance of errors and because of the availability of timing amplification techniques [22], the attack is more feasible for DTLS than it is for TLS.

In summary, Lucky 13 refers to the most recent padding oracle attack that can sometimes be mounted against TLS. Once again, it has demonstrated that the AtE approach is susceptible, and that the EtA approach would be superior (at least from a security perspective). As mentioned several times so far, this has led people to suggest the use of an AEAD cipher or a TLS extension mechanism (Section C.2.19). This will mitigate Lucky 13 and all other padding oracle attacks against TLS. In the meantime, however, it is recommended to use implementations that require constant time to execute (under all circumstances). This has turned out to be challenging (e.g., [23]), and there have been several attacks that work in spite of constant-time programming attempts, such as *Lucky microseconds*[23] mounted against Amazon's s2n TLS implementation, *LuckyMinus20* or *LuckyNegative20*

23 https://eprint.iacr.org/2015/1129.

mounted against OpenSSL,[24] and many more. Constant-time programming has turned out to be difficult, certainly much more difficult than originally anticipated. In fact, it has become an active research area.

A.3 BEAST

In Section 2.4, we mentioned another vulnerability of CBC encryption first reported by Bard in 2004 [24]. This vulnerability goes back to an observation made by Phillip Rogayway in 1995.[25] He observed that CBC is not secure against CPA, if the adversary knows or is somehow able to predict the IV that is going to be used to encrypt the next plaintext block,[26] and that this ability enables a simple chosen-plaintext distinguishing attack; that is an attack in which an adversary is able to distinguish the encryptions of two plaintext blocks using an encryption oracle.

In Rogayway's chosen-plaintext distinguishing attack, an adversary selects two plaintext blocks P_0 and P_1 and is challenged with a ciphertext block C that can either be the encryption of P_0 or the encryption of P_1. The adversary has to respond with the correct plaintext block P_b, where b can be 0 or 1. The adversary knows that CBC encryption employs IV chaining, meaning that some known previously transmitted ciphertext block C_0 is used for the encryption of P_b, and that C_1 is going to be the chaining value for the encryption of the next block. If the adversary assumes $P_b = P_0$, then he or she can proceed as illustrated in Figure A.7. First, the oracle is used to encrypt the plaintext block P_0, where the result is $C_1 = E_K(P_0 \oplus C_0)$. Second, this value is chained into the encryption of the next plaintext block that is constructed as $P_0 \oplus C_0 \oplus C_1$. The result is $C_2 = E_K(P_0 \oplus C_0 \oplus C_1 \oplus C_1) = E_K(P_0 \oplus C_0)$. This means that C_1 and C_2 must be the same, if the assumption $P_b = P_0$ holds. Otherwise (i.e., if C_1 and C_2 are distinct), then P_b must also be

24 The vulnerability and exploit is documented in CVE-2016-2107. A respective explanation is also available at https://web-in-security.blogspot.com/2016/05/curious-padding-oracle-in-openssl-cve.html.

25 http://web.cs.ucdavis.edu/∼rogaway/papers/draft-rogaway-ipsec-comments-00.txt.

26 In theory, CBC requires a fresh and unpredictable one-block IV for every message that is encrypted. In an SSL/TLS setting, a message refers to a record, so, in principle, every record requires such an IV. The designers of the SSL protocol took a different approach: They argued that a message can span multiple records, and that all of these records together represent the message. According to this position, it is obvious that only the first record requires an IV that fulfills the freshness and unpredictability requirement, and that for all other records the preceding record's last block may serve as an IV (this design decision is sometimes reported as a flaw, but—taking this different perspective—it is in line with CBC encryption for a message that spans multiple records). The consequence of this IV chaining structure is that an adversary can always predict the IV that is used next (i.e., to encrypt the next record). This applies to SSL 3.0 and TLS 1.0, but not to all later versions of the TLS protocol.

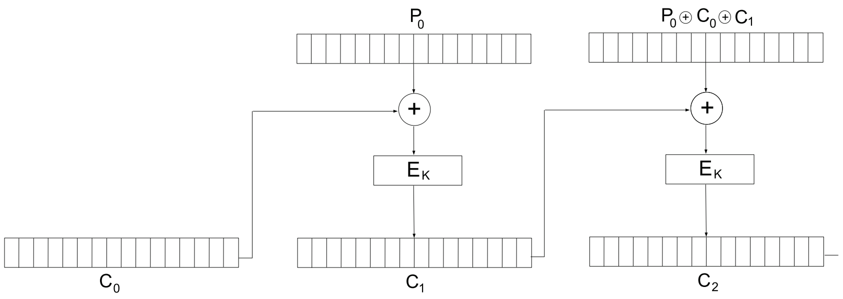

Figure A.7 Rogayway's chosen-plaintext distinguishing attack.

different from P_0, and this means that $P_b = P_1$. So having the oracle encrypt two plaintext blocks and comparing the resulting ciphertext blocks C_1 and C_2, the adversary can distinguish the encryptions of P_0 and P_1. This is an interesting observation, but it cannot be directly turned into a meaningful attack.

Several years later, Rogaway's distinguishing attack was turned into a real-world attack by Wei Dai[27] (for SSH2) and Moeller (for SSL/TLS). This is the setting we are interested in. Assume an adversary who has eavesdropped a sequence of SSL/TLS records with ciphertext blocks $C_0, C_1, \ldots$ (where C_0 represents the IV) and now wants to find out whether or not a particular plaintext block P_i ($1 \leq i \leq n$) is equal to $P^\star$. This setting is illustrated in Figure A.8.

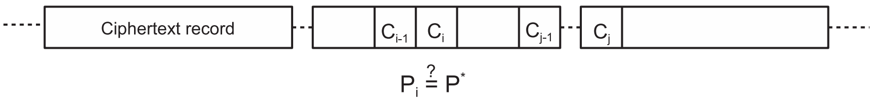

Figure A.8 The setting for the Moeller attack.

Contrary to what we have seen so far (in terms of padding oracle attacks), the attacker can now mount a blockwise CPA, meaning that he or she can artificially craft the next to be encrypted plaintext block P_j as

$$P_j = C_{j-1} \oplus C_{i-1} \oplus P^\star$$

27 http://www.weidai.com/ssh2-attack.txt.

This block gets encrypted to

$$\begin{aligned} C_j &= E_k(P_j \oplus C_{j-1}) \\ &= E_k(C_{j-1} \oplus C_{i-1} \oplus P^\star \oplus C_{j-1}) \\ &= E_k(C_{i-1} \oplus P^\star) \\ &= E_k(P^\star \oplus C_{i-1}) \end{aligned}$$

Comparing this formula with the normal CBC encryption formula for P_i (i.e., $C_i = E_k(P_i \oplus C_{i-1})$) reveals the fact that $C_i = C_j \leftrightarrow P_i = P^\star$. So if the guess $P^\star$ is correct (for P_i), then the newly generated ciphertext block C_j must also be equal to C_i. This can be tested, and hence the validity of $P^\star$ can be verified. If the test fails, then the adversary must try another value for $P^\star$. This procedure can be iterated, until the correct value for P_i is found.

In the general case, the attack is not feasible, because it requires an exhaustive search over all possible values for P_i. For a typical block length of 128 bits, this means that 2^{128} possibilities need to be tested. Needless to say that this is beyond the capabilities of a normal adversary. So either P_i is only a low-entropy value, or the adversary must try to modify the attack in a way that allows him or her to attack each byte within a block individually. Either strategy would bring the attack complexity down to something that is feasible.

- The first strategy (i.e., targeting only low-entropy values) was originally proposed by Bard [24–26].
- The second strategy (i.e., trying to attack each byte individually) was followed by Thai Duong and Juliano Rizzo who developed an attack tool named BEAST that was presented at the 2011 Ekoparty security conference[28] to perform a live attack against a Paypal account.[29] Similar attack techniques had previously been used by Duong and Rizzo to implement padding oracle attacks [17, 18].

Targeting only low-entropy values has a somehow restricted applicability, and trying to attack each byte individually is therefore advantageous. To enable it, Duong and Rizzo introduced the notion of a *chosen-boundary blockwise CPA*. In such an attack, the adversary is able to fill blocks with arbitrary data and hence to move the boundaries of a block at some arbitrary position. This means that the adversary can generate a plaintext block P_i in which all but one bytes are fixed (and hence known in advance), and in which the only unknown byte appears at the end of the block. If the secret to be decrypted is a bearer token (i.e., a cookie) transmitted in an HTTP

28 http://netifera.com/research/beast/beast_DRAFT_0621.pdf.

29 The vulnerability and the attack tool are documented in CVE-2011-3389.

request header, then the adversary can insert dummy characters to make the first character of the cookie appear as the last byte of block P_i. If he or she knows all other bytes of P_i, then he or she can mount a chosen-boundary blockwise CPA to find out the correct value for the last byte. Using the notation introduced above, all $k - 1$ bytes (if k is the block length of the cipher in use) of $P^\star$ are known to the adversary, and the adversary is able to find out the last byte of $P^\star$ (i.e., $P^\star[k]$). On average, this requires $2^8/2 = 2^7 = 128$ tries.

What enables the attack and makes it efficient is the fact that all tries to find the correct value of a byte can be automated, meaning that an adversary can manipulate the browser to mount a chosen-boundary blockwise CPA (as described above) without user interaction and without the user being aware of it.[30] A manipulated browser can thus generate and send out the respective ciphertexts, and the adversary can capture them from the network traffic. In contrast to the padding oracle attacks discussed so far, there is no server that needs to be targeted. Such an attack is difficult to detect while it is going on. The BEAST tool implemented the attack with malicious JavaScript[31] code that generated the appropriate plaintext blocks with the proper boundaries in place. The tool also captured the resulting ciphertext blocks to mount the attack.

When Bard published his findings in 2004, it became clear that the implicit IV used so far had to be replaced with an explicit IV. This means that for every TLS record that is encrypted with a block cipher, there must be an appropriately sized IV that is randomly chosen and sent along with the respective `TLSCiphertext` fragment.[32] This changes the TLS record protocol, and the respective change became part of TLS 1.1. But the topic was still inaccessible to the public and largely went unnoticed. This suddenly changed when the BEAST tool was released in 2011. This tool attracted significant industry and media attention, and the underlying vulnerability (discovered by Rogaway, Dai, and Bard) suddenly became a hot topic.

The record protocol change in TLS 1.1 successfully mitigated the BEAST attack. For those who did not want to upgrade to TLS 1.1, however, there were a few other possibilities to mitigate it. Using another mode of operation than CBC was certainly a theoretical option. In practice, however, almost all cipher suites that employed a block cipher were using CBC. Another possibility was to use a stream cipher, such as RC4, but this seriously restricts the number of cipher suites.

30 To actually achieve this, the attack requires fine-grained control over chosen plaintexts. The BEAST therefore exploited a zero-day vulnerability.

31 This is in contrast to Bard's publications that suggest the use of Java applets.

32 The `TLSCiphertext` fragment comprises the data that is authenticated and encrypted. If a stream cipher is used for encryption, then this means that the fragment consists of some data and a MAC that are collectively encrypted. If a block cipher is used for encryption, then the fragment consists of some data, a MAC, and some padding that are collectively encrypted, as well as an IV that is prepended to the fragment in the clear.

So practitioners were looking for more general and less restrictive techniques to mitigate the BEAST attack inside a browser. A respective technique known as *record splitting* was developed and immediately became a standard feature of most browsers to deal with SSL 3.0 and TLS 1.0.

As its name suggests, the idea is to split every record that is sent into two records: A first record that is basically empty and contains no data and a second record that contains the actual data. When a BEAST-like attack tool feeds in data to be encrypted, a first record is created that is empty but still has a MAC. The last block of this MAC then serves as the IV for the encryption of the second record. Because the MAC value is pseudorandom and cannot be predicted by the adversary, this simple technique mitigates any chosen-boundary blockwise CPA, including the BEAST attack. Because many browsers have problems with empty records, the technique was later changed to have the first record comprise one byte of data and the second record comprise the remaining $n - 1$ bytes (if the total length of the data is n bytes). This special case of record splitting is known as $1/n - 1$ *record splitting*, and it is widely deployed in the field. Due to this technique and some other changes in modern browsers, the BEAST attack does no longer work (or it may only work in some exceptional cases against some outdated browser versions). But the acronym BEAST still stands for itself and makes people nervous when talking about the security of SSL/TLS.

A.4 POODLE

In Section A.2, we mentioned that TLS employs PKCS #7 padding, whereas the padding scheme employed by SSL is much simpler (Figure A.2). In the aftermath of the BEAST attack, it was shown by Möller,[33] Duong, and Krzysztof Kotowicz in 2014 that the SSL padding scheme is susceptible to a simple but highly devastating padding oracle attack known as POODLE.[34] Together with the attacks against RC4 that popped up around the same time, the POODLE attack finally led to the recommendation to deprecate SSL 3.0 and hence all versions of the SSL protocol [27].

Let us assume an adversary who has eavesdropped a ciphertext that consists of n ciphertext blocks $C_1, \ldots, C_n$ encrypted with a block cipher in CBC mode. The block cipher has a block length of k bytes (typically $k = 16$) and uses an initialization vector C_0 that is transmitted along with the ciphertext. For the sake of simplicity, we assume that there is an entire block of padding, meaning that the last byte of C_n (i.e., $C_n[k-1]$ if we start numbering the bytes with zero) comprises

33 https://www.openssl.org/~bodo/ssl-poodle.pdf.

34 The vulnerability and respective attack is documented in CVE-2014-3566.

the value $k - 1$, whereas all other bytes of C_n (i.e., $C_n[0]$, $C_n[1]$, ..., $C_n[k-2]$) comprise random values (in Figure A.2 this means that PL equals $k - 1$, whereas all RB values are arbitrary). Let us further assume that the adversary knows that the ciphertext block C_i comprises the secret he or she is looking for, such as a bearer token (e.g., a cookie), an HTTP authorization header, or something similar. So C_i is the target of the attack, and hence the adversary's goal is to decrypt and later reuse the information comprised in C_i.

Under normal circumstances, the recipient of the ciphertext blocks $C_1, \ldots, C_n$ can use K and C_0 to retrieve the plaintext blocks $P_1, \ldots, P_n$. Using the CBC mode of operation, the formula to decrypt ciphertext block C_i $(i = 1, \ldots, n)$ is given in (A.2). After having decrypted all ciphertext blocks, the recipient can check and remove the padding from the last block, and verify the MAC. If everything is fine, then the plaintext blocks are forwarded to the appropriate application software for further processing.

Note what happens if the adversary—acting as a MITM—replaces the ciphertext block C_n with C_i, and hence constructs a sequence of blocks that looks as follows (the repeated block C_i is underlined here):

$$C_1, C_2, \ldots, C_{i-1}, \underline{C_i}, C_{i+1}, \ldots, C_{n-1}, \underline{C_i}$$

When the recipient decrypts this sequence, everything works fine until the last block (which is now C_i instead of C_n). Using (A.2), C_i decrypts to $P_i = D_K(C_i) \oplus C_{n-1}$. This equation holds at the block level, but it also holds at the byte level, meaning that $P_i[j] = D_K(C_i)[j] \oplus C_{n-1}[j]$ for all $j = 0, \ldots, k-1$. So when the recipient checks the padding, he or she verifies whether the last byte of P_i (i.e., $P_i[k-1]$) is equal to $k - 1$. There are two cases here:

- With a probability of $1/256$,[35] $P_i[k-1]$ is equal to $k - 1$. In this case, C_i is properly padded, meaning that the MAC is computed from the correct position, and the subsequent MAC verification (and hence also the protocol execution) is going to succeed.
- With a much larger probability of $255/256$, however, the opposite is true and $P_i[k-1]$ does not equal $k - 1$. In this case, C_i is not properly padded, meaning that the MAC is computed from a wrong position, and the subsequent MAC verification (and hence also the protocol execution) is going to fail. This failure, in turn, can easily be detected, because an alert message is sent back.

From the adversary's viewpoint, it is simple and straightforward to distinguish the two cases: In the first case, the protocol executes as usual (and does not abort),

35 This probability is $1/256$, because there are 256 equally probable byte values and only one of them is equal to $k - 1$.

whereas in the second case, the protocol aborts and an alert message is sent back. If the adversary is able to send arbitrarily crafted HTTPS (request) messages to a target site (for example, by having compromised a browser through malicious JavaScript code), then he or she can mount the POODLE attack that exploits this distinguishing feature.

More specifically, let us assume an adversary who has compromised a browser and can now prepare and craft plaintext messages as HTTPS requests of his or her choice. In particular, he or she can craft a message in which the first byte of the secret (to be determined) appears as the last byte of ciphertext block C_i (i.e., $C_i[k-1]$) and the last ciphertext block C_n comprises a full block of padding. If, for example, the adversary knows that the browser has a cookie and wants to retrieve the actual value of it, then he or she can have the browser craft an HTTP request message in which the ith ciphertext block C_i ends with the encryption of the first cookie byte, such as `Cookie: SESSID=t` for the cookie `test`, and the last ciphertext block C_n comprises a full block of padding. The browser can achieve this by blowing up the message with arbitrary (but valid) characters in the path identifier or other HTTP headers. The important thing is that the encryption of the secret occurs at the right position.

Referring to the construction mentioned above, the adversary can then replace C_n with C_i in the ciphertext C and send the now modified ciphertext C' to the target site. After the decryption of C', one of the two cases mentioned above occurs (i.e., either the padding is correct or it is not).

- In the more likely case (when the second possibility occurs), the padding is not correct, the MAC verification fails, the protocol aborts, and an alert message is sent back. In this case, the adversary has to repeat the attack.
- In the less likely case (when the first possibility occurs), the padding is correct and the adversary can decrypt $C_i[k-1]$. A correct padding means that $C_i[k-1]$ decrypts to $k-1$. Applying (A.2) at the byte level (and in our setting where $i-1$ is $n-1$), this means that

$$D_K(C_i)[k-1] \oplus C_{n-1}[k-1] = k-1$$

In this formula, C_i can be replaced with $E_K(P_i \oplus C_{i-1})$, so

$$D_K(E_K(P_i \oplus C_{i-1}))[k-1] \oplus C_{n-1}[k-1] = k-1$$

and hence

$$(P_i \oplus C_{i-1})[k-1] \oplus C_{n-1}[k-1] = k-1$$

or

$$P_i[k-1] \oplus C_{i-1}[k-1] \oplus C_{n-1}[k-1] = k-1$$

In the end, this equation can be solved for $P_i[k-1]$:

$$P_i[k-1] = k-1 \oplus C_{i-1}[k-1] \oplus C_{n-1}[k-1]$$

This means that the adversary can determine $P_i[k-1]$, and that this value refers to the first byte of the secret.

The adversary can expect to successfully decrypt $C_i[k-1]$ (that represents the first byte of the secret) after 256 tries. But there is nothing special about this byte, and the same procedure can be applied to decrypt all bytes of the secret. To decrypt the second byte of the secret, for example, the adversary must craft a ciphertext, in which this byte appears as the last byte of C_i. The same is true for the third byte, the fourth byte, and so on and so forth, until the entire secret is decrypted. For every byte to decrypt, the adversary has to mount the attack $256/2 = 128$ times on the average case. This is efficient and perfectly feasible. In fact, it is equally efficient than other padding oracle attacks, but it has the big advantage that wrong and correct guesses can easily be distinguished and told apart from incorrect ones. If a guess is wrong, then the protocol execution immediately aborts (otherwise, the execution continues). In other padding oracle attacks, one usually has to time the behavior of the implementation to decide whether a guess is correct or wrong. This is more expensive and generally more error-prone. This makes the POODLE attack so powerful and devastating.

Due to its unique padding, the POODLE attack only applies to SSL 3.0. Hence, the adversary may try to enforce the use of this protocol version by performing a *downgrade dance* that is essentially a protocol version downgrade attack. If, for example, the client suggests the use of TLS 1.2 in a CLIENTHELLO message, then the adversary acting as a MITM can block the message so the server does not receive it and does not respond. The client then retries with TLS 1.1, TLS 1.0, and finally SSL 3.0. In each unless the last try, the adversary blocks the message, so the only message that goes through is the message suggesting SSL 3.0. This way, the adversary can enforce the use of SSL 3.0 and mount the POODLE attack afterwards. In Section 2.4, we already mentioned the signaling cipher suite TLS_FALLBACK_SCSV that the client may add to the set of supported suited suite to signal that it willingly falls back to the use of SSL 3.0 [28]. If the server supports a version higher, then it knows that something fishy is going on and can abort the connection (and send back a fatal `inappropriate_fallback` alert message).

In spite of the fact that the POODLE attack only applies to SSL 3.0 in theory, practice reveals another picture: Some TLS implementations use a padding similar to

SSL, and hence they have shown to be vulnerable as well. People use the term *POODLE over TLS* for this phenomenon, and they have found vulnerabilities that can be exploited in specific attacks, such as *Zombie POODLE* and *GOLDENDOODLE*,[36] as well as *Sleeping POODLE* and *0-Length OpenSSL*.[37] Some of these attacks are particularly related to OpenSSL. The details are subtle and not further addressed here.

A.5 RENEGOTIATION ATTACKS

In 2009, Marsh Ray and Steve Dispensa published a paper[38] in which they describe a vulnerability[39] that can be exploited to mount a MITM attack against an optional but then frequently used feature of the TLS protocol, namely to renegotiate a session.[40] The reasons that may make it necessary to renegotiate a session were discussed in Section 2.2.2. It may be the case that cryptographic keys need to be refreshed, cryptographic parameters need to be changed, or—most importantly—certificate-based client authentication needs to be invoked. For example, if a client needs to authenticate using a certificate but wants to somehow hide its identity, then it may first establish an unauthenticated session to the server and then use this session to renegotiate another session that is now authenticated with a certificate. Since the renegotiation messages are transmitted within the first session, the client certificate is encrypted during transmission and is not revealed to an eavesdropper. But keep in mind that a session renegotiation is not the same as a session resumption: While a new session with a new session ID is established in a renegotiation, an already existing and previously established session is reused in a resumption. Technically speaking, a client-initiated renegotiation can be started by having the client send a new CLIENTHELLO message to the server, whereas a server-initiated renegotiation can be started by having the server send a HELLOREQUEST message to the client. In either case, the party that receives the message can either accept or refuse the renegotiation request. If it accepts the request, then a new handshake

36 The GOLDENDOODLE attack may affect several implementations, such as the Cavium cryptographic-module firmware used in several products (CVE-2015-4458).

37 The attack is documented in CVE-2019-1559.

38 When the paper was published, Ray and Dispensa were working for PhoneFactor, a then leading company in the realm of multifactor authentication. The publication appeared as a technical report of this company. In 2012, PhoneFactor was acquired by Microsoft, and hence the technical reports of PhoneFactor are no longer available on the internet. However, there are still many internet repositories that distribute them. You may simply search for “renegotiating TLS.”

39 The vulnerability is also documented in CVE-2009-3555.

40 The vulnerability was independently found and described by Martin Rex in a post to the mailing list of the IETF TLS WG (http://www.ietf.org/mail-archive/web/tls/current/msg03928.html).

is initiated. If, however, it refuses the request, then no handshake is initiated and a `no_renegotiation` alert message (code 100) is returned instead.

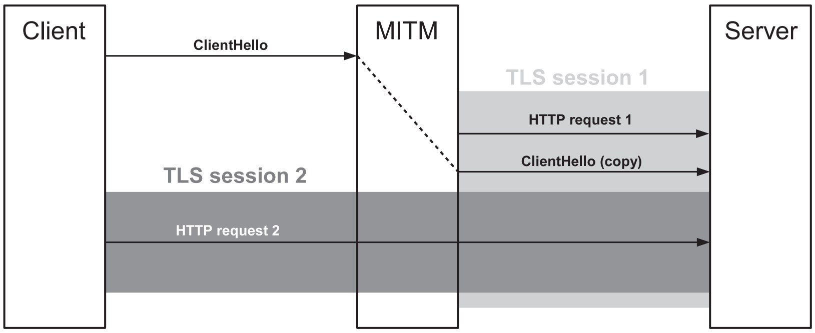

Figure A.9 The TLS renegotiation attack (overview).

A renegotiation attack is essentially a plaintext injection attack, meaning that the MITM tries to inject some additional data into an application data stream. If the injected data is delivered together with the rest of the data stream to the application, then some unexpected things my happen that can be exploited in an attack. Let us consider an exemplary setting in which TLS—and hence HTTPS—is used to secure a web application. Such a setting and the outline of a renegotiation attack are overviewed in Figure A.9. The adversary representing the MITM is located between the client (left side) and the server (right side). The client wants to establish a TLS session and sends a respective CLIENTHELLO message to the server. This message is captured by the MITM. Before forwarding it (on the client's behalf), the MITM establishes a first TLS session to the server. This session 1 is used to send an HTTP request message to the server (denoted HTTP request 1 in Figure A.9). Let us assume that this request comprises the following two header lines (where only the first line is terminated with a new line character):

```
GET /orderProduct?deliverTo=Address-1
X-Ignore-This:
```

The meaning of these header lines will soon become clear. Many SSL/TLS implementations don't pass decrypted data immediately to the application. Instead, they wait for other data to arrive, and pass all data in one chunk to the application. This behavior is exploited in the attack.

Immediately after having sent this HTTP message to the server, the MITM forwards the client's original CLIENTHELLO message to the server, again using session 1. From the server's perspective, this message looks as if the client that originates the message wants to renegotiate the session. So a second TLS session is established between the client and the server, and the use of TLS renegotiation suggests that this second TLS session is tunneled through the first TLS session. This also means that the handshake of session 2 is oblivious to the MITM, and that the MITM can only send back and forth respective handshake messages (through session 1). If the handshake succeeds, then a second TLS session is established between the client and the server, and this session is now end-to-end protected, meaning that the adversary cannot decrypt the transmitted data. Let us assume that the server authenticates the client, and that the server issues a cookie as a bearer token for the client. Due to the end-to-end protection, the adversary cannot access the cookie to misuse it directly. But see what happens if the client uses the end-to-end protected TLS session 2 to send another HTTP request message—labeled "HTTP request 2" in Figure A.9—to the server. In this case, the two messages may be concatenated and collectively passed to the application. From the application's viewpoint, it is therefore no longer possible to distinguish the two messages—they both appear to originate from the same source (i.e., the client). If, in our example, HTTP request 2 started with the following two header lines

```
GET /orderProduct?deliverTo=Address-2
Cookie: 7892AB9854
```

then the two messages would be concatenated as follows:

```
GET /orderProduct?deliverTo=Address-1
X-Ignore-This: GET /orderProduct?deliverTo=Address-2
Cookie: 7892AB9854
```

The `X-Ignore-This:` header is an invalid HTTP header and is therefore ignored by most implementations. Since it is not terminated with a new line character, it is concatenated with the first line of HTTP request 2. This basically means that the entire line is going to be ignored and hence that the requested product is going to be delivered to `Address-1` instead of `Address-2`. Note that the cookie is provided by the client and is therefore valid, so there is no reason not to serve the request. The problem is not the cookie, but the fact that some unauthenticated data is mixed with authenticated data and passed together to the application. This leads to a situation, in which the application falsely suggests that either data is authenticated. Also note that there are many possibilities to exploit this vulnerability and to come up with respective (renegotiation) attacks. Examples are given in the original publication and many related articles, as well as a recommended reading that was posted shortly after the original publication.[41]

Renegotiation attacks are conceptually related to cross-site request forgery (CSRF) attacks. So, in general, any protection mechanism put in place against CSRF attacks may also help mitigating renegotiation attacks. Furthermore, because the renegotiation feature is optional (as mentioned above), a simple and straightforward possibility to mitigate such attacks is to disable the feature and not support renegotiation in the first place. Less strictly speaking, it may be sufficient to only disable client-initiated renegotiation (as a side effect, this also protects the server against some DoS attacks). But since disabling renegotiation is not always possible, people have come up with other protection mechanisms. Most importantly, the IETF TLS WG has developed a mechanism that provides handshake recognition, meaning that it must be confirmed that when renegotiating both parties have the same view of the previous session (i.e., the one that is being renegotiated). As outlined in Section C.2.43, there is a distinct TLS extension; that is the `renegotiation_info` extension specified in [29], that serves this purpose. In essence, it binds the renegotiated session to the previous one. In our example from Figure A.9, session 2 has no previous session, but if it had, then it would be different from session 1 and unknown to the adversary. This seems to mitigate the attack.

To invoke the secure renegotiation mechanism, the client and the server must both include the `renegotiation_info` extension in their hello messages. In the initial handshake, the extensions are empty, meaning that the extensions comprise no data. The client and server only signal to each other that they support the extension. If, at some later point in time, the client wants to securely renegotiate, it adds another `renegotiation_info` extension to its CLIENTHELLO message. This time, the extension comprises data, namely the client-side `verify_data` field of the FINISHED message that was sent in the handshake for the session it wants to renegotiate. Remember from Section 3.2.4 that the `verify_data` field comprises a hash value of all messages previously sent in a handshake before the FINISHED message. It thus stands for a particular handshake and a particular session. If the server is willing to renegotiate this session, then it sends back a SERVERHELLO message with a `renegotiation_info` extension that comprises both the client-side `verify_data` field of the client's FINISHED message and the `verify_data` field of the server's FINISHED message—both from the previous handshake. Intuitively, by including one or two `verify_data` fields from the previous handshake, the parties ensure that they both have the same view of the previous handshake, and hence that they renegotiate the same session.

There is an interoperability issue here: Because some SSL 3.0, TLS 1.0, and sometimes even TLS 1.1 implementations have problems gracefully ignoring empty

41 http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html.

extensions at the end of hello messages, people have come up with another possibility to let a client signal to a server that it supports secure renegotiation. Instead of sending an empty `renegotiation_info` extension in the CLIENTHELLO message, the client can include a special signaling cipher suite value (SCSV) (i.e., `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` with byte code 0x00FF) in the list of supported cipher suites. This also tells the server that it is willing and able to support secure renegotiation. The actual secure renegotiation then remains the same (and hence the secure renegotiation extension still needs to be supported by either party). The only point in using `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` is to avoid an empty extension at the end of a hello message, and hence to make some implementations more stable.

The goal of the `renegotiation_info` extension is to link the session that is being renegotiated to the (handshake of the) previous session. This is to mitigate the renegotiation attack and its variants. A client that is tricked into renegotiation does not automatically provide the `verify_data` field from the previous handshake. Also, if a MITM modified the CLIENTHELLO message to supply the `renegotiation_info` extension, then the respective message modification and integrity loss would be detected at the TLS level.

Between 2010 and 2014, it was believed that the renegotiation attack was successfully mitigated by the `renegotiation_info` extension. In 2014, however, it was shown that this was not really the case, and that a more sophisticated renegotiation attack remains feasible. The attack employs three handshakes and is therefore called a *triple handshake* or *3SHAKE attack* [30].[42] In this attack, the MITM cleverly exploits the following two weaknesses of the TLS handshake protocol:

- First, the MITM can establish two sessions—one between the client and the MITM and the other between the MITM and the server—that share the same master key and session ID. If, for example, RSA is used for key exchange, then the MITM can simply decrypt the premaster secret from the client and reencrypt it with the public key of the server. The random values are left unchanged, and the session ID from the server is relayed back to the client. After the handshake, the master keys and session IDs of the two sessions are the same. But while the client is tricked into thinking that it has established a session to the server (where in fact it has a session with the MITM), the server thinks that it is connected with an unauthenticated client and has no way of detecting the existence of the MITM. Technically speaking, this type of attack is an *unknown key-share attack* [31].
- Second, if a session established with an unknown key-share attack is resumed (not renegotiated), then the resulting two sessions do not only share the same

42 https://www.mitls.org/pages/attacks/3SHAKE.

master key and session ID, but also the same `verify_data` fields in the respective FINISHED messages. This is mainly due to the fact that certificates are not used (and hence do not appear) in a session resumption.

Combining these two weaknesses, the MITM can establish two sessions—one to the client and another to the server—that use the same `verify_data` field in the FINISHED messages and hence also share the same `renegotiation_info` values. This, in turn, means that the renegotiation attack can be mounted again (as a 3SHAKE attack), and that the `renegotiation_info` extension does not really solve the problem.

Referring to its name, the 3SHAKE attack comprises three steps (or handshakes, respectively) that can be summarized as follows:

- In step 1, the MITM mounts an unknown key-share attack to establish two sessions that share the same master key and session ID. As is usually the case in a MITM attack, the first session is between the client and the MITM, and the second session is between the MITM and the server.
- In step 2, the MITM waits until the client resumes its session. In contrast to a renegotiation that requires the `verify_data` from the FINISHED message of the previous handshake, only the session ID and the master key are required to resume a session. The MITM can do this also for his or her session to the server. In the end, there are two fully synchronized sessions that share the same `verify_data` fields. After these preparatory steps, the MITM is ready to mount the renegotiation attack.
- In step 3, the MITM first sends HTTP request 1 to the server and then has the server trigger a renegotiation that requires client-side authentication using, for example, a certificate. The respective handshake then takes place between the client and the server. To mitigate normal renegotiation attacks, the client adds the `renegotiation_info` extension to its CLIENTHELLO message. The data of this extension refers to the `verify_data` field of the previous session to the MITM. According to what we have said before, this is the same value as the `verify_data` field of the previous session between the MITM and the server. So it can be verified by the server, and the server thus believes that the previous session to the MITM is the one that is being renegotiated. When the now authenticated client sends an HTTP request 2 to the server, both requests are concatenated and collectively passed to the application. This means that the attack becomes feasible, and hence that we are back where we started.

The bottom line is that triple handshake attacks can still be mounted in spite of the `renegotiation_info` extension. The underlying problem is that an unknown key-share attack combined with a subsequent session resumption allows a MITM to establish two fully synchronized sessions. This is true even if the sessions are bound to different server certificates (the session from the client to the MITM is bound to the MITM's certificate, whereas the connection from the MITM to the server is bound to the server certificate). One way to mitigate the unknown key-share attack is to make the generation of the master key not only depend on the premaster secret and random values selected by the client and the server, but also on the server certificate. Even more generally, it is reasonable to make the master secret depend on all handshake messages that have been exchanged previously. Because the CERTIFICATE message comprises the server certificate, this ensures that the master secret also depends on the server certificate, and hence that an unknown key-share attack is infeasible to mount.

To mitigate an unknown key-share attack and hence also a 3SHAKE attack, people have proposed another TLS extension called `extended_master_secret` [32] (Section C.2.43). Its goal is to ensure that the master secret that is generated—let us call it *extended master secret*—is unique, meaning that different sessions established to different servers have different extended master secrets. Remember from Section 3.1.1 that a master secret is normally generated as

```
master_secret =
   PRF(pre_master_secret,"master secret",
       client_random + server_random)
```

If the `extended_master_secret` extension is negotiated, then the master secret is generated in a slightly different way:

```
master_secret =
   PRF(pre_master_secret,"extended master secret",
       session_hash)
```

Note that this construction uses another label (i.e., `extended master secret` instead of `master secret`) and that the client and server random values are replaced with a session hash. This is a hash value computed over the concatenation of all handshake messages (including their type and length fields) sent or received, starting with the CLIENTHELLO message and up to and including the CLIENT-KEYEXCHANGE message. This includes, among other things, the client and server random values and the server certificate. Since TLS 1.2, the hash function is the same that is also used to compute the FINISHED message. For all previous versions of the TLS protocol, the hash function employs the concatenation of MD5 and SHA-1. We have seen and discussed this construction before.

If a TLS session is established with both the `renegotiation_info` and `extended_master_secret` extensions in place, then it is reasonable to assume that the respective session is going to be secure against all types of renegotiation attacks. At least nobody has found a possibility to mount yet another attack in this setting. The two extensions are also required for token binding; for example, to complement the `token_binding` extension (Section C.2.20). Since session resumption and renegotiation are no longer supported in TLS 1.3, renegotiation attacks are no longer an issue here.

A.6 COMPRESSION-RELATED ATTACKS

In the early 2010s, there were several attacks against the TLS protocol that exploit the combined use of compression and encryption. The respective vulnerability was first observed and pointed out by John Kelsey in 2002 [33]. The problem is that compression may reduce the size of a plaintext in a specific way and that this reduction in size may leak information about the underlying plaintext (especially in the case of a lossless compression). Under some circumstances, this information is sufficient to reveal some parts of sensitive data, such as session cookies. This is particularly true if a stream cipher is used, but it may also be true if a block cipher is used.

One year after the release of the BEAST tool (Section A.3), Rizzo and Duong presented another attack tool named *compression ratio info-leak made easy* (CRIME) at the 2012 Ekoparty security conference.[43] It basically turns the vulnerability found by Kelsey into a side-channel attack against any security protocol that combines encryption and compression, such as the SSL/TLS protocols or SPDY. Note that the CRIME attack addresses compression that occurs at the SSL/TLS level (which can either be null or DEFLATE[44]) and that this type of compression must be supported on the client and server side. In contrast to HTTP-level compression, SSL/TLS-level compression is more seldom used and can be easily deactivated (for example, to protect against the CRIME attack). For the sake of completeness, we mention that Rizzo and Duong already mentioned in their Ekoparty presentation that the same type of attack can also be mounted against HTTP-level compression (that was demonstrated in later attacks).

The idea of the attack is relatively simple: Let us assume a setting in which a web browser communicates with a server over an SSL/TLS connection. So, the adversary does not see the HTTP messages in the clear. But the adversary has some

43 The vulnerability and exploit are documented in CVE-2012-4929.

44 Remember that the DEFLATE compression method combines LZ77 and Huffman. Zlib and gzip are also based on DEFLATE.

a priori knowledge about the client authentication that is being used. For the sake of simplicity, we assume a situation in which the client authenticates itself with a bearer token sent in a `Cookie:` header together with a value. A respective header line may, for example, look as follows:

```
Cookie: SessionToken=TEST
```

In this example, the cookie (that is intuitively named `SessionToken`) is assigned the artificially short value `TEST`. In practice, this value is usually much longer and encoded in a character set that comprises also other characters than just capital letters. Anyway, in each HTTP request message that is sent from the browser to the server, this `Cookie:` header line is repeated (this is due to the stateless nature of HTTP). As usual, we assume the adversary to have some control over the browser, so that he or she can have the browser generate arbitrary HTTP request messages (that are then sent over the SSL/TLS connection). This level of control can, for example, be achieved through injection of malicious JavaScript code from an evil site (using some form of drive-by infection).

Similar to the POODLE attack, the CRIME attack targets each character of the secret value (i.e., the cookie) individually. So the first character to attack is a "T." The adversary therefore generates a series of HTTP request messages that all carry a different one-character path parameter to the server. The respective HTTP request messages are all equal and only differ at one character:

```
... GET /SessionToken=A ...
... GET /SessionToken=B ...
    .
    .
    .
... GET /SessionToken=Z ...
```

All of these messages are injected into the browser, and the browser encrypts and sends them over the SSL/TLS connection. If the adversary observes the encrypted messages and measures their respective length, then he or she can recognize that the message that comprises the string `GET /SessionToken=T` compresses most (meaning that it is the shortest message among all messages). This is because the string that repeats (i.e., `SessionToken=T`) is one character longer than in all other cases (where only `SessionToken` repeats), and this means that the respective message can be compressed more, and—due to the nature of LZ77—that the resulting message is a little bit shorter.

After having tried all 26 possibilities, the adversary now knows that the first character of the cookie is a "T." He or she can apply the same procedure for the second character. More specifically, he or she generates a series of HTTP request messages that carry a two-character path parameter to the server, where only the second characters differ:

```
... GET /SessionToken=TA ...
... GET /SessionToken=TB ...
     .
     .
     .
... GET /SessionToken=TZ ...
```

Again, the adversary injects these messages into the browser, and measures the lengths of the encrypted messages sent over the SSL/TLS connection. In this case, he or she will recognize that the message that comprises the string

```
... GET /SessionToken=TE ...
```

compresses most, and hence that the second character of the cookie is likely "E." This procedure can be repeated for the third and fourth character, just to find out that the cookie in this example is "TEST." In this example, the workload is 4 (the length of the cookie) times 26 (the number of possible characters), and this means that $4 \cdot 26 = 104$ messages must be injected into the browser and the resulting strings must be analyzed. This number is not impossibly high, and hence the attack represents a real threat (even for longer cookies and larger character sets).

In practice, however, the attack is more difficult to mount than it seems to be at first sight, and there are a few caveats to consider. Most importantly, the DEFLATE compression method does not only comprise LZ77, but it also comprises Huffman encoding that may obfuscate the effects of LZ77 to some extent (Section 3.4.5). Remember that Huffman encoding ensures that frequently occurring characters are more strongly compressed than less frequently occurring ones. This, in turn, may lead to a situation in which an incorrect but frequently occurring character may cause the compressed data to be as small (or even smaller) than the compressed data for the correct guess. This makes the attack more difficult to mount, but Duong and Rizzo showed that it remains feasible.

Since the publication of Kelsey in general, and the demonstration of the CRIME tool in particular, people know that the combined use of encryption and compression may lead to subtle security problems. An obvious approach to solve these problems is to disable (and not use) TLS-level compression. This works and was actually done in TLS 1.3 and has no major negative impact (in contrast to disabling HTTP-level compression as discussed below). A less obvious approach is to hide the true length of any encrypted message, for example, by using padding. Unfortunately, this does not disable the attack entirely; it only makes it a little bit slower. By repeating messages and averaging their sizes, an adversary can still learn the true length of a message (e.g., [34]).

After the publication of the CRIME attack, two variants were publicly announced in the following year: TIME and BREACH. Instead of exploiting TLS-level compression, the attacks exploit application-level (i.e., HTTP-level) compression. Many web servers routinely deliver secrets (e.g., access tokens) in the bodies of the HTTP response messages that may be compressed. If a request message is sent to the server that comprises a substring that matches some part of such a secret, then the respective HTTP response message is going to have a better compression ratio. This can be detected either because the message is shorter (as in the case of BREACH) or because its transmission time is shorter (as in the case of TIME).

- At the 2013 European Black Hat conference,[45] Amichai Shulman and Tal Be'ery presented a variant of the CRIME attack named *timing info-leak made easy* (TIME). The variant refers to two points: On the one hand (as mentioned above), TIME targets HTTP-level compression (instead of TLS-level compression), and on the other hand, TIME measures the timing of messages (instead of their respective sizes). So TIME can be best characterized as a timing attack against HTTP-level compression.
- A few months later, Angelo Prado, Neal Harris, and Yoel Gluck announced a similar attack they named *browser reconnaissance and exfiltration via adaptive compression of hypertext* (BREACH[46]) at the normal 2013 Black Hat conference.[47] Like the TIME attack, the BREACH attack also targets compression in HTTP response messages. But unlike the TIME attack (and similar to the CRIME attack), the BREACH attack is not a timing attack; instead it measures the actual size of HTTP messages. In contrast to the TIME attack (that largely went unnoticed in the press), the BREACH attack attracted a lot of media attention.

Note that the possibility of exploiting HTTP-level compression (instead of TLS-level compression) was already mentioned by Rizzo and Duong, and it largely increases the attack surface.

Because compression-related attacks are relatively efficient, it is legitimate to ask if and how these attacks can be mitigated.

With regard to TLS-level compression (as exploited by CRIME), it is simple and straightforward to disable it, meaning that the only compression method that is supported is the null compression method. Because TLS-level compression has never been widely used in the field, this does not negatively impact the use of the SSL/TLS protocols. This approach is thus followed by TLS 1.3.

45 https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf.

46 http://breachattack.com.

47 The vulnerability and attack are documented in CVE-2013-3587.

With regard to HTTP-level compression (as exploited by TIME and BREACH), the situation is more involved and not supporting compression is impossible here. On the one hand, HTTP-level compression is important and widely used in the field for performance reasons. On the other hand, it is difficult if not impossible to control the use of compression in application-layer protocols, such as HTTP, in some meaningful way. As mentioned above, trying to hide the true length of messages does not help either. There are currently two approaches that look promising when it comes to mitigating compression-related attacks at the HTTP level:

- The first approach is to make sure that the target secret does not remain the same between HTTP requests. If the secret is S, then a one-time pad P may be generated, and $P\|(P \oplus S)$ may be used instead of S. This expression can be easily decoded but still ensures that the secret does not remain the same between requests. This mitigates the attack, but requires a major rewrite of the application. To make things worse, there is also a performance penalty to pay (because the secret is doubled in size and cannot be compressed).
- The second approach is to monitor the volume of data traffic per user and to detect the situation in which a user requests a huge number of resources in a relatively short amount of time.

Since these approaches are independent and not mutually exclusive, they can also be combined at will. Unfortunately, this is not (yet) the case, and we don't see them implemented and deployed in the field. This means that many applications that are in widespread use today are supposed to be susceptible to compression-related attacks.

A.7 KEY EXCHANGE DOWNGRADE ATTACKS

Throughout the book, we have mentioned key exchange downgrade attacks, like FREAK [35] and Logjam [36], several times. Both attacks are MITM attacks, in which an adversary (acting as an MITM) tries to downgrade the key exchange method used to something that is exportable, and thus breakable. The two attacks have clearly demonstrated that continuing support for exportable cipher suites (and all other cipher suites that comprise outdated cryptography) is dangerous and should be avoided under all circumstances. This general rule of thumb applies to all versions of the SSL/TLS protocols, but it certainly applies most to elder protocol versions (because they more likely also provide support for outdated cryptography).

A.7.1 FREAK

The FREAK attack[48] was published in March 2015 [35]. It targets an RSA key exchange and exploits an implementation bug. More specifically, the FREAK attack exploits the fact that some browsers misbehave in the sense that they support and accept exportable ciphers, even though they have been configured not to accept them (so there is an implementation bug in place that enables the attack).[49] The attack starts with a client that sends a CLIENTHELLO handshake message to the server, in which it asks for normal (i.e., nonexportable) cipher suites. The MITM captures this message and changes it to ask for an exportable cipher suite, typically one that employs an ephemeral RSA key exchange. In such a key exchange, a 512-bit RSA key used for key exchange is typically digitally signed with a longer key, such as a 1,024-bit RSA key, and sent to the client in a SERVERKEYEXCHANGE message (Section 2.2.2.5). Normally, the client would now abort the protocol, because the server has chosen a cipher suite that was not part of the client-supported cipher suites. But due to the implementation bug, the client nevertheless accepts the exportable cipher suite and continues the execution of the protocol. This means that it pseudorandomly selects a premaster secret and encrypts it with the 512-bit RSA key used for key exchange. The resulting ciphertext is then sent to the server in a CLIENTKEYEXCHANGE message. If the MITM manages to break this RSA encryption in a reasonable amount of time (i.e., by factoring the 512-bit modulus), then he or she can decrypt the premaster secret and use it to reconstruct all keys needed to either decrypt all messages or even generate new messages on the client's behalf.[50] To mount a FREAK attack, the MITM must be able to factorize a 512-bit integer in almost real time. Because this integer represents the RSA modulus that is distinct for every user, this computation needs to be done for every RSA key individually (meaning that—in contrast to the Logjam attack—there is hardly any precomputation that can be done to speed up the attack significantly).

48 https://www.smacktls.com.

49 The vulnerabilities exploited by the FREAK attack are documented in CVE-2015-0204 for OpenSSL, CVE-2015-1637 for Microsoft's Secure Channel (SChannel), and CVE-2015-1067 for Apple's Secure Transport.

50 Note that SSL provides some protection mechanism against an adversary changing the CLIENTHELLO message as required by the FREAK attack. In fact, a changed CLIENTHELLO message leads to changed FINISHED messages (because the CLIENTHELLO message is hashed together with all other handshake messages that are exchanged between the client and the server). But because the adversary knows all keys, he or she can properly change the FINISHED messages that need to be exchanged between the client and the server. So the attack cannot be detected.

A.7.2 Logjam

The Logjam attack[51] was also published in 2015 (a few months later than FREAK). It is similar to the FREAK attack but still different in some details. For example (and as mentioned above), it targets a DHE key exchange (instead of RSA) and it does not depend on an implementation bug.

The adversary who again represents an MITM has to wait until the client sends a CLIENTHELLO handshake message to the server in which it proposes a DHE-based cipher suite (among others). The adversary then replaces the entire list of cipher suites with a single DHE-based cipher suite that is suitable for export and forwards the modified message to the server. If the server supports export-grade DHE, then it sends back to the client a respective SERVERHELLO message. The MITM changes this message on the fly, replacing the export-grade DHE with the originally supported DHE. So the client is fully unaware of the fact that it's using export-grade DHE only (but since the protocol is the same in either case, the client has no reason to stop the execution of the protocol at this point in time). In the CERTIFICATE and SERVERKEYEXCHANGE handshake messages that follow, the server provides a certificate for its signature key and its DH parameters that are digitally signed with this key. Similarly, the client provides its DH parameters (for the same prime p) in the respective CLIENTKEYEXCHANGE message. If the MITM has done the precomputation for p, then he or she can now easily compute the discrete logarithm required to break the DHE key exchange. If he or she succeeds, then he or she is able to retrieve the premaster secret (that is the result of the key exchange). From there, the attack is identical to the FREAK attack.

References

[1] Kaliski, B., "PKCS #1: RSA Encryption Version 1.5," RFC 2313, March 1998.

[2] Bleichenbacher, D., "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1," *Proceedings of CRYPTO '98,* Springer-Verlag, LNCS 1462, August 1998, pp. 1–12.

[3] Davida, G. I., "Chosen Signature Cryptanalysis of the RSA (MIT) Public Key Cryptosystem," TR-CS-82-2, Deptartment of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, 1982.

[4] Alexi, W., et al., "RSA and Rabin Functions: Certain Parts are as Hard as the Whole," *SIAM Journal on Computing,* Vol. 17, No. 2, 1988, pp. 194–209.

[5] Håstad, J., and M. Näslund, "The Security of all RSA and Discrete Log Bits," *Journal of the ACM,* Vol. 51, No. 2, March 2004, pp. 187–230.

51 https://weakdh.org.

[6] Aviram, N., et al., "DROWN: Breaking TLS with SSLv2," *Proceedings of the 25th USENIX Security Symposium,* USENIX Association, 2016, pp. 689–706.

[7] Böck, H., J. Somorovsky, and C. Young, "Return Of Bleichenbacher's Oracle Threat (ROBOT)," *Proceedings of the 27th USENIX Security Symposium,* USENIX Association, 2018, pp. 817–849.

[8] Ronen, E., et al., "The 9 Lives of Bleichenbacher's CAT: New Cache Attacks on TLS Implementations," *Proceedings of the 2019 IEEE Symposium on Security and Privacy,* IEEE, 2019, pp. 435–452.

[9] Klíma, V., O. Pokorný, and T. Rosa, "Attacking RSA-Based Sessions in SSL/TLS," *Proceedings of Cryptographic Hardware and Embedded Systems (CHES),* Springer-Verlag, September 2003, pp. 426–440.

[10] Kaliski, B., and J. Staddon, "PKCS #1: RSA Cryptography Specifications Version 2.0," RFC 2437, October 1998.

[11] Bellare, M., and P. Rogaway, "Optimal Asymmetric Encryption," *Proceedings of EUROCRYPT '94,* Springer-Verlag, LNCS 950, 1994, pp. 92–111.

[12] Manger, J., "A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS#1 v2.0," *Proceedings of CRYPTO '01,* Springer-Verlag, August 2001, pp. 230–238.

[13] Jonsson, J., and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," RFC 3447, February 2003.

[14] Moriarty, K. (Ed.), et al., "PKCS #1: RSA Cryptography Specifications Version 2.2," RFC 8017, November 2016.

[15] Vaudenay, S., "Security Flaws Induced by CBC Padding—Applications to SSL, IPSEC, WTLS...," *Proceedings of EUROCRYPT '02,* Amsterdam, the Netherlands, Springer-Verlag, LNCS 2332, 2002, pp. 534–545.

[16] Canvel, B., et al., "Password Interception in a SSL/TLS Channel," *Proceedings of CRYPTO '03,* Springer-Verlag, LNCS 2729, 2003, pp. 583–599.

[17] Rizzo, J., and T. Duong, "Practical Padding Oracle Attacks," *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT 2010),* held in conjunction with the 19th USENIX Security Symposium, USENIX Association, Berkeley, CA, 2010, Article No. 1–8.

[18] Duong, T., and J. Rizzo, "Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET," *Proceedings of the IEEE Symposium on Security and Privacy,* Berkeley, CA, 2011, pp. 481–489.

[19] AlFardan, N.J., and K.G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," *Proceedings of the IEEE Symposium on Security and Privacy,* May 2013, pp. 526–540.

[20] Merkle, R.C., "One Way Hash Functions and DES," *Proceedings of CRYPTO '89,* Springer-Verlag, LNCS 435, 1989, pp. 428–446.

[21] Damgård, I.B., "A Design Principle for Hash Functions," *Proceedings of CRYPTO '89,* Springer-Verlag, LNCS 435, 1989, pp. 416–427.

[22] AlFardan, N.J., and K.G. Paterson, "Plaintext-Recovery Attacks against Datagram TLS," *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012),* The Internet Society, February 2012.

[23] Ronen, E., K.G. Paterson, and A. Shamir, "Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure," *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS 2018),* ACM Press, New York, NY, October 2018, pp. 1397–1414.

[24] Bard, G.V., "Vulnerability of SSL to Chosen-Plaintext Attack," Cryptology ePrint Archive, Report 2004/111, 2004.

[25] Bard, G.V., "A Challenging But Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL," Cryptology ePrint Archive, Report 2006/136, 2006.

[26] Bard, G.V., "Blockwise-Adaptive Chosen-Plaintext Attack and Online Modes of Encryption," *Proceedings of the 11th IMA International Conference on Cryptography and Coding '07,* Springer-Verlag, LNCS 4887, 2007, pp. 129–151.

[27] Barnes, R., et al., "Deprecating Secure Sockets Layer Version 3.0," RFC 7568, June 2015.

[28] Moeller, B., and A. Langley, "TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks," RFC 7507, April 2015.

[29] Rescorla, E., S. Dispensa, and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension," RFC 5746, February 2010.

[30] Bhargavan, K., et al., "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS," *Proceedings of the 2014 IEEE Symposium on Security and Privacy,* IEEE Computer Society, 2014, pp. 98–113.

[31] Blake-Wilson, S., and A. Menezes, "Unknown Key-Share Attacks on the Station-to-Station (STS) Protocol," *Proceedings of the Second International Workshop on Practice and Theory in Public Key Cryptography (PKC '99),* Springer, LNCS 1560, 1999, pp. 154–170.

[32] Bhargavan, K. Ed., et al., "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension," RFC 7627, September 2015.

[33] Kelsey, J., "Compression and Information Leakage of Plaintext," *Proceedings of the 9th International Workshop on Fast Software Encryption (FSE 2002),* Springer, LNCS 2365, 2002, pp. 263–276.

[34] Tezcan, C., and S. Vaudenay, "On Hiding a Plaintext Length by Preencryption," *Proceedings of the 9th International Conference on Applied Cryptography and Network Security (ACNS 2011),* Springer, LNCS 6715, 2011, pp. 345–358.

[35] Beurdouch, B., et al, "A Messy State of the Union: Taming the Composite State Machines of TLS," *Proceedings of the 36th IEEE Symposium on Security and Privacy,* 2015, pp. 535–552.

[36] Adrian, D., et al., "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," *Proceedings of the ACM Conference in Computer and Communications Security,* ACM Press, New York, NY, 2015, pp. 5–17.

Appendix B

TLS Cipher Suites

This appendix itemizes the TLS cipher suites that are registered by the IANA at the time of this writing. This is a snapshot, and a currently valid list can be downloaded from the respective IANA repository[1] (with a few changes described in [1]). For each cipher suite, the two-byte reference code (in hexadecimal notation) is provided in the first column, the official name in the second column, an indication (X) whether it can be used for DTLS in the third column, and some reference RFC numbers in the fourth column (where number X refers to RFC X). The RFCs that refer to official specifications of 1.0 (RFC 2246), TLS 1.1 (RFC 4346), TLS 1.2 (RFC 5246), TLS 1.3 (RFC 8446), DTLS 1.0 (RFC 4347), DTLS 1.2 (RFC 6347), and DTLS 1.3 (RFC 9147) are omitted from the table (but the RFC documents that are made obsolete by them are still listed for reference). The cipher suites written in bold are meant to be recommended to support by the IETF consensus process.

00 00	TLS_NULL_WITH_NULL_NULL	X	
00 01	TLS_RSA_WITH_NULL_MD5	X	
00 02	TLS_RSA_WITH_NULL_SHA	X	
00 03	TLS_RSA_EXPORT_WITH_RC4_40_MD5		
00 04	TLS_RSA_WITH_RC4_128_MD5		
00 05	TLS_RSA_WITH_RC4_128_SHA		
00 06	TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	X	
00 07	TLS_RSA_WITH_IDEA_CBC_SHA	X	5469
00 08	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	X	
00 09	TLS_RSA_WITH_DES_CBC_SHA	X	5469
00 0A	TLS_RSA_WITH_3DES_EDE_CBC_SHA	X	
00 0B	TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	X	
00 0C	TLS_DH_DSS_WITH_DES_CBC_SHA	X	5469

1 https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4.

00 0D	TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	X	
00 0E	TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	X	
00 0F	TLS_DH_RSA_WITH_DES_CBC_SHA	X	5469
00 10	TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	X	
00 11	TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	X	
00 12	TLS_DHE_DSS_WITH_DES_CBC_SHA	X	5469
00 13	TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	X	
00 14	TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	X	
00 15	TLS_DHE_RSA_WITH_DES_CBC_SHA	X	5469
00 16	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	X	
00 17	TLS_DH_anon_EXPORT_WITH_RC4_40_MD5		
00 18	TLS_DH_anon_WITH_RC4_128_MD5		
00 19	TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	X	
00 1A	TLS_DH_anon_WITH_DES_CBC_SHA	X	5469
00 1B	TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	X	
00 1E	TLS_KRB5_WITH_DES_CBC_SHA	X	2712
00 1F	TLS_KRB5_WITH_3DES_EDE_CBC_SHA	X	2712
00 20	TLS_KRB5_WITH_RC4_128_SHA	X	2712
00 21	TLS_KRB5_WITH_IDEA_CBC_SHA	X	2712
00 22	TLS_KRB5_WITH_DES_CBC_MD5	X	2712
00 23	TLS_KRB5_WITH_3DES_EDE_CBC_MD5	X	2712
00 24	TLS_KRB5_WITH_RC4_128_MD5		2712
00 25	TLS_KRB5_WITH_IDEA_CBC_MD5	X	2712
00 26	TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA	X	2712
00 27	TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA	X	2712
00 28	TLS_KRB5_EXPORT_WITH_RC4_40_SHA		2712
00 29	TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5	X	2712
00 2A	TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5	X	2712
00 2B	TLS_KRB5_EXPORT_WITH_RC4_40_MD5		2712
00 2C	TLS_PSK_WITH_NULL_SHA	X	4785
00 2D	TLS_DHE_PSK_WITH_NULL_SHA	X	4785
00 2E	TLS_RSA_PSK_WITH_NULL_SHA	X	4785
00 2F	TLS_RSA_WITH_AES_128_CBC_SHA	X	
00 30	TLS_DH_DSS_WITH_AES_128_CBC_SHA	X	
00 31	TLS_DH_RSA_WITH_AES_128_CBC_SHA	X	
00 32	TLS_DHE_DSS_WITH_AES_128_CBC_SHA	X	
00 33	TLS_DHE_RSA_WITH_AES_128_CBC_SHA	X	
00 34	TLS_DH_anon_WITH_AES_128_CBC_SHA	X	
00 35	TLS_RSA_WITH_AES_256_CBC_SHA	X	
00 36	TLS_DH_DSS_WITH_AES_256_CBC_SHA	X	
00 37	TLS_DH_RSA_WITH_AES_256_CBC_SHA	X	
00 38	TLS_DHE_DSS_WITH_AES_256_CBC_SHA	X	
00 39	TLS_DHE_RSA_WITH_AES_256_CBC_SHA	X	
00 3A	TLS_DH_anon_WITH_AES_256_CBC_SHA	X	

00 3B	TLS_RSA_WITH_NULL_SHA256	X	
00 3C	TLS_RSA_WITH_AES_128_CBC_SHA256	X	
00 3D	TLS_RSA_WITH_AES_256_CBC_SHA256	X	
00 3E	TLS_DH_DSS_WITH_AES_128_CBC_SHA256	X	
00 3F	TLS_DH_RSA_WITH_AES_128_CBC_SHA256	X	
00 40	TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	X	
00 41	TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	X	5932
00 42	TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA	X	5932
00 43	TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA	X	5932
00 44	TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	X	5932
00 45	TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	X	5932
00 46	TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA	X	5932
00 67	TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	X	
00 68	TLS_DH_DSS_WITH_AES_256_CBC_SHA256	X	
00 69	TLS_DH_RSA_WITH_AES_256_CBC_SHA256	X	
00 6A	TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	X	
00 6B	TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	X	
00 6C	TLS_DH_anon_WITH_AES_128_CBC_SHA256	X	
00 6D	TLS_DH_anon_WITH_AES_256_CBC_SHA256	X	
00 84	TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	X	5932
00 85	TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA	X	5932
00 86	TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA	X	5932
00 87	TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	X	5932
00 88	TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	X	5932
00 89	TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA	X	5932
00 8A	TLS_PSK_WITH_RC4_128_SHA		
00 8B	TLS_PSK_WITH_3DES_EDE_CBC_SHA	X	4279
00 8C	TLS_PSK_WITH_AES_128_CBC_SHA	X	4279
00 8D	TLS_PSK_WITH_AES_256_CBC_SHA	X	4279
00 8E	TLS_DHE_PSK_WITH_RC4_128_SHA		4279
00 8F	TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA	X	4279
00 90	TLS_DHE_PSK_WITH_AES_128_CBC_SHA	X	4279
00 91	TLS_DHE_PSK_WITH_AES_256_CBC_SHA	X	4279
00 92	TLS_RSA_PSK_WITH_RC4_128_SHA		4279
00 93	TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA	X	4279
00 94	TLS_RSA_PSK_WITH_AES_128_CBC_SHA	X	4279
00 95	TLS_RSA_PSK_WITH_AES_256_CBC_SHA	X	4279
00 96	TLS_RSA_WITH_SEED_CBC_SHA	X	4162
00 97	TLS_DH_DSS_WITH_SEED_CBC_SHA	X	4162
00 98	TLS_DH_RSA_WITH_SEED_CBC_SHA	X	4162
00 99	TLS_DHE_DSS_WITH_SEED_CBC_SHA	X	4162
00 9A	TLS_DHE_RSA_WITH_SEED_CBC_SHA	X	4162
00 9B	TLS_DH_anon_WITH_SEED_CBC_SHA	X	4162
00 9C	TLS_RSA_WITH_AES_128_GCM_SHA256	X	5288

00 9D	TLS_RSA_WITH_AES_256_GCM_SHA384	X	5288
00 9E	**TLS_DHE_RSA_WITH_AES_128_GCM_SHA256**	X	5288
00 9F	**TLS_DHE_RSA_WITH_AES_256_GCM_SHA384**	X	5288
00 A0	TLS_DH_RSA_WITH_AES_128_GCM_SHA256	X	5288
00 A1	TLS_DH_RSA_WITH_AES_256_GCM_SHA384	X	5288
00 A2	TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	X	5288
00 A3	TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	X	5288
00 A4	TLS_DH_DSS_WITH_AES_128_GCM_SHA256	X	5288
00 A5	TLS_DH_DSS_WITH_AES_256_GCM_SHA384	X	5288
00 A6	TLS_DH_anon_WITH_AES_128_GCM_SHA256	X	5288
00 A7	TLS_DH_anon_WITH_AES_256_GCM_SHA384	X	5288
00 A8	TLS_PSK_WITH_AES_128_GCM_SHA256	X	5487
00 A9	TLS_PSK_WITH_AES_256_GCM_SHA384	X	5487
00 AA	**TLS_DHE_PSK_WITH_AES_128_GCM_SHA256**	X	5487
00 AB	**TLS_DHE_PSK_WITH_AES_256_GCM_SHA384**	X	5487
00 AC	TLS_RSA_PSK_WITH_AES_128_GCM_SHA256	X	5487
00 AD	TLS_RSA_PSK_WITH_AES_256_GCM_SHA384	X	5487
00 AE	TLS_PSK_WITH_AES_128_CBC_SHA256	X	5487
00 AF	TLS_PSK_WITH_AES_256_CBC_SHA384	X	5487
00 B0	TLS_PSK_WITH_NULL_SHA256	X	5487
00 B1	TLS_PSK_WITH_NULL_SHA384	X	5487
00 B2	TLS_DHE_PSK_WITH_AES_128_CBC_SHA256	X	5487
00 B3	TLS_DHE_PSK_WITH_AES_256_CBC_SHA384	X	5487
00 B4	TLS_DHE_PSK_WITH_NULL_SHA256	X	5487
00 B5	TLS_DHE_PSK_WITH_NULL_SHA384	X	5487
00 B6	TLS_RSA_PSK_WITH_AES_128_CBC_SHA256	X	5487
00 B7	TLS_RSA_PSK_WITH_AES_256_CBC_SHA384	X	5487
00 B8	TLS_RSA_PSK_WITH_NULL_SHA256	X	5487
00 B9	TLS_RSA_PSK_WITH_NULL_SHA384	X	5487
00 BA	TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256	X	5932
00 BB	TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256	X	5932
00 BC	TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256	X	5932
00 BD	TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256	X	5932
00 BE	TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	X	5932
00 BF	TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256	X	5932
00 C0	TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256	X	5932
00 C1	TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256	X	5932
00 C2	TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256	X	5932
00 C3	TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256	X	5932
00 C4	TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256	X	5932
00 C5	TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256	X	5932
00 C6	TLS_SM4_GCM_SM3	X	8998
00 C6	TLS_SM4_CCM_SM3	X	8998
00 FF	TLS_EMPTY_RENEGOTIATION_INFO_SCSV	X	5746

13 01	**TLS_AES_128_GCM_SHA256**	X	
13 02	**TLS_AES_256_GCM_SHA384**	X	
13 03	**TLS_CHACHA20_POLY1305_SHA256**	X	
13 04	**TLS_AES_128_CCM_SHA256**	X	
13 05	TLS_AES_128_CCM_8_SHA256	X	
56 00	TLS_FALLBACK_SCSV	X	7507
C0 01	TLS_ECDH_ECDSA_WITH_NULL_SHA	X	8422
C0 02	TLS_ECDH_ECDSA_WITH_RC4_128_SHA		8422
C0 03	TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	X	8422
C0 04	TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	X	8422
C0 05	TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	X	8422
C0 06	TLS_ECDHE_ECDSA_WITH_NULL_SHA	X	8422
C0 07	TLS_ECDHE_ECDSA_WITH_RC4_128_SHA		8422
C0 08	TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	X	8422
C0 09	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	X	8422
C0 0A	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	X	8422
C0 0B	TLS_ECDH_RSA_WITH_NULL_SHA	X	8422
C0 0C	TLS_ECDH_RSA_WITH_RC4_128_SHA		8422
C0 0D	TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	X	8422
C0 0E	TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	X	8422
C0 0F	TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	X	8422
C0 10	TLS_ECDHE_RSA_WITH_NULL_SHA	X	8422
C0 11	TLS_ECDHE_RSA_WITH_RC4_128_SHA		8422
C0 12	TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	X	8422
C0 13	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	X	8422
C0 14	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	X	8422
C0 15	TLS_ECDH_anon_WITH_NULL_SHA	X	8422
C0 16	TLS_ECDH_anon_WITH_RC4_128_SHA		8422
C0 17	TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	X	8422
C0 18	TLS_ECDH_anon_WITH_AES_128_CBC_SHA	X	8422
C0 19	TLS_ECDH_anon_WITH_AES_256_CBC_SHA	X	8422
C0 1A	TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA	X	5054
C0 1B	TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA	X	5054
C0 1C	TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA	X	5054
C0 1D	TLS_SRP_SHA_WITH_AES_128_CBC_SHA	X	5054
C0 1E	TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA	X	5054
C0 1F	TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA	X	5054
C0 20	TLS_SRP_SHA_WITH_AES_256_CBC_SHA	X	5054
C0 21	TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA	X	5054
C0 22	TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA	X	5054
C0 23	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	X	5289
C0 24	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	X	5289
C0 25	TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	X	5289
C0 26	TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	X	5289

C0 27	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	X	5289
C0 28	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	X	5289
C0 29	TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	X	5289
C0 2A	TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	X	5289
C0 2B	**TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256**	X	5289
C0 2C	**TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384**	X	5289
C0 2D	TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	X	5289
C0 2E	TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	X	5289
C0 2F	**TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256**	X	5289
C0 30	**TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384**	X	5289
C0 31	TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	X	5289
C0 32	TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	X	5289
C0 33	TLS_ECDHE_PSK_WITH_RC4_128_SHA		5489
C0 34	TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA	X	5489
C0 35	TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA	X	5489
C0 36	TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA	X	5489
C0 37	TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256	X	5489
C0 38	TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384	X	5489
C0 39	TLS_ECDHE_PSK_WITH_NULL_SHA	X	5489
C0 3A	TLS_ECDHE_PSK_WITH_NULL_SHA256	X	5489
C0 3B	TLS_ECDHE_PSK_WITH_NULL_SHA384	X	5489
C0 3C	TLS_RSA_WITH_ARIA_128_CBC_SHA256	X	6209
C0 3D	TLS_RSA_WITH_ARIA_256_CBC_SHA384	X	6209
C0 3E	TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256	X	6209
C0 3F	TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384	X	6209
C0 40	TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256	X	6209
C0 41	TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384	X	6209
C0 42	TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256	X	6209
C0 43	TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384	X	6209
C0 44	TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256	X	6209
C0 45	TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384	X	6209
C0 46	TLS_DH_anon_WITH_ARIA_128_CBC_SHA256	X	6209
C0 47	TLS_DH_anon_WITH_ARIA_256_CBC_SHA384	X	6209
C0 48	TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256	X	6209
C0 49	TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384	X	6209
C0 4A	TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256	X	6209
C0 4B	TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384	X	6209
C0 4C	TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256	X	6209
C0 4D	TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384	X	6209
C0 4E	TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256	X	6209
C0 4F	TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384	X	6209
C0 50	TLS_RSA_WITH_ARIA_128_GCM_SHA256	X	6209
C0 51	TLS_RSA_WITH_ARIA_256_GCM_SHA384	X	6209
C0 52	TLS_DHE_RSA_WITH_ARIA_128_GCM_SHA256	X	6209

C0 53	TLS_DHE_RSA_WITH_ARIA_256_GCM_SHA384	X	6209
C0 54	TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256	X	6209
C0 55	TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384	X	6209
C0 56	TLS_DHE_DSS_WITH_ARIA_128_GCM_SHA256	X	6209
C0 57	TLS_DHE_DSS_WITH_ARIA_256_GCM_SHA384	X	6209
C0 58	TLS_DH_DSS_WITH_ARIA_128_GCM_SHA256	X	6209
C0 59	TLS_DH_DSS_WITH_ARIA_256_GCM_SHA384	X	6209
C0 5A	TLS_DH_anon_WITH_ARIA_128_GCM_SHA256	X	6209
C0 5B	TLS_DH_anon_WITH_ARIA_256_GCM_SHA384	X	6209
C0 5C	TLS_ECDHE_ECDSA_WITH_ARIA_128_GCM_SHA256	X	6209
C0 5D	TLS_ECDHE_ECDSA_WITH_ARIA_256_GCM_SHA384	X	6209
C0 5E	TLS_ECDH_ECDSA_WITH_ARIA_128_GCM_SHA256	X	6209
C0 5F	TLS_ECDH_ECDSA_WITH_ARIA_256_GCM_SHA384	X	6209
C0 60	TLS_ECDHE_RSA_WITH_ARIA_128_GCM_SHA256	X	6209
C0 61	TLS_ECDHE_RSA_WITH_ARIA_256_GCM_SHA384	X	6209
C0 62	TLS_ECDH_RSA_WITH_ARIA_128_GCM_SHA256	X	6209
C0 63	TLS_ECDH_RSA_WITH_ARIA_256_GCM_SHA384	X	6209
C0 64	TLS_PSK_WITH_ARIA_128_CBC_SHA256	X	6209
C0 65	TLS_PSK_WITH_ARIA_256_CBC_SHA384	X	6209
C0 66	TLS_DHE_PSK_WITH_ARIA_128_CBC_SHA256	X	6209
C0 67	TLS_DHE_PSK_WITH_ARIA_256_CBC_SHA384	X	6209
C0 68	TLS_RSA_PSK_WITH_ARIA_128_CBC_SHA256	X	6209
C0 69	TLS_RSA_PSK_WITH_ARIA_256_CBC_SHA384	X	6209
C0 6A	TLS_PSK_WITH_ARIA_128_GCM_SHA256	X	6209
C0 6B	TLS_PSK_WITH_ARIA_256_GCM_SHA384	X	6209
C0 6C	TLS_DHE_PSK_WITH_ARIA_128_GCM_SHA256	X	6209
C0 6D	TLS_DHE_PSK_WITH_ARIA_256_GCM_SHA384	X	6209
C0 6E	TLS_RSA_PSK_WITH_ARIA_128_GCM_SHA256	X	6209
C0 6F	TLS_RSA_PSK_WITH_ARIA_256_GCM_SHA384	X	6209
C0 70	TLS_ECDHE_PSK_WITH_ARIA_128_CBC_SHA256	X	6209
C0 71	TLS_ECDHE_PSK_WITH_ARIA_256_CBC_SHA384	X	6209
C0 72	TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256	X	6367
C0 73	TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384	X	6367
C0 74	TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256	X	6367
C0 75	TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384	X	6367
C0 76	TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	X	6367
C0 77	TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384	X	6367
C0 78	TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256	X	6367
C0 79	TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384	X	6367
C0 7A	TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 7B	TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 7C	TLS_DHE_RSA_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 7D	TLS_DHE_RSA_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 7E	TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256	X	6367

C0 7F	TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 80	TLS_DHE_DSS_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 81	TLS_DHE_DSS_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 82	TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 83	TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 84	TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 85	TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 86	TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 87	TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 88	TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 89	TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 8A	TLS_ECDHE_RSA_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 8B	TLS_ECDHE_RSA_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 8C	TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 8D	TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 8E	TLS_PSK_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 8F	TLS_PSK_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 90	TLS_DHE_PSK_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 91	TLS_DHE_PSK_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 92	TLS_RSA_PSK_WITH_CAMELLIA_128_GCM_SHA256	X	6367
C0 93	TLS_RSA_PSK_WITH_CAMELLIA_256_GCM_SHA384	X	6367
C0 94	TLS_PSK_WITH_CAMELLIA_128_CBC_SHA256	X	6367
C0 95	TLS_PSK_WITH_CAMELLIA_256_CBC_SHA384	X	6367
C0 96	TLS_DHE_PSK_WITH_CAMELLIA_128_CBC_SHA256	X	6367
C0 97	TLS_DHE_PSK_WITH_CAMELLIA_256_CBC_SHA384	X	6367
C0 98	TLS_RSA_PSK_WITH_CAMELLIA_128_CBC_SHA256	X	6367
C0 99	TLS_RSA_PSK_WITH_CAMELLIA_256_CBC_SHA384	X	6367
C0 9A	TLS_ECDHE_PSK_WITH_CAMELLIA_128_CBC_SHA256	X	6367
C0 9B	TLS_ECDHE_PSK_WITH_CAMELLIA_256_CBC_SHA384	X	6367
C0 9C	TLS_RSA_WITH_AES_128_CCM	X	6655
C0 9D	TLS_RSA_WITH_AES_256_CCM	X	6655
C0 9E	**TLS_DHE_RSA_WITH_AES_128_CCM**	X	6655
C0 9F	**TLS_DHE_RSA_WITH_AES_256_CCM**	X	6655
C0 A0	TLS_RSA_WITH_AES_128_CCM_8	X	6655
C0 A1	TLS_RSA_WITH_AES_256_CCM_8	X	6655
C0 A2	TLS_DHE_RSA_WITH_AES_128_CCM_8	X	6655
C0 A3	TLS_DHE_RSA_WITH_AES_256_CCM_8		6655
C0 A4	TLS_PSK_WITH_AES_128_CCM	X	6655
C0 A5	TLS_PSK_WITH_AES_256_CCM	X	6655
C0 A6	**TLS_DHE_PSK_WITH_AES_128_CCM**	X	6655
C0 A7	**TLS_DHE_PSK_WITH_AES_256_CCM**	X	6655
C0 A8	TLS_PSK_WITH_AES_128_CCM_8	X	6655
C0 A9	TLS_PSK_WITH_AES_256_CCM_8	X	6655
C0 AA	TLS_PSK_DHE_WITH_AES_128_CCM_8	X	6655

C0 AB	TLS_PSK_DHE_WITH_AES_256_CCM_8	X	6655
C0 AC	TLS_ECDHE_ECDSA_WITH_AES_128_CCM	X	7251
C0 AD	TLS_ECDHE_ECDSA_WITH_AES_256_CCM	X	7251
C0 AE	TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8	X	7251
C0 AF	TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8	X	7251
C0 B0	TLS_ECCPWD_WITH_AES_128_GCM_SHA256	X	8492
C0 B1	TLS_ECCPWD_WITH_AES_256_GCM_SHA384	X	8492
C0 B2	TLS_ECCPWD_WITH_AES_128_CCM_SHA256	X	8492
C0 B3	TLS_ECCPWD_WITH_AES_256_CCM_SHA384	X	8492
C0 B4	TLS_SHA256_SHA256	X	9150
C0 B5	TLS_SHA384_SHA384	X	9150
C1 00	TLS_GOSTR341112_256_WITH_KUZNYECHIK_CTR_OMAC		9189
C1 01	TLS_GOSTR341112_256_WITH_MAGMA_CTR_OMAC		9189
C1 02	TLS_GOSTR341112_256_WITH_28147_CNT_IMIT		9189
C1 03	TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_L		
C1 04	TLS_GOSTR341112_256_WITH_MAGMA_MGM_L		
C1 05	TLS_GOSTR341112_256_WITH_KUZNYECHIK_MGM_S		
C1 06	TLS_GOSTR341112_256_WITH_MAGMA_MGM_S		
CC A8	**TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256**	X	7905
CC A9	**TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256**	X	7905
CC AA	**TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256**	X	7905
CC AB	TLS_PSK_WITH_CHACHA20_POLY1305_SHA256	X	7905
CC AC	**TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256**	X	7905
CC AD	**TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256**	X	7905
CC AE	TLS_RSA_PSK_WITH_CHACHA20_POLY1305_SHA256	X	7905
D0 01	**TLS_ECDHE_PSK_WITH_AES_128_GCM_SHA256**	X	8442
D0 02	**TLS_ECDHE_PSK_WITH_AES_256_GCM_SHA384**	X	8442
D0 03	TLS_ECDHE_PSK_WITH_AES_128_CCM__8_SHA256	X	8442
D0 05	**TLS_ECDHE_PSK_WITH_AES_128_CCM_SHA256**	X	8442

Reference

[1] Salowey, J., and S. Turner, "IANA Registry Updates for TLS and DTLS," RFC 8447, August 2018.

Appendix C

TLS Extensions

In this appendix, we overview and explain the TLS extensions that are registered by the IANA at the time of this writing. Note that this is just a snapshot in time, and that a currently valid and probably more comprehensive list of TLS extensions can be downloaded from the respective IANA repository.[1]

C.1 OVERVIEW

In this section, we overview the TLS extensions in tabular form. For each extension, the (decimal) value is provided in the first column, the extension name in the second column, and some additional references in terms of complementary RFCs in the third column. To keep the table as simple as possible, the RFCs that refer to the official specifications of TLS 1.0 (RFC 2246), TLS 1.1 (RFC 4346), TLS 1.2 (RFC 5246), TLS 1.3 (RFC 8446), DTLS 1.0 (RFC 4347), DTLS 1.2 (RFC 6347), and DTLS 1.3 (RFC 9147) are omitted from the list (but some RFCs that are made obsolete by these specifications are still listed for the sake of completeness).

Value	Extension Name	Additional References
0	`server_name`	RFC 6066 [1], RFC 9261 [2]
1	`max_fragment_length`	RFC 6066 [1], RFC 8449 [3]
2	`client_certificate_url`	RFC 6066 [1]
3	`trusted_ca_keys`	RFC 6066 [1]
4	`truncated_hmac`	RFC 6066 [1]
5	`status_request`	RFC 6066 [1]
6	`user_mapping`	RFC 4681 [4]

1 https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml.

7	client_authz	RFC 5878 [5]
8	server_authz	RFC 5878 [5]
9	cert_type	RFC 6091 [6]
10	supported_groups	RFC 8422 [7], RFC 7919 [8]
11	ec_point_formats	RFC 8422 [7]
12	srp	RFC 5054 [9]
13	signature_algorithms	
14	use_srtp	RFC 5764 [10]
15	heartbeat	RFC 6520 [11]
16	application_layer_protocol_negotiation	RFC 7301 [12]
17	status_request_v2	RFC 6961 [13]
18	signed_certificate_timestamp	RFC 6962 [14]
19	client_certificate_type	RFC 7250 [15]
20	server_certificate_type	RFC 7250 [15]
21	padding	RFC 7685 [16]
22	encrypt_then_mac	RFC 7366 [17]
23	extended_master_secret	RFC 7627 [18]
24	token_binding	RFC 8472 [19]
25	cached_info	RFC 7924 [20]
26	tls_lts	Internet-Draft
27	compress_certificate	RFC 8879 [21]
28	record_size_limit	RFC 8449 [3]
29	pwd_protect	RFC 8492 [22]
30	pwd_clear	RFC 8492 [22]
31	password_salt	RFC 8492 [22]
32	ticket_pinning	RFC 8672 [23]
33	tls_cert_with_extern_psk	RFC 8773 [24]
34	delegated_credentials	Internet-Draft
35	session_ticket	RFC 5077 [25], RFC 8447 [26]
36	TLMSP	ETSI TS 103 523-2 [27]
37	TLMSP_proxying	ETSI TS 103 523-2 [27]
38	TLMSP_delegate	ETSI TS 103 523-2 [27]
39	supported_ekt_ciphers	RFC 8870 [28]
41	pre_shared_key	
42	early_data	
43	supported_versions	
44	cookie	
45	psk_key_exchange_modes	
47	certificate_authorities	

48	`oid_filters`	
49	`post_handshake_auth`	
50	`signature_algorithms_cert`	
51	`key_share`	
52	`transparency_info`	RFC 9162 [29]
54	`connection_id`	RFC 9146 [30]
55	`external_id_hash`	RFC 8844 [31]
56	`external_session_id`	RFC 8844 [31]
57	`quic_transport_parameters`	RFC 9001 [32]
58	`ticket_request`	RFC 9149 [33]
59	`dnssec_chain`	RFC 9102 [34]
65281	`renegotiation_info`	RFC 5746 [35]

C.2 DETAILED EXPLANATIONS

In this section, we go through the list provided in the previous section and provide some advanced and more detailed explanations for each TLS extension (referenced by the extension name and its value in brackets). We exclude from our discussion the extensions that are specified in Internet-Drafts (i.e., `tls_lts` (26) and `delegated_credentials` (34)). These extensions are not stable and still subject to change.

C.2.1 **`Server_name`** (0) Extension

Virtual hosting is a commonly used method to host multiple servers (e.g., web servers) with different domain names on the same machine, possibly using a single IP address. Virtual hosting is a basic requirement for the ongoing trend toward server virtualization and cloud computing. In the case of HTTP, for example, it is common to host many (virtual) web servers on the same machine or cluster of machines. If, for example, a user directs his or her browser to `www.esecurity.ch`, then a DNS lookup reveals that the respective server runs on a machine with a particular IP address (e.g., 49.12.165.202). The browser then establishes a TCP connection to port 80 of this machine, and the client and server use HTTP to talk to each other. In particular, since the browser wants to contact `www.esecurity.ch`, the client sends an HTTP request message in which it specifies the web server's DNS host name in a special host header. This may look as follows:

```
Host: www.esecurity.ch
```

This allows the server machine to distinguish the HTTP requests that target different web servers, and hence to host multiple web servers on that machine. This works perfectly fine with HTTP.

Consider what happens if one wants to invoke SSL/TLS and use HTTPS instead of HTTP. Then an SSL/TLS connection must be established (typically to port 443), before HTTP can be invoked. At this point in time, however, it is not clear how to distinguish between different (virtual) web servers that reside on the same machine. For a long time, there was no analog to an HTTP `Host` header, and hence the only possibility was to use a unique IP address for each SSL/TLS-enabled web server. This did not scale well, and hence people defined a TLS server name extension known as *server name indication* (SNI) [1] that is conceptually similar to an HTTP `Host` header. Using SNI, a client can send a CLIENTHELLO message with an extension of type `server_name` to communicate to the server the (fully qualified) DNS hostname of the web server it wants to connect to. So there is no need to use distinct IP addresses, and a server machine (with a unique IP address) can now host multiple SSL/TLS-enabled web servers. This is important for the large-scale deployment of the TLS protocol.[2]

The SNI extension allows one to specify an HTTPS server, but it is transmitted in the clear (as part of a CLIENTHELLO message). This means that it leaks information about what web server a client currently wants to retrieve data from. To mitigate this information leakage and potential privacy problem, people have come up with the notions of an *encrypted SNI* (ESNI) and—more recently—an encrypted CLIENTHELLO (ECH) message.

- The idea of ESNI is to have the client encrypt the SNI with an ESNI public key that is found in the DNS record of the respective web server, and to use some complementary security technologies, like DNSSEC, DNS over TLS, and/or DNS over HTTPS (DoH). This, by the way, is also where the complementary RFC from above (i.e., [2]) comes into play (it is not further addressed here).
- The idea of ECH is to broaden the scope of privacy protection to other extensions (than the SNI extension), and to encrypt the relevant parts of the CLIENTHELLO message. This introduces a number of difficulties and challenges that have not all been solved yet.

ESNI and ECH are primarily used in the realm of TLS 1.3 and discussed in this context (Section 3.5). While SNI is well established, ESNI and ECH are still in an experimental stage.

2 At the Black Hat 2014 conference, it was shown that virtual hosting also introduces new security problems, and that the proper configuration of virtual hosts using SNI is not trivial and thus error-prone (see `https://bh.ht.vc/vhost_confusion.pdf`).

C.2.2 `Max_fragment_length` (1) and `Record_size_limit` (28) Extensions

Section 2.2 mentioned that the maximum fragment length of an SSL record is 2^{14} bytes. According to Section 3.1, this restriction also applies to TLS. In many situations, it is reasonable to work with fragments of that length. There are, however, also situations in which the clients are constrained and need to operate on smaller fragments. Also, there may be bandwidth constraints that may require smaller fragments (as mentioned in Chapter 4, this is particularly true for the DTLS protocol). This is where the `max_fragment_length` extension [1] comes into play. It can be used by a client to communicate to the server that it needs to negotiate a smaller maximal fragment length. The actual maximum fragment length is sent in the data field of the `max_fragment_length` extension. Supported values are 1 (standing for 2^9 bytes), 2 (standing for 2^{10} bytes), 3 (standing for 2^{11} bytes), and 4 (standing for 2^{12} bytes).

As outlined in [3], the `max_fragment_length` extension suffers from several design problems, and hence a new extension type (i.e., `record_size_limit`) is proposed as a replacement (basically deprecating `max_ fragment_length`). Instead of restricting the maximum fragment length to a few supported values, such as 2^9, 2^{10}, 2^{11}, or 2^{12} bytes, the data field of the `record_size_limit` extension can now specify any value that refers to the maximum number of bytes in a TLS record that is accepted (note that the value must not be smaller than 64 bytes). In TLS 1.2 and earlier versions, this limit refers to the fragment of a TLS record (without padding added as part of encryption), whereas in TLS 1.3, this limit refers to plaintext of a protected record (including the content type and padding, i.e., the complete length of the TLS inner plaintext).

In either case (i.e., independent from whether the `max_fragment_length` or `record_size_limit` extension is used), the negotiated length applies for the entire duration of a TLS session, including all session resumptions. Hence, it is fixed and cannot be changed dynamically at will. If another length is required, then a new session must be negotiated (using a suitable value for the maximum fragment length).

C.2.3 `Client_certificate_url` (2) Extension

Instead of including a full-fledged client certificate (or certificate chain, respectively) in a CERTIFICATE message sent to the server, it is sometimes advantageous to only include a *client certificate URL* (that informs the server where it can retrieve the certificate). This is computationally less expensive and requires less memory. Other possibilities to reduce the size of a certificate are enabled by the `cached_info` and `compress_certificate` extensions addressed below.

If a client wants to use a certificate URL (instead of a full-fledged client certificate), then it can include an extension of type `client_certificate_url` [1] in the CLIENTHELLO message sent to the server. If the server is willing to support the mechanism, then it returns the same extension in the SERVERHELLO message. The client and server then know that they can both use the client certificate URL mechanism in the handshake. But the use of the mechanism remains optional, meaning that a client can still either send a CERTIFICATE message (type 11) or a CERTIFICATEURL message (type 21) to the server.

From a security viewpoint, the client certificate URL mechanism has a major problem: If it is naively implemented, then it can be misused to direct a server to a malicious website where a drive-by infection may take place. This problem must be taken into account, when the mechanism is invoked on the server side. In fact, it is highly recommended to implement and put in place some complementary protection mechanisms, such as data content inspection, before the extension is activated. It should never be activated automatically and by default.

C.2.4 **`Trusted_ca_keys`** (3) Extension

In a normal SSL/TLS handshake, the server does not know in advance what root CAs the client is willing to accept when it has to provide a certificate (chain) in the CERTIFICATE message. So it may happen that the client does not accept the certificate provided by the server, and hence that the handshake must be repeated, possibly many times. This is not efficient, especially if the client is configured with only a few root CAs (for example, due to memory constraints).

Against this background, the `trusted_ca_keys` extension [1] can be included in the CLIENTHELLO message to inform the server what root CAs the client is willing to accept. The root CAs are encoded in the data field of the extension, using one of the many possible ways of identifying them. If the server is willing to take advantage of this extension, then it sends back a SERVERHELLO message with the `trusted_ca_keys` extension and an empty data field (so the server only signals support for the extension). As usual, the actual certificate signed by one of the root CAs accepted by the client is then provided later in the handshake (i.e., in the CERTIFICATE message provided by the server).

As mentioned below, the `trusted_ca_keys` extension is no longer used in TLS 1.3, and a `certificate_authorities` extension is used instead.

C.2.5 `Truncated_hmac` (4) Extension

The output of the HMAC construction is equally as long as the output of the hash function in use; that is 128 bits for MD5, 160 bits for SHA-1, and 256 bits for SHA-256. If the HMAC construction is used in constrained environments, then the output of the HMAC construction may be truncated to, for example, 80 bits. The extension type `truncated_hmac` can be used for this purpose [1]. If the client and server both support it, then they can start using truncated HMAC values instead of full-length values. Note, however, that this affects only the computation of the HMAC values used for authentication (and does not affect the way the HMAC construction is used in the TLS PRF for master secret and key derivation). Due to some cryptanalytical results, the use of truncated HMAC values (and hence the `truncated_hmac` extension) is strongly discouraged. Combined with the fact that the use of an AEAD cipher is highly recommended (and such a cipher is not based on the HMAC construction), the `truncated_hmac` extension is only seldom used in the field.

C.2.6 `Status_request` (5) and `Status_request_v2` (17) Extensions

When an entity (client or server) receives a certificate in an SSL/TLS handshake, it is usually this entity's task to verify the validity of the certificate. As further addressed in Chapter 6, this can be done by retrieving a CRL and checking that the certificate is not included, or by making use of OCSP. This is simple and straightforward and normally does not pose any problem. Only if the certificate-receiving entity is a constrained client, it may be the case that retrieving and checking a CRL or making an OCSP query is too expensive. In this case, an alternative that burdens the server (instead of the client) may be preferred, and this is where the extension type `status_request` [1] comes into play. It can be used by a client to signal to the server that it wishes to receive some status information about the certificate, such as an OCSP response (with some additional information encoded in the extension's data field). If the server is willing to provide the requested certificate status information, then it sends a respective CERTIFICATESTATUS message (type 22) back to the client. Due to the fact that the certificate status information is strongly related to OCSP, the mechanism enabled by this extension is also known as *OCSP stapling*. It means that the server queries the OCSP status of its certificate (instead of the client). This basically moves the workload from the client to the server.

OCSP stapling as enabled by the `status_request` extension supports only one OCSP response and can be used to check the revocation status of a single server certificate. This is a problem, if such a certificate comes with a chain, and hence multiple certificates need to be checked collectively. This limitation is addressed with a multiple certificate status extension specified in RFC 6961 [13]. The respective extension type is `status_request_v2`. Consequently, the `status_request` and `status_ request_v2` extensions only differ in the number of server certificates that can be handled simultaneously, and hence the `status_request_v2` extension improves the efficiency of OCSP stapling considerably.

C.2.7 `User_mapping` (6) Extension

The user mapping extension is the first extension that is not introduced in [1]. Instead, it is specified in [4] that is based on a generic mechanism provided in RFC 4680 [36]. The generic mechanism can be used to have a TLS handshake support the exchange of supplemental data. The respective TLS handshake message flow is illustrated in Figure C.1. The client and server exchange extended hello messages, before they can exchange supplemental data in a SUPPLEMENTALDATA message (type 23).

Using the generic mechanism, [4] specifies the extension `user_mapping` and a payload format for the SUPPLEMENTALDATA message that can be used to accommodate mapping of users to their accounts when client authentication is invoked. More specifically, the client can send the extension in its CLIENTHELLO message, and the server can confirm it in its SERVERHELLO message. The client can afterward include the user mapping data in a SUPPLEMENTALDATA message sent to the server. It is then up to the server to interpret and make use of this data in some appropriate way. Needless to say that this is beyond the TLS extension mechanism.

C.2.8 `Client_authz` (7) and `Server_authz` (8) Extensions

In its native form, the TLS protocol does not provide authorization—it only provides authentication (and client authentication only in the case of mTLS). There are, however, two TLS extensions (i.e., `client_authz` and `server_authz` [5]) that can be used for this purpose. Using the `client_authz` extension, the client can signal what authorization data formats it currently supports, and the server can do the same using the `server_authz` extension. Both extensions are sent in respective hello messages. Similar to the user mapping mechanism, the actual authorization data (in one of the mutually agreed formats) can be provided in a SUPPLEMENTALDATA message as illustrated in Figure C.1. Instead of directly providing this data, it is also possible to provide a URL that can be used to retrieve the information from an online repository. This is similar to the client certificate URL extension explained above.

In theory, there are many authorization data formats that can be used here. In practice, however, it is mainly about attribute certificates (introduced in Chapter 6)

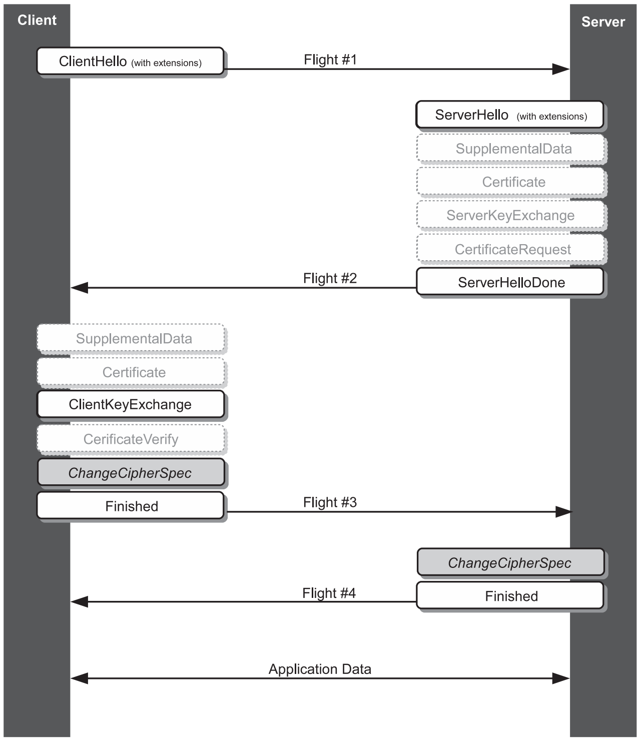

Figure C.1 The TLS handshake protocol supporting the exchange of supplemental data.

and security assertion markup language (SAML) assertions. A comprehensive list of authorization data formats that are commonly supported is available in the respective IANA repository.[3]

C.2.9 `Cert_type` (9) Extension

As further addressed in Section 6.1, there are two competing standards for public key certificates: X.509 and OpenPGP. While the SSL/TLS protocols natively support only X.509 certificates, there are situations in which X.509 certificates are not available or non-X.509 certificates are more appropriate. To accommodate situations like

3 https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#authorization-data.

these, the IETF TLS WG has released RFC 6091 [6] that defines the extension type `cert_type` and specifies how non-X.509 certificates (i.e., OpenPGP certificates) can be used in a TLS setting.

If a client wants to signal its support for non-X.509 certificates, then it must include a `cert_type` extension in the CLIENTHELLO message. The data field of the extension must carry a list of supported certificate types, sorted by client preference. As the only type values currently supported are X.509 (0) and OpenPGP (1), this list is going to be short. If the server receives a CLIENTHELLO message with the `cert_type` extension and chooses a cipher suite that requires a certificate, then it must either select a certificate type from the list of client-supported certificate types or terminate the connection with an `unsupported_certificate` (code 43) alert message that is fatal (in TLS 1.2 or before). In the first case, the server must encode the selected certificate type in an extended SERVERHELLO message with a respective `cert_type` extension.

It goes without saying that the certificate type that is negotiated also influences the certificates that are sent in the CERTIFICATE messages. If the use of X.509 certificates is negotiated, then the certificates are of this type. This is going to be the normal case. If, however, the use of OpenPGP certificates is negotiated, then the certificates must be OpenPGP certificates. In this case, there is a subtle remark to make with regard to the CERTIFICATEREQUEST message that may be sent from the server to the client: Such a message usually specifies the certificate types and list of CAs that are accepted by the server. In the case of OpenPGP certificates, however, the list of CAs must be empty (because OpenPGP certificates are usually issued by peer entities instead of CAs).

C.2.10 `Supported_groups` (10) and `ec_point_formats` (11) Extensions

Because ECC has many advantages (especially in realm of SSL/TLS), RFC 4492 [37] originally introduced five ECC-based key exchange algorithms and two TLS extensions (i.e., `elliptic_curves` and `ec_point_formats`) to negotiate the use of ECC. If a client wants to signal support for ECC to the server, then it has to include one (or both) extension(s) in its CLIENTHELLO message.

- Using the `elliptic_curves` extension, the client can inform the server what elliptic curves it is capable and willing to support. A list of possible curves as originally specified by the Standards for Efficient Cryptography

Group (SECG[4]) is provided in [37]. Many of these curves are also recommended by other standardization bodies, such as ANSI and NIST.[5] Among these curves, secp224r1, secp256r1, and secp384r1 are the most widely used ones. In addition, the use of Curve25519 and Curve448 is specified in [38], and the use of the Brainpool curves in [39] (and [40] for TLS 1.3, respectively).

- Using the `ec_point_formats` extension, the client can inform the server that it wants to make use of compression for some curve parameters. This is primarily a theoretical option that is hardly ever used in practice.

In either case, it is up to the server to select an elliptic curve and to decide whether it wants to use compression (if supported by the client), and to return the selection in an extended SERVERHELLO message to the client. The client and server can then start making use of ECC.

More recently, RFC 8422 [7] was published to deprecate [37]. According to this document, the `elliptic_curves` extension is replaced with a `supported_groups` extension. Also, a complementary RFC 7919 [8] was published that elaborates on how to negotiate finite field Diffie-Hellman ephemeral parameters for TLS. Finally, note that TLS 1.3 mandates the use of ECC and ECDHE, and hence ECC is therefore always invoked.

C.2.11 `srp` (12) Extension

Due to its vulnerability to MITM attacks, a Diffie-Hellman key exchange must always be authenticated in one way or another. In general, there are many possibilities to do so, but the use of passwords is particularly simple and straightforward. So there are many proposals for a password-authenticated Diffie-Hellman key exchange, but many of these proposals are susceptible to dictionary attacks (where an adversary can try out all possible password candidates, until he or she finds the correct one). Against this background, Steven M. Bellovin and Michael Merritt introduced the notion of an *encrypted key exchange* (EKE) in the early 1990s to defeat dictionary attacks [41, 42]. In the most general form of EKE, at least one party encrypts an ephemeral (one-time) public key using a password, and sends it to a second party, who decrypts it and uses it to negotiate a shared key with the first party. The password is not susceptible to dictionary attacks, because it is used to encrypt a randomly

4 http://www.secg.org.

5 As of this writing, the elliptic curves recommended by the NIST are considered suspicious, because there is no explanation of how the parameters were selected, and hence people are afraid of backdoors. This is a major concern, because it was publicly revealed that such a backdoor had been incorporated into a pseudorandom bit generator standardized by the NIST (i.e., Dual_EC_DRBG).

looking value, and hence an adversary is not able to decide whether a password candidate is correct.

The notion of an EKE was later refined by many researchers, and one of the resulting proposals is known as the *Secure Remote Password* (SRP) protocol [43, 44].[6] The SRP protocol has many use cases, also in the realm of TLS client authentication [9]—as an alternative to public key certificates, preshared keys, and Kerberos. The SRP protocol is a variant of the Diffie-Hellman key exchange, so it basically represents a key exchange algorithm that can be combined with any cipher and hash function to form a cipher suite. All cipher suites with a name that begins with `TLS_SRP_` employ the SRP protocol.

If a client wants to use the SRP protocol for key exchange, then it sends an extended CLIENTHELLO message with the `srp` extension to the server. If the server also supports it, then it returns the extension in its SERVERHELLO message. The client and server then exchange SRP-specific SERVERKEYEXCHANGE and CLIENTKEYEXCHANGE messages, and in the end, they are able to compute a premaster secret (while the SRP protocol ensures that an eavesdropper cannot mount a dictionary attack against the password). In spite of its security and efficiency advantages, the SRP has seen surprisingly little use in the field.

C.2.12 `Signature_algorithms` (13) Extension

There are multiple signature algorithms that may be supported by different clients. The IANA registries itemize the currently available hash[7] and signature algorithms[8] with their respective code values. Note, however, that not all signature algorithms can be paired with all hash algorithms. DSA, for example, only works with SHA-1. Anyway, the client can use the extension type `signature_algorithms` with the respective code values to tell the server which hash and signature algorithms it supports. If the extension is not present, then the server assumes support for SHA-1 and infers the supported signature algorithms from the client's offered cipher suites.

C.2.13 `Use_srtp` (14) Extension

In the realm of real-time communications, SRTP [45] and SRTCP are omnipresent. SRTP does not provide key management functionality, but instead depends on external key management to exchange secret keys, and to negotiate the algorithms and parameters for use with those keys. Against this background, [10] provides a mechanism to use DTLS for the management of SRTP keys—sometimes referred to

6 http://srp.stanford.edu.

7 https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-18.

8 https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-16.

as DTLS-SRTP. The key points of DTLS-SRTP are that application data is protected using SRTP, and that a DTLS handshake is used to establish keying material, algorithms, and parameters for SRTP. The DTLS handshake, in turn, employs the (D)TLS extension `use_srtp`.

C.2.14 `Heartbeat` (15) Extension

The `heartbeat` extension goes back to a 2012 PhD thesis[9] that proposes a mechanism (called Heartbeat) to discover the PMTU[10] and to keep alive a secure connection. The mechanism originally targeted DTLS, but it is also applicable to TLS (note that TLS does not provide a feature to keep a connection alive without continuous data transfer). In the same year, RFC 6520 [11] was published and the Heartbeat mechanism was submitted to the internet standards track.

The client and server can signal support for the Heartbeat mechanism with the `heartbeat` extension. If both support it, then they can subsequently send heartbeat request and response messages to each other. The IANA has assigned the content type value 24 for such messages (this complements the content type values introduced in Section 2.2.1). This means that heartbeat can be seen as yet another TLS subprotocol, in addition to change cipher spec (20), alert (21), handshake (22), and application data (23).

Heartbeat is an extension supported by many implementations, including OpenSSL,[11] where it is enabled by default. Hardly anybody knew about its availability until April 2014, when it was discovered that the implementation suffered from a serious flaw that allowed the extraction of sensitive data from the server's process memory. More specifically, the client can send a Heartbeat payload (together with an indication about the payload length) to the server, and the server is tasked to send back the same payload. If the client lied about the payload length (i.e., specifying a length that exceeds the actual payload[12]), then the implementation flaw would have the server read out and send back to the client some contents of the memory cells that had not been properly assigned to the payload. Depending on the data that had been stored in these memory cells, the server could leak some sensitive data, such as cryptographic keys. To make things worse, the client could iteratively send multiple requests to the server without leaving any traces in the respective log files. Due to

9 http://duepublico.uni-duisburg-essen.de/servlets/DerivateServlet/Derivate-31696/dissertation.pdf.

10 The MTU refers to the size of the largest data unit that can be sent as a whole between two entities. When two entities communicate directly, then they can exchange their MTUs and agree on the shorter one. If, however, communication goes over multiple hops, then it is sometimes necessary to discover the largest possible MTU by sending progressively larger data units. The result is then called PMTU.

11 http://www.openssl.org.

12 The maximum length is $2^{14} = 16{,}384$ bytes.

the criticality of the information leakage, the resulting attack was figuratively named *Heartbleed*. The announcement of Heartbleed shocked the security community in general, and the open source community in particular. It is simple to mitigate the risks imposed by Heartbleed: One can either patch the software or disable the Heartbeat extension altogether. As a side effect, many developers of open source cryptographic software have launched new projects, such as LibreSSL,[13] that are targeted to provide more secure SSL/TLS implementations. But meanwhile, OpenSSL has also improved its security considerably in the past few years.

C.2.15 `Application_layer_protocol_negotiation` (16) Extension

According to Section 2.1, there are two strategies to stack an application-layer protocol on top of SSL/TLS: the separate port strategy and the upward negotiation strategy. The first strategy requires separate ports for all application-layer protocols, whereas the second strategy requires the application-layer protocol to support an upward negotiation feature. If there are multiple application-layer protocols that need to be supported on the same TCP or UDP port, but these protocols don't support an upward negotiation feature, then it is possible to have the application layer negotiate within the TLS handshake which protocol is going to be used in the end. This is where the `application_layer_protocol_negotiation` (ALPN) extension [12] comes into play. With ALPN, a TLS-enabled server running on port 443 can support HTTP 1.1 by default, but also allow the negotiation of other application-layer protocols, such as SPDY[14] or HTTP/2. A client can send the `application_layer_protocol_negotiation` extension in its CLIENTHELLO message and thereby submit a list of supported application-layer protocols. The server can then decide and use the same extension in its SERVERHELLO message to inform the client about what application-layer protocol it should use.[15] As a side-effect, the ALPN extension plays a major role in mitigating cross-protocol attacks, such as ALPACA (Section 3.8).

C.2.16 `Signed_certificate_timestamp` (18) and `transparency_info` (52) Extensions

As introduced and discussed in Section 6.6.4, CT is a technology—or rather a framework—that was originally developed by Google to implement an allowlist for legitimately issued certificates. All such certificates are routinely written to an

13 http://www.libressl.org.

14 SPDY (pronounced "speedy") was a predecessor protocol of HTTP/2.

15 If the server doesn't support any of the protocols advertised by the client, then it should respond with the newly introduced `no_application_protocol` alert (120) that is fatal.

append-only, tamper-resistent (by the means of cryptography), and publicly auditable data structure known as a log. This log can then be queried to decide whether a particular certificate has been issued in a legitimate way. More specifically, an SCT may serve as proof that a certificate is appended to a log, and hence that the certificate can be considered to be legitimate. The idea is that a certificate is accompanied by one or several SCTs. This can be accomplished in many ways, one of them using TLS extensions to transport the SCTs. In fact, there are two TLS extensions that can be used for this purpose: The `signed_certificate_timestamp` extension [14] in CT version 1 and the `transparency_info` extension [29] in CT version 2. With the success of CT, the use of SCTs and the two TLS extensions is already widespread and will become even more widespread in the future. For obvious reasons, this is particularly for the `transparency_info` extension (that is more recent).

C.2.17 `Client_certificate_type` (19) and `Server_certificate_type` (20) Extensions

There are situations in which public keys are needed in native form, meaning that the large overhead that is typically involved in a public key certificate is not needed (this applies to X.509 and OpenPGP certificates, but it also applies to self-signed certificates). It might be that there are only a few public keys involved, and that these keys never change. It might also be that there are memory constraints that prohibit the storage of data that is not needed. The only field that is required is the public key, and hence all other fields that are usually found in a public key certificate can sometimes be omitted. This means that the resulting raw public key is as small as it can possibly get.

Against this background, Standards Track RFC 7250 [15] specifies how to use raw public keys in (D)TLS, and provides two respective extensions; that is `client_certificate_type` (for the client) and `server_certificate_type` (for the server). Either or both extensions can be used in a TLS handshake when raw public keys for authentication need to be negotiated. Note, however, that the keys must still be associated with entities, otherwise they cannot be used for authentication. There are several possibilities that can be used for this purpose, such as DANE in conjunction with DNSSEC (Section 6.6). As of this writing, raw public keys are seldom used on the internet.

C.2.18 `Padding` (21) Extension

Successive TLS versions have added support for more cipher suites and more TLS extensions. This has caused the size of the respective CLIENTHELLO messages to grow, and the additional size has caused some implementation bugs. At least one of these implementation bugs can be ameliorated by increasing the length of the CLIENTHELLO message (this is desirable given that fully comprehensive patching of affected implementations is difficult to achieve). Against this background, Standards Track RFC 7685 [16] defines a `padding` extension that can be used to pad a CLIENTHELLO message to a desired length in order to avoid implementation bugs (that may be caused by some specific lengths).

C.2.19 **`Encrypt_then_mac`** (22) Extension

As mentioned several times throughout the book, the appropriate way of combining authentication and encryption is EtA and not AtE (as used in SSL/TLS until TLS 1.2). However, changing the order of the authentication and encryption operations in records processing is not trivial and requires changes in the respective implementations. So it will happen that some implementations support the changed order, whereas others don't, and hence there is room for a mechanism to negotiate the order. This can be done with the TLS extension `encrypt_then_mac` specified in Standards Track RFC 6377 [17]: The client and server can negotiate the use of EtA (instead of AtE) by exchanging hello messages that include this extension. As long as a block cipher in CBC mode is used, this extension is important. It will become obsolete once AEAD ciphers become omnipresent, such as in the case of TLS 1.3.

C.2.20 **`Token_binding`** (24) Extension

The basic idea and notion of token binding is introduced in Section 3.1.2. In short, it aims at providing protection against MITM attacks by cryptographically binding (authentication) tokens, like HTTP cookies, to TLS connections and respective long-lived public key pairs (so that they cannot be submitted on other TLS connections than the ones they have been issued for). The use of token binding can be negotiated in TLS via the `token_binding` extension specified in RFC 8472 [19]. In addition to this RFC document, RFC 8471 [46] specifies a simple token binding protocol (version 1.0), and RFC 8473 [47] specifies its major use case, namely token binding over HTTP. All three documents deal with TLS up to version 1.2 (whereas token binding in TLS 1.3 is still work in progress).

More specifically, a client and a server can negotiate the use of token binding (together with some parameters) with `token_binding` extensions exchanged in CLIENTHELLO and SERVERHELLO messages. If the client and server both support token binding and agree on common parameters, then the client can establish a token binding by generating a unique and long-lived public key pair for that server,[16] providing the public key to the server (as part of a token binding ID), and proving possession of the corresponding private key on every TLS connection. In the case of HTTP, for example, this proof involves signing the EKM of the TLS connection and submitting the signature (together with the token binding ID) to the server in every HTTP request message, using the `Sec-Token-Binding` request header. Whenever the server issues a token for the client, it includes the client's token binding ID (or a cryptographic hash value thereof) in the token. This token is presented by the client to the server at some later point in time. The server can then verify that the token binding ID matches the one established with the client in the first place. If it does, then the server can also verify the signature for the EKM and thus ensure that the token is in fact bound to and submitted on a genuine TLS connection.

Note that TLS 1.2 and older TLS versions are susceptible to the 3SHAKE attack (Section A.5) that allows an adversary to synchronize keying material between TLS connections, and hence may enable the adversary to replay bound tokens, the `token_binding` extension must not be negotiated with these TLS versions, unless the `renegotiation_info` and `extended_master_secret` extensions (Section C.2.43) have also been negotiated.

C.2.21 `Cached_info` (25) Extension

As part of a (D)TLS handshake, the client and server often exchange fairly static information, such as the server certificate and a list of trusted CAs. This information can be of considerable size, particularly if the server certificate is bundled with a complete certificate chain. Against this background, Standards Track RFC 7924 [20] defines the `cached_info` extension that allows a client to inform a server about cached information, thereby enabling the server to reuse already available information. Needless to say that this helps improve the efficiency of a (D)TLS handshake considerably.

C.2.22 `Compress_certificate` (27) Extension

In a (D)TLS handshake, certificate chains often take up the majority of the bytes that need to be transmitted. This is a performance bottleneck, and hence Standards Track RFC 8879 [21] introduces the TLS `compress_certificate` extension and explains how it can be used to compress the length of certificate chains. Note

16 Ideally, the key pair is generated within a secure environment, such as a hardware security module (HSM), a trusted platform module (TPM), or a TEE. The fact that it is long-lived means that it may encompass multiple TLS sessions and connections between a given client and server.

that certificate compression does not enable compression-related attacks (Section A.6), so it can be used at will.

C.2.23 Pwd_protect (29), pwd_clear (30), and password_salt (31) Extensions

TLS uses certificate-based authentication by default (at least for the server side). There are, however, many settings in which the use of passwords for authentication is advantageous. To empower these settings and provide guidance, informational RFC 8492 [22] defines several new cipher suites for the TLS protocol that support certificateless, secure authentication using only a simple, low-entropy password. The result is a password-authenticated key exchange (PAKE) that is similar to SRP. In the realm of TLS, the PAKE of choice is a protocol named *Dragon* [48] (that is also used in WPA3). The use of Dragon in TLS-PWD is discussed in [22], together with the TLS extensions `pwd_protect`, `pwd_clear`, and `password_salt` that refer to the secure usage of passwords.

C.2.24 Ticket_pinning (32) Extension

As an alternative to public key pinning (Section 6.6), it may be useful to work with tickets (instead of public keys). RFC 8672 [23] therefore proposes a TLS `ticket_pinning` extension that supports opaque tickets as a way to pin the server's identity. In contrast to many RFC documents discussed in this appendix, this RFC is experimental, meaning that its use and deployment is open.

C.2.25 Tls_cert_with_extern_psk (33) Extension

In TLS 1.3, a server can authenticate itself with a certificate or a PSK that can either be established by a previous handshake (resumption PSK) or by any other means (external PSK). Against this background, RFC 8773 [24] defines a TLS 1.3 `tls_cert_with_extern_psk` extension that allows certificate-based server authentication to be combined with an external PSK. Again, this RFC is just experimental and not (yet) submitted to the internet standards track.

C.2.26 Session_ticket (35) and ticket_request (58) Extension

As introduced and discussed in Section 2.2.2, the SSL handshake protocol can be simplified considerably to resume a session (Figure 2.6). To invoke this 1-RTT mechanism, the server must keep per-client session state. If there is a huge quantity of simultaneous clients, then the server may easily run out of state. So there is incentive to have a 1-RTT mechanism that does not require per-client session state on the server side. Instead, the session state information may be sent to the client as a *session ticket* that can then be returned to the server to resume the session at some later point in time. This idea is similar to HTTP cookies and Standards Track RFC 5077 [25] introduces the TLS `session_ticket` extension that can be used for this purpose.

If a client supports session tickets, then it includes the `session_ticket` extension in the CLIENTHELLO message sent to the server. If it does not already possess a session ticket, then the data field of the extension is empty. (Otherwise, it may include the ticket in an abbreviated handshake.) If the server does not support session tickets, then it simply ignores the extension and continues as if the CLIENTHELLO message did not comprise a `session_ticket` extension. However, if the server also supports session tickets, then it sends back a SERVERHELLO message with an empty `session_ticket` extension. Since the session state is not yet determined, the server cannot include the actual session ticket in the data field of the extension. Instead, it must postpone the delivery of the session ticket to some later point in time in the TLS protocol execution. As illustrated in Figure C.2, there is a new handshake message called NEWSESSIONTICKET (type 4) that serves this purpose. It is sent toward the end of the handshake, immediately before the server sends its CHANGECIPHERSPEC and FINISHED messages to the client.

A session ticket comprises the encrypted session state, including, for example, the cipher suite and the master secret in use, as well as a ticket lifetime that informs the server how long the ticket should be stored (in seconds). A value of zero indicates that the lifetime of the ticket is not defined. Because the session ticket is encrypted by the server (with a server-selected key or pair of keys), it is opaque to the client, and there are no interoperability issues to consider.

After a client has received a NEWSESSIONTICKET message, it caches the session ticket along with the master secret and the other parameters of the respective session. When it thereafter wishes to resume the session, it includes the ticket in the `session_ticket` extension of the CLIENTHELLO message (so the extension is no longer empty). The server decrypts and authenticates the ticket (using the proper keys), retrieves the session state, and uses this state to resume the session. If the server wants to renew the ticket, then it can initiate an abbreviated handshake with an empty `session_ticket` extension in the SERVERHELLO message and a NEWSESSIONTICKET message that is sent immediately after the SERVERHELLO message. The respective message flow is illustrated in Figure C.3. If the server does not want to renew the ticket, then it can be used right away (and neither the `session_ticket` extension in the SERVERHELLO message nor a NEWSESSIONTICKET message is needed).

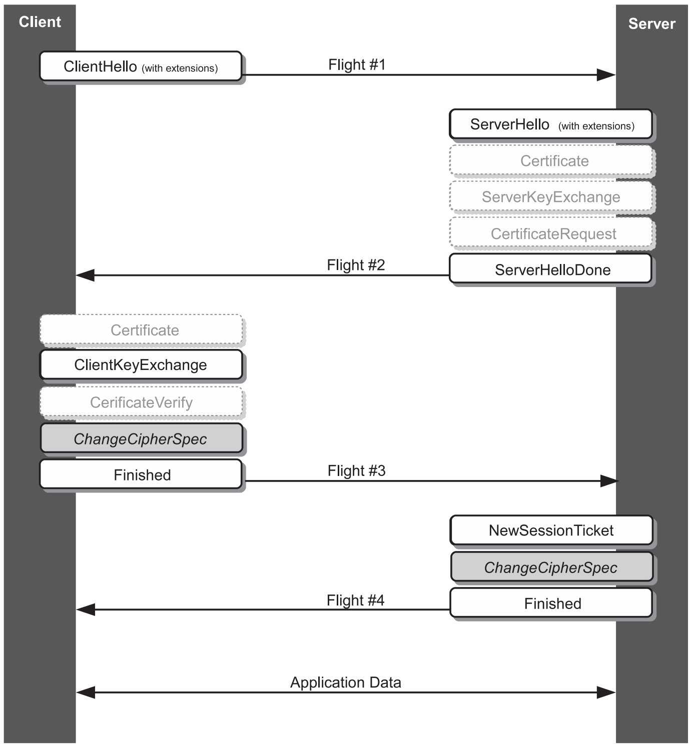

Figure C.2 The message flow of the TLS handshake protocol issuing a new session ticket.

For TLS 1.3 and DTLS 1.3, a new `ticket_request` extension is specified in Standards Track RFC 9149 [33]. Using this extension, clients can express their desired number of session tickets, and servers can use it as a hint for the number of NEWSESSIONTICKET messages to be sent.

Last but not least, it is important to note that the use of session tickets (as discussed so far) in some senses breaks forward secrecy provided by DHE or ECDHE. If an adversary is somehow able to retrieve the key that is used by the server to encrypt the session tickets, then he or she can also use this key to decrypt the session state that, among other things, includes the key that can be used to decrypt the data that is transmitted. So there is a choice to make: If one wants to have forward secrecy, then one either has to stay away from session tickets entirely

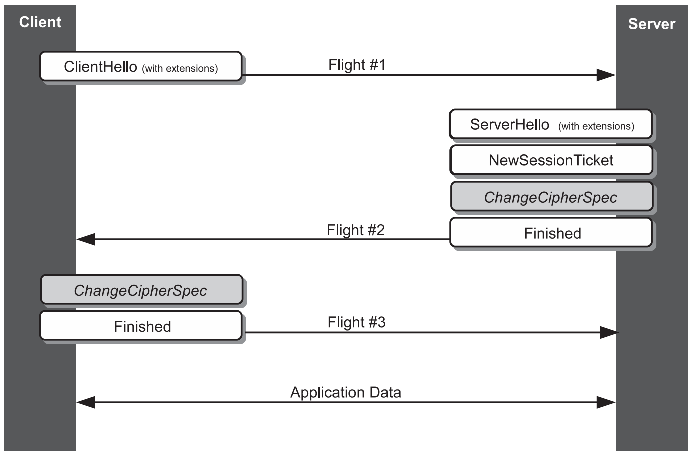

Figure C.3 The message flow for an abbreviated TLS handshake protocol using a new session ticket.

or use a more sophisticated approach (which still has to be researched and has not been standardized yet). In either case, it is a good idea to regularly change the ticket encryption key (e.g., once per week) and to destroy the encryption keys at the end of their validity period.

C.2.27 `TLMSP` (36), `TLMSP_proxying` (37), and `TLMSP_delegate` (38) Extensions

As introduced and discussed in Sections 3.5 and 5.4, the ETSI has specified a TLS variant called MSP or TLMSP. In spite of the fact that the use of (TL)MSP in the field is controversially discussed in the community, there are still three TLS extensions that can be used here: `TLMSP`, `TLMSP_proxying`, and `TLMSP_delegate`. There are no RFC documents that explain the use of the extensions. Instead, the IETF refers to the respective ETSI documents (referenced in the sections mentioned above).

C.2.28 `Supported_ekt_ciphers` (39) Extension

SRTP is frequently used in decentralized conferences, where a common key must be distributed to all conference endpoints. This is where encrypted key transport (EKT) comes into play. Standards Track RFC 8870 [28] defines the DTLS `supported_ekt_ciphers` extension that can be used to securely transport SRTP master keys, rollover counters, and other information within SRTP.

C.2.29 `Pre_shared_key` (41) Extension

As mentioned in Section 3.5, TLS 1.3 supports PSKs (the same is true for DTLS 1.3). In particular, a TLS 1.3 client can use the TLS `pre_shared_key` extension to specify a list of PSKs and respective labels that are known to it, and hence express its wish to use a certain PSK to establish a new connection. It is then up to the server to select and accept the PSK.

C.2.30 `Early_data` (42) Extension

If the `pre_shared_key` extension and a respective PSK are used in (D)TLS 1.3, then a client can also use the `early_data` extension to announce the transmission of some early application data in the first flight (i.e., before the server sends back a SERVERHELLO message). In this case, the data is protected with a key derived from the PSK (and not in a way that provides forward secrecy). Hence, the use of early data and the respective `early_data` extension should always be taken with a grain of salt.

C.2.31 `Supported_versions` (43) Extension

As explained in Section 3.5.1, TLS 1.3 uses a new version negotiation mechanism that is distinct from its predecessors. Instead of having the client send its highest supported version number in the `version` field of the CLIENTHELLO message and having the server decide what version to use, the CLIENTHELLO message now includes a `legacy_version` field (always set to 0x0303 referring to TLS 1.2) and the actual version negotiation takes place via the `supported_versions` extension. More specifically, the CLIENTHELLO message comprises such an extension to inform the server about the client-supported TLS versions in preference order, with the most preferred first. Again, it is up to the server to decide what version to use, and this decision is communicated to the client via the `supported_versions` extension in the respective SERVERHELLO message. The default (and fall back) value here is TLS 1.2.

As mentioned in Section 4.4.1, DTLS 1.3 uses a similar version negotiation mechanism, in which negotiation takes place through the `supported_versions` extension and the version field in the DTLS 1.3 record header serves no real purpose. It is therefore filled with the `legacy_record_version` value 0xFEFD or 254,253 (standing for DTLS 1.2).

C.2.32 `Cookie` (44) Extension

The use of cookies to protect a server against DDoS attacks (from randomly chosen IP addresses) was fully addressed in Section 4.2.2.4. To signal support for cookies and to actually transmit cookies, the TLS `cookie` extension is used.

C.2.33 `Psk_key_exchange_modes` (45) Extension

In TLS 1.3, this extension complements the `pre_shared_key` extension sent by the client. It indicates whether the PSK is used alone (`psk_ke`) or in conjunction with (EC)DHE (`psk_dhe_ke`). This choice has to be driven by the need to provide forward secrecy. If forward secrecy is needed, then the `pre_shared_key` extension must be set to `psk_dhe_ke`; otherwise, `psk_ke` is perfectly fine.

C.2.34 `Certificate_authorities` (47) Extension

In TLS 1.3, the `certificate_authorities` extension can be used to indicate the CAs (by their DNs) supported by an entity. The client may send the extension in a CLIENTHELLO message, whereas the server may send it in a CERTIFICATEREQUEST message. The `trusted_ca_keys` extension that serves a similar purpose in previous versions of the SSL/TLS protocols, is no longer used in TLS 1.3.

C.2.35 `Oid_filters` (48) Extension

This extension allows a server to specify a set of OIDs and allowed values for client certificates. Examples include OIDs for the key usage and extended key usage extensions of X.509 certificates. The `oid_filters` extension thus works as a filter; it is only provided by the server in a CERTIFICATEREQUEST message.

C.2.36 `Post_handshake_auth` (49) Extension

In TLS 1.3, a client can use the `post_handshake_auth` extension to indicate to the server that it is willing to perform an authentication after the handshake has taken place. This is partly needed, because renegotiation is no longer supported in TLS 1.3. Anyway, note that `post_handshake_auth` is a client-side extension that must not be sent by servers.

C.2.37 `Signature_algorithms_cert` (50) Extension

This optional extension is similar to the `signature_algorithms` extension, but it only applies to signatures in certificates. If it is missing, that is no `signature_algorithms_cert` extension is being used and put in place, then the `signature_algorithms` extension also applies to signatures in certificates.

C.2.38 `Key_share` (51) Extension

In TLS 1.3, the `key_share` extension is very important, as it can be used to send an (EC)DHE key share for a supported group. If more than one group is supported, then the extension can comprise multiple key shares (one of each of these groups). It goes without saying that this is certainly the most important extension that allows to opportunistically start the ephemeral Diffie-Hellman key exchange after the first flight.

C.2.39 `Connection_id` (54) Extension

Standards Track RFC 9146 [30] specifies the CID construct originally proposed for DTLS 1.2.[17] To negotiate the use of CIDs, the `connection_id` extension is used as further explained in Section 4.3.

C.2.40 `External_id_hash` (55) and `external_session_id` (56) Extensions

The `external_id_hash` and `external_session_id` extensions can be used to mitigate some unknown key-share attacks against DTLS-SRTP and related settings. The attacks and the use of the extensions to mitigate them are further addressed in Standards Track RFC 8844 [31].

C.2.41 `Quic_transport_parameters` (57) Extension

As mentioned in Section 1.3 and further explained in Standards Track RFC 9001 [32], TLS can be used to secure the Quick UDP Internet Connections (QUIC)

17 Note that this is the only extension discussed here that is DTLS-only, meaning that it can only be used in DTLS (and not in TLS).

protocol. The respective parameters can be negotiated via the `quic_transport_parameters` extension. Because the QUIC protocol is not the focus of this book, this extension is not further addressed here.

C.2.42 `Dnssec_chain` (59) Extension

In situations where the retrieval of DNS records is unavailable to the client or may incur undesirable additional latency, the use of the TLS `dnssec_chain` extension as described in RFC 9102 [34] may be used. It provides support for the inband transport of a complete set of RRs required for DNSSEC, and thus enables a client to perform DANE authentication of a TLS server without the need to perform out-of-band DNS lookups. Note that [34] is experimental and not submitted to the internet standards track.

C.2.43 `Renegotiation_info` (65281) and `extended_master_secret` (23) Extensions

Section A.5 explained how the TLS renegotiation mechanism can be exploited in a sophisticated MITM attack known as the renegotiation attack. The attack can be mitigated either by disabling TLS renegotiation or by using a TLS extension that cryptographically binds the handshake of the renegotiated session to the initial one. A respective TLS extension named `renegotiation_info` is specified in RFC 5746 [35] that is submitted to the internet standards track. The cryptographic binding is achieved by having the `renegotiation_info` extension of the CLIENTHELLO message of the renegotiated handshake carry the `verify_data` field of the FINISHED message of the initial handshake.[18]

Unfortunately, the `renegotiation_info` extension does not fully mitigate the renegotiation attack, and there is an even more sophisticated attack known as the triple handshake or 3SHAKE attack that can still be mounted, even if the `renegotiation_info` extension is put in place. The 3SHAKE attack cleverly exploits two weaknesses that allow an MITM to establish two connections that share the same value for the respective `verify_data` field. This, in turn, means that the `renegotiation_info` extension doesn't protect against all types of renegotiation attacks.

To mitigate the triple handshake attack, people have come up with another TLS extension named `extended_master_secret` that is specified in Standards Track RFC 7627 [18]. The extension ensures that every TLS connection has a

18 In SSL 3.0, this field is 36 bytes long (Section 2.2.2.11), whereas it is only 12 bytes long in all currently deployed versions of the TLS protocol (Section 3.2.4).

distinct and—as it is hoped—unique master key, and hence that an unknown key-share attack (Section A.5) can no longer be mounted. This, in turn, mitigates the 3SHAKE attack, and it is hoped that it protects against all types of renegotiation attacks.

Both the `renegotiation_info` and `extended_master_secret` extensions are mandatory to use in TLS 1.2, but they are no longer required in TLS 1.3 (because renegotiation is partially replaced by post-handshake authentication and key update mechanisms).

References

[1] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions," RFC 6066, January 2011.

[2] Sullivan, N., "Exported Authenticators in TLS," RFC 9261, July 2022.

[3] Thomson, M., "Record Size Limit Extension for TLS," RFC 8449, August 2018.

[4] Santesson, S., A. Medvinsky, and J. Ball, "TLS User Mapping Extension," RFC 4681, October 2006.

[5] Brown, M., and R. Housley, "Transport Layer Security (TLS) Authorization Extensions," RFC 5878, May 2010.

[6] Mavrogiannopoulos, N., and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication," RFC 6091, February 2011.

[7] Nir, Y., S. Josefsson, and M. Pegourie-Gonnard, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier," RFC 8422, August 2018.

[8] Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)," RFC 7919, August 2016.

[9] Taylor, D., et al., "Using the Secure Remote Password (SRP) Protocol for TLS Authentication," RFC 5054, November 2007.

[10] McGrew, D., and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)," RFC 5764, May 2010.

[11] Seggelmann, R., M. Tuexen, and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension," RFC 6520, February 2012.

[12] Friedl, S., et al., "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension," RFC 7301, July 2014.

[13] Pettersen, Y., "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension," RFC 6961, June 2013.

[14] Laurie, B., A. Langley, and E. Kasper, "Certificate Transparency," RFC 6962, June 2013.

[15] Wouters, P. (ed.), et al., "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 7250, June 2014.

[16] Langley, A., “A Transport Layer Security (TLS) ClientHello Padding Extension,” RFC 7685, October 2015.

[17] Gutmann, P., “Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS),” RFC 7366, September 2014.

[18] Bhargavan, K. (ed.), et al., “Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension,” RFC 7627, September 2015.

[19] Popov, A. (Ed.), M. Nystroem, and D. Balfanz, “Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation,” RFC 8472, October 2018.

[20] Santesson, S., and H. Tschofenig, “Transport Layer Security (TLS) Cached Information Extension,” RFC 7924, July 2016.

[21] Ghedini, A., and V. Vasiliev, “TLS Certificate Compression,” RFC 8879, December 2020.

[22] Harkins, D. (Ed.), “Secure Password Ciphersuites for Transport Layer Security (TLS),” RFC 8492, February 2019.

[23] Sheffer, Y., and D. Migault, “TLS Server Identity Pinning with Tickets,” RFC 8672, October 2019.

[24] Housley, R., “TLS 1.3 Extension for Certificate-Based Authentication with an External Pre-Shared Key,” RFC 8773, March 2020.

[25] Salowey, Y., et al., “Transport Layer Security (TLS) Session Resumption without Server-Side State,” RFC 5077, January 2008.

[26] Salowey, J., and S. Turner, “IANA Registry Updates for TLS and DTLS,” RFC 8447, August 2018.

[27] ETSI TS 103 523-2, “Middlebox Security Protocol, Part 2: Transport layer MSP, Profile for Fine Grained Access Control,” 2021.

[28] Jennings, C., et al., “Encrypted Key Transport for DTLS and Secure RTP,” RFC 8870, January 2021.

[29] Laurie, B., E. Messeri, and R. Stradling, “Certificate Transparency Version 2.0,” RFC 9162, December 2021.

[30] Rescorla, E. (Ed.), H. Tschofenig (Ed.), and A. Kraus, “Connection Identifier for DTLS 1.2,” RFC 9146, March 2022.

[31] Thomson, M., and E. Rescorla, “Unknown Key-Share Attacks on Uses of TLS with the Session Description Protocol (SDP),” RFC 8844, January 2021.

[32] Thomson, M. (Ed.), and S. Turner (Ed.), “Using TLS to Secure QUIC,” RFC 9001, May 2021.

[33] Pauly, T., D. Schinazi, and C.A. Wood, “TLS Ticket Requests,” RFC 9149, April 2022.

[34] Dukhovni, V., at al., “TLS DNSSEC Chain Extension,” RFC 9102, August 2021.

[35] Rescorla, E., S. Dispensa, and N. Oskov, “Transport Layer Security (TLS) Renegotiation Indication Extension,” RFC 5746, February 2010.

[36] Santesson, S., "TLS Handshake Message for Supplemental Data," RFC 4680, September 2006.

[37] Blake-Wilson, S., et al., "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)," RFC 4492, May 2006.

[38] Langley, A., M. Hamburg, and S. Turner, "Elliptic Curves for Security," RFC 7748, January 2016.

[39] Merkle, J., and M. Lochter, "Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS)," RFC 7027, October 2013.

[40] Bruckert, L., J. Merkle, and M. Lochter, "Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS) Version 1.3," RFC 8734, February 2020.

[41] Bellovin, S.M., and M. Merritt, "Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks," *Proceedings of the IEEE Symposium on Security and Privacy,* IEEE Computer Society, 1992, p. 72.

[42] Bellovin, S.M., and M. Merritt, "Augmented Encrypted Key Exchange: A Password-based Protocol Secure Against Dictionary Attacks and Password File Compromise," *Proceedings of 1st ACM Conference on Computer and Communications Security,* Fairfax, VA, November 1993, pp. 244–250.

[43] Wu, T., "The Secure Remote Password Protocol," *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium,* San Diego, CA, March 1998, pp. 97–111.

[44] Wu, T., "The SRP Authentication and Key Exchange System," RFC 2945, September 2000.

[45] Baugher, M., et al., "The Secure Real-time Transport Protocol (SRTP)," RFC 3711, March 2004.

[46] Popov, A. (Ed.), et al., "The Token Binding Protocol Version 1.0," RFC 8471, October 2018.

[47] Popov, A., et al., "Token Binding over HTTP," RFC 8473, October 2018.

[48] Harkins, D. (Ed.), et al., "Dragonfly Key Exchange," RFC 7664, November 2015.

Abbreviations and Acronyms

AA	attribute authority
AAI	authentication and authorization infrastructure
ABAC	attribute-based access control
AC	attribute certificate
ACM	Association for Computing Machinery
ACME	automated certificate management environment
AE	authenticated encryption
AEAD	authenticated encryption with associated data
AES	Advanced Encryption Standard
ALPACA	application layer protocols allowing cross-protocol attacks
ALPN	application-layer protocol negotiation
ALTS	application layer transport security
ANSI	American National Standards Institute
AOL	America Online
API	application programming interface
APT	advanced persistent threat
ASCII	American Standard Code for Information Interchange
ASN.1	abstract syntax notation one
AtE	authenticate-then-encrypt
BCP	best current practice
BEAST	browser exploit against SSL/TLS
BER	basic encoding rules
BREACH	browser reconnaissance and exfiltration via adaptive compression of hypertext
BSI	German Federal Office for Information Security
CA	certification authority
CAA	certification authority authorization
CAT	cache-like attack

CBC	cipherblock chaining
CCA	chosen ciphertext attack
CCA2	adaptive CCA
CCADB	common CA database
CCM	counter with CBC-MAC mode
CDN	content delivery network
CERT	computer emergency response team
CERT/CC	CERT coordination center
CFB	cipher feedback
CFRG	Crypto Forum Research Group
CID	connection ID
CISA	Cybersecurity & Infrastructure Security Agency
CNSA	commercial national security algorithm
CoAP	constrained application protocol
COMPUSEC	computer security
COMSEC	communication security
CP	certificate policy
CPA	chosen plaintext attack
CPS	certificate practice statement
CRC	cyclic redundancy check
CRIME	compression ratio info-leak made easy
CRL	certificate revocation list
CSM	client-side middlebox
CSP	certification service provider
CSR	certificate signing request
CSRF	cross-site request forgery
CT	certificate transparency
cTLS	compact TLS
CTR	counter mode encryption
CVE	common vulnerabilities and exposures
CVSS	common vulnerability scoring system
DAC	discretionary access control
DANE	DNS-based authentication of named entities
DCCP	datagram congestion control protocol
DER	distinguished encoding rules
DES	data encryption standard
DH_anon	anonymous Diffie-Hellman key exchange
DH	Diffie-Hellman key exchange
DHE	ephemeral Diffie-Hellman key exchange

DIT	directory information tree
DLP	data loss prevention
DN	distinguished name
DNS	domain name system
DNSSEC	DNS security
DoC	Department of Commerce
DoD	Department of Defense
DoH	DNS over HTTPS
DoS	denial of service
DROWN	decrypting RSA with obsolete and weakened encryption
DSA	digital signature algorithm
DSS	digital signature standard
DTLS	datagram TLS
DV	domain validation
E&A	encrypt-and-authenticate
E2EE	end-to-end encryption
ECB	electronic code book
ECC	elliptic curve cryptography
ECDH	elliptic curve Diffie-Hellman
ECDHE	elliptic curve ephemeral Diffie-Hellman
ECDSA	elliptic curve digital signature algorithm
ECH	encrypted CLIENTHELLO
EFF	Electronic Frontier Foundation
EFGH	end-to-end fine grained HTTP
EKE	encrypted key exchange
EKM	exported keying material
EKT	encrypted key transport
ESNI	encrypted SNI
EtA	encrypt-then-authenticate
eTLS	enterprise TLS
ETS	Enterprise Transport Security
ETSI	European Telecommunications Standards Institute
EV	extended validation
FIPS	Federal Information Processing Standard
FQDN	fully qualified domain name
FREAK	factoring attack on RSA export keys
FSUIT	Federal Strategy Unit for Information Technology
FTP	file transfer protocol

FYI	for your information
GCM	Galois/Counter Mode
GMT	Greenwich Mean Time
GNU	GNU’s not Unix
GPL	general public license
GREASE	generate random extensions and sustain extensibility
GUI	graphical user interface
HA	high assurance
HKDF	HMAC-based KDF
HMAC	hashed MAC
HPKP	HTTP public key pinning
HSM	hardware security module
HSTS	HTTP strict transport security
HTTP	Hypertext Transfer Protocol
IACR	International Association for Cryptologic Research
IAM	identity and access management
IANA	Internet Assigned Numbers Authority
ICSI	International Computer Science Institute
ID	identity (identifier)
IDEA	international data encryption algorithm
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IESG	Internet Engineering Steering Group
IFIP	International Federation for Information Processing
IIOP	Internet InterORB protocol
IKE	internet key exchange
IKM	input keying material
IM	identity management
IMAP	internet message access protocol
INFOSEC	information security
IoT	Internet of Things
IP	Internet Protocol
IPsec	IP security
IRC	internet relay chat
IRTF	Internet Research Task Force
ISO	International Organization for Standardization

ISOC	Internet Society
ISRG	Internet Security Research Group
IT	information technology
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
IV	initialization vector
JSSE	Java secure socket extensions
JTC1	Joint Technical Committee 1
KDC	key distribution center
KDF	key derivation function
KEA	key exchange algorithm
LAN	local area network
LDAP	Lightweight Directory Access Protocol
LRA	local registration authority (or agent)
LSB	least significant bit
LURK	limited usage of remote key
MAC	message authentication code
MALCOLM	measuring active listeners, connection observers, and legitimate monitors
maTLS	middlebox-aware TLS
mbTLS	middlebox TLS
mcTLS	multi-context TLS
MIME	multipurpose internet mail extensions
MITM	man-in-the-middle
MMD	maximum merge delay
MPL	Mozilla Public License
MSB	most significant bit
MSP	middlebox security protocol
mTLS	mutual TLS
MTU	maximum transmission unit
NAT	network address translation
NCSA	National Center for Supercomputing Applications
NIST	National Institute of Standards and Technology
NNTP	Network News Transfer Protocol
NPN	next protocol negotiation

NSA	National Security Agency
NSS	network security services
OAEP	optimal asymmetric encryption padding
OBC	origin-bound certificate
OCSP	online certificate status protocol
OFB	output feedback
OID	object identifier
OIDC	OpenID Connect
OKM	output keying material
OMA	Open Mobile Alliance
OpenPGP	open pretty good privacy
OSI	open systems interconnection
OTP	one-time password
OV	organization validation
OWASP	open web application security project
PAKE	password-authenticated key exchange
PCAP	packet capture
PCT	private communication technology
PGP	pretty good privacy
PITM	person-in-the-middle
PKCS	public key cryptography standard
PKI	public key infrastructure
PKIX	Public-Key Infrastructure X.509
PL	padding length
PMTU	path MTU
POODLE	padding oracle downgraded legacy encryption
POP3	post office protocol
PRBG	pseudorandom bit generator
PRF	pseudorandom function
PSK	preshared key
PSRG	Privacy and Security Research Group
PSS	probabilistic signature scheme
PUB	publication
QUIC	Quick UDP Internet Connections
RA	registration authority
RB	random byte

RBAC	role-based access control
RC2	Rivest Cipher 2
RC4	Rivest Cipher 4
RFC	request for comments
ROBOT	return of Bleichenbacher's oracle threat
RSA	Rivest, Shamir, and Adleman
RSASSA	RSA signature scheme with appendix
RTT	round-trip time
s2n	signal to noise
SAML	security assertion markup language
SAN	subject alternative name
SCADA	supervisory control and data acquisition
SChannel	secure channel
SCSV	signaling cipher suite value
SCT	signed certificate timestamp
SCTP	stream control transmission protocol
SCVP	server-based certificate validation protocol
SDSI	simple distributed security infrastructure
SECG	Standards for Efficient Cryptography Group
SGC	server gated cryptography
SGX	software guard extensions
SHA	secure hash algorithm
SHS	secure hash standard
SHTTP	secure HTTP
S-HTTP	secure HTTP (also known as SHTTP)
SIP	session initiation protocol
SLOTH	security loss due to the use of obsolete and truncated hash constructions
SM	ShangMi (Chinese cipher)
S/MIME	secure MIME
SMTP	simple mail transfer protocol
SNI	server name indication
SOA	service-oriented architecture
SPI	security parameter index
SPKI	simple public key infrastructure
SRP	Secure Remote Password
SRTCP	Secure Real-time Transport Control Protocol
SRTP	Secure Real-time Transport Protocol
SSH	Secure Shell

SSL	secure sockets layer
SSM	server-side middlebox
STLP	Secure Transport Layer Protocol
STUN	session traversal utilities for NAT
TACK	trust assurances for certificate keys
TC11	Technical Committee 11
TCP	Transmission Control Protocol
TDX	trust domain extensions
TEE	trusted execution environment
TEK	token encryption key
TIME	timing info-leak made easy
TLMSP	transport layer MSP
TLS	transport layer security
TLSA	TLS authentication
TOFU	trust-on-first-use
TPM	trusted platform module
TTP	trusted third party
UC	University of California
UDP	User Datagram Protocol
URL	uniform resource locator
UTA	using TLS in applications
UTC	Universal Time Coordinated
VoIP	voice over IP
VPN	virtual private network
W3C	World Wide Web Consortium
WAF	web application firewall
WAP	wireless application protocol
WebRTC	Web Real-Time Communication
WEP	wired equivalent privacy
WG	working group
WLAN	wireless local area network
WTLS	wireless TLS
WTS	web transaction security
WWW	World Wide Web

XML	Extensible Markup Language
XOR	exclusive or

About the Author

Rolf Oppliger[1] received an MSc and a PhD in computer science from the University of Berne, Switzerland, in 1991 and 1993, respectively. After spending a year as a postdoctoral researcher at the International Computer Science Institute (ICSI[2]) of the University of California (UC) Berkeley, he joined the Swiss National Cyber Security Centre (NCSC[3]) in 1995 and continued his research and teaching activities at several universities in Switzerland and Germany. In 1999, he received the venia legendi for computer science from the University of Zurich, Switzerland, where he still serves as an adjunct professor. Also in 1999, he founded eSECURITY Technologies Rolf Oppliger[4] to provide scientific and state-of-the-art consulting, education, and engineering services related to information security and began serving as the editor of Artech House's Information Security and Privacy Series. Dr. Oppliger has published numerous papers, articles, and books, holds a few patents, regularly serves as a program committee member of internationally recognized conferences and workshops, and was a member of the editorial board of some prestigious periodicals in the field. He has been a senior member of the Association for Computing Machinery (ACM), the Institute of Electrical and Electronics Engineers (IEEE) and its Computer Society, as well as a member of the IEEE Computer Society and the International Association for Cryptologic Research (IACR). Besides, he also served as the vice-chair of the International Federation for Information Processing (IFIP) Technical Committee 11 (TC11) Working Group 4 (WG4) on network security. His full curriculum vitae is available online.[5]

1 https://rolf-oppliger.ch and https://rolf-oppliger.com.
2 https://www.icsi.berkeley.edu.
3 https://www.ncsc.admin.ch.
4 https://www.esecurity.ch.
5 https://www.esecurity.ch/Flyers/cv.pdf.

Index