ARTECH HOUSE

INFORMATION SECURITY AND PRIVACY SERIES

SSL and TLS

Theory and Practice

Second Edition

ROLF OPPLIGER

SSL and TLS

Theory and Practice

Second Edition

SSL and TLS

Theory and Practice

Second 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 John Gomes

ISBN 13: 978-1-60807-998-8

10 9 8 7 6 5 4 3 2 1

To Lara

Contents

Preface

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

— Albert Einstein

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 met in one way or another. If they want to manifest that they are tech-savvy, then they bring in acronyms like SSL or 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 security problems. This is arguably not the case and largely exaggerates the role SSL/TLS can play in the security arena.

But SSL/TLS is still the most widely used and most important technology to secure e-$*$ applications or certain aspects thereof. 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 (e-mail), instant messaging, file transfer, terminal access, Internet banking, money transfer, electronic voting (e-voting), and online gaming. Many of these applications and respective application protocols are nowadays routinely layered on top of SSL and TLS protocols to provide some basic security services to their users.

Considering the wide deployment of the SSL/TLS protocols, it is important to teach e-$*$ 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 is not enough to design and develop secure applications.[1] 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 to 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 may provide is questionable or even illusive.

When I started to compile a teaching module about SSL/TLS more than 10 years ago, I was surprised to learn that the few books that were available then either addressed the technology only superficially or—maybe worse—were out-of-date. This was particularly true for the two reference books used in the field [1, 2]. They both appeared in 2000—which was almost a decade ago. Against this background, I decided to take my lecture notes and compile a new book that would not only address the fundamental principles of the SSL/TLS protocols, but would also try to explain the rationale behind their current designs. The resulting book appeared in 2009. The triumphant success of SSL/TLS and many recent developments have 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 hot and fastly evolving field: Books become outdated even sooner than is normally the case. So I have taken the opportunity to update the book and to bring it in line with some of the more recent cryptanalytical results and developments. Some of these results were known when the first edition of the book

1 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. Maybe most importantly, it was revealed in the same year that some elder 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.

went to press. However, they were not properly addressed, because there was no evidence that the underlying theoretical vulnerabilities could actually be exploited in practice. This has changed fundamentally, and terms like BEAST, CRIME, TIME, BREACH, POODLE, FREAK, Logjam, and Lucky 13 have made a lot of headlines and frightened both application developers and users. Keeping the old saying that "attacks always get better; they never get worse" in mind, one can reasonably expect many interesting attacks to be mounted in the future.[2] The situation remains risky and scary.

More recently, a book on SSL/TLS that is more in line with this book has appeared [3]. It also addresses the fundamentals of SSL/TLS and the abovementioned attacks. Unlike this book, however, major parts of [3] address complementary topics (that are not directly related to security), such as implementation, deployment, performance optimization, and configuration issues—sometimes even related to specific products. For the reasons discussed below, *SSL/TLS: Theory and Practice, Second Edition,* takes a different approach and delves into neither implementation issues nor configuration details of particular implementations. Also, it addresses some practically important topics that are entirely neglected in other books including [3], such as TLS extensions, TLS version 1.3, and datagram TLS (DTLS), and discusses cryptanalytical attacks and techniques to protect against them and mitigate the respective risks.

In addition to providing a basic introduction and discussion of the SSL/TLS protocols, another important goal of this second edition is to provide enough background information to understand, discuss, and put into perspective the latest cryptanalytical attacks and attack tools. This goal is ambitious, because the attacks are not at all simple and straightforward. Instead, they require quite a lot of background knowledge in cryptography. This is one of the reasons I have changed the outline of the book a little bit: Instead of providing a cryptography primer and starting from scratch, I assume that readers already have some basic knowledge about cryptography and start from there. Thus, readers who want to seriously delve into the security of SSL/TLS should acquire this knowledge first. Fortunately, they can use many books for this purpose, including [4].

Except for the omission of the cryptography primer, I have tried to stay as close as possible to the original outline of the first edition of the book. This edition is again intended for anyone who wants to get a deep understanding of the SSL/TLS protocols and their proper use—be they theorists or practitioners. As mentioned above, implementation issues and respective details are not addressed or only addressed superficially. 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 book.

2 As an example of such attacks, you may refer to the blog entry "The POODLE has friends," which is available at https://vivaldi.net/en-US/blogs/entry/the-poodle-has-friends.

They are modified and updated too frequently. The most popular open-source implementations are OpenSSL,[3] GnuTLS,[4] Bouncy Castle,[5] and MatrixSSL,[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 technicians and not 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 mentioned in footnote 1), and hence there are also a few open-source implementations that 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 some SSL/TLS implementations that are dual-licensed, meaning that they are available either as an open source or under a commercial license. Examples include mbed TLS[12] (formerly known as PolarSSL[13]), wolfSSL[14] (formerly known as CyaSSL), and cryptlib.[15] In addition, all major software manufacturers have SSL/TLS implementations and libraries of their own that they routinely embed in their products. Examples include 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. If you want to use the SSL/TLS protocols practically (e.g., to secure an e-$*$ application), then you have to delve into the documentation and technical specification of the application or development environment that you are currently using. This book is not a replacement for these documents; it is only aimed at providing the basic

3 http://www.openssl.org.
4 http://www.gnutls.org.
5 http://www.bouncycastle.org.
6 http://www.matrixssl.org.
7 http://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 web sites may be hidden from noise by the use of strong cryptography.
10 https://github.com/awslabs/s2n.
11 http://www.mitls.org.
12 https://tls.mbed.org.
13 https://polarssl.org.
14 http://yassl.com.
15 http://www.cryptlib.com.
16 Note that the NSS is also available as open source under a special Mozilla Public License (MPL).

knowledge to properly understand them—you still have to capture and read them. In the case of OpenSSL, for example, you may use [5] or Chapter 11 of [3] as a reference. Keep in mind, though, that, due to Heartbleed, the reputation of OpenSSL is discussed controversially in the community. In the case of another library or development environment, you have to read the original documentation.

In addition to cryptography, this book also assumes some basic familiarity with the TCP/IP protocols and their working principles. Again, this assumption is reasonable, because anybody not familar 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 of the many books about TCP/IP. Among these books, I particularly recommend the classic books of Richard Stevens [6] and Douglas Comer [7], but there are many other (or rather complementary) books available.

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 a book on TCP/IP networking. It is also explained in RFC 2026 [8] and updated on a web page hosted by the Internet Engineering Task Force (IETF).[17] For each protocol specified in an RFC document, we are going to say whether it has been submitted to the Internet standards track or specified for experimental or informational use. This distinction is important and highly relevant in practice.

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 a freely available open-source software tool. With regard to SSL/TLS, it is sufficiently complete, meaning that it can be used to analyze SSL/TLS-based data exchanges. We don't reproduce screenshots in this book, mainly because the graphical user interfaces (GUIs) of tools like Wireshark are highly nonlinear, and the corresponding screenshots are difficult to read and interpret if only single screenshots are available. When we use Wireshark output, we provide it in textual form. This is visually less stimulating, but generally more appropriate and therefore more useful in practice.

SSL/TLS: Theory and Practice, Second Edition, is organized and structured in seven chapters, described as follows.

- 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.

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

- 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.
- 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.
- Chapter 6, *Public Key Certificates and Internet Public Key Infrastructure (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. We have to look behind the scenes to understand what is going on.
- Chapter 7, *Concluding Remarks,* rounds off the book.

In addition, the book includes Appendix A, which summarizes the registered TLS cipher suites; Appendix B, a primer on padding oracle attacks; Appendix C, a list of abbreviations and acronyms; my biography; and an index.

At various places, the book refers to common vulnerabilities and exposures (CVEs). This 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.

Referring to the introductory quote of Albert Einstein, it is sometimes important to make a clear distinction between what is available in theory and what is achievable in practice. The second edition of this book makes this distinction more clear. While the first edition mainly focuses on the theoretical security of the SSL/TLS protocols, the second edition delves more deeply into the practically oriented security considerations as exemplified by the many attacks and attack tools mentioned above. In theory, they are a minor concern, because they can be easily mitigated. In practice, however, they are a major concern, because the mitigation is not always done or done properly. This simplifies an adversary's job considerably.

Again, I hope that *SSL/TLS: Theory and Practice, Second Edition*, serves your needs. The ultimate goal of a technical book is to 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 http://books.esecurity.ch/ssltls2e.html. I use this page periodically to post errata lists, additional information, and complementary material. I look forward to hearing from you in one way or another.

References

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

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

[3] 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.

[4] Oppliger, R., *Contemporary Cryptography*, 2nd edition. Artech House Publishers, Norwood, MA, 2011.

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

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

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

[8] Bradner, S., "The Internet Standards Process—Revision 3," Request for Comments 2026 (BCP 9), October 1996.

Acknowledgments

Many people have contributed to the writing and publication of both editions of this book, but particularly for the second edition. First, I thank the designers and developers of the SSL/TLS protocols. They have turned theory into practice and provided the basis for many network security solutions in use today. Second, I thank the buyers and readers of the first edition, especially the ones who provided feedback and approached me with questions, comments, and hints. It is always a pleasure to see a book being used in the field or even providing some controversy. Third, I thank all people who have answered specific questions, reviewed (parts of) the manuscript, or discussed interesting issues with me. All of these people have provided their valuable and highly appreciated contribution. Also, the staff at Artech House has been enormously helpful in producing and promoting the book. Among these people, I am particularly grateful to Aileen Storry, Molly Klemarczyk, and Lindsay Moore who supervised the entire production process. Last but not least, I am indebted to my family—my wife, Isabelle, and our beloved children, Lara (to whom this book is dedicated) and Marc. They have supported the entire book 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. More specifically, it starts with a brief introduction into information and network security in Section 1.1, delves more deeply into transport layer security and the evolution of the SSL/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 available in RFC 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 distinction between COMPUSEC and COMSEC has been blurred to some extent by recent developments in the realm of middleware and service-oriented architectures (SOAs). The distinction is no longer precise, and hence it may not be as useful as it was 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, organizational, personnel, or legal 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.

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 provide them) in some standardized way. One standard that can be used here is the *security architecture* that has been 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 only has four. 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.

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 (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, 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 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

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 web site is available at https://irtf.org.

2 This work has been abandoned.

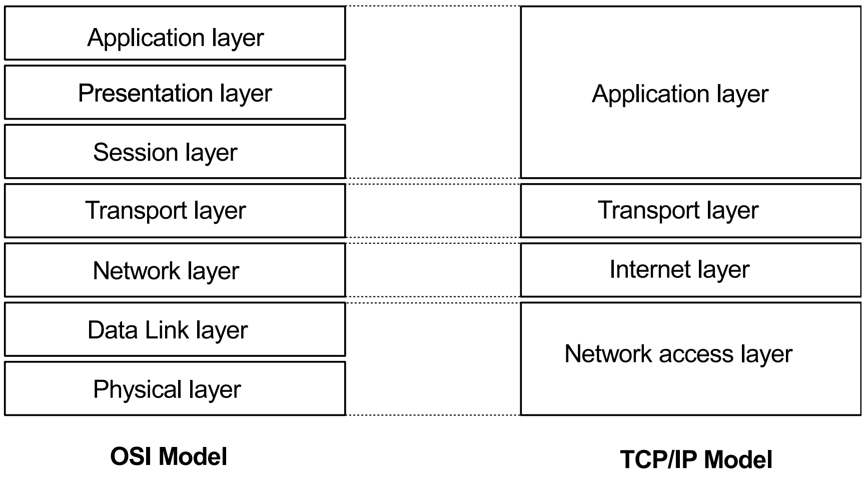

Figure 1.1 Layers of the OSI and TCP/IP models.

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 interrelationships. 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 are not addressed at all, but in any real-world application setting, these services may still be important. Take electronic payment or electronic voting (e-voting) systems as examples to make this point.

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

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.

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

the OSI reference model, so do services in the OSI security architecture. A security service hence provides a particular security functionality.

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 described as follows.

- 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 performing an unauthorized replay. 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 needs to be complemented with a data integrity service (as discussed in Section 1.1.1.4).

Authentication services are very important in practice, and they often represent a prerequisite for the provision of authorization, access control, and accountability 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 access rights, and accountability refers to the property that actions of an entity can be traced uniquely to this particular entity. All of these services are important for the overall security of a system.

1.1.1.2 Access Control Services

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 the resource does not have the proper privileges or permissions needed to do so. As such, access control services are typically the most commonly thought of services in computer and network security. However, 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 increasingly important in practice.

1.1.1.3 Data Confidentiality Services

In general parlance, data confidentiality refers to the property that data is not made available or disclosed to unauthorized entities, and hence *data confidentiality services* protect data from unauthorized disclosure. There are several forms of such services:

- A *connection confidentiality service* provides confidentiality for all data transferred over a connection.
- A *connectionless confidentiality service* provides confidentiality for individual data units (i.e., the 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 in 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 from traffic analysis.

The first three confidentiality services can be provided by standard cryptographic techniques and mechanisms, whereas these techniques and mechanisms—at least in their native form—do not help to provide traffic flow confidentiality services. Complementary security mechanisms are needed here.

1.1.1.4 Data Integrity Services

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

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

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 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 raises 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 relevant for the SSL/TLS protocols. Section 2.1 discusses what services are relevant for the SSL protocol, and the same argument also applies to the TLS protocol. This introduction aims to familiarize the readers with the terms and convey a common understanding for their meaning.

1.1.2 Security Mechanisms

In addition to the security services mentioned above, the OSI security architecture 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.

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 the data.

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. They 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). 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 (e.g., [6, 7]). Today, people are elaborating on using attributes to simplify the assignment of access rights in so-called attribute-based access controls (ABACs).[5]

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 (e.g., smartcards). The use of biometrics in a nontrivial setting also yields a strong authentication exchange mechanism for human users.

6. *Traffic padding mechanisms* can be used to protect against traffic analysis. It works by having the data originator generate and transmit randomly composed data hand in hand with the actual data. Only the data 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.

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,

5 http://csrc.nist.gov/projects/abac.

or destination. The assurance is provided by a trusted party—sometimes also called a trusted third party (TTP)—in a testifiable manner.

As explained later, all specific security mechanisms except access control, traffic padding, 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 traffic padding and routing control mechanisms are better placed underneath the transport layer.

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 itemized 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* is about functionality that can be 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 itemized above.
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).

3. It is increasingly important to complement preventive security mechanisms with detective and even 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.
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. Similar to event detection, security recovery largely depends on heuristics.

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 the case today, 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 particular 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 [8] or Chapter 5 of [9] 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 [10] on the Internet layer, the SSL/TLS protocols on the transport layer,[6] and some variation of a secure messaging scheme [11] on the application layer are the most appropriate possibilities. There is a *end-to-end argument* in system design that strongly speaks in favor of providing security services at a higher layer [12]. The argument basically says the following:

- 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;
- These intermediaries are incapable of ensuring that the data is processed correctly.

The bottom line is that, whenever possible, communications 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 [13], it particularly applies to the provision of network security services.

Following the end-to-end argument and design principle, the IETF chartered a web transaction security (WTS) 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 [14–16]. 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 [16]. As such, S-HTTP is conceptually similar to today's specifications of the World Wide Web Consortium (W3C) related to extensible markup language (XML) encryption and XML signatures. It was submitted to the web transaction discussion in 1994, and—due to its strong initial support from the software industry—it seemed to be only a question of time until it would become the key player in the field.

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.

7 http://www.ietf.org/html.charters/OLD/wts-charter.html.

Things evolved differently, however. Independent from the end-to-end argument and the S-HTTP proposal, the software developers at Netscape Communications[8] 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* (SSL), and its job 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 interwinded 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 datagram protocol (UDP),[9] and it has only been recently that the TLS protocol has 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) 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 provide data integrity protection. 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 all types of replay attacks. Later on, the designers of SSL 1.0 added sequence numbers and

8 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.

9 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.

checksums, but still used an overly simple cyclic redundancy check (CRC) instead of a cryptographically strong hash function to protect the integrity of data.

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. Netscape Communications then released the Netscape Navigator that implemented SSL 2.0 together with a few other products that also 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.[10] 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*[11]). The SSL patent was granted in August 1997 and was assigned to 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.

With the release of the Netscape Navigator (supporting the newly specified SSL 2.0 protocol), the Internet and 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 the 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.[12] In particular, the two protocols used a compatible record format, meaning that a server could easily support both protocols. 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 still support it, but outside the world of Microsoft the PCT protocol has never been used and has silently sank into oblivion. All one needs to know is the acronym and what it stands for (i.e., 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 (still in the year 1995). The SSL 3.0 protocol was specified by Alan O. Freier and Philip Karlton from Netscape Communications with the support of an

10 http://tools.ietf.org/html/draft-hickman-netscape-ssl-00.
11 U.S. Patent No. 5,657,390.
12 http://graphcomp.com/info/specs/ms/pct.htm.

independent consultant named Paul C. Kocher (who later founded Cryptography Research[13]). Also, around this time, Netscape Communications hired several security professionals, including Taher Elgamal—the inventor of the Elgamal public key cryptosystem [17]. These distinguished security professionals helped 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 [18] 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 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 has been complemented with SHA-1, and the MAC construction has been made 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 [19]. 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 four years later in 2015 [20].

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 its huge installed base pushing PCT. The company also

13 http://www.cryptography.com.

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 are now included in the TLS protocol). In this situation, an IETF Transport Layer Security (TLS) Working Group[14] was formed in 1996 to resolve the issue and to 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 [21] 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).
- Second, at the April 1995 IETF meeting in Danvers, Massachusetts, the IESG adopted the so-called *Danvers doctrine* [22], 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 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 around 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 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

14 http://www.ietf.org/html.charters/tls-charter.html.

in general and 3DES in particular. After some heated discussions between the IESG and the IETF TLS WG, 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 (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.[15] In fact, the first version of the TLS protocol (i.e., TLS 1.0) was specified in RFC 2246 [23] 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, 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. 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 (TLS 1.1) was specified in Standards Track RFC 4346 [24], making RFC 2246 [23] obsolete. As discussed in the following two chapters, there were some cryptographic problems that needed to be resolved in TLS 1.1. In addition to TLS 1.1, the IETF TLS WG also released the first version of the DTLS protocol to secure UDP-based applications (DTLS 1.0) in Standards Track RFC 4347 [25]. After another two-year revision period, the TLS protocol version 1.2 (TLS 1.2) was specified in Standards Track RFC 5246 in 2008 [26]. This document not only made RFC 4346 obsolete, but also a few other RFC documents (most of them referring to TLS extensions). Four years later, in January 2012, version 1.2 of the DTLS protocol was officially released in Standards Track RFC 6347 [27]. Today, the IETF TLS WG is actively working on version 1.3 of the TLS protocol (TLS 1.3) [28].[16] This version is just the latest step in the evolution of the SSL/TLS protocols, but it is not going to be the end of the story (and not the end of the protocol evolution, respectively).

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

16 In spite of the fact that it is usually inappropriate to reference Internet drafts, we make an exception for TLS 1.3. The protocol specification is fairly stable and highly relevant for the topic of this book.

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 we think that there are two major reasons for their success:

- On one hand, they can be used to secure any application layer protocol that is stacked on 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.[17] This simplifies the deployment of the protocols considerably.

The SSL/TLS protocols are cryptographic security protocols, meaning that 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* [29]. It basically says that the 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 on the network. 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. For example, many contemporary attacks are either based on malware or they 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 protcols 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 definitively have a deep impact on the

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

design of future versions of the TLS protocol. The bottom line is that one always has to be careful when people make claims about the security of a protocol. One always 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 for the SSL/TLS protocols.

References

[1] Shirey, R., “Internet Security Glossary, Version 2,” Informational 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 Publishers, Norwood, MA, 2001.

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

[7] Coyne, E.J., and J.M. Davis, *Role Engineering for Enterprise Security Management*. Artech House Publishers, Norwood, MA, 2008.

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

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

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

[11] Oppliger, R., *Secure Messaging on the Internet*, Artech House Publishers, Norwood, MA, 2014.

[12] 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.

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

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

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

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

[17] 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.

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

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

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

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

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

[23] Dierks, T., and C. Allen, “The TLS Protocol Version 1.0,” Standards Track RFC 2246, January 1999.

[24] Dierks, T., and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.1,” Standards Track RFC 4346, April 2006.

[25] Rescorla, E., and N. Modadugu, “Datagram Transport Layer Security,” Standards Track RFC 4347, April 2006.

[26] Dierks, T., and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” Standards Track RFC 5246, August 2008.

[27] Rescorla, E., and N. Modadugu, “Datagram Transport Layer Security Version 1.2,” Standards Track RFC 6347, January 2012.

[28] Rescorla, E., “The Transport Layer Security (TLS) Protocol Version 1.3,” Internet-Draft, October 2015.

[29] 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] mentioned in the title of this book (i.e., the SSL protocol that is specified with retroactive effect in RFC 6101 [1]).[2] More specifically, we provide an introduction in Section 2.1, overview the SSL protocol and its subprotocols in Section 2.2, outline the transcript of a protocol execution in Section 2.3, elaborate on the security analyses and attacks that are available in Section 2.4, and conclude with some final remarks in Section 2.5. The bottom line is that—due to the recent POODLE attack—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 the TLS and DTLS protocols. This allows us to shorten the chapters on TLS and DTLS considerably, and to put everything into the right perspective. The SSL protocol is really key to properly understanding the TLS and DTLS protocols.

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.

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 as a reference.

Referring to the terminology introduced in Section 1.1, the SSL protocol 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, it does not provide nonrepudiation services—neither nonrepudiation with proof of origin nor nonrepudiation with proof of delivery. This is in sharp contrast to S-HTTP and XML signatures that are able (and have been specifically designed) to provide such services.

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 (usually called a 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) 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 is optionally compressed, authenticated, encrypted,[3] prepended with a header, and transmitted to the recipient. 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, decompressed, and reassembled, before the data is actually delivered to the higher-layer—typically the application-layer—protocol.

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

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 include padding verification.

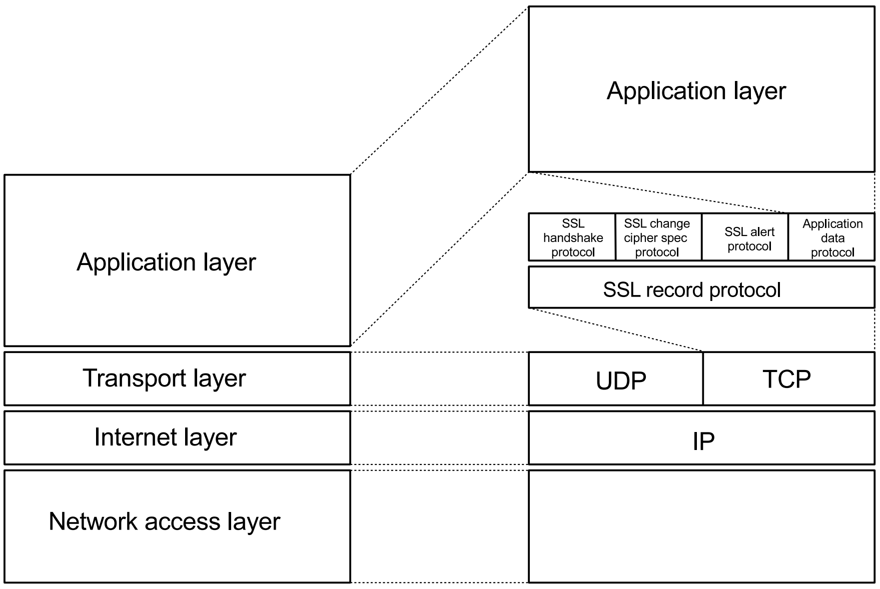

Figure 2.1 The SSL with its (sub)layers and (sub)protocols.

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 TCP in the case of the TCP/IP protocol suite.[6] This layer basically comprises the *SSL record protocol* that is used (together with the SSL application data protocol) for the second function mentioned above (i.e., the encapsulation 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 (i.e., establishment of a secure

6 This is in contrast to the DTLS protocol that is stacked on top of UDP. The DTLS protocol is addressed in Chapter 4.

connection). More specifically, it allows the communicating peers to authenticate each other and to negotiate a cipher suite and a compression method. As its name suggests, 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.
- 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 (i.e., 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 simultaneously. When we refer to a specific subprotocol, we usually employ its full name (but sometimes also leave the prefix "sub" aside).

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 an SSL 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 an SSL record is additionally tagged with a (message) length field that refers to the length of this particular message. Note and keep in mind that multiple SSL messages may be carried inside a single SSL record.

7 Note that recent attacks have shown that compressing data in SSL/TLS is dangerous and should be avoided in the first place. This topic is further addressed in Section 3.8.2.

One 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 accomodate 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 the following two strategies to achieve this:

- 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 both on the original port and the new (secure) port, for any connection that arrives on the secure port; SSL can then be invoked automatically and transparently.
- In contrast to this, in an *upward negotiation strategy*, a single port is used for both versions 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, both strategies 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 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 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, 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,

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

most protocol designers have made a choice in favor of the separate port strategy. For example, until the SSL 3.0 protocol specification was officially released in 1996, the Internet Assigned Numbers Authority (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` effectively became `nntps`. 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. Among these examples, `ldaps`, `ftps` (and `ftps-data`), `imaps`, and `pop3s` are particularly 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. Just for the sake of completeness, we note that STARTTLS is based on the SMTP extensions mechanism specified in RFC 1869 [5] and that multiple implementations of STARTTLS originally had a flaw that is documented in CVE-2011-0411 and could be exploited in a STARTTLS command injection attack. Because STARTTLS and STARTTLS command injection attacks are outside the scope of this book, they are not further addressed here.

Table 2.1
Port Numbers Reserved for Applicaton 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
…	…	…

9 http://www.iana.org/assignments/port-numbers.

The separate port strategy has the disadvantage that it effectively halves the number of available ports on the server side (because two ports must be reserved for each application-layer protocol). During an IETF meeting in 1997, the applications 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 mechansism for HTTP/1.1) has been available for almost a decade and has been submitted to the IETF standards track, there has hardly been any 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 most widely deployed, and this is not likely to change anytime soon in the future.

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 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 specific 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 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 muliple-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 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 SSL 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,[10] 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. 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.

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

10 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.

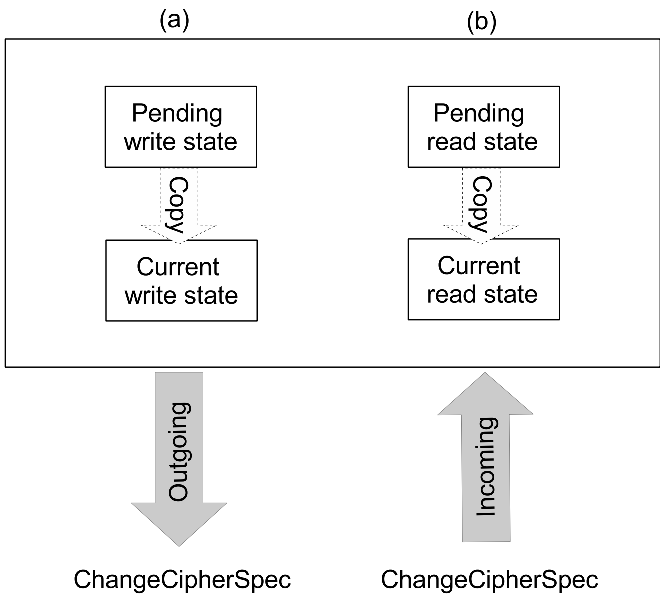

Figure 2.2 The SSL state machine.

occurs when a CHANGECIPHERSPEC message is sent or received during an SSL handshake. The rules are outlined 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.
- 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.

When the SSL handshake negotiaton is complete, the client and server exchange CHANGECIPHERSPEC messages,[11] and hence the pending states are copied into the current states. This means that they can now use the newly agreed-upon

11 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.

cryptographic (and other) parameters. The FINISHED messages are then the first SSL handshake messages that are protected according to these new parameters.

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.

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

Where appropriate and available (in the 1990s when the protocol was designed), the developers of the SSL protocol tried to follow standards. For example, RSA signatures are always performed using public key cryptography standard (PKCS) #1 block type 1,[12] 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].[13] As discussed in Section 2.4 (and more thoroughly addressed in Appendix B.1), PKCS #1 version 1.5 turned out to be susceptible to an adaptive chosen ciphertext attack (CCA2) that was due to Daniel Bleichenbacher, and hence it was replaced with a more secure version 2.0 in 1998 [7]. Later on, more subtle vulnerabilities led to another revision of PKCS #1, so the current version is 2.1 [8]. We postpone the treatment of these vulnerabilities and the respective attacks to Section 2.4. Instead, we now continue with a more detailed overview and explanation of the SSL protocol and its subprotocols.

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

13 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.

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 $2^{64}-1$.

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 encapsulation 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 (cf. Table 2.2) and the cryptographic parameters and elements of the SSL connection state (cf. Table 2.3). The result is sent to the recipient in the fragment field of the respective SSL record.

The SSL record processing consists of five steps, illustrated in Figure 2.3. The first four steps refer to *fragmentation*, *compression*, *message authentication*, and

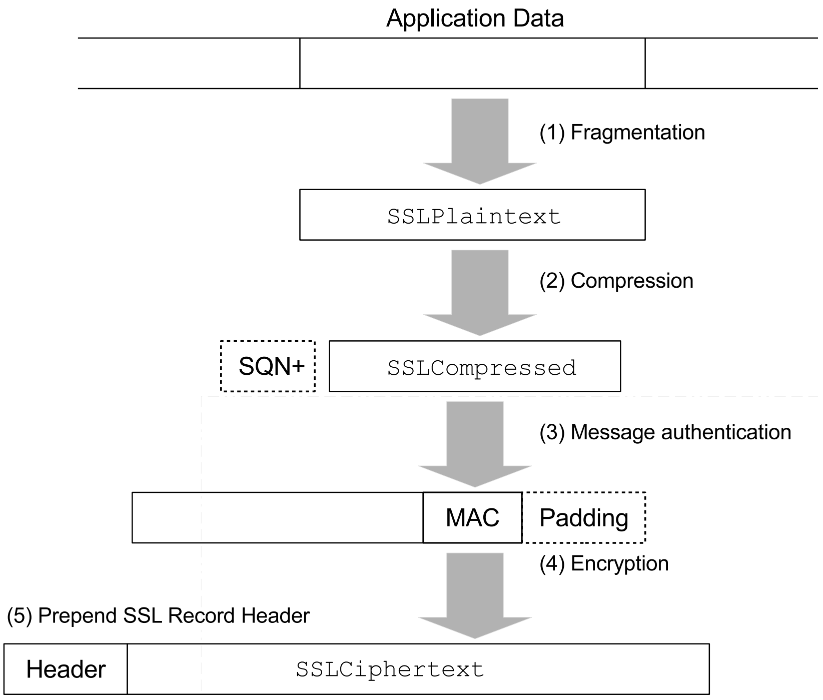

Figure 2.3 The SSL record processing (overview).

encryption, whereas an SSL record header is prepended in the fifth step. 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 (i.e., message authentication and encryption) is called `SSLCiphertext`. This is the one that is actually included in an SSL record's *fragment* field. In addition to the fragment field, each data structure comprises an 8-bit type field, a 16-bit version field, and another 16-bit length field. Let us briefly overview the contents of these fields:

1. The *type* field 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.

2. The *version* field refers to the version of the SSL protocol in use. It is a two-byte value that consists of a major and a minor version number.[14] 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 on, 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).
3. The *length* field 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 concatenated into a single SSL record). The length field comprises two bytes, so in theory an SSL record could be up to $2^{16} - 1 = 65,535$ bytes in length. According to the SSL protocol specification, however, the maximum record length should not exceed $2^{14} - 1 = 16,384$ bytes, but there are always implementations that do not follow this recommendation and use larger records.

In all SSL record processing steps, the contents of these fields are copied from one SSL data structure to the next one. This means that the data structures illustrated in Figure 2.3 comprise all fields (i.e., type, version, length, and fragment), but that this subdivision is not visually replicated. Let us now briefly go through the five steps mentioned above.

2.2.1.1 Fragmentation

In the first step, the SSL record protocol fragments the higher-layer protocol data—typically the application-layer data—into blocks of 2^{14} bytes or less. Each block is packed into an `SSLPlaintext` structure. More specifically, the block is written into the fragment field of the `SSLPlaintext` structure that is augmented with appropriate values for the type, version, and length fields.

14 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.

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 that is specified for the SSL session. In the case of SSL 3.0, it is initially set to null, so no compression is 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). The bottom line is that the designers of the SSL protocol made a reasonable and wise decision when they enabled the SSL record protocol to compress data before it is encrypted.

Today, the situation is slightly 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 a lot of sense (due to the line of argumentation given above), but on the other hand, the combined use of compression and encryption is known to be dangerous, as it introduces some new vulnerabilities that may be exploited by specific attacks (cf. Section 3.8.2). So the use of compression in SSL/TLS 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 be lossless and that it not increase the length of the fragment field by more than 1,024 bytes.[15] However, no compression method other than null was defined for SSL 3.0. This has since changed, and at least a few compression methods have been specified for TLS 1.2 (cf. 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. Support for compression has been entirely removed in TLS 1.3.

In either case (i.e., independent from whether the data is compressed or not), the output of this step is an `SSLCompressed` structure with distinct type, version, length, and fragment fields. If the SSL protocol uses null compression, then the

15 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.

compression method represents the identity operation, and hence the fragment fields of the `SSLPlaintext` and `SSLCompressed` structures are the same. The type and version fields of the structures also remain the same, whereas the length field of the `SSLCompressed` structure has a smaller value, if compression is invoked and applied. If null compression is used, then the length fields of the two structures are the same, as well.

2.2.1.3 Cryptographic Protection

In the third and fourth steps, the SSL record protocol protects an `SSLCompressed` structure as defined in the cipher spec of the SSL session state. 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 is 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 recommendations, people basically refer to cipher suites.

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.[16] 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 its components in terms of key exchange algorithm, cipher (i.e., symmetric encryption system), and cryptographic hash function. For example, SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA refers to the cipher suite that comprises an RSA-authenticated fixed Diffie-Hellman key exchange, Triple DES (3DES) in CBC mode for encryption,[17] 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 as summarized in Appendix A. There are a few differences. For example, the FORTEZZA key exchange algorithm KEA is no longer supported in TLS. So the respective cipher suites from Table 2.4 are no longer present in

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

17 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.

Appendix A (and the respective places are now occupied by some cipher suites that employ Kerberos for key exchange).

Table 2.4
SSL Cipher Suites (According to [1])

Name of the Cipher Suite	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	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

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 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 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 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 possible and appropriate.

There is always an active cipher suite. It is initially set to SSL_NULL_WITH_NULL_NULL, which does not provide any security 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 one itemized in Table 2.4.

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:

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 before 2001 that Hugo Krawczyk and Ran Canetti showed 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. However, this is clearly not the case and changing the order of the message authentication and encryption operations in retrospective is difficult (to say the least). So people have stayed with the AtE approach even for the TLS protocol. According to the results of Krawczyk and Canetti, this shouldn't be a problem, because they also showed that—at least in theory and if properly implemented—AtE is also 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 many possibilities to attack an AtE implementation, and many recent padding oracle attacks against SSL/TLS bear witness to this fact. We revisit this topic several times in this book.

Let us now have a closer look at the key exchange, message authentication, and encryption as they are 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 bilaterally between the client and the server. In the case of SSL, this material is derived from a 48-byte premaster secret (that is called `pre_master_secret` in the SSL protocol specification). There are three key exchange algorithms that can be used to establish such a premaster secret: RSA, Diffie-Hellman, and FORTEZZA.[18] Some of these algorithms combine a key exchange with peer entity authentication and hence actually refer to authenticated key exchange. To make this distinction explicit, a key exchange without peer entity authentication is called *anonymous*. SSL provides the following possibilities 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 long-termed and retrieved from a public key certificate, or short-termed and provided for a particular key exchange. As outlined in Sections 2.2.2.4 and 2.2.2.5, the situation is slightly more involved, due to the fact that an RSA key exchange can also be exportable (if sufficiently short keys are 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) represents 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.

18 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.

– In a *fixed Diffie-Hellman key exchange* (DH), the parameters that are needed for the Diffie-Hellman key exchange are fixed and part of the respective public key certificates. This applies to the server, but it may also apply to the client. This means that if client authentication is required, then the client's Diffie-Hellman parameters are fixed and part of the client certificate. Otherwise (i.e., if client authentication is not required), the parameters are generated on the fly and provided in appropriate SSL handshake messages (i.e., CLIENTKEYEXCHANGE messages).

– In an *ephemeral Diffie-Hellman key exchange* (DHE), the parameters that are 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). Here, a controversy is going on whether a standard group should be used for the Diffie-Hellman key exchange or not.[19]

  ∗ On the one hand, it has been shown that cross-protocol attacks are feasible, if arbitrary (i.e., nonstandard) groups are used [11].

  ∗ On the other hand, it has been shown that breaking the Diffie-Hellman key exchange by computing the respective discrete logarithm can be simplified by using precomputation that is feasible for many standard groups.

  A unique answer can be given only if the Diffie-Hellman key exchange is done on elliptic curves. In this case, it is clearly advantageous to use standard groups (but sometimes people are also afraid of backdoors that may be existent in some of these groups). In either case, 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 verification key is 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,

19 Standard groups are, for example, specified by the IETF in RFC 4419 and RFC 5114 and by many other standardization bodies in similar documents.

it may be anybody spoofing a legitimate participant. The key exchange is inherently anonymous, and as such it is susceptible to man-in-the-middle (MITM) attacks.

DHE appears to be the most secure version of the Diffie-Hellman key exchange, mainly because it yields temporary keys that are authenticated in some meaningful way. 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). But the single most important security advantage of DHE is its ability to provide *forward secrecy* that is sometimes also called *perfect forward secrecy* (PFS).[20] Forward secrecy means that the compromise of any long-term key does not automatically compromise all session keys. Note what happens in the case of RSA (that does not provide PFS): 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 protect 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 (or PFS) has a clear 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, and hence that the use of DHE is highly recommended.

- 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 an NSA approved security microprocessor called Capstone chip (sometimes also known as MYK-80) that also implemented a cipher called Skipjack. The FORTEZZA KEA was actually a way to provide the keying material necessary for Skipjack in a way that allowed a trusted third party to retrieve the encryption keys if needed and

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

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, key escrow and respective techniques have silently sunk into oblivion.

In the past, RSA has been the predominant SSL key exchange method. This is about to change, mainly because—as mentioned above—DHE provides forward secrecy and has security advantages. So the ongoing trend toward cipher suites that comprise DHE is likely to continue in the future. This is particularly true for TLS 1.3 that mandates any key exchange to be ephemeral.

The result of the key exchange is a premaster secret. Once a premaster secret is established, it can be used to construct a master secret that is called `master_secret` in the SSL protocol specification. According to Table 2.2, the master secret represents an SSL session state element. It is constructed 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. Interestingly, the construction does not use either MD5 or SHA-1, but combines the two cryptographic hash functions (probably to compensate for any deficiency). The combined use of MD5 and SHA-1 in the way specified above yields a unique and nonstandard pseudorandom function (PRF). As discussed in Chapter 3, the TLS protocol uses another—somehow more standardized—PRF to derive a master secret from a premaster secret.

An MD5 hash value is 16 bytes long, so the total length of the master secret is $3 \cdot 16 = 48$ bytes. Its construction is the same for the RSA, Diffie-Hellman, or FORTEZZA key exchange algorithms (in the case of FORTEZZA, 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 the premaster secret can be safely deleted from memory once the master secret has been constructed. This is certainly a good idea, because something that is nonexistent cannot be attacked in the first place.

Equipped with the master secret, essentially the same handcrafted PRF as given above can also be used to generate an arbitrarily long key block, termed `key_block`. In this PRF 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 of 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 optional). 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 contruction of FORTEZZA. It did not apply to the construction of the encryption keys and IVs for FORTEZZA—these values were generated inside the security token of the client and securely transmitted to the server in a respective key exchange message. Since the FORTEZZA key exchange has never been widely used and is no longer an alternative, it is not discussed further here. The same is true for the exportable encryption algorithms (used in exportable cipher suites). They require some additional processing to derive the final encryption keys and IVs. Two recent attacks have shown that exportable cipher suites are inherently dangerous and should no longer be supported:

- The *FREAK attack*[21] was published in March 2015 [12];

21 The acronym FREAK stands for “factoring attack on RSA export keys,” or something similar.

- The *Logjam attack*[22] was announced two months later, in May 2015 [13].

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. Because both attacks have been discovered recently and also apply to TLS, we postpone their treatment to Section 3.8.4.

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 HMAC construction that is specified in RFC 2104 [14] and is in widespread use. 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 comparable to that of the HMAC construction. Remember that 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 any representative of 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, $\oplus$ the bit-wise addition modulo 2, and $\parallel$ the concatenation operation.

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

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

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

22 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.[23] The sequence number represents an SSL connection state element. To keep things sufficiently simple, the sequence number and all other data that is input to the SSL MAC is termed SQN+ in Figure 2.3. In the end, the resulting SSL MAC is appended to the `SSLCompressed` structure, before the structure is handed over to the encryption process.

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 discussed in Section 2.4, the use of RC4 is no longer allowed in the realm of the SSL/TLS protocols.
- If a block cipher is used, then things are more involved, mainly for 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.[24] 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

23 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.

24 In the case of 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.

(specifying the length of the padding) is repeated for all 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 explained in Section 2.4 exploits the overly simple padding scheme of SSL. It has brought SSL to the end of its life cycle.

– Second, an IV is needed in some encryption modes. In the case of the CBC mode, for example, the SSL handshake protocol must provide an IV that also represents an SSL connection state element (as shown in 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 (i.e., the BEAST attack explained in Section 3.3.1) can be mounted against SSL 3.0 and TLS 1.0.

According to Table 2.4, SSL 3.0 envisioned the use of the block ciphers 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 above-mentioned Skipjack cipher (named FORTEZZA). It goes without saying that, in principle, any other block cipher can be used, as well.

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 recent results in the cryptanalysis of RC4, this has changed again, and—as mentioned above—the use of RC4 is no longer recommended (see Section 2.4).

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, so the total length of the `SSLCiphertext` fragment (i.e., encrypted data and MAC) should never exceed $2^{14} + 2,048$ bytes. The last last step is to prepend a header to complete the SSL record.

2.2.1.4 SSL Record Header

Like any SSL structure, an SSL record comprises a type, version, length, and fragment field. This is illustrated in Figure 2.4. The first three fields represent the SSL record header and are defined as described above. The fragment field of the

SSL record comprises an `SSLCiphertext` structure (optionally compressed) with a MAC in possibly encrypted form. Note that the `SSLCiphertext` structure may also comprise some padding (depending on the cipher suite in use). Anyway, the entire SSL record is sent to the recipient in a TCP segment. Again, if multiple SSL records need to be sent to the same recipient, then these records may be sent in a single TCP segment.

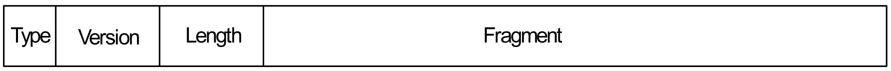

Figure 2.4 The outline of an SSL record.

2.2.2 SSL Handshake Protocol

The SSL handshake protocol is layered on top of the SSL record protocol. It allows a client and server to authenticate each other 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 square brackets 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 message is therefore illustrated but written in italics. 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 both hexadecimal and decimal notation).

The SSL handshake protocol comprises four sets of messages—sometimes called *flights*—that are exchanged between the client and the server. All messages of a 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 that may be sent from the server to the client to initiate an SSL handshake. This message, however, is seldom used in practice and 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. Sending messages in a different and unexpected order must always result in an error that is fatal.[25]

- The first flight comprises a single ClientHello message (type value 0x01 or 1) that is sent from the client to the server.

25 In the realm of SSL/TLS, a fatal error must lead to an abortion of the respective protocol execution.

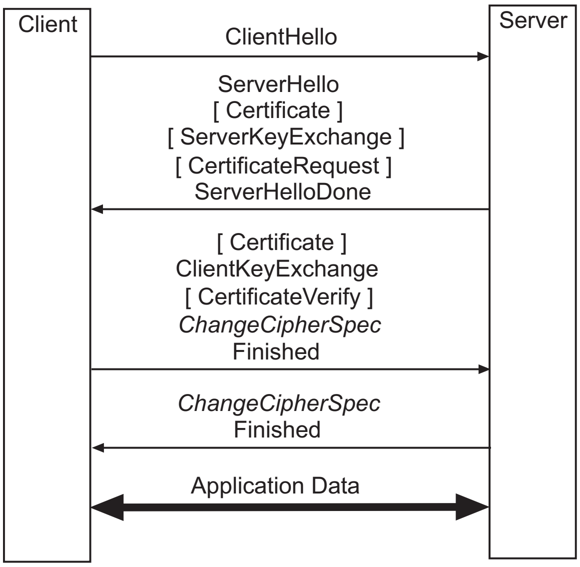

Figure 2.5 The SSL handshake protocol.

- The second flight comprises two to five messages that are sent back from the server to the client:

  1. A SERVERHELLO message (type value 0x02 or 2) is sent in response to the CLIENTHELLO message.
  2. If the server is to authenticate itself (which is generally the case), it may send 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.

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 for further use.

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

  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[26] 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 exchanging application-layer data—typically using the SSL application data protocol.

Most SSL sessions start with a handshake, proceed with an exchange of application data, and are terminated at some later point in time. If more data needs to be exchanged (between the same client and server), then there are basically two

26 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).

possibilities: Either a new (full) handshake takes place to renegotiate a new session, or a simplified handshake takes place to resume an old (i.e., previously established) session. In the first case, we are talking about a *session renegotiation*, whereas in the second case, we are talking about a *session resumption*. Both terms need to be clearly distinguished, as they lead to different protocol executions.

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 *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. This is known as *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 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 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) as 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 used for message authentication) is about to overflow. (Note, however, that this seldom occurs, because it is a 64-bit number.) 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 some new vulnerabilities that may be exploited in a specific attack—the so-called *renegotiation attack*. We postpone the elaboration of this attack and some possibilities to protect against it to Section 3.8.1, mainly because the attack was published in 2009 in the realm of the TLS protocol. So we simply ignore the topic as far as SSL is concerned.

If a handshake has already been performed recently, then the respective session may be resumed in 1 round-trip time (1-RTT). So session resumption is much more efficient than session renegotiation. If a client and server are willing to resume a previously established 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 this 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

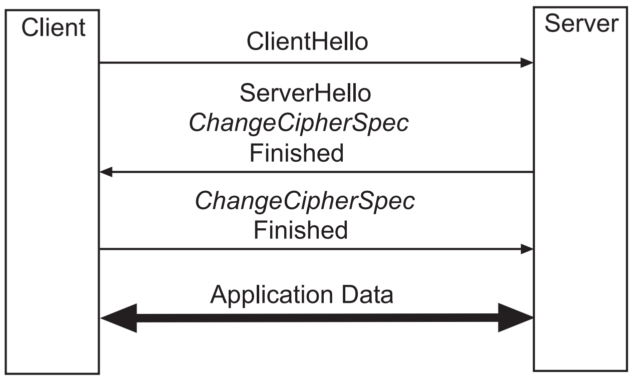

Figure 2.6 The simplied SSL handshake protocol (to resume a session).

found, then the server must generate a new session ID and the client and server must go through a full SSL handshake, meaning that they have to fall back to a session renegotiation. So session resumption may be seen as the more efficient version of a session renegotiation.

For the sake of completeness, we add the remark 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 further addressed in Section 3.4.1.18. It is not relevant for the SSL protocol, and hence we don't address it in this chapter.

Let us now 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. The general structure of such a message 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 (standing for 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 this 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 sent in a separate record. However, keep in mind that this need not be the case, and that—depending on

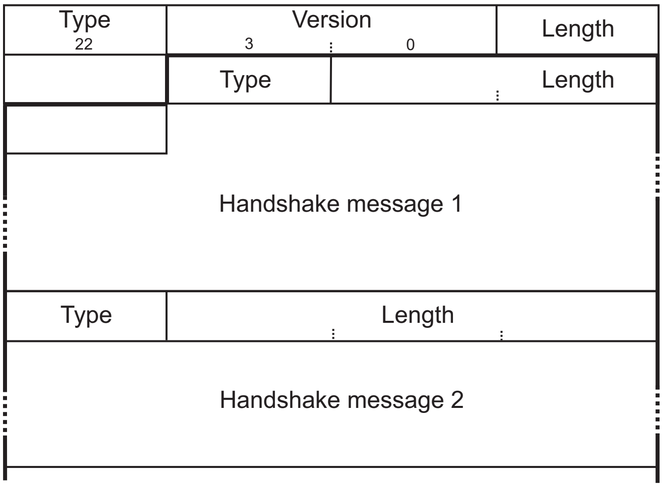

Figure 2.7 The general structure of an SSL handshake protocol message.

the implementation—multiple handshake messages may be sent in a single record. 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. Also, keep in mind that the length field value of a record header can only be indicated if the message is sent in a record of its own. (Otherwise the value is 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 to the servers. If, for example, an SSL connection has been in use for so long that its security is put in question, then the server can send a HELLOREQUEST message to force the client to renegotiate a new session and establish new keys.

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

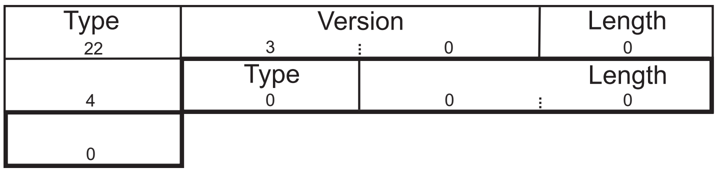

Figure 2.8 An SSL HELLOREQUEST message.

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, because it represents a flight of 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 starts with the usual 5-byte SSL record header, a type field value of one (referring to a CLIENTHELLO message), and a 3-byte message length field value. 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 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`. It basically 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

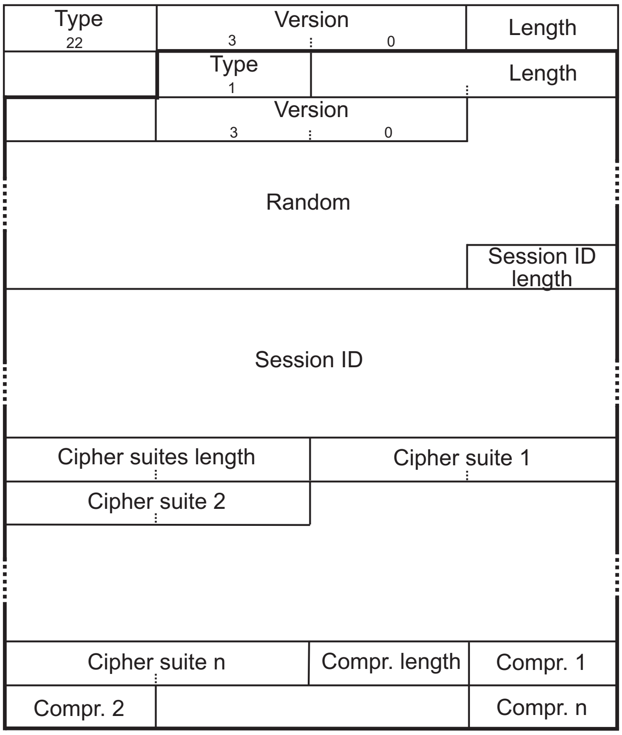

Figure 2.9 An SSL CLIENTHELLO message.

Coordinated Universal Time (UTC[27]) of January 1, 1970, not counting leap seconds[28] according to the sender's internal clock.[29]

– 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. Consequently, it is required that it is unpredictable to some extent, and hence that a cryptographically strong

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

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

29 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.

random or pseudorandom bit generator is used to generate the second part of it.

- 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, its actual value must be specified.
- 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 otherwise it must be ignored. 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, exploited by the extension mechanism used in TLS 1.2 (cf. Section 3.4.1).

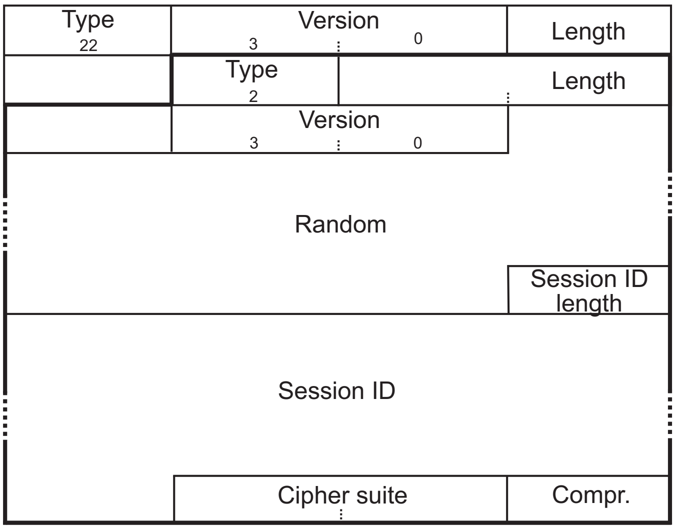

Figure 2.10 An SSL SERVERHELLO message.

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 the session.

More specifically, an SSL SERVERHELLO message starts with the usual 5-byte SSL record header, a type field value 0x02 (referring 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 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 is then identifying 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 in the field name). 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 (note the singular form). Again, for resumed sessions, the value for the compression method field must be copied from the resumed session state.

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 to establish 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. (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., in the same flight). The same message type occurs 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 as in the case of the FORTEZZA key exchange algorithm).

As illustrated in Figure 2.11, an SSL CERTIFICATE message starts with the usual 5-byte SSL record header, a type field value of value 11 (referring to an SSL CERTIFICATE message), and a 3-byte message length field. As mentioned above, the body of the message then contains a 3-byte length field (that contains a value that is three less than the message length) and an actual certificate chain. Each certificate

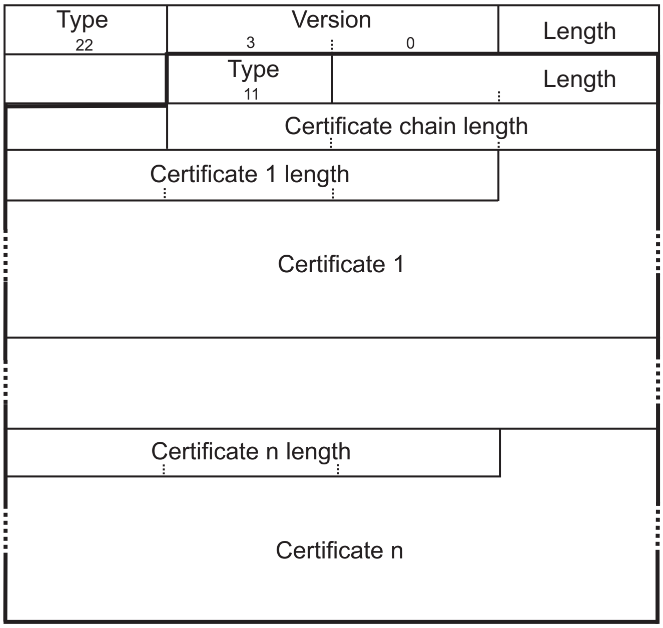

Figure 2.11 An SSL CERTIFICATE message.

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 CERTIFICATE message may be very long.

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 CAs, such as VeriSign, and contained a special attribute in the extended key usage (`extKeyUsage`) field. In fact, an International Step-Up certificate included the OID 2.16.840.1.113730.4.1, 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 support International Step-Up and SGC.

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 or 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*[30] 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 important. Since 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 (cf. Section 3.8.4), the notion of export-grade cryptography has become important again. In fact, it is highly recommended today to disable all export-grade cipher suites by default. The mere existence of these cipher suites has turned out to be dangerous.

2.2.2.5 SERVERKEYEXCHANGE Message

If RSA is used for key exchange, then the client can retrieve the public key from the server certificate and encrypt the premaster secret with this key. Similarly, if a fixed

30 http://www.fortify.net.

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. In all of these cases, the server's CERTIFICATE message is sufficient and no additional information is required for the client to securely communicate a premaster secret to the server. In particular, no SERVERKEYEXCHANGE message is needed. In all other cases, however, the client needs some additional information, and this information must be delivered by the server in such a SERVERKEYEXCHANGE message. Most importantly, this applies for the DHE key exchange that has become important in practice.

A special case occurs if RSA_EXPORT is used for key exchange. In this case, a former U.S. export law may apply, according to which RSA keys larger than 512 bits could not directly be used for key exchange in software exported from the United States. Instead, these RSA keys could be used (as signature-only keys) to sign temporary shorter RSA keys for key exchange. Consequently, temporary 512-bit RSA keys were used, and these keys were signed with the larger RSA keys (found in the certificate). Needless to say, this extra step is obsolete if the original RSA keys are 512 bits long or shorter. The bottom line is that one has to distinguish between two cases:

- If RSA_EXPORT is used for key exchange and the public key in the server certificate is longer than 512 bits, then the extra step must be taken and the SERVERKEYEXCHANGE message (that includes a signed shorter RSA key) must be sent.
- If, however, RSA_EXPORT is used for key exchange and the public key in the server certificate is 512 bits long or shorter, then the extra step need not be taken and the SERVERKEYEXCHANGE message need not be sent.

In either case, keep in mind that an export-strength RSA key is restricted to 512 bits, and that this key length is insufficient to withstand more recent attacks. As demonstrated by the FREAK attack (cf. Section 3.8.5), an adversary who is able to factorize a 512-bit RSA key can mount a MITM attack and sometimes break the SSL protection. This should be kept in mind, and from today's perspective, there is really no reason why exportable cipher suites should be supported in the first place. They are relics from the past.

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

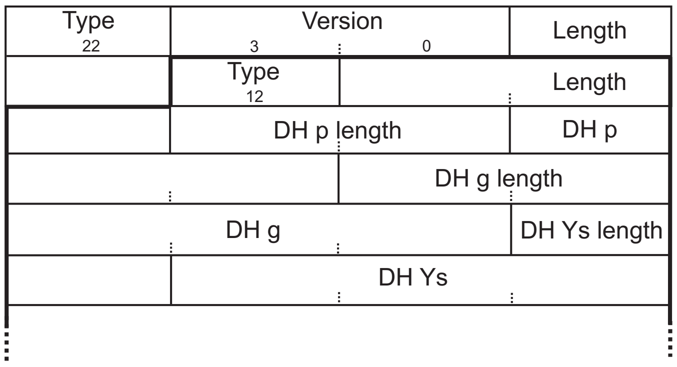

Figure 2.12 The beginning of an SSL SERVERKEYEXCHANGE message using Diffie-Hellman.

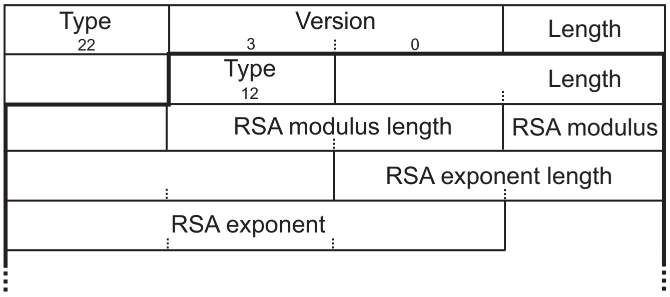

Figure 2.13 The beginning of an SSL SERVERKEYEXCHANGE message using RSA.

- 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, generator g, and public exponent Y_s, as well as a digital signature for the parameters. 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 RSA is used but the server has a signature-only RSA key, then the client cannot send a premaster secret encrypted with the server's public key. Instead, the server must create a temporary RSA public key pair and use the SERVERKEYEXCHANGE message to deliver the public key to the client. The

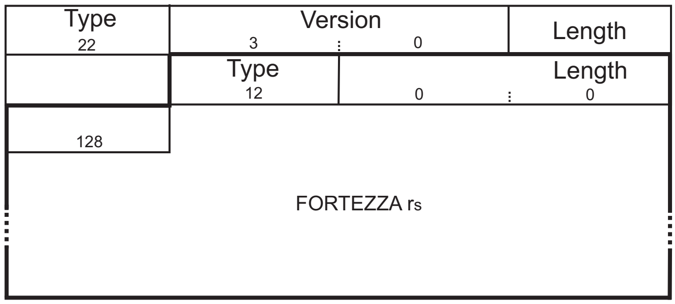

Figure 2.14 An SSL SERVERKEYEXCHANGE message using FORTEZZA.

SERVERKEYEXCHANGE message then includes the two parameters that together define a temporary RSA public key: the modulus and the exponent. Again, these parameters must come along with a digital signature. The beginning of such a message is illustrated in Figure 2.13 (without the signature part). Note again that the fields for the RSA parameters have a variable length (consistently set to three in Figure 2.13).

- If FORTEZZA is used, then the SERVERKEYEXCHANGE message only carries the server's r_s value that is required by the FORTEZZA KEA. 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.

In the first two cases, the SERVERKEYEXCHANGE message may include a signature part. If server authentication is not part of a particular SSL session, then no signature part is required, and the SERVERKEYEXCHANGE message ends with the Diffie-Hellman or RSA parameters. If the server is not acting anonymously and has sent a CERTIFICATE message, however, then the signed parameters format depends on the signature algorithm indicated in the server's certificate (RSA or DSA).

- If the server's certificate is for RSA signing, then the signed parameters consist of the concatenation of two hash values: an MD5 hash value and a SHA-1 hash value. Note that the two hash values are not individually signed, but one signature is generated for the combined hashes.
- If the server's certificate is for DSA signing, then the signed parameters consist solely of a SHA-1 hash value.

In either case, the input to the hash functions is a string that consists 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. (All components are concatenated.) The random values are included so that old signatures and temporary keys cannot be replayed. The server key parameters refer to either the Diffie-Hellman parameters of Figure 2.12 or the RSA parameters of Figure 2.13. As mentioned above, no signed parameters are included in a FORTEZZA key exchange.

2.2.2.6 CERTIFICATEREQUEST Message

A nonanonymous server may optionally authenticate a client[31] by sending a CERTIFICATEREQUEST message to it. This message not only asks the client to send a certificate (and sign data using its corresponding private signing key later on), but it also informs the client about what types of certificates (among the ones itemized in Table 2.5) it is going to accept.

As illustrated in Figure 2.15, an SSL CERTIFICATEREQUEST message starts with the usual 5-byte SSL record header, a type field value of value 13 (referring to a CERTIFICATEREQUEST message), and a 3-byte message length field. The actual body of the CERTIFICATEREQUEST 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 large, and hence the entire CERTIFICATEREQUEST message may be long.

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 type field value of value 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

31 Note that an anonymous server must not request a certificate from the client.

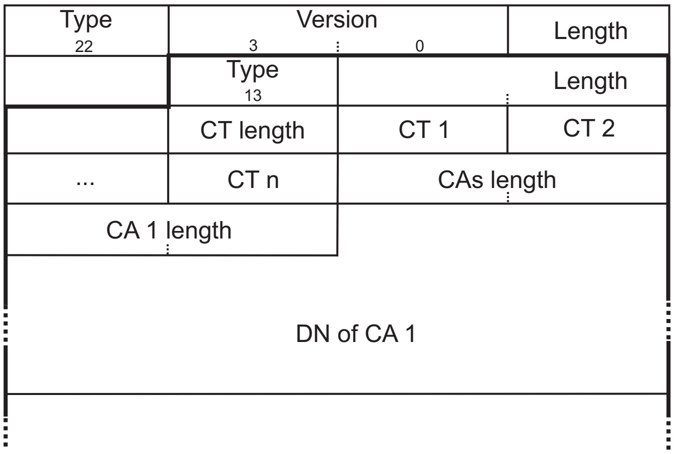

Figure 2.15 An SSL CERTIFICATEREQUEST message.

Table 2.5
SSL Certificate Type Values

Value	Name	Description
1	rsa_sign	RSA signing and key exchange
2	dss_sign	DSA signing only
3	rsa_fixed_dh	RSA signing with fixed DH key exchange
4	dss_fixed_dh	DSA signing with fixed DH key exchange
5	rsa_ephemeral_dh	RSA signing with ephemeral DH key exchange
6	dss_ephemeral_dh	DSA signing with ephemeral DH key exchange
20	fortezza_kea	FORTEZZA signing and key exchange

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 in the SERVERHELLO message are acceptable. If everything is fine, the client sends a couple of messages to the server as part of the third flight. If the server requested a certificate, then the client would first send a CERTIFICATE message to the server.

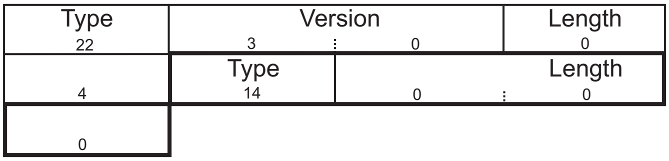

Figure 2.16 An SSL SERVERHELLODONE message.

This message would be structurally the same as the message sent from the server to the client (see Section 2.2.2.4). If the Diffie-Hellman key exchange algorithm is used, then the client-side Diffie-Hellman parameters must be compliant 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 an auxiliary message with regard to client authentication. The actual authentication takes place when the client sends a CERTIFICATEVERIFY message to the server (as addressed in Section 2.2.2.10).

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 depends on the key exchange algorithm in use. In either case, it starts with the usual 5-byte SSL record header, a type field value of value 16 (referring to a CLIENTKEYEXCHANGE message), and a 3-byte message length field. The body of the message then depends on the key exchange algorithm in use.

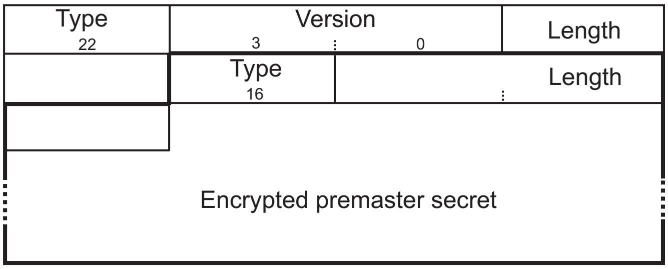

Figure 2.17 An SSL CLIENTKEYEXCHANGE message using RSA.

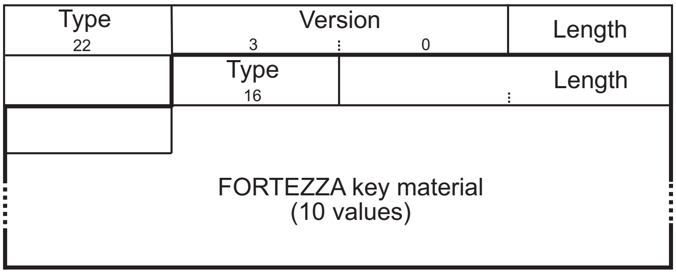

Figure 2.18 An SSL CLIENTKEYEXCHANGE message using FORTEZZA.

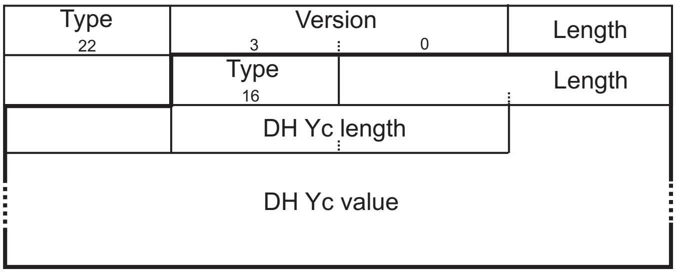

Figure 2.19 An SSL CLIENTKEYEXCHANGE message using Diffie-Hellman.

- 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`) that is sent from the client to the server. To detect version rollback attacks, the first 2 bytes from the 48 bytes refer to the latest (newest) version supported by the client and offered in the corresponding CLIENTHELLO message (note that this need not be the version that is actually in use[32]). 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 under the public RSA key from the server's certificate or temporary RSA key from the SERVERKEYEXCHANGE message. A respective SSL CLIENTKEYEXCHANGE message using RSA is illustrated in Figure 2.17.

32 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.

- In the case of FORTEZZA, the KEA is used to derive a 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 very important and should be taken with a grain of salt.

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 ephemeral or anonymous Diffie-Hellman is used, then the CLIENTKEYEXCHANGE message comprises the client's public Diffie-Hellman parameter Y_c. Such a message is illustrated in Figure 2.19. If, however, fixed Diffie-Hellman is used, then the client's public Diffie-Hellman parameters were already sent in a CERTIFICATE message, and hence a CLIENTKEYEXCHANGE message is not needed.

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.

2.2.2.10 CertificateVerify Message

If the client has provided a certificate with signing capabilities[33] in a Certificate message, then it must still prove that it possesses the corresponding private key. (The certificate alone does not provide client authenication, 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.

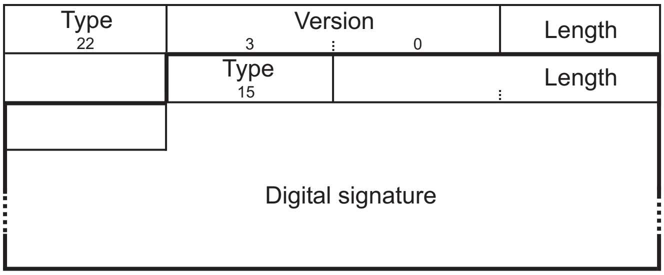

Figure 2.20 An SSL CertificateVerify message.

As illustrated in Figure 2.20, such an SSL CertificateVerify message starts with the usual 5-byte SSL record header, a type field value of value 15 (referring to a CertificateVerify message), and a 3-byte message length field. The body of the CertificateVerify message comprises a digital signature, where the exact format of the signature depends on whether the client's certificate is for RSA or DSA.

- For RSA certificates, two separate hash values are combined and signed: an MD5 hash value and a SHA-1 hash value. The signature covers both values (there are not two separate signatures).
- For DSA certificates, only a SHA-1 hash value is signed.

In either case, the information that serves as input to the hash functions is the same. If the term $handshake\_messages$ refers to the concatenation of all SSL handshake messages that have been exchanged so far, then the hash value is computed as follows:

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

33 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).

In this notation, h refers to MD5 or SHA-1, k to the master secret, and $ipad$ and $opad$ to the values introduced earlier in this chapter. Again, the two values are repeated 48 times for MD5 and 40 times for SHA-1. In either case, the resulting hash value is digitally signed using either RSA or DSA.

In case of an RSA certificate, the two hash values (one using MD5 and the other using SHA-1) are concatenated to form a 36-byte or 288-bit string, and this string is then digitally signed using the appropriate signing RSA key. In case of a DSA certificate, only one hash value is computed (using SHA-1), and this value is digitally signed using the appropriate signing DSA key. In either case, the server can verify the signature with the public key that can be extracted from the client's 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 discussed in 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 acknowledgment is required, meaning that the parties may begin sending encrypted data immediately after having sent a FINISHED message.

As illustrated in Figure 2.21, an SSL FINISHED message starts with the usual 5-byte SSL record header and then continues with a body part that is cryptographically protected, meaning that it is encrypted most of the time (depending on the cipher suite in use). The fact that the FINISHED message is encrypted is new in Figure 2.21. All previous figures showing SSL handshake messages have not shown the cryptographic protection, including the MAC (that—strictly speaking—does not belong to the handshake message). With this remark, we note that the encrypted body part comprises a header for the FINISHED message, with a type field value of value 20 (referring to a CERTIFICATEVERIFY message) and a 3-byte message length field, a 16-byte MD5 hash value, a 20-byte SHA-1 hash value, and a 16- or 20-byte MAC (the MAC length actually depends on the hash function in use). Both the MD5 and SHA-1 hash values depend on a key and actually refer to a MAC. These MACs, however, are distinct from the MAC that is computed for the SSL record and appended to the message. The MD5 and SHA-1 hash values 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[34] (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. Otherwise, if the server sends the message, then this value is 0x53525652. 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 indicated in Figure 2.21. Keep in mind, however, that these values comprise the cryptographic protection of the SSL record.

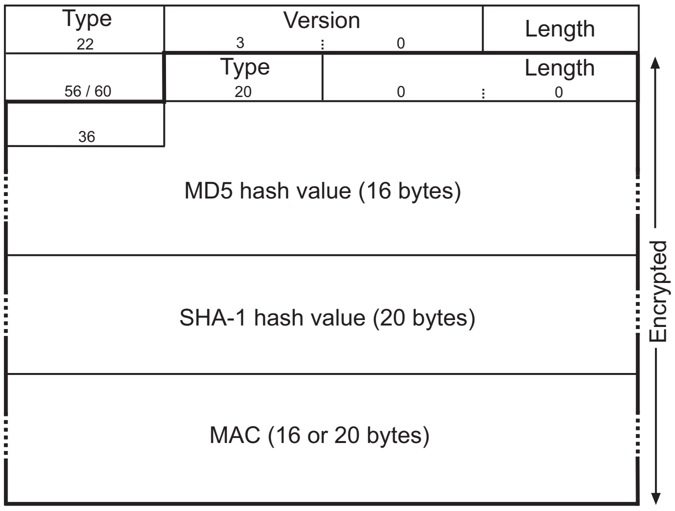

Figure 2.21 An SSL FINISHED message.

2.2.3 SSL Change Cipher Spec Protocol

As mentioned above, the SSL change cipher spec protocol is a protocol of its own that allows the communicating peers to signal transitions in ciphering strategies.

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

The protocol itself is very simple. It consists of a single message (i.e., a CHANGE-CIPHERSPEC message) that is compressed and encrypted according to the current (not pending) cipher spec. 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.

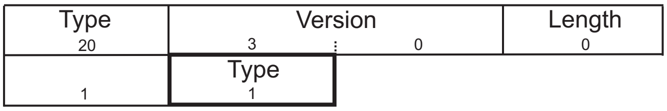

Figure 2.22 An SSL CHANGECIPHERSPEC message.

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 single type byte. This byte, in turn, is a placeholder that can currently only have a single value of 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 the SSL protocol implementation, and is therefore discussed controversially in the community.

2.2.4 SSL Alert Protocol

As mentioned above, the SSL alert protocol is another SSL (sub)protocol that allows the communicating peers to exchange alert messages. Each alert message carries an alert level and an alert description:

- The *alert level* comprises 1 byte, where the value 1 stands for “warning” and the value 2 stands for “fatal.” For all errors messages for which a particular alert 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 discretion whether to treat this as

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.

a fatal error. Anyway, all messages that are transmitted with an alert level of fatal must be treated appropriately, and this means that they must result in the immediate termination of the connection.

- The *alert description* also comprises 1 byte, where a numeric code refers to a specific situation. The alert messages, codes, and brief descriptions of the SSL protocol specification are summarized in Table 2.7. For example, code 0 stands for the closure alert `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.

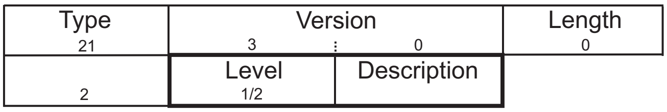

Figure 2.23 An SSL ALERT message.

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 the alert level and the other byte referring to the alert description code. So both the SSL change cipher spec and the SSL alert protocols are very simple.

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, cryptographic protection, and SSL record encapsulation. The resulting SSL records are sent to the recipient, where the application data is decrypted, verified, decompressed, and reassembled.

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 type field (this time referring to 23 standing for the SSL application data protocol), a version field, and a length field. Everything after the SSL record header is encrypted and can only be decrypted using the appropriate key. This applies to the actual application data, but it also applies to the MAC (that is either 16 or 20 bytes long). Remember that the MAC is appended to the application data prior to encryption (that may or may not invoke padding).

This way of encapsulating application data in an SSL record is simple and straightforward for a stream cipher. It is slightly more involved, when a block cipher

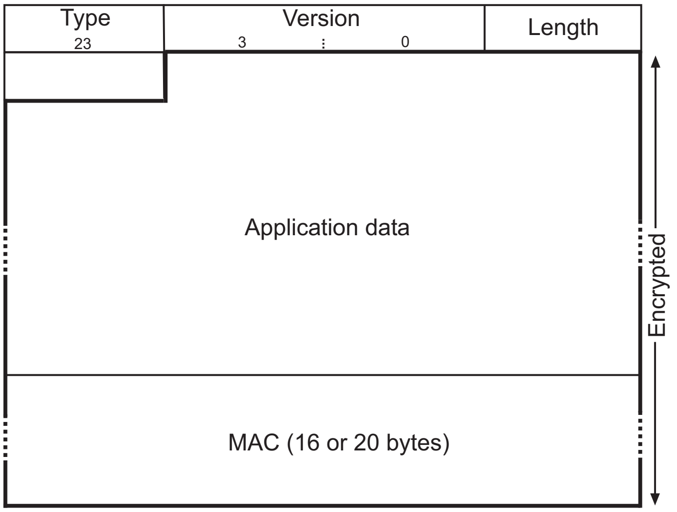

Figure 2.24 Application data encapsulated in an SSL record (stream cipher).

is used for encryption. In this case, some message padding must be appended to the SSL record prior to encryption, and the last byte in the record must then refer to the padding length. The corresponding format of an SSL record for a block cipher is illustrated in Figure 2.25. We revisit the notion of padding when we address padding oracle attacks against SSL/TLS later in this chapter. (You may also refer to Appendix B for a comprehensive treatment of padding oracle attacks.)

2.3 PROTOCOL EXECUTION TRANSCRIPT

To illustrate the working principles and functioning 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 is well suited to show 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 exist between the client and server.

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

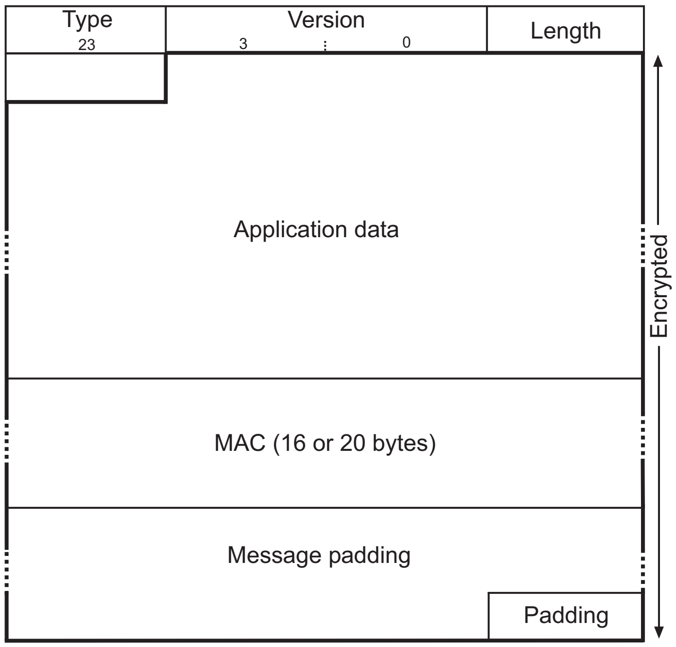

Figure 2.25 Application data encapsulated in an SSL record (block cipher).

```
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 following 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. 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. 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 then merged into a single SSL record and transmitted in a single TCP segment to the client. In our example, such an SSL record comprsies a SERVERHELLO, a CERTIFICATE, and a SERVERHELLODONE message. The corresponding SSL record starts with the following byte sequence:

```
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 above-mentioned 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 is TLS_RSA_WITH_RC4_128_SHA), 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 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 (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 4 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. It 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 in our example). 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 comprises a data fragment.

2.4 SECURITY ANALYSES AND ATTACKS

As a result of the SSL protocol's success, many researchers have investigated its security in the past two decades. 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 bit generator (and hence the method to generate the premaster secret) was cryptographically weak,[35] meaning that the premaster secrets that were generated could be predicted 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 a problem of how the protocol was implemented by Netscape Communications.[36] Anyway, the resulting vulnerability became a press headline and casted a damning light on the security of the then-evolving SSL protocol. The problem could easily be fixed by strengthening the pseudorandom bit generator in the browser. This was quickly done by Netscape Communications, but the story still illustrated the well-known fact that even a secure protocol can be implemented in a rather insecure way.

Later in 1996, Wagner and Bruce Schneier were the first who did a (yet informal) security analysis of the SSL protocol versions 2 and 3 [15]. As a result of their analysis, they reported a few weaknesses and possibilities to mount active attacks against SSL 2.0. For example, the MAC did not protect the padding length field, and this could, in principle, be exploited in an attack against the integrity of the protected data. Also, the lack of authenticity and integrity protection of the cipher suites included in a CLIENTHELLO message could be exploited in a cipher suite rollback attack, in which the adversary modifies the CLIENTHELLO message to only include weak cipher suites. The server then has no other choice than to reject the connection or accept a weak cipher suite. These and some other weaknesses finally led the IETF to publish a Standards Track RFC in 2011 to prohibit the use of SSL 2.0

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

36 More recently, a similar problem was reported for a pseudorandom bit generator employed by the OpenSSL library for Debian (https://www.debian.org/security/2008/dsa-1571).

entirely [16]. Due to ongoing competition between Netscape Communications and Microsoft in these early days of SSL (cf. Section 1.2), most of the weaknesses of SSL 2.0 had already been corrected in SSL 3.0. So when Wagner and Schneier did their analysis in 1996, they found only a few attacks that could still be mounted against SSL 3.0, such as a key exchange algorithm confusion or a version rollback attack. More recently, it has, for example, been shown that SSL 3.0 and all versions of TLS are still vulnerable against certain types of version rollback attacks, and attacks like FREAK and Logjam have gained a lot of media attention (cf. Section 3.8.4). In their 1996 security assessment, Wagner and Schneier concluded that the remaining attacks against SSL were not devastating and hence that the overall security was still positive. In fact, they wrote that "on the whole SSL 3.0 is a valuable contribution towards practical communications security" [15].

In the aftermath of the Wagner-Schneier analysis, a few other researchers tried to do more formal analyses by applying formal methods for the security evaluation of SSL 3.0 [17, 18]. 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. However, every theoretical result is only as good as its underlying model, and this is where things get more involved.

In practice, a secure protocol can be implemented insecurely or at least provide some hooks from where a cryptanalytical attack can start. In 1998, for example, Daniel Bleichenbacher found such a hook in the way the premaster secret is padded before it is RSA encrypted in a CLIENTKEYEXCHANGE message [19]. According to the SSL 3.0 protocol specification, this padding and encryption must conform to PKCS #1 version 1.5 [6]. Bleichenbacher found a way to mount an adaptive chosen ciphertext attack (CCA2[37]) against the PKCS #1 padding and encoding of the premaster secret (so the attack is basically more an attack against PKCS #1 than an attack against the SSL protocol). If the adversary is successful and can decrypt the premaster secret, then he or she can derive all keying material required to break the security of the SSL session in question. The resulting *Bleichenbacher attack* was the first example of a so-called *padding oracle attack* (i.e., an attack 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 about this ciphertext, namely whether the plaintext that results from it after decryption is properly padded or not. One problem of the Bleichenbacher attack is that the padding oracle reveals only one bit of information in every request. This means that an adversary requires a huge quantity (i.e., roughly one million) of oracle queries, and the Bleichenbacher attack is therefore also known as the *million*

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

message attack. Because a huge quantity of oracle queries is required, the attack can usually be detected quite easily in an online setting. So detection seems to be simple, but prevention is not. To prevent the attack, it is required that a server does not leak any information about the correctness of the padding (including any timing information that correlates with the padding). The simplest way is to treat mispadded messages as if they were properly padded (e.g., [20]). So when a server receives a CLIENTKEYEXCHANGE message with a flawed padding, it does not return an error message but randomly generates a premaster secret and continues the protocol as usual. The client and server then end up with cryptographic keys that are distinct, and hence the SSL session cannot be established, but no other information about the padding is leaked. The technical details of padding oracle attacks in general, and the Bleichenbacher or million message attack in particular, are explained in Appendix B. Here, we only want to discuss the following two implications of the attack:

- First, the attack clearly demonstrated that CCAs in general, and CCA2s in particular, are practical and something to consider seriously. Before the attack was published, it was often argued that such attacks are purely theoretical and cannot 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 and to come up with a new padding scheme. In the same year as the attack was published, a technique known as *optimal asymmetric encryption padding* (OAEP) [21] was adopted in PKCS #1 version 2.0 [7]. Unlike ad hoc padding schemes, such as the one employed in PKCS #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 prior to encryption (so RSA-OAEP is the acronym of the resulting encryption system). 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 cryptographic community. There are asymmetric encryption systems that are provably secure against CCA2 in the standard model, such as a system proposed by Ronald Cramer and Victor Shoup [22], but the use of such systems has disadvantages and is avoided in practice. This may change one day, but as of this writing OAEP is much more widely deployed. It is highly efficient and provides a reasonable level of security and resistance against CCA2.

In the aftermath of the Bleichenbacher attack, many researchers tried to find optimizations, variations, and extensions of the attack. For example, in 2001, James Manger found a timing channel that can be used to mount a CCA against some implementations of PKCS #1 version 2.0 [23]. At first sight, this sounds impossible, because RSA-OAEP is known to be secure against CCAs—at least in the random oracle model. However, once again, we have to say that a security proof in theory does not mean that the respective system (or an implementation thereof) cannot be attacked in practice. So PKCS #1 had to be updated again (to reduce the likelihood that a Manger attack can be successfully mounted), and this was done in PKCS #1 version 2.1, which was released in 2003 [8]. In 2012, PKCS #1 was updated once again (in version 2.2), but this update has no impact on SSL/TLS security. Similar to Manger, three Czech cryptologists—Vlastimil Klíma, Ondrej Pokorný, and Tomás Rosa—found a variation of the Bleichenbacher attack that 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 in 2003 [24]. Also, there are a few Bleichenbacher-related side-channel attacks that have been developed and proposed in the past decade. Some of these attacks can even be mounted remotely against network servers (e.g., [25, 26][38]). Implementing public key cryptography and protecting these 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 optimizations, variations, and extensions, some researchers have found other (mostly subtle) security problems in block ciphers operated in CBC mode. Most importantly, Serge Vaudenay published a paper in 2002, in which he explained how CBC padding may induce a side channel that can be exploited in a CCA [27]. This publication did not come along with a feasible attack. But only one year later, in 2003, Vaudenay et al. published a follow-up paper in which they showed that the CBC padding problem can actually be turned into a feasible attack [28]. The attack is known as *Vaudenay attack*, and it is explained in detail in Appendix B.2. As the attack was published after the official release of TLS 1.0, SSL 3.0 has no built-in countermeasures in place. Such countermeasures had to be designed and retrofitted into TLS 1.1 after the publications of Vaudenay. The same is true for another vulnerability that affects block ciphers operated in CBC mode that was found by Gregory Bard in 2004. To some extent, this vulnerability is worse, because it can be exploited in a blockwise chosen-plaintext attack (CPA) [29]. Note that a CPA is generally much simpler to mount than a CCA, especially in the realm of public key cryptography where public keys are available per se. Both vulnerabilities are addressed in TLS 1.1, and hence their detailed description is postponed to Chapter 3.

Here, we only address a subtlety of SSL that can be exploited in a very efficient variation of the Vaudenay attack. While TLS employs PKCS #7 padding, the padding scheme employed by SSL is much simpler. PKCS #7 requires a byte

38 The vulnerability that enables [25] is documented in CVE-2003-0147.

representing the padding length (PL) that is repeatedly written, but SSL only requires the last byte to refer to the padding length. All other padding bytes are random bytes (RBs), and hence they cannot be checked for authenticity and integrity (also because the padding is not part of the MAC). This means that SSL padding is not deterministic, and this simplifies padding oracle attacks considerably.

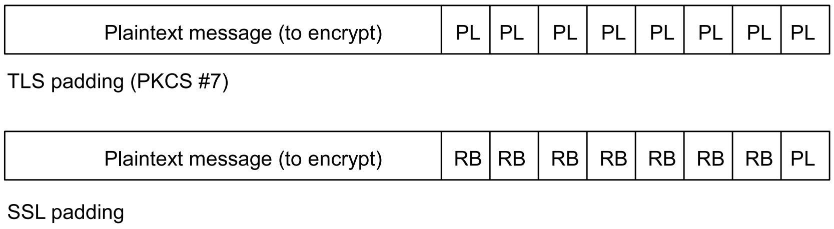

Figure 2.26 TLS and SSL padding.

The two padding schemes are illustrated in Figure 2.26. Note that all PL bytes refer to the same byte value, whereas all RB bytes can be different and arbitrary. As shown by Bodo Möller, Thai Duong, and Krzysztof Kotowicz, the SSL padding format is highly vulnerable to a specific type of padding oracle attack known as *padding oracle downgraded legacy encryption* (POODLE) attack.[39] The vulnerability that is exploited by the POODLE attack is documented in CVE-2014-3566, whereas a related F5 Networks' implementation bug is documented in CVE-2014-8730. The POODLE attack is devastating and has brought SSL to the end of its life cycle (this is why we discuss it here and not in the following chapter). Today, it is generally recommended to disable all versions of SSL and to no longer support it [30]. This also defeats all types of protocol downgrade attacks that may otherwise still be feasible.

To explain the POODLE attack, let us assume an adversary who has eavesdropped a sequence $C_1, \ldots, C_n$ of ciphertext blocks that are encrypted with a block cipher in CBC mode (with block length k and some initialization vector C_0). For the sake of simplicity, we assume that an entire block of padding, meaning that the last byte of C_n, i.e., $C_n[k-1]$, comprises the value $k-1$, whereas all other bytes of C_n (i.e., $C_n[0]$, $C_n[1]$, …, $C_n[k-2]$) may comprise random values. In Figure 2.26 this means that PL is $k-1$, whereas all other RB values are arbitrary. In this starting position, the adversary knows that ciphertext block C_i is the target of the attack. This block may comprise a bearer token, such as an HTTP cookie, an HTTP

39 https://www.openssl.org/$\sim$bodo/ssl-poodle.pdf.

authorization header, or something similar. In either case, the adversary's goal is to decrypt and possibly reuse the information carried in C_i.

Under normal circumstances, the recipient uses K and C_0 to decrypt the sequence of ciphertext blocks $C_1, \ldots, C_n$ and get the corresponding plaintext blocks. Using CBC mode of operation, the formula to decrypt ciphertext block C_i (for $i = 1, \ldots, n$) is as follows:

$$P_i = D_K(C_i) \oplus C_{i-1} \tag{2.1}$$

After decrypting the entire sequence of ciphertext blocks, the recipient checks and removes the padding from the last block, and finally verifies the MAC. If everything is fine, then the plaintext blocks are handed over to the appropriate application process for further processing.

Now see what happens if the adversary replaces the ciphertext block C_n with C_i, and hence constructs a sequence of blocks that looks as follows (the now repeated block C_i is underlined):

$$C_1, C_2, \ldots, C_{i-1}, \underline{C_i}, C_{i+1}, \ldots, C_{n-1}, \underline{C_i}$$

When the recipient decrypts this sequence, everything works normally until the last block C_i (instead of C_n). Using (2.1), this block is decrypted to $P_i = D_K(C_i) \oplus C_{n-1}$. When the recipient checks the padding, he or she actually verifies whether the last byte of P_i (i.e., $P_i[k-1]$) is equal to $k-1$. Because all byte values are equally probable, the probability that this is true (i.e., $P_i[k-1] = k-1$) is only $1/256$. With a much larger probability of $255/256$, $P_i[k-1]$ is not equal to $k-1$, meaning that the block is not properly padded. In this case, the MAC is taken from a wrong position, and hence the subsequent MAC verification step is very likely going to fail. This failure, in turn, can be detected (by observing a respective alert message or measuring the timing behavior) and exploited in an attack.

To mount the attack, the adversary must compromise the victim's browser and be able to send arbitrarily crafted HTTPS (request) messages to the target site. In particular, this means that the adversary must be able to ensure that all messages that are sent to the target site are exactly as long as an entire block of padding is required (this block then represents C_n), and that the first byte to be determined appears as the last byte of some earlier block (this block then represents C_i). The adversary then replaces C_n with C_i and sends the respective sequence of ciphertext blocks (in a modified SSL record) to the target site. Following the line of argumentation given above, the server rejects the record (because its MAC cannot be verified) with a probability of $255/256$. However, the adversary can now start modifying the value of $C_i[k-1]$, until he or she has found the correct value. In the worst case, he or she has to try out all 256 possible values. On the average, he or she will be done in $256/2 = 128$ tries.

The interesting question is what the adversary has achieved when he or she has found the correct value for $C_i[k-1]$. In this case, we know that this value correctly decrypts to $k-1$ (which is the proper padding length value). This means that

$$D_K(C_i)[k-1] \oplus C_{n-1}[k-1] = k-1$$

[since the CBC decryption formula (2.1) also applies on the byte level (i.e., $P_i[k-1] = D_K(C_i)[k-1] \oplus C_{n-1}[k-1]$)]. If we replace $D_K(C_i)[k-1]$ with $P_i[k-1] \oplus C_{i-1}[k-1]$ (what it is actually standing for), then we get

$$P_i[k-1] \oplus C_{i-1}[k-1] \oplus C_{n-1}[k-1] = k-1$$

This equation can finally 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 has determined $P_i[k-1]$. This is the first previously unknown byte that is now decrypted. He or she can proceed to the next byte $P_i[k-2]$, for example, by changing the lengths of the components of the respective HTTPS (request) messages, and so on and so forth. For every byte to be determined, the attack requires 128 (on the average) or 256 (in the worst case) messages to be sent to the target site. So the attack is very efficient, and this means that block ciphers in CBC mode should no longer be used in SSL.

Instead of using a block cipher in CBC mode, one can always use a stream cipher like RC4. In fact, it has been frequently recommended to use RC4 to mitigate all attacks against SSL that have become public in the past. Unfortunately, this is not a particularly good idea, because RC4 has security problems of its own [31–33].[40] 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. Against SSL/TLS, this means attacking many connections collectively and simultaneously. In fact, we are talking about millions of messages that must be sent to a server, a lot of bandwidth, and a lot of time. This is

40 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.

not trivial, and the resulting attacks are highly involved and somehow academic. Keep in mind, however, that attacks always get better and that the most recent results are on the verge of becoming practically exploitable [33]. This does not mean that it can be done in real time. In November 2013, Jacob Appelbaum, an independent security researcher, nevertheless claimed that the NSA can break RC4 in real time. If this were true, then RC4 would have more serious weaknesses than the statistical ones (that do not allow the breaking of RC4 in real time). However, due to Appelbaum's statement, many people have disabled RC4 in their preferred cipher suites, and this advice has also been adopted in respective recommendations issued by software vendors, such as Microsoft,[41] and standardization bodies, such as the IETF [34] and the NIST [35].

More recently, a group of researchers has shown that padding oracle attacks (of the Bleichenbacher or Vaudenay type) can also be mounted against cryptographic 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 widely used in the field [36, 37]. Since then, the cryptanalytical power of padding oracle attacks has been widely recognized, and designing and coming up with implementations that are resistant against such attacks has become an important goal in research and development. The story is very likely to be continued here.

Due to the POODLE attack and the (known) vulnerabilities of RC4, it is generally recommended today to disable SSL 3.0 altogether [30]. If this is not possible, then it should at least be ensured that any protocol downgrade only happens with the client's blessing. This is where the term TLS_FALLBACK_SCSV comes into play. The acronym SCSV stands for signaling cipher suite value, so an SCSV actually refers to a "dummy cipher suite." It may be included in the list of supported cipher suites to allow a client to signal some information to the server.[42] 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 supports (because the last one failed for some reason). In either case, it is up to the server to decide whether the fallback is appropriate and acceptable. Otherwise (i.e., if the fallback is not appropriate), the server must abort the connection and yield a fatal error. The specification of the TLS_FALLBACK_SCSV for preventing protocol downgrade attacks is a work

41 http://blogs.technet.com/b/srd/archive/2013/11/12/security-advisory-2868725-recommendation-to-disable-rc4.aspx.

42 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).

in progress within the IETF TLS WG.[43] A respective summary is provided, for example, in Chapter 6 of [38].

2.5 FINAL REMARKS

This chapter introduced, overviewed, and detailed the SSL protocol. This will help us to better understand the TLS protocol 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 PFS.) There are only a few details that can be discussed controversially, such as the use of a separate content type for CHANGECIPHERSPEC messages, and these details may even change in the future. 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 good. During the first decade, no serious vulnerability was found. This only changed, when Bleichenbacher, Vaudenay, and Bard published their results, and—even more—when some attacks and attack tools that exploited the vulnerabilities they found were presented at security conferences worldwide. Most importantly, the POODLE attack that was announced in 2014 brought the SSL protocol to the end of its life cycle. In fact, the IETF deprecated SSL 3.0 in June 2015 [30] and has since then recommended its replacement with more recent versions of the TLS protocol. The browser vendors slowly followed this recommendation and disabled SSL 3.0 by default. If a browser still supports SSL 3.0, it is almost always possible to disable it manually in the configuration settings. The details depend on the browser in use and are not addressed here.

Like any other security technology, the SSL protocol has a few disadvantages and pitfalls (some of which also apply to the TLS protocol). 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. This is because the content screener only “sees” encrypted data in which it cannot efficiently find malicious content. 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 (see 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 before, 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,

43 https://tools.ietf.org/html/draft-ietf-tls-downgrade-scsv-03.

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

References

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

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

[3] Rescorla, E., “HTTP Over TLS,” Informational RFC 2818, May 2000.

[4] Hoffman, P., “SMTP Service Extension for Secure SMTP over TLS,” Standards Track RFC 2487, January 1999.

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

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

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

[8] Jonsson, J., and B. Kaliski, “Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1,” Informational Request for Comments 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 ’01*, Springer-Verlag, LNCS 2139, 2001, pp. 310–331.

[11] Mavrogiannopoulos, N., et al., “A Cross-Protocol Attack on the TLS Protocol,” *Proceedings of the ACM Conference in Computer and Communications Security,* ACM Press, New York, NY, 2012, pp. 62–72.

[12] 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.

[13] 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.

[14] Krawczyk, H., M. Bellare, and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” Informational RFC 2104, February 1997.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

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

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

[31] 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.

[32] Gupta, S., et al., “(Non-)Random Sequences from (Non-)Random Permutations—Analysis of RC4 Stream Cipher,” *Journal of Cryptology,* Vol 27, 2014, pp. 67–108.

[33] 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, August 2015.

[34] Popov, A., “Prohibiting RC4 Cipher Suites,” Standards Track RFC 7465, February 2015.

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

[36] 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.

[37] Bardou, R., et al., “Efficient Padding Oracle Attacks on Cryptographic Hardware,” Cryptology ePrint Archive, Report 2012/417, 2012.

[38] 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.

Chapter 3

TLS Protocol

This chapter introduces, discusses, and puts into perspective the second transport layer security protocol[1] mentioned in the title of the book—the TLS protocol. We assume the reader to be familiar with the SSL protocol (as outlined in Chapter 2), and we confine ourselves to the differences between the SSL protocol and the various versions of the TLS protocol (see Section 1.2 for a historical outline of the TLS protocol evolution, comprising versions 1.0 to 1.3 [1–4]) . More specifically, we provide an introduction in Section 3.1, focus on the various versions of the TLS protocol in Sections 3.2–3.5, elaborate on HTTP strict transport security (HSTS) in Section 3.6, go through a TLS protocol execution transcript in Section 3.7, analyze the security of the TLS protocol and overview the relevant attacks in Section 3.8, and conclude with some final remarks in Section 3.9. Mainly due to the richness of the TLS extensions and the many relevant attacks that can be mounted against the TLS protocol, this chapter is quite long.

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 protocols as the SSL protocol (with the only difference that the prefix “SSL” in the protocols’ names is replaced with “TLS”):

1 As mentioned in Section 1.2, 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, each of these data structures comprises a one-byte *type* field, a two-byte *version* field, another two-byte *length* field, and a variable-length (up to 2^{14}=16,384 bytes) *fragment* field. The type, version, and length fields represent the TLS record header, whereas the fragment field represents the payload of a 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);
  - The *TLS alert protocol* (21);
  - The *TLS handshake protocol* (22);
  - The *TLS application data protocol* (23).

  Each protocol is identified with a unique content type value that is appended in brackets (i.e., 20, 21, 22, and 23). To allow future extensions, additional record types may be defined and supported by the TLS record protocol. As further addressed in Section 3.4.1.13, such an additional record type has, for example, been assigned to the Heartbeat extension (with a content type value of 24). The content type is the value that is written into the type field of the TLS record header, so each record can carry arbitrarily many messages of a particular protocol.

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

Like the SSL protocol, the TLS protocol employs sessions and connections, where multiple connections may refer to a single 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, while the security parameters and elements for the pending states are negotiated and set during the execution of the TLS handshake protocol. This also means that the SSL state machine illustrated in Figure 2.2 also applies to the TLS protocol.

The state elements of a TLS session are essentially the same as the state elements of an SSL session (cf. Table 2.2), so we don't have to 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 summarized in Table 3.1 and the state elements summarized in Table 3.2, the SSL protocol does not make this distinction and only considers state elements (cf. Table 2.3). However, taking the security parameters of Table 3.1 and the state elements of Table 3.2 together, the differences between SSL and TLS connections are rather small and irrelevant. In addition to the security parameters summarized in Table 3.1, the PRF algorithm is yet another security parameter that has been introduced in TLS 1.2 [3].[2] It is not shown in Table 3.1.

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

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
`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)

A major difference between the SSL protocol and the TLS protocol lies in the way the keying material that is needed is actually generated. In Section 2.1, we have seen that SSL uses an ad hoc and somehow handcrafted construction to generate the master secret and the key block (from which the keying material is taken). 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 on how the

2 The PRF algorithm has to be specified in TLS 1.2, because prior versions of the TLS protocol use a different PRF.

TLS protocol generates the keying material that is needed. There are some subtle differences between the TLS PRF used in versions 1.0 and 1.1 and the TLS PRF used in versions 1.2 and 1.3. We start with the former and postpone the discussion of the latter to Section 3.4.

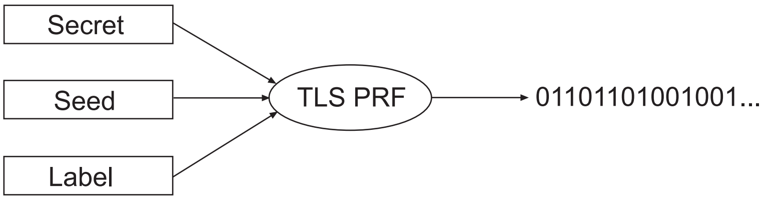

Figure 3.1 Overview of the TLS PRF.

3.1.1 TLS PRF

The TLS PRF is overviewed in Figure 3.1 (from a bird's eye perspective). The function takes as input a secret, a seed, and a label—sometimes termed *identifying label*—and it generates as output an arbitrarily long bit sequence. To make the TLS PRF as secure as possible, the TLS PRF combines the two cryptographic hash functions MD5 and SHA-1. The idea is that the resulting PRF should be secure as long as at least one of the underlying hash functions remains secure. This is actually a good idea that is often applied in cryptography. The combined use of MD5 and SHA-1 is true for TLS versions 1.0 and 1.1, but it is no longer true for TLS versions 1.2 and 1.3 (these versions use the supposedly stronger hash function SHA-256).

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 MD5 or SHA-1 in TLS 1.0 and 1.1, and SHA-256 in TLS 1.2 and 1.3) to expand a secret and a seed into an arbitrarily long output value. In particular, 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))
```

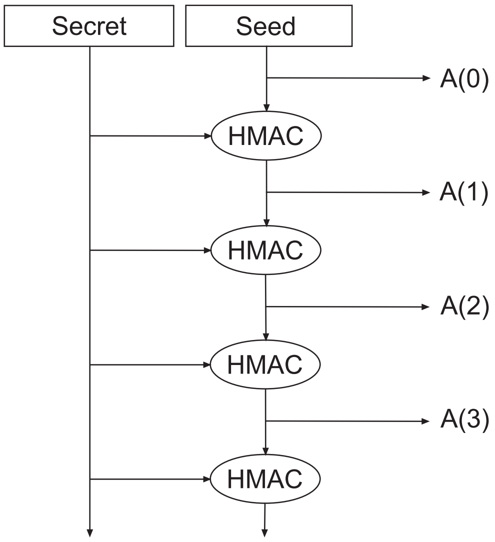

Figure 3.2 The A-function of the TLS PRF.

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

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.

Putting everything together, we are ready to explain the TLS PRF (as used for TLS 1.0 and 1.1). As illustrated in Figure 3.3, the secret is 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.[3] 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 subject to a bitwise addition modulo 2 (XOR). This construction 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

3 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.

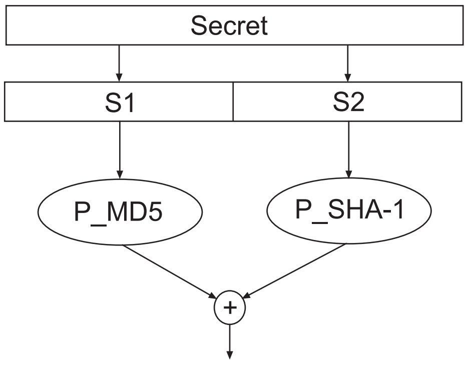

Figure 3.3 The internal structure of the TLS PRF (as used for TLS 1.0 and TLS 1.1).

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 and 1.3 employs the same expansion function `P_hash(secret,seed)` but a single hash function 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.

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 some TLS connection state). The construction is as follows:[4]

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

4 In Section 3.4.1.19, we will see that there is a new way of generating a master secret that protects against an attack known as the *triple handshake attack*. To make the distinction clear, the respective master secret is then called *extended master secret*.

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` in 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 all 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 generation 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 the following values by splitting it into blocks of appropriate sizes:

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

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 (or 104 bytes, respectively). Different cipher suites may have different requirements regarding the length of the key block that needs to be generated.

In former times (when the U.S. export controls were still in place), people were used to invoke 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`. Without further explanation, we note that the respective constructions were as follows:

```
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 the string "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 (cf. Section 3.8.4), 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 important. We simply refer to the TLS 1.0 protocol specification [1] for an example of how to actually generate the keying material for RC2 with a 40-bit key. (That is an example of an exportable block cipher.)

Sometimes, people use applications on top of TLS (or DTLS) that require keying material for their own purposes. These applications then need a mechanism to import keying material from TLS (or DTLS) and to securely agree on the context where this *exported keying material* (EKM) can be used afterward. Such a mechanism and respective keying material exporters are specified in standards track RFC 5705 [5]. Without going into the details, we mention that the construction of the EKM is conceptually similar to the construction of the key block (given above) and that it uses the same TLS PRF.

In addition to the TLS PRF and the generation of the keying material, 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 in Sections 3.2–3.5 (which address the four currently available versions of the TLS protocol). For each difference, we give some background information and the rationale that has led to the particular design. We begin with TLS version 1.0.

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 be also viewed as SSL version 3.1. This viewpoint is also 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 already suggests that TLS 1.0 is conceptually the same as 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 the keying material. Also, the TLS protocol distinguishes between security parameters and state elements for TLS connections, whereas the SSL protocol only considers state elements. In addition to these obvious differences, there are a few differences that are more subtle and require some further explanation. These differences refer to the cipher suites, certificate management, alert messages, and some other differences, addressed in Sections 3.2.1–3.2.4.

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 (cf. Table 2.4). However, the following three cipher suites, which employ FORTEZZA, are no longer supported and have no counterpart in TLS 1.0.[5]

- 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 names 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

5 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.

substantial change with regard to its meaning. However, there are still a few subtle changes that need to be mentioned here. They refer to the way messages are authenticated and encrypted. With the exception of FORTEZZA no longer being supported, the key exchange algorithms have not changed and remain the same. Accordingly, they are not further addressed here.

3.2.1.1 Message Authentication

The MAC construction employed by the SSL protocol (as outlined in 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 concatentation of the 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 actually generated.[6] 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 types of replay attacks.

6 The fact that 13 bytes are prepended is exploited in a recent padding oracle attack against the TLS protocol employing a block cipher in CBC mode that is known as *Lucky 13*. We revisit this topic toward the end of this chapter.

3.2.1.2 Data Encryption

SSL 3.0 was specified prior to the enactment of the new 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,[7] and hence 3DES represented the strongest possible 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 15 years after its promotion in TLS 1.0. As explained below, the only problem is the CBC mode of encryption that is susceptible to several padding oracle attacks.

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 have this requirement. 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 basic protection against simple traffic analysis attacks. However, keep in mind that—from today's perspective—the use of CBC mode is no longer recommended and that there are better modes of operation for a block cipher.

In addition to the move from DES to 3DES, a complementary RFC 4132 was first released in 2005[8] that proposed a couple of cipher suites that comprise the Camellia block cipher.[9] In 2010, RFC 4132 was updated in standards track RFC 5932 [6], and the list of cipher suites was expanded to also use SHA-256 (in addition to SHA-1). The respective cipher suites together with their reference code values are summarized in Table 3.3. 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

7 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.

8 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.

9 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 more recently (see Section 3.9).

Table 3.3
The Camellia-Based Cipher Suites for TLS (According to [6])

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` }

can be applied to all versions of the TLS protocol (it is even complemented by an informational RFC 6367 that introduces new cipher suites that 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 strong reason or incentive to replace AES-based cipher suites with Camellia-based ones (unless non-U.S. ciphers are preferred). The same line of argumentation also applies to the ARIA-based cipher suites mainly used in Korea.

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:

Table 3.4
TLS 1.0 Certificate Type Values

Value	Name	Description
1	rsa_sign	RSA signing
2	dss_sign	DSA signing
3	rsa_fixed_dh	RSA signing with fixed Diffie-Hellman key exchange
4	dss_fixed_dh	DSA signing with fixed Diffie-Hellman key exchange

- 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 that is trusted). 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 summarized in Table 2.5), SSL 3.0 supports many certificate types. This is no longer true for TLS 1.0. As summarized in Table 3.4, TLS 1.0 only supports the first four certificate types itemized in Table 2.5, namely RSA (1) and DSA signing (2), RSA signing with a fixed Diffie-Hellman key exchange (3), and DSA signing with a fixed Diffie-Hellman key exchange (4). The numbers in brackets refer to the respective type values. So TLS 1.0 no longer supports RSA signing with ephemeral Diffie-Hellman key exchange (5), DSA signing with ephemeral Diffie-Hellman key exchange (6), and FORTEZZA signing and key exchange (20). The first two certificate types (5 and 6) are not really needed, because a certificate that can be used to generate (RSA or DSA) signatures can also be used to sign ephemeral Diffie-Hellman keys. Also, the last certificate type (20) is not needed anymore, because the FORTEZZA-type cipher suites have been removed from TLS 1.0 entirely.

Later in this chapter, we will see that the certificate types that are missing in TLS 1.0 have been reintroduced in TLS 1.1 as reserved values. So the question of what certificate types must be supported heavily depends on the version of the TLS protocol.

Table 3.5
TLS 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.

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).[10] Due to its reserved status, it is not included in Tables 3.5 and 3.6.

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

Table 3.6
TLS 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.

3.2.4 Other Differences

The CERTIFICATEVERIFY and FINISHED messages of SSL 3.0 are explained in Sections 2.2.2.10 and 2.2.2.11. Because the respective constructions are involved, they were simplified in TLS 1.0 and brought in line with the TLS PRF construction where possible and appropriate.

- The body of the CERTIFICATEVERIFY message comprises a digital signature for the concatenated handshake messages that have been exchanged so far. The signing key is the one that corresponds to the public key that is found in the client's certificate.
- The body of the FINISHED message comprises 12 bytes (96 bits) of data that are 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[11] 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." So the construction of the data can formally be expressed as follows:

```
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 to come up with something that is inherently more difficult to compromise than if only one hash function were used. The resulting 12 bytes actually 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 distinct (because the two labels are distinct).

This finishes our exposition of the differences between SSL 3.0 and TLS 1.0. We are now ready to go one step further in the evolution of the TLS protocol and to address 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 officially released [2]. Following the notation of SSL 3.0 and TLS 1.0, the official version number of TLS 1.1 is 0x0302, where 0x03 refers to the major version 3 and 0x02 refers to the minor version 2. 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 that have been exploited by specific attacks against block ciphers operated in CBC mode. The subtleties and the respective protocol changes are explained below. Before that, we want to emphasize that TLS 1.1 also introduced a new way of specifying parameters and parameter values. Instead of incorporating

11 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.

them into the protocol specification of the respective RFC document, the IANA[12] 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.[13] 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 (BCP 26) [7], 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 by specification required, and yet another block for parameter values assigned by private use.

Before we delve into the differences between TLS 1.0 and TLS 1.1, we start with the cryptographic subtleties mentioned above. Remember that these subtleties are necessary to understand the protocol changes made in TLS 1.1, so it makes a lot of sense to discuss them first.

3.3.1 Cryptographic Subtleties

Section 2.4 mentioned that some researchers have found some subtle vulnerabilities and respective attacks against the way SSL/TLS employs padding in CBC encryption. In the case of SSL 3.0, these findings have led to the POODLE attack and to the recommendation not to use SSL anymore. In the case of TLS, Vaudenay published

12 The IANA is responsible for the global coordination of the DNS root, IP addressing, and other Internet protocol resources.

13 http://www.iana.org/assignments/tls-parameters.

a padding oracle attack in 2002 [8]. This attack is introduced and fully explained in Appendix B.2 (so this is a perfect moment in time to interrupt and read Appendix B.2). The attack works in theory, mainly because there are two distinct alert messages to signal either a decryption failure (i.e., `decryption_failed` with the code value 21) or a MAC verification failure (i.e., `bad_record_mac` with the code value 20). 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 the Vaudenay attack.

Mounting the Vaudenay attack, the adversary basically sends 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. So the victim serves as a padding oracle. As outlined in Appendix B.2, this attack works fine in theory. In practice, however, it is not immediately clear whether such an attack can be mounted at all. In the case of SSL/TLS, for example, the output of the padding oracle is encrypted and not accessible to the adversary. So the adversary cannot directly exploit the alert messages. Also, an SSL/TLS 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. However, there are situations in which a Vaudenay attack may still be feasible. In a 2003 follow-up paper, Vaudenay et al. showed how such an attack can actually be mounted against a password transmitted over an SSL/TLS connection [9]. Such a situation occurs, for example, if an IMAP client uses SSL/TLS to securely connect to an IMAP server. The attack cleverly combines the following 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 of Vaudenay et al. was practically relevant and became a 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 the Vaudenay attack.

An obvious protection mechanism is to make it impossible for an adversary to distinguish the two above-mentioned alert messages. This was proposed by Bodo Möller in a 2003 posting to the OpenSSL mailing list.[14] 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 suppressing `decryption_failed` alert messages, Vaudenay et al. suggested making alert messages time-invariant by simulating a MAC verification even if a padding error has occured and even to add random noise to the time delay [9]. In the TLS 1.1 specification published in 2006, 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 was adapted to require 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."

As explained in Section 3.8, the assessment that the timing channel is too small to be exploitable in practice has turned out to be wrong, as there are recent and very subtle attacks that do exactly this (e.g., the Lucky 13 attack). This has once again demonstrated the fact that the correct way to handle the Vaudenay attack is to use EtA instead of AtE. The security community has learned this lesson the hard way, and it is currently also moving in this direction (cf. Section 3.4.1.17).

The other security problem of CBC encryption mentioned in Section 2.4 is the vulnerability found by Bard in 2004 [10]. Strictly speaking, it is a consequence of an observation that was made earlier by Phillip Rogayway[15] in the realm of IPsec and that was later applied to TLS by Wei Dai and Möller. In theory, CBC requires a fresh

14 http://www.openssl.org/∼bodo/tls-cbc.txt.

15 A respective unpublished manuscript is available at http://web.cs.ucdavis.edu/∼rogaway/papers/draft-rogaway-ipsec-comments-00.txt.

and unpredictable (and hence randomly chosen) one-block IV for each message that is encrypted. In SSL 3.0 and TLS 1.0, however, only the initial IV is chosen (pseudo-)randomly, and all subsequent IVs are taken from the final block of the preceding ciphertext. Taking this IV chaining structure into account, an adversary can trivially predict the IV that is used to encrypt the subsequent message.

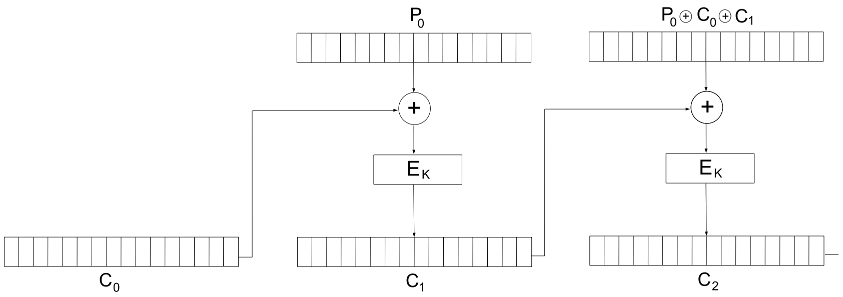

Figure 3.4 A chosen-plaintext distinguishing attack.

The ability to predict the IV enables a simple chosen-plaintext distinguishing attack (i.e., an attack in which the adversary has to distinguish the encryptions of two plaintext blocks using an encryption oracle). More specifically, the adversary can select two plaintext blocks P_0 and P_1 and is challenged with a ciphertext block $C = C_1$ that can either be the encryption of P_0 or the encryption of P_1. The adversary has to respond with the proper plaintext block P_b (where b can be 0 or 1). The adversary knows that the CBC encryption employs IV chaining, meaning that some known previous 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 plaintext block. If the adversary assumes $P_b = P_0$, then he or she can mount a chosen-plaintext distinguishing attack as illustrated in Figure 3.4. 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 $P_0 \oplus C_0 \oplus C_1$. Again, 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 are the same, if the assumption $P_b = P_0$ holds. Otherwise, if C_1 and C_2 are distinct, then P_b must also be 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 actually distinguish the encryptions of P_0 and P_1. This is an interesting observation, but it cannot be directly turned into an attack that is relevant in practice.

Bard was the first to notice that the vulnerability that can be exploited in this chosen-plaintext distinguishing attack (i.e., IV chaining) can also be exploited in a blockwise[16] CPA [11, 12]. The attack allows an adversary to validate a guess as to the value of a particular plaintext block. Besides the fact that this already violates the definition of a secure encryption, it allows an adversary to determine the value of a low-entropy string, such as a user-generated password or PIN.

Assume an adversary who has observed a ciphertext $C = C_1, C_2, \ldots$ that may comprise multiple records and who wants to verify a guess as to whether a particular plaintext block P_j $(j > 1)$ has a particular value $P^\star$. The adversary can mount a blockwise CPA, meaning that he or she has access to an encryption oracle. So the adversary can repeatedly have the oracle encrypt any message P' of his or her choice. In particular, he or she can construct a message whose initial block P'_1 equals $C_{j-1} \oplus C_l \oplus P^\star$, where

- C_{j-1} refers to the ciphertext block that immediately precedes the block under attack;
- C_l refers to the last ciphertext block of the record that immediately precedes the record in which C_j is transmitted;
- $P^\star$ refers to the adversary's guess (as mentioned above).

Note that the adversary can compile this message and send it the oracle for encryption. When the oracle encrypts the first block P'_1, it actually computes C'_1. This computation can be expressed as follows:

$$\begin{aligned} C'_1 &= E_k(P'_1 \oplus C_l) \\ &= E_k(C_{j-1} \oplus C_l \oplus P^\star \oplus C_l) \\ &= E_k(C_{j-1} \oplus P^\star) \\ &= E_k(P^\star \oplus C_{j-1}) \end{aligned}$$

If we compare this equation [i.e., $C'_1 = E_k(P^\star \oplus C_{j-1})$] with the general equation for CBC encryption [i.e., $C_j = E_k(P_j \oplus C_{j-1})$], then we can easily recognize that $C'_1 = C_j$ can only occur if and only if $P_j = P^\star$. This means that the adversary can verify his or her guess $P^\star$ by comparing C'_1 with C_j. If the two values are the same, then $P^\star$ must be the correct value for P_j. Otherwise, for example, if C'_1 and C_j are different, then the same procedure can be repeated with another guess $P^\star$. This can be repeated arbitrarily many times, until the correct value of P_j is found. In the simplest case, if the adversary knows that P_j is one of two possible values, then he

16 In contrast to a "normal" CPA where messages are considered to be atomic, in a blockwise CPA an adversary is assumed to have the additional ability to insert plaintext blocks within some longer message that is encrypted.

or she can determine the value by executing the attack a single time. More generally, if the adversary knows that P_j is one of n possible values, then it is sufficient to repeat the attack $n/2$ times (on the average). To mount such an attack, the adversary must know which plaintext block j contains the desired information. This, in turn, means that the adversary must know the exact format of the data transmission that is subject to SSL/TLS protection. Also, the adversary must know C_l and C_{j-1}. This is generally simple, because these ciphertexts are transmitted over the Internet. Technically, the most challenging part of the attack is to insert a plaintext block into the first block of the next message to be transmitted. In his original publication, Bard suggested the use of a malicious plugin for the browser (and argued why this is simpler than requesting the user to install malware that implements a keylogger in the first place). However, any other possibility to insert a plaintext block is fine and may work equally well.

Due to Bard's publications, the designers of TLS 1.1 had to replace the implicit IV used previously with an explicit IV (so the IV is no longer the last ciphertext block of the immediately preceding record). This means that for every TLS record that is encrypted with a block cipher, there is an appropriately sized IV that is randomly chosen and sent along with the corresponding `TLSCiphertext` fragment.[17] In spite of the fact that Bard's publication led to a revision of the way TLS uses IVs, it largely went unnoticed in the public. This changed immediately when Thai Duong and Juliano Rizzo presented a tool named *browser exploit against SSL/TLS* (BEAST) at the Ekoparty security conference in September 2011 and performed a live attack against a Paypal account.[18] The vulnerability that is exploited by the BEAST tool is documented in CVE-2011-3389. The attack implemented by the BEAST tool is conceptually similar to the one proposed by Bard, but it uses JavaScript (instead of Java applets). The BEAST tool attracted significant industry and media attention, and since its demonstration, the underlying vulnerability has become known to be severe and devastating. In fact, it is often recommended not to use SSL 3.0 and TLS 1.0 anymore. (As already mentioned in Section 2.4, the more recent POODLE attack has made it even necessary to deprecate SSL 3.0.)

For those who still need SSL 3.0 or TLS 1.0, there is a well-established workaround for the BEAST attack known as *record splitting*. The idea is to split every record that is sent into two records, with the first containing less than one cipher block worth of plaintext. In the simplest and most widely deployed case, the

17 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.

18 There is an unpublished manuscript written by Duong and Rizzo that is entitled "Here Come The $\oplus$ Ninjas" and that can easily be found on the Internet using the title as a search string.

first byte is sent in the first record, and the remaining $n - 1$ bytes are sent in the second record. The resulting record splitting technique is therefore also known as $1/n - 1$ *record splitting*. It defeats the possibility to construct a first single-block message as described above (because this block will be sent in two records). As $1/n - 1$ record splitting is implemented in most modern browsers, the attack no longer works in the field (it may only work in some exceptional cases against some old versions of browsers), and the BEAST tool is actually defeated. However, the acronym still makes people nervous about the security of SSL/TLS.

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` }

3.3.2 Cipher Suites

The cryptographic subtleties outlined above have led to a number of changes in the TLS 1.1 specification [2]. Most of these changes work, but some don't work perfectly. For example, the use of explicit IVs (that are sent along with the respective `TLSCiphertext` fragments) works, but the changes to defeat the Vaudenay attack do not work perfectly, as demonstrated, for example, by the Lucky 13 attack. Here, the long-term solutions will be to either move away from block ciphers operated in

Table 3.8
TLS 1.1 Kerberos-Based Cipher Suites

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

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 }

CBC mode or to use EtA instead of AtE. In either case, there is still a long way to go.

Except for these changes in the way a block cipher operates, there are only a few differences related to cipher suites. First of all, all cipher suites that comprise an export-grade key exchange algorithm or cipher may still be offered for backward compatibility, but they should not be negotiated anymore. (In fact, the FREAK and Logjam attacks mentioned in Section 3.8.5 have clearly shown that exportable cipher suites should no longer be supported.) This applies to all cipher suites written in italics in Table 2.4 (except SSL_NULL_WITH_NULL_NULL) and the export-grade Kerberos-based cipher suites from RFC 2712 [13]. All other Kerberos- and AES-based cipher suites proposed in RFC 2712 [13] and RFC 3268 [14] have been

Table 3.10
TLS 1.1 Certificate Type Values

Value	Name	Description
1	rsa_sign	RSA signing and key exchange
2	dss_sign	DSA signing only
3	rsa_fixed_dh	RSA signing with fixed DH key exchange
4	dss_fixed_dh	DSA signing with fixed DH key exchange
5	rsa_ephemeral_dh_RESERVED	RSA signing with ephemeral DH key exchange
6	dss_ephemeral_dh_RESERVED	DSA signing with ephemeral DH key exchange
20	fortezza_dms_RESERVED	FORTEZZA signing and key exchange

incorporated into TLS 1.1. The resulting cipher suites are summarized in Tables 3.7–3.9 (together with their respective code values). The Camellia-based cipher suites itemized in Table 3.3 still apply and can also be used in TLS 1.1. Again, refer to Appendix A for a complete list of the cipher suites that are currently registered for TLS and their respective code values.

3.3.3 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, the certificate management of TLS 1.1 remains essentially the same as in SSL 3.0 and TLS 1.0.

3.3.4 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, and the latter has become obsolete due to the reasons discussed earlier in this section (i.e., the distinction between `bad_record_mac` and `decryption_failed` enables specific padding oracle attacks).

3.3.5 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 may simplify the management of TLS sessions and 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) was ready and could be released officially in RFC 5246 [3].

The biggest change in TLS 1.2 is its extension mechanism that allows additional functionality to be incorporated into TLS without having to change the underlying protocol. We therefore discuss the TLS extension mechanism in detail before we delve into the other—mostly minor—differences between TLS 1.1 and TLS 1.2. 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 has been abandoned since TLS 1.2. In fact, a single—and, one hopes, more secure—cryptographic hash function is now being used. The respective PRF construction is simple and straightforward. It 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 exactly the same way (cf. 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 (which 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 the extensions, whereas the actual definitions are provided in a complementary document (i.e., standards track RFC 6066) [15].[19]

In general, TLS extensions are enabled by specific extensions that are sent along with the client and server hello messages. The extensions are backward-compatible, meaning that communication is possible between clients that support the extensions and servers that do not support the extensions, and vice versa. This means that 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. 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" any existing TLS server. A TLS 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 "breaks" existing TLS clients. 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 an extended CLIENTHELLO message. This is not the case yet.

Each extension is structurally identical and consists of two fields: a two-byte type field and a variable-length data field (that may also be empty). If, for example, a client wants to initially signal support for the secure renegotiation extension to the server, then it appends the five additional bytes 0xFF, 0x01, 0x00, 0x01, and 0x00 to the end of its CLIENTHELLO message. In this encoding, the first two bytes, 0xFF and 0x01, refer to the type of the extension (that corresponds to the binary value 11111111 11111111 00000000 00000001 and the decimal value $15 \cdot 16^3 + 15 \cdot 16^2 + 1 = 65,281$), the next two bytes, 0x00 and 0x01, refer to the length of the extension (that corresponds to 1), and the last byte refers to the length of the extension data field (that corresponds to 0). Other extensions are encoded similarly, and in many cases the data fields are not empty.

19 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.

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 are available for use. 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 summarized in Table 3.11 and further explained below. Note that RFC 6066 [15] also introduces a number of new TLS alert messages that are itemized in Table 3.12. Meanwhile, the `unsupported_extension` alert message with code 110 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).

Table 3.11
TLS 1.2 Extension Types and Values

Extension Type	Value	Description	Reference
`server_name`	0	Server name indication	[15]
`max_fragment_length`	1	Maximum fragment length negotiation	[15]
`client_certificate_url`	2	Client certificate URL	[15]
`trusted_ca_keys`	3	Trusted CA keys	[15]
`truncated_hmac`	4	Truncated HMAC	[15]
`status_request`	5	Certificate status request	[15]
`user_mapping`	6	User mapping	[16]
`client_authz`	7	Client authorization	[17]
`server_authz`	8	Server authorization	[17]
`cert_type`	9	Certificate types	[18]
`elliptic_curves`	10	Elliptic curve cryptography	[19]
`ec_point_formats`	11	Elliptic curve cryptography	[19]
`srp`	12	SRP protocol	[20]
`supported_signature_algorithms`	13	Signature algorithms	[3]
`use_srtp`	14	Key establishment for the SRTP	[21]
`heartbeat`	15	Heartbeat	[22]
`application_layer-protocol_negotiation`	16	Application-layer protocol negotiation	[23]
`status_request_v2`	17	Certificate status request version 2	[24]
`signed_certificate_timestamp`	18	Certificate transparency	[25]
`client_certificate_type`	19	Raw public keys	[26]
`server_certificate_type`	20	Raw public keys	[26]
`encrypt_then_mac`	22	Use EtA instead of AtE	[27]
`extended_master_secret`	23	Secure renegotiation (revisited)	[28]
`session_ticket`	35	Session tickets	[29]
`renegotiation_info`	65281	Secure renegotiation	[30]

Let us now briefly go through the TLS extensions that are available today. The first six extensions are specified in RFC 6066 [15], whereas all other extensions are specified in distinct RFC documents. These documents are referenced in the last column of Table 3.11.

Table 3.12
New TLS Alert Messages Introduced in [15]

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.

3.4.1.1 Server Name Indication

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. 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., 88.198.39.16). 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 hostname in a special host header (i.e., `Host: www.esecurity.ch`). This allows the server machine to distinguish the HTTP requests that target different web servers. 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 may reside on the same server 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) 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` (value 0) 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 very important for the large-scale deployment of the TLS protocol. However, SNI is a late addition to TLS, so there are still products that don't support it, and virtual hosting is still not globally available for all SSL/TLS-enabled web sites. More recently, it has even been shown that virtual hosting introduces some new security problems, and that the proper configuration of virtual hosts using SNI is far away from being trivial.[20]

3.4.1.2 Maximum Fragment Length Negotiation

Section 2.2 showed that the maximum fragment length of an SSL record is 2^{14} bytes. This also applies to TLS. In many situations, it is reasonable to work with fragments of exactly this length. There are, however, situations in which the clients are constrained and need to operate on fragments of smaller length. Also, there may be bandwidth constraints that require smaller fragments. This is where the extension type `max_fragment_length` (value 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 extension. Supported values are 1 (standing for 2^9 bytes), 2 (2^{10} bytes), 3 (2^{11} bytes), and 4 (2^{12} bytes). The negotiated length applies for the duration of a TLS session, including all session resumptions, so it is not dynamic and cannot be changed on the fly.

3.4.1.3 Client Certificate URL

Instead of including a full-fledged client certificate (or certificate chain, respectively) in a CERTIFICATE message that is 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. If a client wants to use this mechanism, then it includes an extension of type `client_certificate_url` (value 2) in the CLIENTHELLO message that is 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 subsequent handshake. 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

20 https://bh.ht.vc/vhost_confusion.pdf.

21) to the server. CERTIFICATEURL is one of two message types that are newly introduced in [15]—the other type is CERTIFICATESTATUS and is mentioned below.

From a security viewpoint, the client certificate URL mechanism has a problem. If it is naively implemented, then it can be misused to direct a server to a malicious web site where a drive-by infection can take place. This problem must be taken into account, when the mechanism is invoked. 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 on the server side. By no means should it be activated by default.

3.4.1.4 Trusted CA Keys

In a “normal” SSL/TLS handshake, the server does not know in advance what root CAs the client is going 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, 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 extension type `trusted_ca_keys` (value 3) can be included in the CLIENTHELLO message to inform the server what root CAs the client is going 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 its support for the extension). The actual certificate signed by one of the root CAs accepted by the client is still provided in the CERTIFICATE message.

3.4.1.5 Truncated HMAC

The output of the HMAC construction is equally long as the output of the hash function in use (i.e., 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` (value 4) can be used for this purpose. 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 that it does not affect the way the HMAC construction is used in the TLS PRF for master secret and key derivation.

Unfortunately, and due to some recent cryptanalytical results [31], the use of the `truncated_hmac` extension is no longer recommended, and most standards and best practices nowadays prohibit its use.

3.4.1.6 Certificate Status Request

When a party receives a certificate in an SSL/TLS handshake, it is usually this party's task to verify the validity of the certificate. As further addressed in Chapter 6, this can be done by retrieving a certificate revocation list (CRL) and checking that the certificate is not included in the list, or by using the online certificate status protocol (OCSP). This is simple and straightforward and normally does not pose a problem. Only if the certificate-receiving party 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` (value 5) 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. As mentioned above, this is the other message type that has been newly introduced in [15]. Due to the fact that the certificate status information is strongly related to OCSP, the mechanism enabled by this extension is sometimes also known as *OCSP stapling*.

OCSP stapling as enabled by the `status_request` extension supports only one OCSP response and can be used to check the revocation status of only a single server certificate. This limitation is addressed in RFC 6961 [24], where support for multiple OCSP responses is added. The respective extension type is `status_request_v2` (value 17). As of this writing, this updated version of the certificate status request extension is not yet well supported and deployed in client and server software, but this is likely to change in the future.

3.4.1.7 User Mapping

The user mapping extension is the first extension in Table 3.11 that is not specified in RFC 6066 [15]. It is based on a general mechanism to exchange supplemental data in a TLS handshake as originally proposed and specified in standards track RFC 4680 [32]. This mechanism and the respective TLS message flow are illustrated in Figure 3.5. The client and server exchange extended hello messages, before they can send supplemental data in SUPPLEMENTALDATA messages (type 23) to each other.

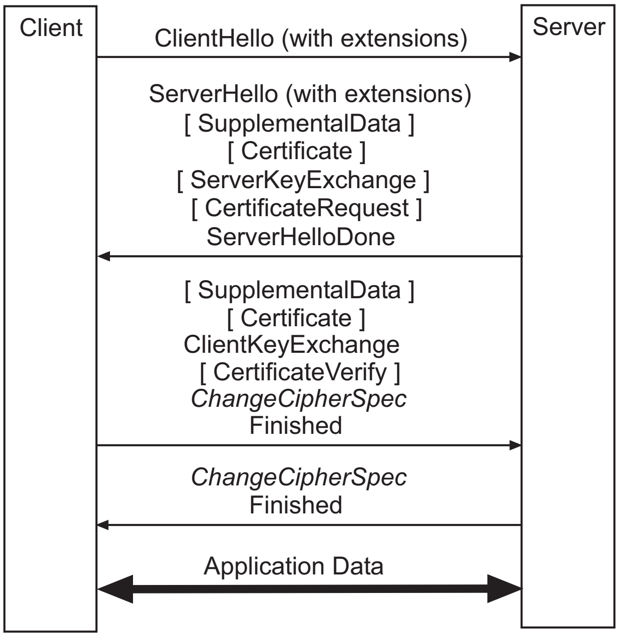

Figure 3.5 The TLS handshake protocol supporting the exchange of supplemental application data.

Using this general mechanism, standards track RFC 4681 [16] specifies the TLS extension `user_mapping` (value 6) 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.

3.4.1.8 Authorization

In its native form, the TLS protocol does not provide authorization—it only provides authentication. There are, however, two TLS extensions—`client_authz` (value 7) and `server_authz` (value 8)—specified in an experimental RFC [17] 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) is provided in a SUPPLEMENTALDATA message. 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 general, there are many authorization data formats that are possible. In practice, however, it is mainly about attribute certificates (ACs) [33] and security assertion markup language (SAML) assertions [34]. As introduced in Chapter 6, an AC is a digitally signed data structure that certifies some attributes of its holder (instead of a public key). It is linked to a public key certificate by a common subject field. A SAML assertion is conceptually similar to an AC, but it uses a distinct syntax. For a comprehensive list of authorization data formats, you may refer to the respective IANA repository.

3.4.1.9 Certificate Types

As further addressed in Section 6.1, there are two competing standards for public key certificates used on the Internet: X.509 and (Open)PGP. 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 these, the IETF TLS WG has released an informational RFC [18] that defines the extension type `cert_type` (value 9) and specifies how non-X.509 certificates (i.e., OpenPGP certificates) can actually be used in a TLS environment.

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), the 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 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 in 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 (cf. Section 2.2.2.6): 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 issued by peer entities instead of CAs).

3.4.1.10 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is a widely implemented and deployed technology that allows the use of short keys (for the same level of security), and that is particularly interesting for constrained environments. The IETF TLS WG has specified an informational RFC [19] to incorporate ECC into TLS. In particular, the RFC introduces five ECC-based key exchange algorithms that can be used in TLS. All of them use the elliptic curve version of the Diffie-Hellman (ECDH) key exchange algorithm, and they differ only in the lifetime of the ECDH keys (long-term versus ephemeral) and the mechanism used for authentication (ECDSA versus RSA). Combining these possibilities yields four key exchange algorithms. The fifth algorithm is anonymous and uses no authentication at all. These algorithms are described as follows.

- ECDH_ECDSA uses long-term ECDH keys and ECDSA-signed certificates. More specifically, the server's certificate must contain a long-term ECDH public key signed with ECDSA, and hence a SERVERKEYEXCHANGE message need not be sent. The client generates an ECDH key pair on the same elliptic curve as the server's long-term public key and may send its own public key in the CLIENTKEYEXCHANGE message. Both client and server then perform an ECDH key exchange and use the result as the premaster secret.
- ECDHE_ECDSA uses ephemeral ECDH keys and ECDSA-signed certificates. More specifically, the server's certificate must contain an ECDSA public key signed with ECDSA. The server sends its ephemeral ECDH public key and a specification of the corresponding elliptic curve in a SERVERKEYEXCHANGE message. The parameters are digitally signed with ECDSA using the private key corresponding to the public key in the server's certificate. The client generates another ECDH key pair on the same curve and sends its public key to the server in a CLIENTKEYEXCHANGE message. Again, both the client and the server perform an ECDH key exchange and use the result as the premaster secret.

- ECDH_RSA uses long-term ECDH keys and RSA-signed certificates. This key exchange algorithm is essentially the same as ECDH_ECDSA, except that the server's certificate is signed with RSA instead of ECDSA.
- ECDHE_RSA uses ephemeral ECDH keys and RSA-signed certificates. This key exchange algorithm is essentially the same as ECDHE_ECDSA, except that the server's certificate must contain an RSA public key authorized for signing, and the signature in the SERVERKEYEXCHANGE message must be generated with the corresponding private RSA key. Also, the server certificate must be signed with RSA instead of ECDSA.
- ECDH_anon uses an anonymous ECDH key exchange without any authentication. This basically means that no signature must be provided, and hence no certificate must be in place. The ECDH public keys are exchanged in SERVERKEYEXCHANGE and CLIENTKEYEXCHANGE messages.

The ECDHE_ECDSA and ECDHE_RSA key exchange algorithms provide PFS, whereas all others do not provide PFS. Each of the five key exchange algorithms can be combined with any cipher and hash function to form a cipher suite. (Again, all possible cipher suites are itemized in Appendix A.)

According to Table 3.11, there are two TLS extensions (described as follows) that refer to the use of ECC in TLS: `elliptic_curves` (value 10) and `ec_point_formats` (value 11). When a client wants to signal support for ECC to the server, then it includes 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[21]) [35] is provided in [19]. Many of these curves are also recommended by other standardization bodies, such as ANSI and NIST.[22] Among these curves, secp224r1, secp256r1, and secp384r1 are the most widely used ones. More recently, the IRTF has chartered a Crypto Forum Research Group (CFRG) as a general forum for discussing and reviewing uses of cryptographic mechanisms, such as ECC. The CFRG is currently working on recommendations for elliptic curves to be used in TLS. As of this writing, the elliptic curves recommended by the CFRG are known as

21 http://www.secg.org.

22 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).

Curve25519[23] and Ed448-Goldilocks.[24] Alternative curves—known as the Brainpool curves—have been specified for TLS in informational RFC 7027 [36]. In fact, there are 256-bit, 384-bit, and 512-bit versions of such curves, i.e., brainpoolP256r1, brainpoolP384r1, and brainpoolP256r1.

- 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 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.

3.4.1.11 SRP Protocol

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 [37, 38]. 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 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 [39, 40].[25] The SRP protocol has many use cases, also in the realm of TLS. There is, for example, an informational RFC [20] that specifies the use of the SRP protocol for client authentication in a TLS handshake. As such, it complements the use of public key certificates, preshared keys, or Kerberos for client authentication. The SRP protocol is a variant of the Diffie-Hellman key exchange, so it basically

23 The mathematical formula that specifies Curve25519 is $y^2 \equiv x^3 + 486662x^2 + x \pmod{p}$ with $p = 2^{255} - 19$. The curve has been proposed by Dan Bernstein.

24 The mathematical formula that specifies Ed448-Goldilocks is $x^2 + y^2 \equiv 1 - 39081x^2y^2 \pmod{2^{448} - 2^{224} - 1}$. The curve has been proposed by Mike Hamburg.

25 http://srp.stanford.edu.

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 (value 12) 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 master 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 is still not widely used in practice. This is surprising and difficult to explain.

3.4.1.12 Signature Algorithms

There are multiple hash and signature algorithms that may or may not be supported by different clients. The extension type `supported_signature_algorithms` (value 13) is defined in the TLS 1.2 protocol specification [3]. It is optional and can be used by a client 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.

The currently supported hash and signature algorithms are specified in respective IANA registries. In the case of hash algorithms, these are none (0),[26] MD5 (1), SHA-1 (2), SHA-224 (3), SHA-256 (4), SHA-384 (5), and SHA-512 (6). In the case of signature algorithms, these are anonymous (0), RSA (1),[27] DSA (2), and ECDSA (3). The corresponding code values are appended in brackets. In principle, each pair of algorithms specifies a way to generate and verify digital signatures. Note, however, that not all pairs are possible and equally reasonable. For example, DSA only works with SHA-1.

3.4.1.13 Heartbeat

In 2012, Robin Seggelmann submitted his Ph.D. thesis[28] entitled "Strategies to Secure End-to-End Communication" to the University of Duisburg-Essen. In his thesis, he suggested the use of a mechanism called Heartbeat to keep alive secure connections (even if they run on a connectionless transport layer protocol like UDP) and to

26 The "none" value is provided for future extensibility, in case of a signature algorithm that does not require hashing before signing.

27 The "RSA" value actually refers to RSA using PKCS version 1.5.

28 http://duepublico.uni-duisburg-essen.de/servlets/DerivateServlet/Derivate-31696/dissertation.pdf.

discover the path maximum transmission unit (PMTU).[29] The mechanism originally targeted DTLS, but it was recognized that it was also applicable to TLS (note that TLS does not provide a feature to keep a connection alive without continuous data transfer). So in the same year, Seggelmann and his colleagues published RFC 6520 [22], which introduced and specified a respective TLS extension that was submitted to the Internet standards track. Support for Heartbeat is advertised by both the client and the server via the `heartbeat` extension (value 15) and the mode that is encoded as a parameter. Afterward, the client and the server can send heartbeat request and response messages to each other. The IANA has assigned the content type value 24 for heartbeat messages (this complements the content type values introduced in Section 2.2.1.4). This means that heartbeat can be seen as another TLS subprotocol, in addition to change cipher spec (20), alert (21), handshake (22), and application data (23).

Heartbeat is an extension that was developed in academia and is not the result of an evolutionary process in practice. However, it is still supported by many implementations, including, for example, OpenSSL,[30] 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[31]), 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 senstive 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 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

29 The maximum transmission unit (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.

30 http://www.openssl.org.

31 The maximum length is $2^{14} = 16{,}384$ bytes.

new projects, such as LibreSSL.[32] It is expected that LibreSSL will be more securely implemented than OpenSSL.

3.4.1.14 Application-Layer Protocol Negotiation

According to Section 2.1, there are two strategies to stack an application-layer protocol on top of 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 application-layer protocol negotiation (ALPN)—formerly called *next protocol negotiation* (NPN)—comes into play. ALPN is enabled by the TLS extension `application_layer_protocol_negotiation` (value 16) specified in standards track RFC 7301 [23]. Supporting ALPN, a TLS-enabled server running on port 443 can support HTTP 1.0 by default, but also allow the negotiation of other application-layer protocols, such as HTTP 2.0 or SPDY[33] (pronounced "SPeeDY"). 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 its decision.

3.4.1.15 Certificate Transparency

Section 6.6 discusses a project that Google has launched called *Certificate Transparency* (CT).[34] The project goal is to overcome some problems of the currently deployed Internet PKI. One of the primary results of the CT project is a mechanism that allows a CA to submit every certificate it issues to a public log server (that represents a white list for certificates), and to get in return a proof of submission—called a *signed certificate timestamp* (SCT)—that it can provide to the relying parties. Obviously, there are many possibilities to provide and hand over an SCT to a relying party. The possibility that is favored by the CT project is to use a new TLS extension called `signed_certificate_timestamp` (value 18) that is specified in an experimental RFC [25]. Due to the fact that Google is promoting CT, there are good odds that the SCT extension will be used in the future.

32 http://www.libressl.org.

33 http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft1.

34 http://www.certificate-transparency.org.

3.4.1.16 Raw Public Keys

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 in the first place (this applies to X.509 and OpenPGP certificates, but it also applies to self-signed certificates that comprise the same fields as “normal” 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 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 [26] specifies two TLS extensions: the `client_certificate_type` (value 19) extension for the client and the `server_certificate_type` (value 20) extension for the server. Either or both extensions can be used in an extended TLS handshake when raw public keys for authentication need to be negotiated. The exact details of the negotiations are provided in [26]. We note that possible keys are for RSA, DSA, or ECDSA and that these keys must still be associated with entities. (Otherwise they cannot reasonably be used for authentication.) DNS-based authentication of named entities (DANE) in conjunction with DNSSEC can be used here (cf. Section 6.6). In spite of the fact that TLS extensions for raw public keys exist, they are still not widely used on the Internet.

3.4.1.17 EtA Instead of AtE

As mentioned several times so far, the appropriate way of combining authentication and encryption is EtA and not AtE (as currently used in SSL/TLS). 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 must be a mechanism to negotiate the order. This can be achieved by the TLS extension `encrypt_then_mac` (value 22) that has been proposed for this purpose (i.e., use EtA instead of AtE) in standards track RFC 7366 [27]. So the client and server can negotiate the use of EtA instead of AtE by exchanging hello messages that include this extension. As long as CBC encryption is used, this extension is important. It may, however, become obsolete when AEAD ciphers become more widely deployed. (As we will see later on, AEAD ciphers are mandatory in TLS 1.3.)

3.4.1.18 Session Tickets

Section 2.2.2 introduced and discussed the SSL handshake protocol and its simplified version that can be used to resume a session (cf. 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 in place 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 is well documented in [41, 42]. Standards track RFC 5077 [29] specifies a `session_ticket` extension (value 35) 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 3.6, 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. The format is irrelevant, but [29] still recommends a format.

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

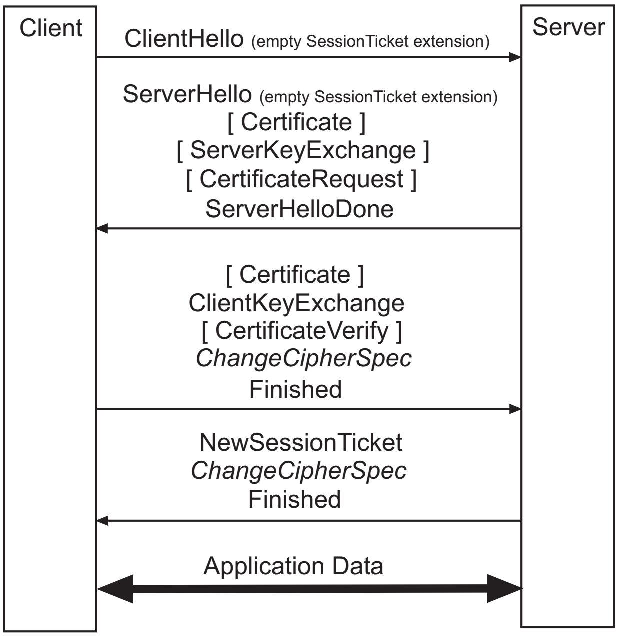

Figure 3.6 The message flow of the TLS handshake protocol issuing a new session ticket.

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 3.7. 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).

Last but not least, it is important to note that the use of session cookies (as discussed so far) in some senses breaks PFS 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 PFS, then one either has to stay away from session tickets entirely or use a more sophisticated approach (which has still to be researched and has not been standardized so far). Such an approach may, for example, look similar to secure cookies that have been proposed for HTTP [43]. So when using session tickets, the notion of PFS needs to be looked at carefully.

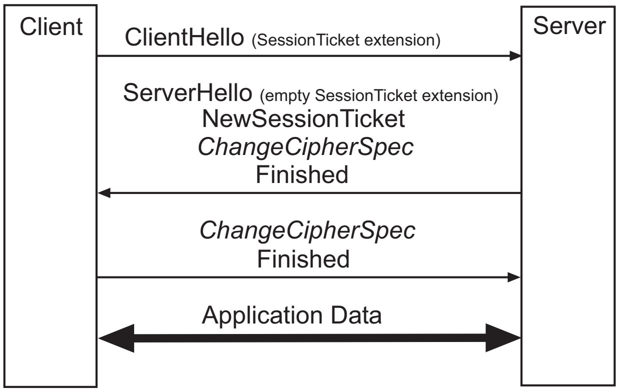

Figure 3.7 The message flow for an abbreviated TLS handshake protocol using a new session ticket.

3.4.1.19 Secure Renegotiation

Section 3.8.1 discusses how the TLS renegotiation mechanism can be exploited in a sophisticated attack known as the *renegotiation attack* and explains that the TLS extension `renegotiation_info` (value 65281) has been introduced in Standards Track RFC 5746 [30] to protect against this attack. Unfortunately, the protection is not foolproof, and people have come up with another attack—the so-called *triple handshake attack*—that can still be used to mount a renegotiation attack (even if the `renegotiation_info` extension is used). In an attempt to protect against the triple handshake attack (and to solve the underlying problem that the TLS protocol is susceptible to an unknown key-share attack), people have come up with another TLS extension called `extended_master_secret` (value 23) in RFC 7627 [28]. This extension is to ensure that every TLS connection has a distinct and, one hopes, unique master key, so that an unknown key-share attack cannot be mounted. It is hoped that this really protects against all types of renegotiation attacks. Again, we refer to Section 3.8.1 for the details.

3.4.1.20 Summary

In this section, we have seen that TLS 1.2 introduces and comes along with a huge quantity of possible 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 come up and 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 messages have no counterparts in the SSL protocol or any prior version of the TLS protocol.

Some of the extensions, such as `server_name` and `elliptic_curves`, are already widely used in the field, while others are likely to follow. Other extensions will not be deployed (or even disabled) and will silently sink into oblivion. As of this writing, however, it is too early to tell for which extensions this will be the case. Anyway, there is a lot of room for TLS evolution using extensions. Because some of these extensions may be sensitive and require protection, TLS 1.3 is going to introduce a feature that allows extensions to be exchanged in the encrypted part of a TLS handshake (cf. Section 3.5). This feature is particularly important for future extensions.

3.4.2 Cipher Suites

All cipher suites that employ DES or IDEA are no longer recommended[35] and have been removed from the TLS 1.2 protocol specification. The remaining cipher suites employ either 3DES or AES in the case of block ciphers and RC4 in the case of stream ciphers. They are itemized in Tables 3.13–3.15. Note that some cipher suites have been extended to also support SHA-256 (in addition to SHA-1). Similar to all previous versions of SSL/TLS, TLS 1.2 also supports the default cipher suite TLS_NULL_WITH_NULL_NULL. In the absence of an application profile standard specifying otherwise, a TLS-compliant application must implement and support the cipher suite TLS_RSA_WITH_AES_128_CBC_SHA.

Table 3.13
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` }

35 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 3.14
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 standards track RFC 5116 [44] that introduces algorithms for *authenticated encryption with additional data* (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 AEAD ciphers are required in TLS 1.3). 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, one hopes, more efficient and also more secure than the sequential application of the respective basic transformations. AEAD is a relatively new area in cryptographic research. It is, for example, possible to use a block cipher (e.g., AES) in a specific mode of operation to yield an AEAD cipher. Such modes of operation are the counter with CBC-MAC mode (CCM) [45] and the Galois/counter mode (GCM) [46]. The use of AES in CCM is specified in [47], whereas the use of AES in GCM is specified in [48, 49].[36]

36 Note that only the first RFC is submitted to the Internet standards track, whereas the second RFC is just informational.

Table 3.15
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` }

AEAD ciphers are usually nonce-based, meaning that the cryptographic transformation takes a nonce as an additional and auxiliary input (in addition to the plaintext and 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 additional data that is not encrypted. This is why it is called "authenticated encryption with additional data" and not only "authenticated encryption." This additional data can be some header information that cannot be encrypted by default. In the case of TLS, the additional data comprises the sequence number and the type, version, and length[37] fields of 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 always needs to 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 [50] specifies the following three sets of cipher suites for the TLS protocol that support authentication based on *preshared keys* (PSKs):

- 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.

37 This is true for TLS 1.2. In TLS 1.3, however, the length field is excluded from the additional data. This is because it presents a problem for ciphers with data-dependent padding, such as CBC.

- 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 addressed in Section 3.4.1.11 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 [51] complements RFC 4279 with regard to newer hash functions (SHA-256 and SHA-384) and AES in GCM mode, RFC 5489 [52] elaborates on ECDHE_PSK cipher suites, and RFC 4785 [53] specifies the use of PSK without encryption.

3.4.3 Certificate Management

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 fixed ECDH key exchange (with RSA used for authentication in the first case and ECDSA used for authentication 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. Also, it must be signed with an appropriate hash and signature generation algorithm pair.

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 new TLS extensions mechanism (cf. Section 3.4.1). It is sent by a client that receives an extended SERVERHELLO message containing an extension that it did not put in the respective CLIENTHELLO. Some complementary TLS alert messages that were introduced in [15] are summarized in Table 3.12.

3.4.5 Other Differences

Compression algorithms for TLS are specified in standards track RFC 3749 [54]. In addition to value 0 (referring to null compression), this RFC also introduces value 1 (referring to the DEFLATE compression method and encoding format specified in [55]). In short, the DEFLATE compression method combines LZ77 and Huffman encoding:

- LZ77 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 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 practice, and—due to the existence of some compression-related attacks (cf. Section 3.8.2)—the general recommendation is not to use it at all.

3.5 TLS 1.3

Since the official release of TLS 1.2 in 2008, the IETF TLS WG has been working hard on the next version of the TLS protocol. The resulting protocol is known as TLS 1.3 (with version number 3,4) and it is specified in an Internet draft [4] that will soon be submitted to the IETF.[38] It incorporates a few substantial changes, such as a new message flow and a focus on strong cryptography. Let us begin with the new message flow. The focus on strong cryptography is addressed in Section 3.5.1.

A normal SSL/TLS handshake requires at least two RTTs, and it gets even worse if certificates are verified. (Either CRLs need to be downloaded and checked or OCSP needs to be invoked.) In an attempt to improve the latency of TLS, Google introduced and deployed in its Chrome browser an optimistic technology known as *False Start* in 2010.[39] The idea of TLS False Start was to have a client start sending application data immediately after having sent its FINISHED message to the server (instead of waiting until the server sends back a FINISHED message). Under certain conditions, this may save one RTT. Unfortunately, the consistent implementation and

38 It is possible and very likely that this submission will have already occured when this book hits the shelves of the bookstores.

39 There is an experimental Internet-Draft entitled “Transport Layer Security (TLS) False Start” that will probably be published in an RFC document.

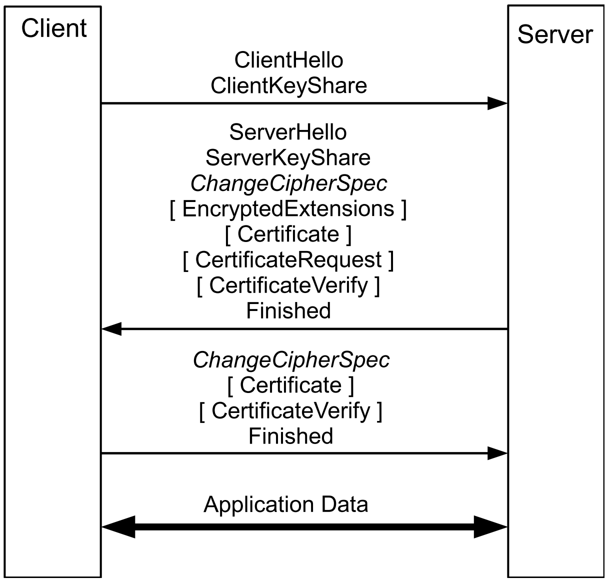

Figure 3.8 The TLS 1.3 message flow (overview).

deployment of TLS False Start has turned out to be difficult, and due to this difficulty, the technology has not been adapted in the standardization of TLS. However, when people started to design TLS 1.3, they remembered TLS False Start and wanted to employ similar optimistic technologies to reduce the number of RTTs required for a handshake. The result was a new message flow that is simplified and streamlined considerably. It is illustrated in Figure 3.8.

In TLS 1.3, there are only three flights that are needed to set up a TLS connection. This is similar to a TCP connection establishment that also requires three messages to be exchanged. If combined with OCSP stapling, this allows a very fast establishment of a secure connection. In the first flight, the client sends a CLIENTHELLO message immediately followed by a CLIENTKEYSHARE message to the server. The CLIENTKEYSHARE message (type 18) replaces the former CLIENTKEYEXCHANGE message (type 16) that is now reserved. As discussed below, TLS 1.3 no longer supports static key exchange, so every key exchange must be ephemeral to provide PFS. The CLIENTKEYSHARE message contains an appropriate set of parameters for zero or more key exchange methods supported by

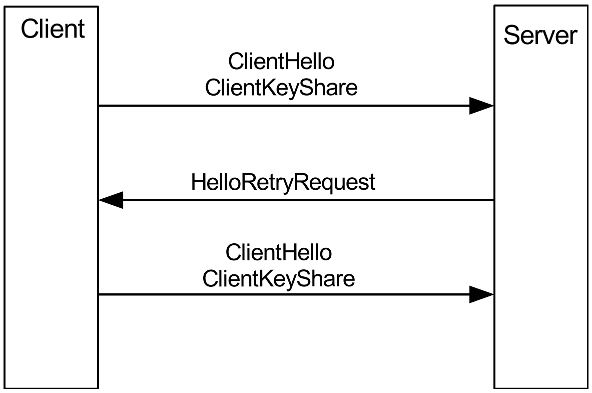

Figure 3.9 The TLS 1.3 handshake start with mismatched parameters.

the client. If the parameters are unacceptable or unsupported by the server, then the server sends back a HELLORETRYREQUEST message to the client, and the client has to restart the handshake from scratch with another CLIENTKEYSHARE message (the rest of the handshake remains the same). A respective handshake start with mismatched parameters is illustrated in Figure 3.9.

Once the client has provided an appropriate CLIENTKEYSHARE message, the server responds with a SERVERHELLO message. As usual, this message contains the server's choice of the protocol version, session ID, and cipher suite, a random value, and the server's response to the extensions offered by the client. In TLS 1.3 as opposed to previous versions of the protocol, the server's extensions are split between the SERVERHELLO message (in which the extensions are not encrypted) and the ENCRYPTEDEXTENSIONS message addressed below (in which the extensions are encrypted). The server then generates its keying material (i.e., the parameters used for key exchange) and sends it in a SERVERKEYSHARE message to the client. Similar to the CLIENTKEYSHARE message, the SERVERKEYSHARE message (type 17) replaces the former SERVERKEYEXCHANGE message (type 12) that is now reserved. The server then computes the shared secret, sends a CHANGECIPHERSPEC message to the client, and copies the pending write state into the current write state. The remainder of the server's messages are then encrypted under this key. Except for the FINISHED message, all of the following messages are optional:

- The server may send an ENCRYPTEDEXTENSIONS message that contains a response to any client's extensions that are not necessary to establish a key. Contrary to the server's extensions included in the SERVERHELLO message, the extensions sent in this message are encrypted (as the message name suggests).
- The server may send its certificate in a CERTIFICATE message if it is to be authenticated.
- The server may request a certificate from the client by sending a CERTIFICATEREQUEST message to it.
- If the server is to be authenticated, then it sends a CERTIFICATEVERIFY message to the client that comprises a signature over all handshake messages up to this point. In addition to explicit server authentication, this provides integrity for the handshake.

To finish the flight, the server sends a FINISHED message to the client. As usual, this message is keyed with the proper value to authenticate the handshake.

Once the client has received the SERVERKEYSHARE message, it is also able to compute the premaster secret and to derive the master secret and all keys that are needed. The client sends its CHANGECIPHERSPEC message to the server and copies its pending write state into the current write state. Again, the remainder of the client's messages are then encrypted under the respective key. If the server has sent a CERTIFICATEREQUEST message, then the client must send a CERTIFICATE message to the server. This message may be empty and contain zero certificates. But if the client has sent a certificate, then a digitally signed CERTIFICATEVERIFY message must also be sent to the server (to prove possession of the private key that belongs to the certificate). Finally, the client must send a FINISHED message to the server. Once the handshake is complete, the client and server can start using the now established TLS connection to exchange application data in a secure way.

3.5.1 Cipher Suites

As mentioned above, the other substantial change in TLS 1.3 is the focus on strong cryptography. In particular, this means that TLS 1.3 no longer supports compression (due to the compression-related attacks overviewed in Section 3.8.2), static (RSA and DH) key exchange (due to the lack of PFS), and non-AEAD ciphers. These three changes are described as follows.

- With regard to compression, the list of supported methods in a TLS 1.3 CLIENTHELLO message must contain only the value "null." If a client sends another value, then the server must generate a fatal `illegal_parameter` alert.

- With regard to key exchange, TLS 1.3 no longer supports any static method, such as RSA or DH, and always requires a key exchange that is ephemeral. Hence, the only key exchange algorithms supported by TLS 1.3 are DHE and ECDHE. The respective parameters needed are exchanged in the CLIENTKEYSHARE and SERVERKEYSHARE messages.
- With regard to encryption, TLS 1.3 no longer supports non-AEAD ciphers and always requires AEAD ciphers as introduced in Section 3.4.2. Remember that an AEAD cipher protects both the confidentiality and authenticity (together with integrity) of data in a single cryptographic transformation and that there is room for additional data that is only authenticated and integrity-protected (not encrypted). Because one single cryptographic transformation is applied, the encryption keys and IVs are needed, but the MAC keys can be ignored or need not be generated in the first place. Using an AEAD cipher, the data that is encrypted is the fragment of the `TLSCompressed` (or `TLSPlaintext`) structure,[40] and the data that is additionally authenticated comprises the sequence number as well as the type and version fields of the `TLSPlaintext` structure.

The three changes have a strong impact on the security of the TLS protocol, because they disable almost all cryptographic attacks that are known today. Hence, TLS 1.3 adds a lot of cryptographic strength.

3.5.2 Certificate Management

TLS 1.3 continues to support RSA, DSA, and ECDSA signatures. This means that the respective certificates are still needed and that there is hardly any difference in certificate management between TLS 1.3 and the previous versions.

3.5.3 Alert Messages

The fact that support for compression has been removed in TLS 1.3 means that the respective alert message `decompression_failure` (30) is not needed anymore. It is now obsolete and called `decompression_failure_RESERVED`.

40 Because support for compression is removed from TLS 1.3, the fragment of a TLSCompressed structure and the fragment of a respective TLSPlaintext structure are basically the same.

3.5.4 Other Differences

Because the TLS extension mechanism introduced in TLS 1.2 is a moving target and still subject to change, there are also a few subtle changes regarding the extensions defined for TLS 1.3. These changes are beyond the scope of this book and not further addressed here. You may refer to the TLS 1.3 specification [4] or the IANA registries for a respective update.

There are a few topics that deserve further study before TLS 1.3 can deviate from its predecessors. For example, there is an ongoing controversy about whether client puzzles to protect TLS servers against (D)DoS attacks should be added to the TLS protocol specification. In a respective proposal (that is specified in an Internet draft), the client signals support for client puzzles in a respective extension in its CLIENTHELLO message. If the server also supports client puzzles and wants to make use of one, it sends back a HELLORETRYREQUEST message with the same extension and a puzzle to be solved by the client. When the client recontacts the server with a new CLIENTHELLO message, it must also submit a solution for the puzzle (otherwise the message is not accepted). This protects the server against clients that simply try to flood the server with TLS connection establishments. Obviously, it does not protect the server against more sophisticated DDoS attacks. If the attack is launched in a botnet, then each bot is able to solve the puzzle presented by the server. In this case, the use of client puzzles does not really help and slows down TLS connection establishment in the first place. Hence, the controversy is about whether the advantages of client puzzles outweigh their disadvantages. This is not clear yet.

3.6 HSTS

At the 2009 BlackHat 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 before.[41] The tool acts as a MITM and attempts to remove the use of SSL/TLS by 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). To defeat this 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.[42] HSTS is based on previous work related to ForceHTTPS[43] [56] that is basically a prototype implementation

41 http://www.thoughtcrime.org/software/sslstrip/.

42 As introduced above, the acronym HSTS stands for "strict transport security."

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

of a Firefox extension. In 2012, the IETF officially released the standards track RFC 6797 [57] that specifies HSTS. The aim is to provide a mechanism that enables web sites 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 an SSL/TLS connection. While ForceHTTPS used an HTTP cookie to make the declaration, HSTS uses a special HTTP response header field (i.e., `Strict-Transport-Security`). The deployment of HSTS has grown tremendously in the recent past [58], and it has replaced ForceHTTPS entirely (it is mentioned only for the sake of completeness and for historic reasons).

It is important to note that HSTS is not a full-security solution for the web. It provides some security against specific (passive and active) attacks, but it does not protect against all possible attacks. For example, it does not protect against malware or phishing and related social engineering attacks. Taking into account its relatively narrow scope, the use of HSTS is still recommended and is not known to have any negative impact on security. Hence, there is hardly any reason not to use HSTS (except for the privacy concern related to HSTS supercookies discussed shortly). In fact, increasingly many companies and organizations follow this approach and incorporate HSTS in their web offerings. From a security viewpoint, this development is very much appreciated.

To invoke HSTS, a web server must be configured to declare itself an HSTS host when communicating with a browser. As previously mentioned, this declaration is made via the HTTP `Strict-Transport-Security` response header that must be sent over a secure connection. This means that a browser 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-Security` header field to the browser. The browser 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, browsers 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 deployed and shipped with a list of known HSTS hosts. (This list is sometimes known as the *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 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.

As mentioned above, the release of HSTS was very 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 misused to implement a sophisticated tracking technology (which found its way into the trade press under the name *HSTS supercookies*[44]). The technology employs the fact that HSTS can be invoked for each subdomain individually (instead of using the `includeSubDomains` directive). This means that whether HSTS is invoked or not (for a particular subdomain) reveals one particular 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 a 160-bit number that can be visualized using six characters from the base-32 character set (e.g., t9mxvk). However, any other numbering and encoding scheme is equally fine and can be used instead. If, for example, the HSTS host `www.esecurity.ch` wanted to use supercookies to track its visitors, 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

44 http://www.radicalresearch.co.uk/lab/hstssupercookies.

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 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 much deeper into the details of HSTS supercookies and their implications.

3.7 PROTOCOL EXECUTION TRANSCRIPT

To illustrate the functioning of the TLS protocol, we now provide a transcript of a TLS 1.0 session. We therefore consider the same setting as described in Section 2.3. The client takes the initiative and launches the TLS protocol by sending a CLIENTHELLO message to the server. The message 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` standing for a message length of 61 bytes, and `0x0301` again representing 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 the 11 cipher suites that are supported by the client. Each pair of bytes represents a 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 no compression). No TLS extension is appended.

After having received the 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 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 2,655 bytes). The three above-mentioned 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 ID 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 the null compression).

- Next, the CERTIFICATE message comprises the server's public key certificate. It is quite comprehensive and begins with the followiung 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 above.)

- 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 transmitted 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 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 very large.

3.8 SECURITY ANALYSES AND ATTACKS

Due to the fact that the SSL and TLS protocols have a long history, they have been subject to relatively many security analyses in the past (cf. Section 2.4). More recent theoretical results are reported in [59–61], and implementation issues related to TLS are addressed in [62]. In addition, an informal security analysis of TLS 1.2 is, for example, given in Appendix F of [3]. Some of these analyses have led to modifications incorporated in TLS 1.1. This includes, for example, Vaudenay's padding oracle attack [7, 8], as well as Bard's blockwise CPA [9–11] that is used, for example, in the BEAST tool (cf. Sections 2.4 and 3.3.1). Other analyses and respective insights have led to modifications that are incorporated in later versions of TLS.

In addition to these analyses, many researchers have tried to find alternative ways of attacking the TLS protocol and some of its implementations. A comprehensive and chronological overview is given, for example, in [63, 64]. A less detailed summary is provided in informational RFC 7457 [65].[45] In this book, we only address the most important attacks that are sometimes quite subtle and tricky. Attacks that target the (mis)use of certificates are postponed and further addressed in Chapter 6. Also, we want to emphasize again that the most severe attack against OpenSSL (i.e., Heartbleed) exploited a fairly trivial implementation bug and that similar bugs will probably reoccur in the future. So from a technical perspective, it will be important to patch the respective implementations as soon as possible. This makes patch management a very important topic with regard to the practical security of any implementation.

In the sequel, we focus on the attacks that are not caused by implementation bugs but rather by vulnerabilities and problems that are inherent to the TLS protocol. We address these attacks more or less in chronological order.

45 Unlike most other RFC documents referenced in this book, this RFC document is not provided by the TLS WG of the IETF Security Area but by the Using TLS in Applications (UTA) WG of the Applications Area that was chartered in November 2013 (cf. https://datatracker.ietf.org/wg/uta/).

3.8.1 Renegotiation Attack

In 2009, Marsh Ray and Steve Dispensa published a paper[46] in which they described a MITM attack against an optional but highly recommended feature of the TLS protocol, namely to be able to renegotiate a session (cf. CVE-2009-3555). The reasons that may make it necessary to renegotiate a session are discussed in Section 2.2.2. In short, it may be the case that cryptographic keys need to be changed, that the cipher suite or the level of authentication needs to be augmented, or—maybe most importantly—that a server requires certificate-based client authentication and therefore renegotiates the session. 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 session renegotiation, an already existing and previously established session is reused in a session 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 new HELLOREQUEST message to the client. The party that receives the message can either accept or refuse the renegotiation request. Only in the first case is a new TLS handshake initiated. In the second case, nothing is done and a `no_renegotiation` alert message (with code 100) is sent back as a warning.

The TLS renegotiation attack (as originally proposed by Ray and Dispensa) can be applied in many settings in which SSL/TLS is used. This applies, for example, to a web setting in which HTTP and HTTPS play a dominant role. Such a setting and the outline of a respective attack are overviewed in Figure 3.10. The adversary (representing the MITM) is sitting between the client (left side) and the server (right side). He or she waits until the client initiates a renegotiation by sending a CLIENTHELLO message to the server. (If the renegotiation is server-initiated, then the server has sent a HELLOREQUEST message to the client beforehand, but this is not important here.) The adversary captures the CLIENTHELLO message and postpones its delivery to some later point in time. Instead of immediately delivering the message to the server, the adversary establishes a first TLS connection to the server (labeled TLS connection 1 in Figure 3.10). Once this connection is established, the adversary uses it to send a first HTTP request message (HTTP request 1) to the server. Let us assume, for example, that HTTP request 1 comprises

46 When the paper was published, Ray and Dispensa were working for PhoneFactor, a then leading company in the realm of multifactor authentication. The publication was actually a technical report of this company. In 2012, PhoneFactor was acquired by Microsoft, and hence the technical reports of PhoneFactor are no longer directly available on the Internet. However, there are still many Internet repositories that distribute them. Any search engine may help to find the original publication entitled "Renegotiating TLS."

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 become clear later (it is hoped). At this point in time, it is important to note that many SSL/TLS implementations don't pass decrypted data immediately to the respective application; instead they wait for other data to come and pass them collectively. This behavior is exploited by the attack. In particular, the adversary takes the previously suspended CLIENTHELLO message and forwards it to the server. Based on this message, a session renegotiation is initiated, and if everything works fine, a second TLS connection (TLS connection 2 in Figure 3.10) is established. This connection is encrypted end-to-end, meaning that the adversary cannot compromise data that is sent over this connection. However, note what happens if the client uses this connection to send another HTTP request message (HTTP request 2) to the server. In this case, the two messages are concatenated and collectively delivered to the application. From the application's viewpoint, it is therefore no longer possible to distinguish the two message components—they both appear to originate from the same source. See what would happen in our example if HTTP request 2 started with the following two header lines:

```
GET /orderProduct?deliverTo=Address-2
Cookie: 7892AB9854
```

In this case, 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 product is going to be delivered to `Address-1` instead of `Address-2`. Note that the session cookie is provided by the client and is therefore valid, so there is no reason not to serve the request. Also note that there are many possibilities to exploit the renegotiation vulnerability and to come up with respective attacks. Examples are given in the original publication of Ray and Dispensa and many articles in the trade press, as well as a posting that appeared one week after the original publication was revealed.[47] The attacks work for both server-only authentication and mutual authentication modes of SSL/TLS, and not even the use of client certificates is able to prevent them.

47 http://www.securegoose.org/2009/11/tls-renegotiation-vulnerability-cve.html.

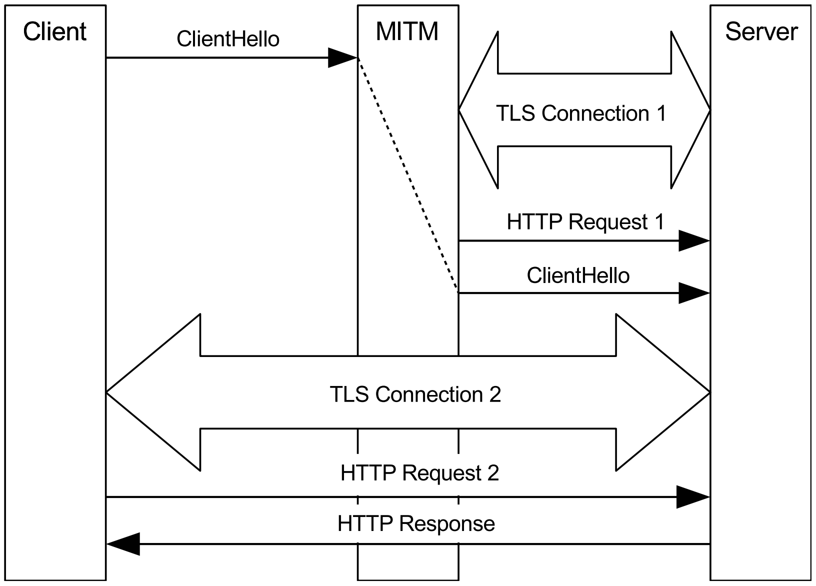

Figure 3.10 The TLS renegotiation attack (overview).

Renegotiation attacks are conceptually similar to cross-site request forgery (CSRF) attacks, so protection mechanisms against CSRF attacks generally help in protecting against renegotiation attacks. Also, because the renegotiation feature is optional (as mentioned above), a simple and straightforward way to defeat renegotiation 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 handshake. As introduced in Section 3.4.1.19 and offcially specified in standards track RFC 5746 [30], there is a distinct TLS extension, i.e., the `renegotiation_info` extension, that serves this purpose. Using this mechanism, the client and server include the `renegotiation_info` extension in their respective CLIENTHELLO and SERVERHELLO messages. If a party wants to signal support for secure renegotiation, then it sends the `renegotiation_info` extension with the empty string as a value. Only if a party wants to renegotiate, it actually includes data that confirms the previous handshake. Referring to Section 3.2.4, the client sends the client-side `verify_data`, whereas the server sends the concatenation of the client-side `verify_data` and the server-side `verify_data` of the respective FINISHED message from the previous handshake (the details vary with the different versions of the SSL/TLS protocols). Intuitively, by including the verify data from the previous handshake, the parties can be sure that they have the same view of the previous handshake and hence that they renegotiate the proper connection.

However, there is an interoperability issue that needs to be mentioned here: Because some SSL 3.0, TLS 1.0, and TLS 1.1 implementations have problems gracefully ignoring empty extensions at the end of hello messages, people have come up with another possibility to have a client signal to a server that it wants to securely renegotiate. Instead of sending an empty `renegotiation_info` extension in the CLIENTHELLO message, the client can also include a special signaling cipher suite, i.e., `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` (with code 0x00FF) in the list of supported cipher suites [30]. This also signals to the server that the client is willing and able to support secure renegotiation. After this initial phase, the actual secure renegotiation remains the same (so the secure renegotiation extension still needs to be supported by either party).

By linking the renegotiated connection to the handshake of the previous connection, the secure renegotiation mechanism seems to effectively thwart the original renegotiation attack and most of its variations. The client does not know that it initiates a renegotiation and hence it is not going to supply `verify_data` from a previous handshake. Also, if the adversary modified the CLIENTHELLO message to supply the data, then the respective message modification and integrity loss would be detected at the TLS level. Unfortunately, however, the protection is not foolproof, and it was shown in 2014 that a renegotiation attack can still be mounted, even though the secure renegotiation mechanism is put in place [66]. The respective MITM attack employs three handshakes and is therefore called *triple handshake attack*. It exploits two weaknesses.

- The first weakness is related to the fact that a MITM can have two TLS connections established—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 RSA is used for key exchange, then the MITM can simply relay handshake messages that are sent back and forth. These messages include, among other things, the premaster secret and the random values that used to generate the master key. Also, the session ID is chosen by the server and simply relayed by the MITM to the client. The feasibility of such an *unknown key-share attack* has been known for a long time (but it was not thought to be a real problem) [67].

- The second weakness is related to the fact that an unknown key-share attack followed by a connection resumption leads to a situation in which both connections not only share the same master key and session ID but also have the same `verify_data` in the respective FINISHED messages.

Exploiting these two weaknesses, the triple handshake attack actually works in three steps, where each step represents a handshake:

- In step 1, the MITM mounts an unknown key-share attack to establish two TLS connections that share the same master key and session ID. As usual in a MITM attack, the first connection is between the client and the MITM, and the second connection is between the MITM and the server.
- In step 2, the MITM waits until the client initiates a session resumption. When a session is resumed, nothing other than the session ID and the master key is required. (This is in contrast to a renegotiation that requires the `verify_data` from the FINISHED message of the previous handshake.) So the MITM can simply relay handshake messages forth and back. In the end, there are two fully synchronized connections that employ the same `verify_data`.
- In step 3, the MITM mounts the actual attack. As in a "normal" renegotiation attack, he or she sends HTTP request 1 to the server and then triggers a renegotiation that requires client-side authentication using a certificate. In spite of the fact that there is the secure renegotiation mechanism in place, the MITM can initiate the renegotiation, because it knows the proper `verify_data` that is needed now.

Once the client is authenticated, the HTTP request 1 is concatenated with the other HTTP requests that originate from the same client before they are passed to the application. This means that HTTP request 1 will be processed under the identity of the client. So we are back where we started, and an adversary can mount a renegotiation attack in spite of the fact that the secure renegotiation mechanism is put in place.

Since the publication of the triple handshake attack, work has been going on within the IETF TLS WG to mitigate the attack and to come up with a renegotiation mechanism that is really secure.[48] The most secure possibility to protect against the

48 https://secure-resumption.com.

triple handshake attack is to refuse any change of certificates during renegotiation. Another possibility is to make sure that an unknown key-share cannot take place. This happens, for example, if all TLS connections use a distinct and—one hopes—unique master secret. So if the master secret does not only depend on the premaster secret, the label "master secret," and the client and server random values, but also takes into account some other parameters, such as, for example, the server certificate, then the master secret that is generated—let us call it *extended master secret*—is likely to be unique. Distinct connections to different servers are then going to use distinct extended master secrets. So in the MITM scenario, the extended master secret for the connection between the client and the MITM and the extended master secret for the connection between the MITM and the server are going to be different. This defeats the unknown key-share attack, and hence it also defeats the triple handshake attack—at least in its current form. There is actually an Internet draft[49] that specifies how to generate an extended master secret and how to use it in a TLS setting. According to Section 3.4.1.19, a special TLS extension called `extended_master_secret` can be used here (i.e., the client sends the extension in its CLIENTHELLO message and the server does the same in its SERVERHELLO message). Section 3.1.1 demonstrated that the master secret is normally generated as follows:

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

However, if the `extended_master_secret` extension is negotiated in a TLS session, then the master secret is generated as follows:

```
master_secret =
   PRF(pre_master_secret,"extended master secret",
       session_hash)
```

Note that this construction uses another label 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 CLIENTKEYEXCHANGE message. This includes, among other things, the client and server random values. 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, however, the hash function employs the concatenation of MD5 and SHA-1.

If a TLS session is protected by an extended master secret, then it is assumed to be secure against all types of renegotiation attacks. However, this is just an

49 https://tools.ietf.org/html/draft-ietf-tls-session-hash-03.

assumption, and the validity of this assumption has yet to be challenged by the cryptographic community (e.g., [68]).

3.8.2 Compression-Related Attacks

In the past few years, there have been a few 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 a 2002 research paper [69]. The problem is that compression may reduce the size of a plaintext in a specific way and that this reduction in size may reveal some 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 (although some additional work is then required to align the block lengths). Let us discuss the compression-related attacks in chronological order.

3.8.2.1 CRIME

One year after the release of the BEAST tool (cf. Section 3.3.1), Rizzo and Duong presented another attack tool—named *compression ratio info-leak made easy* (CRIME)—at the Ekoparty security conference in 2012.[50] The respective vulnerability is documented in CVE-2012-4929. In fact, they presented a way to turn the vulnerability found by Kelsey into a side-channel attack against a 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—according to what has been said above—can either be null or DEFLATE[51]) 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 (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.[52] However, this was demonstrated in later attacks.

When a client sends an HTTP request message to a server, it may also send a session token in a `Cookie:` header together with a respective value. A respective header line may, for example, look as follows:

50 https://www.imperialviolet.org/2012/09/21/crime.html.

51 Zlib and gzip are also based on DEFLATE.

52 See, for example, slide 7 of the presentation that is publicly available from Google Docs. Search for "CRIME attack" and "Google docs."

```
Cookie: SessionToken=ws32456fg
```

In this case, the session token (that is intuitively named `SessionToken`) is assigned the artificially short value `ws32456fg`. The CRIME attack now targets this value and tries to steal the token to hijack the session. There are two requirements that must be fulfilled so that an adversary can mount the attack:

- First, the adversary must be able to eavesdrop on the network traffic between the client and the server.
- Second, the adversary must have some control over the browser used on the client side. This control can, for example, be achieved by injecting some JavaScript code from an evil site (using some form of drive-by infection).

The attack is conceptually simple and straightforward: For each character of the cookie, the adversary sends a series of HTTP request messages with different one-character path parameters to the server (to find the correct character). In our example, the adversary has the client send out a first HTTP request message in which the path parameter is set to `GET /SessionToken=a`. This allows the adversary to check whether the first character of the cookie is an "a." If this were the case, then the LZ77 algorithm of the DEFLATE compression method would have the resulting message to be compressed. (Note that the `Cookie:` header and the respective value are retransmitted with every HTTP request message.) Obviously, this is not the case here, because the first character is not an "a." So the adversary continues with a second HTTP request message in which the path parameter is set to `GET /SessionToken=b`. Again, the resulting message cannot be compressed. This continues until the adversary sends out an HTTP request message in which the path parameter is set to `GET /SessionToken=w`. Now, the string `SessionToken=w` repeats, and the respective HTTP request message can be compressed. This leads to a shorter message that can be detected by the adversary. So the adversary has found the first character of the cookie and can continue with all subsequent characters in a similar way. In fact, the adversary can attack each character individually to rebuild the entire cookie. For each character, the adversary has to try at most 256 characters (or 128 on average), and hence the workload of the attack mainly depends on the number of characters that form the cookie.

The attack works perfectly fine in theory. In practice, however, there are a few caveats that make things slightly more complicated. 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). 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 arguably makes the attack more difficult to mount. Duong and Rizzo proposed to deal with this problem using a so-called "two-tries method": Instead of sending a single HTTP request message for each character guess, the adversary actually sends two of them. In either case, a padding is included (where the padding consists of a set of characters taken from the complement of the alphabet for the target secret[53]). In one message, the padding is prepended to the guess, and in the other message, it is appended. In our example, the two messages for the first guess would be `GET /SessionToken=a{}` and `GET /SessionToken={}a`. If the guess is incorrect (as is the case here), then both messages should be compressed to the same size. If, however, the guess is correct, then the first message has a longer repeated substring, and this means that the message can be compressed more strongly (so `SessionToken=w{}` leads to a slightly bigger compression ratio than `SessionToken={}w`).

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 has no major negative impact (in contrast to disabling HTTP-level compression as discussed in Section 3.8.2.3). 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; 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., [70]). But there is still work going on within the IETF that follows this approach.[54]

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, TIME and BREACH both 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 then 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). 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. Also, given the fact that HTTP compression is very important in practice, mitigation by simply disabling compression does not work here. Instead,

53 For example, if the secret is known to consist of hexadecimal numbers, then a candidate padding is "{}".

54 https://tools.ietf.org/html/draft-pironti-tls-length-hiding-02.

mitigation typically requires some application code to be modified. This is difficult and sometimes prohibitive in some application contexts.

3.8.2.2 TIME

At the European Black Hat conference in March 2013,[55] 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 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. In spite of the fact that TIME is an interesting variant of CRIME, its deployment and use in the field has remained sparse.

3.8.2.3 BREACH

A few months after Shulman and Be'ery presented CRIME, Angelo Prado, Neal Harris, and Yoel Gluck announced a similar attack they named *browser reconnaissance and exfiltration via adaptive compression of hypertext* (BREACH) at the "normal" Black Hat conference that took place in August 2013 in Las Vegas.[56] 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.

All compression-related attacks require a lot of resources to mount in practice. However, if someone is motivated enough and capable to spend them, then the impact of a successful attack may be devastating. Hence, not using TLS-level compression and only carefully using HTTP-level compression seem to be reasonable countermeasures to mitigate the attacks and the respective risks. Unfortunately, not using HTTP-level compression is going to have a rather severe impact on performance. So this is certainly not a solution that is envisioned in the long term. As mentioned above, the same is true for any mitigation strategy that tries to hide the true length of the messages: It is not foolproof and can only slow down the adversary a little bit. In addition to these approaches that are not very effective, there are still a few approaches that are effective and may work in practice:

55 https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf.
56 http://breachattack.com.

- A first approach is to put the user input in a different compression context than the application data (that includes the target secret). Unfortunately, this approach generally requires a major rewrite of the application.
- A second approach is to make sure that the target secret does not remain the same between 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 also 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 will not be compressible).
- A third approach is to monitor the volume of data traffic per user and 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 may also be combined in one way or another. Unfortunately, this is not (yet) the case, and we don't see them implemented and deployed in the field. This means that many applications we use in daily life are still susceptible to compression-related attacks.

3.8.3 More Recent Padding Oracle Attacks

Section 3.3 mentioned a few mechanisms that had been introduced in TLS 1.1 to protect TLS implementations against padding oracle attacks like the Vaudenay attack. Most importantly, compliant implementations must ensure that the record processing time is essentially the same whether or not the padding is correct. The best way to achieve this (and to avoid the respective timing channel) is to compute a MAC even if the padding is incorrect (i.e., a packet is rejected only after a MAC is computed anyway). 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. The respective RFC documents [2, 3] note that "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 Kenneth G. Paterson demonstrated the opposite in 2013 [71]. In fact, they showed empirically that padding oracle attacks are still feasible—even if the defense strategy of the RFC documents 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. They are documented in CVE-2013-0169.

To understand the working principles and functioning of the Lucky 13 attacks, we have to go back to the HMAC construction introduced in Section 3.2.1.1. The aim of this construction is message authentication, so an HMAC value that represents a MAC 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 additional data that are prepended. More specifically, the 13 bytes comprise an 8-byte sequence number field and the usual 5-byte header of a `TLSCompressed` structure (comprising a type, version, and length field). So the point to remember 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 a lucky choice from the adversary's perspective, and hence the attack was named Lucky 13 (maybe also due to a lack of a better and more descriptive name).

Anyway, 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 leads 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 [72, 73].[57] The respective Merkle-Damgård construction uses a compression function that is iteratively applied to sequential 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

57 Both papers were presented at CRYPTO '89.

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 needed. 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 the terminology of cryptanalysis this means that a Lucky 13 attack represents a CCA.

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.[58] 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:

$$\begin{aligned} P_0 &= \underbrace{\text{AB AB} \ \ldots \ \text{AB}}_{32} \ \underbrace{\text{0xFF 0xFF} \ldots\ldots\ldots \text{0xFF}}_{256} \\ P_1 &= \underbrace{\text{AB AB} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{AB}}_{287} \ \text{0x00} \end{aligned}$$

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

58 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.

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.
- 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 SHA-1, 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, 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 likely to apply. 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 perfectly feasible.

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 [71] 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 actually needed to create a significant timing difference. So in the general case, plaintext recovery attacks are quite comprehensive 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^*. (It may be an explicit IV or the last block of the preceding ciphertext.) According to CBC decryption, we know that

$$P^* = D_K(C^*) \oplus C'$$

This is what the adversary is going for. As usual, we assume that he or she is capable of eavesdropping on the communications and of injecting messages of his or her choice into the network. 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 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 header of the TLS record, whereas C_0, C_1, C_2, $C' \oplus \Delta$, and C^* refer to five 16-byte blocks that collectively represent the TLS record's fragment field. C_0 is an IV, whereas all other blocks are non-IV ciphertext blocks. In the construction of $\overline{C}(\Delta)$, C_0, C_1, and C_2 are arbitrary and can be chosen at will. If $\overline{C(\Delta)}$ is sent to the victim, then the TLS record header is silently discarded, and 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 (i.e., $P_4[i] = P^*[i] \oplus \Delta[i]$ for every byte $i = 0, \ldots, 15$). If P_4 ends with two 0x01 bytes, then the plaintext block ends a valid padding of two bytes. In this case, two bytes are removed and the next 20 bytes are interpreted as MAC. This leaves a record of length at most $64 - 2 - 20 = 42$ bytes, meaning that MAC verification is performed on a message of length at most $42 + 13 = 55$ bytes (because a 5-byte header and an 8-byte sequence number are prepended before MAC verification). In all other cases (i.e., if P_4 ends with a 0x00 byte or any other (even invalid) byte pattern), then the MAC verification is performed on a message of length 56 bytes or more. So we have a situation in which the message is at most 55 bytes long, and all other situations in which the message is longer. We mentioned earlier that an HMAC value can be computed with only four evaluations of the compression function if a message is 55 bytes long at most, whereas five evaluations of are required for any longer message. This difference causes a different runtime behavior, and this, in turn, means that a value for the two last bytes of $\overline{C(\Delta)}$ can be detected, in which the underlying message block P_4 ends with two 0x01 bytes. This means that after 2^{16} trials (in the worst case), the adversary has found a value for $\overline{C(\Delta)}$ that triggers this case. The last two bytes of P^* can the be computed as follows:

$$\begin{aligned} P^*[15] &= P_4[15] \oplus \Delta[15] \\ P^*[14] &= P_4[14] \oplus \Delta[14] \end{aligned}$$

Once the two last bytes of P^* have been recovered, the adversary can recover the remaining bytes of P^* in a way that is similar to the original Vaudenay attack (cf. Appendix B.2). 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 now 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 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 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 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 fair assumption for randomly generated texts.

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 (as originally suggested in [9]). 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 [74], the attack is more feasible for DTLS than it is for TLS. So the topic is revisited in Chapter 4 (addressing DTLS).

In summary (and from a bird's eye perspective), Lucky 13 refers to the most recent padding oracle attacks that can be mounted against TLS. Once again, it has demonstrated that the AtE approach is susceptible to padding oracle attacks and that the EtA approach would be the better choice to start with (at least from a security perspective). As mentioned several times, this has led people to propose a protocol modification or at least a TLS extension mechanism (cf. Section 3.4.1.17). The use of this extension mechanism will disable Lucky 13 and all other padding oracle attacks against TLS. However, it will take some time until the mechanism is implemented and widely deployed. In the meantime, it is recommended that one mitigates padding oracle attacks by using cipher suites that employ authenticated encryption. This is mandatory in TLS 1.3.

3.8.4 Key Exchange Downgrade Attacks

Section 2.2.1.3 introduced the FREAK [75] and Logjam [76] attacks. Both attacks are 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 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.

3.8.5 FREAK

The FREAK attack[59] was published in March 2015. 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[60]). The vulnerabilities exploited by the FREAK attack are documented in CVE-2015-0204 for OpenSSL, CVE-2015-1637 for Microsoft's SChannel, and CVE-2015-1067 for Apple's Secure

59 https://www.smacktls.com.

60 Without referring to the implementation bug, the possibility of a "downgrade to export" attack was already mentioned in a note of Bodo Möller entitled "Export-PKC attacks on SSL 3.0/TLS 1.0" that was posted to the IETF TLS WG mailing list on October 10, 1998. Möller, by the way, is one of the researchers who finally revealed the FREAK attack almost 17 years later.

Transport. The attack starts with a client that sends a CLIENTHELLO 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. (The situation is slightly more involved and fully addressed in 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 key 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 this secret to to reconstruct all keys that are needed to either decrypt all messages or even generate new messages on the client's behalf.[61] To mount a FREAK attack, the MITM must be able to factorize a 512-bit integer in near 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).

3.8.6 Logjam

The Logjam attack[62] is similar to the FREAK attack but still different in many 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 a MITM has to wait until the client sends a CLIENTHELLO message to the server in which it proposes a DHE-based cipher suite (among other cipher suites). 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

61 Note that SSL provides some protection mechanism against an adversary changing the CLIENT-HELLO 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.

62 https://weakdh.org.

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 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 similar to the FREAK attack.

3.9 FINAL REMARKS

This chapter overviews, discusses, and puts into perspective the various versions of the TLS protocol (i.e., TLS 1.0, 1.1, 1.2, and 1.3). With its latest modifications and extensions, the TLS protocol is quite comprehensive and has drifted away from the simple and straightforward cryptographic security protocol it used to be. In fact, the more recent versions of the TLS protocol (i.e., TLS 1.2 and TLS 1.3) are quite involved and come along with many features. This helps in making the TLS protocol more flexible and useful in many (maybe nonstandard) situations. This includes, for example, some 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 informational RFC 7435, "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" [77]. If the only available alternative is no security at all, 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 sometimes ignore the fact that the security is only opportunistic and behave as if the data were fully protected. This, in turn, may lead to a wrong user behavior. Opportunistic security can therefore also be seen as a dual-edged sword.

In its current form, the TLS protocol is able to support all technologies and techniques the cryptographic community has come up with in the past. This applies, for example, to the AES (in possibly new modes of operation), 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 or technique into TLS. (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 (see Section 3.4.1.11 and [20]) and a Korean block cipher known as *ARIA* [78, 79]. There is even an informational RFC [80] about how to use TLS (since version 1.2) in full compliance with suite B cryptography of the NSA [81].[63] 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 or technique can be proposed to be used with TLS.) The downside of TLS'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 is to build implementations that provide interoperability. This general rule of thumb applies to any (security) protocol, not necessarily only to TLS. In the realm of IPsec, for example, we have created a similarly bad situation.

As already mentioned several times (most recently in the brief explanation of Lucky 13), the AtE approach followed by SSL/TLS is problematic and enables many sophisticated attacks, such as padding oracle attacks. A simple and straightforward possibility to make SSL/TLS more resistant against such attacks is to change the order of the authenticate and encrypt operations, and hence to follow an EtA approach. If one encrypts before one authenticates an SSL/TLS record, then all previously mentioned padding oracle attacks cannot be mounted anymore. Also, some theoretical investigations give us assurance that the EtA approach really leads to a construction that is inherently more secure. Unfortunately, changing the order of the authenticate and encrypt operations is easier said than done in practice, because it requires significant changes in the respective implementations. Section 3.4.1.17 discusses a TLS extension that can be employed to use EtA instead of AtE. As long as cipher suites are used that employ a block cipher in CBC mode, this extension remains important. Fortunately, this is no longer the case in TLS 1.3, and hence this problem will go away and silently sink into oblivion.

References

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

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

63 In essence, suite B cryptography includes AES-128 and AES-256 for encryption, ECDSA (using curves with 256- and 384-bit prime moduli) for digital signatures, and ECDH (using equally long prime moduli) for key exchange.

[3] Dierks, T., and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” Standards Track RFC 5246, August 2008.

[4] Rescorla, E., The Transport Layer Security (TLS) Protocol Version 1.3,” Internet-Draft, October 2015.

[5] Rescorla, E., “Keying Material Exporters for Transport Layer Security (TLS),” Standards Track RFC 5705, March 2010.

[6] Kato, A., M. Kanda, and S. Kanno, “Camellia Cipher Suites for TLS,” Standards Track RFC 5932, June 2010.

[7] Narten, T., and H. Alvestrand, “Guidelines for Writing an IANA Considerations Section in RFCs,” RFC 2434 (BCP 26), October 1998.

[8] 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.

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

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

[11] Bard, G.V., “A Challenging But Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL,” Cryptology ePrint Archive, Report 2006/136, 2006.

[12] 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.

[13] Medvinsky, A., and M. Hur, “Addition of Kerberos Cipher Suites to Transport Layer Security (TLS),” Standards Track RFC 2712, October 1999.

[14] Chown, P., “Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS),” Standards Track RFC 3268, June 2002.

[15] Eastlake, D., “Transport Layer Security (TLS) Extensions: Extension Definitions,” Standards Track RFC 6066, January 2011.

[16] Santesson, S., A. Medvinsky, and J. Ball, “TLS User Mapping Extension,” Standards Track RFC 4681, October 2006.

[17] Brown, M., and R. Housley, “Transport Layer Security (TLS) Authorization Extensions,” Experimental RFC 5878, May 2010.

[18] Mavrogiannopoulos, N., and D. Gillmor, “Using OpenPGP Keys for Transport Layer Security (TLS) Authentication,” Informational RFC 6091, February 2011.

[19] Blake-Wilson, S., et al., “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS),” Informational RFC 4492, May 2006.

[20] Taylor, D., et al., “Using the Secure Remote Password (SRP) Protocol for TLS Authentication,” Informational RFC 5054, November 2007.

[21] McGrew, D., and E. Rescorla, “Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP),” Standards Track RFC 5764, May 2010.

[22] Seggelmann, R., M. Tuexen, and M. Williams, “Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension,” Standards Track RFC 6520, February 2012.

[23] Friedl, S., et al., “Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension,” Standards Track RFC 7301, July 2014.

[24] Pettersen, Y., “The Transport Layer Security (TLS) Multiple Certificate Status Request Extension,” Standards Track RFC 6961, June 2013.

[25] Laurie, B., A. Langley, and E. Kasper, “Certificate Transparency,” Experimental RFC 6962, June 2013.

[26] Wouters, P. (ed.), et al., “Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS),” Standards Track RFC 7250, June 2014.

[27] Gutmann, P., “Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS),” Standards Track RFC 7366, September 2014.

[28] Bhargavan, K. (ed.), et al., “Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension,” Standards Track RFC 7627, September 2015.

[29] Salowey, Y., et al., “Transport Layer Security (TLS) Session Resumption without Server-Side State,” Standards Track RFC 5077, January 2008.

[30] Rescorla, E., S. Dispensa, and N. Oskov, “Transport Layer Security (TLS) Renegotiation Indication Extension,” Standards Track RFC 5746, February 2010.

[31] Paterson, K., T. Ristenpart, and T. Shrimpton, “Tag Size Does Matter: Attacks and Proofs for the TLS Record Protocol,” *Proceedings of ASIACRYPT 2011*, Springer-Verlag, LNCS 7073, 2011, pp. 372–389.

[32] Santesson, S., “TLS Handshake Message for Supplemental Data,” Standards Track RFC 4680, September 2006.

[33] Farrell, S., R. Housley, and S. Turner, “An Internet Attribute Certificate Profile for Authorization,” Standards Track RFC 5755, January 2010.

[34] OASIS Security Services Technical Committee, “Security Assertion Markup Language (SAML) Version 2.0 Specification Set,” March 2005.

[35] Standards for Efficient Cryptography, “SEC 2: Recommended Elliptic Curve Domain Parameters,” Version 1.0, September 2000.

[36] Merkle, J., and M. Lochter, “Elliptic Curve Cryptography (ECC) Brainpool Curves for Transport Layer Security (TLS),” Informational RFC 7027, October 2013.

[37] 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.

[38] 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.

[39] 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.

[40] Wu, T., “The SRP Authentication and Key Exchange System,” Standards Track RFC 2945, September 2000.

[41] Aura, T., and P. Nikander, “Stateless Connections,” *Proceedings of the First International Conference on Information and Communication Security (ICICS 97),* Springer-Verlag, LNCS 1334, 1997, pp. 87–97.

[42] Shacham, H., D. Boneh, and E. Rescorla, “Client-side caching for TLS,” *Transactions on Information and System Security (TISSEC),* Vol. 7, No. 4, 2004, pp. 553–575.

[43] Park, J.S., and R. Sandhu, “Secure Cookies on the Web,” *IEEE Internet Computing,* Vol. 4, No. 4, 2000, pp. 36–44.

[44] McGrew, D., “An Interface and Algorithms for Authenticated Encryption,” Standards Track RFC 5116, January 2008.

[45] Dworkin, M., *Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality,* NIST Special Publication 800-38C, May 2004.

[46] Dworkin, M., *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC,* NIST Special Publication 800-38D, November 2007.

[47] McGrew, D., and D. Bailey, “AES-CCM Cipher Suites for Transport Layer Security (TLS),” Standards Track RFC 6655, July 2012.

[48] Salowey, J., A. Choudhury, and D. McGrew, “AES Galois Counter Mode (GCM) Cipher Suites for TLS,” Standards Track RFC 5288, August 2008.

[49] Rescorla, E., “TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM),” Informational RFC 5289, August 2008.

[50] Eronen, P., and H. Tschofenig (eds.), “Pre-Shared Key Ciphersuites for Transport Layer Security (TLS),” Standards Track RFC 4279, December 2005.

[51] Badra, M., “Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode,” Informational RFC 5489, March 2009.

[52] Badra, M., and I. Hajjeh, “ECDHE_PSK Cipher Suites for Transport Layer Security (TLS),” Standards Track RFC 5487, March 2009.

[53] Blumenthal, U., and P. Goel, “Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS),” Standards Track RFC 4785, January 2007.

[54] Hollenbeck, S., “Transport Layer Security Protocol Compression Methods,” Standards Track RFC 3749, May 2004.

[55] Deutsch, P., “DEFLATE Compressed Data Format Specification version 1.3,” Informational RFC 1951, May 1996.

[56] Jackson, C., and A. Barth, “ForceHTTPS: Protecting High-Security Web Sites from Network Attacks,” *Proceedings of the 17th International World Wide Web Conference (WWW2008),* 2008.

[57] Hodges, J., C. Jackson, and A. Barth, “HTTP Strict Transport Security,” Standards Track RFC 6797, November 2012.

[58] Garron, L., A. Bortz, and D. Boneh, “The State of HSTS Deployment: A Survey and Common Pitfalls,” 2013, https://garron.net/crypto/hsts/hsts-2013.pdf.

[59] Jager, T., et al., “On the Security of TLS-DHE in the Standard Model,” *Proceedings of CRYPTO 2012*, Springer-Verlag, LNCS 7417, 2012, pp. 273–293.

[60] 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.

[61] 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.

[62] 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.

[63] Meyer, C., and J. Schwenk, *Lessons Learned From Previous SSL/TLS Attacks—A Brief Chronology Of Attacks And Weaknesses*, Cryptology ePrint Archive, Report 2013/049, January 2013.

[64] Meyer, C., *20 Years of SSL/TLS Research—An Analysis of the Internet’s Security Foundation,* Ph.D. Thesis, Ruhr-University Bochum, February 2014.

[65] Sheffer, Y., R. Holz, and P. Saint-Andre, “Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS),” Informational RFC 7457, February 2015.

[66] 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.

[67] 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.

[68] Giesen, F., F. Kohlar, and D. Stebila, “On the Security of TLS Renegotiation,” *Proceedings of 20th ACM Conference on Computer and Communications Security (CCS 2013),* ACM Press, New York, NY, 2013, pp. 387–398.

[69] 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.

[70] 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.

[71] 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.

[72] Merkle, R.C., "One Way Hash Functions and DES," *Proceedings of CRYPTO '89,* Springer-Verlag, LNCS 435, 1989, pp. 428–446.

[73] Damgård, I.B., "A Design Principle for Hash Functions," *Proceedings of CRYPTO '89,* Springer-Verlag, LNCS 435, 1989, pp. 416–427.

[74] 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.

[75] 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.

[76] 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.

[77] Dukhovni, V., "Opportunistic Security: Some Protection Most of the Time," Informational RFC 7435, December 2014.

[78] Lee, J., et al., "A Description of the ARIA Encryption Algorithm," Informational RFC 5794, March 2010.

[79] Kim, W., et al., "Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)," Informational RFC 6209, April 2011.

[80] Salter, M., and R. Housley, "Suite B Profile for Transport Layer Security (TLS)," Informational RFC 6460, January 2012.

[81] NSA, "Fact Sheet Suite B Cryptography," November 2010,
http://www.nsa.gov/ia/programs/suiteb_cryptography/.

Chapter 4

DTLS Protocol

In this chapter, we elaborate on the DTLS protocol that is the UDP version of the TLS protocol, meaning that it can be used to secure UDP-based applications and respective application protocols. More specifically, we introduce the topic in Section 4.1, address the basic properties and distinguishing features of the DTLS protocol in Section 4.2, briefly analyze its security in Section 4.3, and conclude with some final remarks in Section 4.4. Instead of explaining the DTLS protocol from scratch, we assume the reader to be familiar with the SSL/TLS protocols and mainly focus on the differences between the DTLS protocol and the SSL/TLS protocols. This also means that this chapter does not stand for itself, but can only be understood after having read and properly understood Chapters 2 and 3. We think that this is appropriate, because nobody is going to start with the DTLS protocol.

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 the TCP/IP protocol suite, and hence they can be used only to secure TCP-based applications. However, there are increasingly many applications and application protocols that are not TCP-based, but rather use UDP as a transport layer protocol. Examples include media streaming, real-time communications (e.g., Internet telephony and videoconferencing), multicast communications, online gaming, and—maybe most importantly—many application-layer protocols that are used for the Internet of Things (IoT).

In contrast to TCP, UDP provides only a best-effort datagram delivery service that is connectionless and unreliable. So neither the SSL protocol nor any of the TLS protocol versions can be used to secure UDP-based applications and respective protocols. The same is true for other connectionless transport layer protocols that are used in the field, such as the datagram congestion control protocol (DCCP) [1] or the stream control transmission protocol (SCTP) [2]. Both protocols have in common with UDP that applications and application protocols layered on top of them cannot invoke SSL/TLS natively. This is a problem, and the designers and developers of such a protocol generally have the following three options:

- 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, all application protocols that have stringent latency and jitter requirements cannot live with TCP's loss and congestion correction algorithms.
- 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 many disadvantages and are difficult to deploy and use in the field. (Some reasons for this 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 protocol is stacked. Unfortunately, the design, implementation, and deployment of a new protocol is 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 (theoretical) options have severe disadvantages, and hence they are not particularly useful in practice. The most desirable way to secure an application protocol is still to use SSL/TLS or a similar technology that is equally simple and efficient and that also runs entirely in application space, without requiring kernel modifications. In the past, there have been a few proposals, such as Microsoft's STLP (see Section 1.2) or the WAP Forum's wireless TLS (WTLS) protocol. Both proposals have not been successful and have silently sunk into oblivion.

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. The term that was originally coined in a 2004 publication for the yet-to-be-defined protocol was *datagram TLS* (DTLS) [4]. This term was adopted by the IETF TLS WG. When people talk about DTLS, they actually refer to a TLS version that derives as little as possible from TLS but that can be layered on top of UDP (instead of TCP).

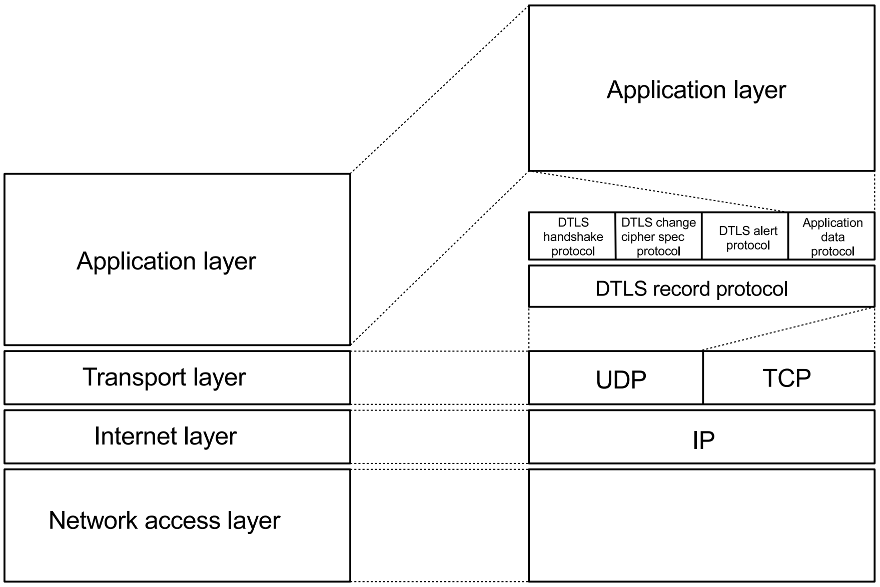

Figure 4.1 The placement of the DTLS protocol in the TCP/IP protocol stack.

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 it is structurally identical. This is illustrated in Figure 4.1 (as compared to Figure 2.1). The only differences refer to the name and the fact that DTLS is layered on top of UDP (instead of TCP). This also means that the DTLS protocol must be able to deal with datagrams that are not reliably transmitted, meaning that they can get lost, reordered, or replayed. Note that the TLS protocol has no internal facilities to handle this type of unreliability, and hence TLS implementations would routinely 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 UDP-based applications, as well as DCCP- and SCTP-based applications. (The use of DTLS to secure DCCP- and SCTP-based applications is addressed in [5] and [6].) So one can reasonably expect that many applications will make use of DTLS in the future. There is no well-known UDP port for DTLS, because—similar to the SSL/TLS protocols—the port number depends on the application protocol that is layered on top of UDP (and DTLS). In the case of OpenSSL, for example, the default port for native DTLS is 4433.

So far, there have been two versions of the DTLS protocol: DTLS version 1.0 specified in RFC 4347 [7], which was officially released (and submitted to the Internet standards track) in 2006, and DTLS version 1.2 specified in RFC 6347 [8], which was offcially released in 2012. Note that there is no documented version 1.1 of the DTLS protocol and that the release of version 1.2 was to bring DTLS standardization in line with TLS standardization. This implies that most things that have been said for TLS 1.2 also apply to DTLS 1.2. In particular, all cipher suites registered by the IANA for TLS, at least in principle, also apply to DTLS. There are only a few exceptional cipher suites that cannot be used for DTLS, such as the family of cipher suites that comprise the stream cipher RC4 (see Appendix A for a comprehensive overview.)

Instead of explaining the DTLS protocol from scratch, we now take advantage of the fact that we are already well familiar with the SSL/TLS protocols; we can therefore focus on the basic properties and distinguishing features of the DTLS protocol. This approach (of restricting the focus on the differences between the DTLS protocol and the SSL/TLS protocols) is also followed by the official DTLS protocol specifications.

4.2 BASIC PROPERTIES AND DISTINGUISHING FEATURES

To understand the key differences between the SSL/TLS protocols and the DTLS protocol, it is necessary to have a look at two problem areas that are specific and that need to be addressed by the DTLS protocol in one way or another.

- First, UDP provides a connectionless best-effort datagram delivery service that operates at the transport layer. This means that a UDP datagram is transmitted and processed independently from all other datagrams, and hence it must also be possible to encrypt and decrypt a DTLS record (that is transmitted in 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 sometimes not processed independently and that there are at least two types of interrecord dependencies.
  - In some cipher suites, cryptographic context is chained between subsequent TLS records. If, for example, a block cipher is used in CBC mode, then SSL 3.0 and TLS 1.0 require that the last ciphertext block be

the IV for the encryption of the next plaintext block. Also, if a stream cipher is used, then the key stream needs to be synchronized between the sender and recipient. In this case, the key stream index represents the cryptographic context that yields an interrecord dependency.

- As addressed in Section 3.2, the TLS protocol provides protection againt replay and message reordering attacks by using a MAC that also comprises a sequence number (which 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 records independently, then all types of interrecord dependencies must be avoided. This can be done by ignoring some cryptographic mechanisms or adding explicit state to the records. If, for example, stream ciphers are not used in DTLS, then stream cipher key state need not be maintained in the first place. This is why RC4-based cipher suites are not part of the DTLS protocol specification.[1] Also, TLS 1.1 has already added explicit CBC state to TLS records (see Section 3.3). In principle, DTLS uses the same mechanism, and an explicit sequence number (consisting of two fields) is added to the DTLS record format.

- Second, the TLS handshake protocol requires a reliable transport channel for the transmission of messages. Due to this requirement, the TLS handshake protocol can be kept comparably simple. If UDP is used instead of TCP, then this requirement is no longer fulfilled, and the DTLS handshake protocol must be modified to compensate for the missing reliability. In fact, the DTLS handshake protocol must incorporate controls for messages that are lost, reordered, or replayed. This makes the resulting protocol more involved.

These two problem areas have led to quite fundamental changes in both the record protocol and the handshake protocol of DTLS. Let us look at these changes separately.

4.2.1 Record Protocol

As a consequence of the first problem area mentioned above, each DTLS record includes an explicit sequence number (in addition to the type, version, length, and fragment fields that are already present in a "normal" SSL/TLS record). To make

1 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).

sure that sequence numbers are distinct, DTLS uses numbers that actually consist of two fields:

- A 16-bit *epoch* field that comprises a countervalue that is incremented on every cipher state change.
- A 48-bit *sequence number* field that comprises an explicit sequence number that is incremented for every DTLS record (sent in a given cipher state).

The epoch field is initially set to zero and is incremented each time a CHANGECIPHERSPEC message is sent. The sequence number field, in turn, always refers to a specific epoch and is incremented on a per-record basis. It is reset to zero after every CHANGECIPHERSPEC message that is sent. The DTLS record numbering scheme is illustrated in Figure 4.2.

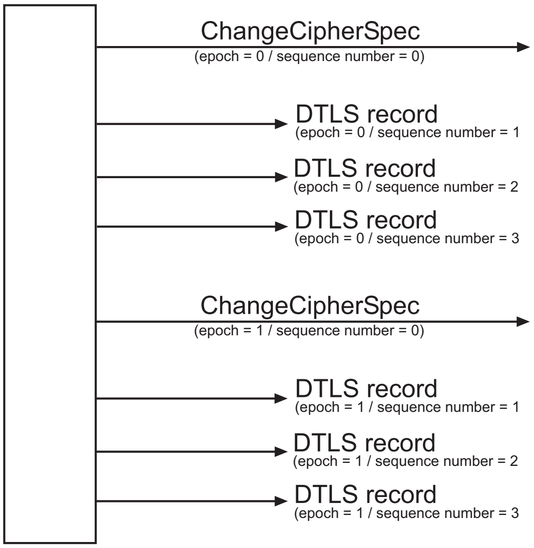
ChangeCipherSpec
(epoch = 0 / sequence number = 0)
DTLS record
(epoch = 0 / sequence number = 1
DTLS record
(epoch = 0 / sequence number = 2
DTLS record
(epoch = 0 / sequence number = 3
ChangeCipherSpec
(epoch = 1 / sequence number = 0)
DTLS record
(epoch = 1 / sequence number = 1
DTLS record
(epoch = 1 / sequence number = 2
DTLS record
(epoch = 1 / sequence number = 3

Figure 4.2 The DTLS record numbering scheme.

The resulting DTLS record format is illustrated in Figure 4.3. Comparing this figure to Figure 2.4 reveals the fact that the only difference between the SSL/TLS and DTLS record formats is the inclusion of the two above-mentioned fields in a DTLS record. Except for that, the two record formats are the same.

With regard to message authentication, we note that the HMAC construction used by TLS also employs a sequence number seq_number (see Section 3.2) but that this number is implicit, meaning that it must be maintained by the communicating

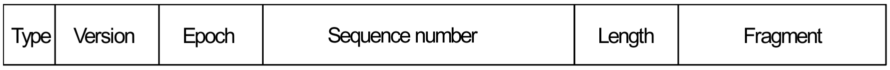

Figure 4.3 The outline of a DTLS record.

peers. In contrast, the sequence number of DTLS is explicit, meaning that it is now part of the DTLS record. If the concatenation of the epoch value and the sequence number value is taken as the new 64-bit value for seq_number, then the formula to compute a MAC in DTLS 1.0 is exactly the same as the formula to compute a MAC in TLS. It can be formally expressed as follows:

$$\begin{aligned} & HMAC_K(DTLSCompressed) = \\ & \quad h(K \parallel opad \parallel h(K \parallel ipad \parallel epoch \parallel sequence\,number \parallel \\ & \quad type \parallel version \parallel length \parallel fragment)) \end{aligned}$$

Again, note that the bitstring $epoch \parallel sequence\ number$ in DTLS refers to seq_number in TLS. In either case, the length of the bitstring is 64. An important difference between TLS and DTLS is that in TLS MAC errors must result in connection termination, whereas in DTLS MAC errors need not result in connection termination. Instead, the receiving DTLS implementation should silently discard the offending record and continue with the transmission. 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 generate a `bad_record_mac` alert (code 20) with level fatal and terminate the connection.

Last but not least, we note that there is a subtle difference between the TLS and DTLS record format that is related to the version field: In a DTLS record this field 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 of 1,0 is 254,255 (or 0xFEFF, respectively). So these bytes are included in the DTLS record's version field. For DTLS 1.2, the 1's complement of 1,2 is 254,253, and hence these bytes are included. So future version numbers of DTLS are decreasing in value, while the true version number is increasing in value. The maximal spacing between TLS and DTLS version values is to ensure that records from the two protocols can be clearly distinguished.

4.2.2 Handshake Protocol

Similar to TLS handshake messages, DTLS handshake messages may be quite large (i.e., up to $2^{24} - 1$ bytes). In contrast (and to avoid IP fragmentation), DTLS records are usually kept smaller than the maximum transmission unit (MTU) or path MTU

(PMTU). In the case of an Ethernet segment, for example, the standard MTU size is 1,500 bytes, and any larger handshake message must be transmitted in multiple DTLS records. Due to the unreliability of UDP, these records can be lost, received out of order, or replayed. So the DTLS handshake protocol must be modified to compensate for these situations. On one hand, the header format is modified to comprise some additional fields. On the other hand, the protocol is modified to support message retransmission. These two differences are addressed next, before a third difference that refers to a stateless cookie exchange to protect against denial of service (DoS) attacks is explained.

4.2.2.1 Header Format

Since a DTLS handshake message may be too large to fit into a single DTLS record, it may span multiple records. This, in turn, makes it necessary to fragment and reassemble the message. To support fragmentation and reassembly, each DTLS handshake message header must comprise three new fields (in addition to the "normal" type and length fields):

- A 16-bit *message sequence* field that comprises a sequence number for the message that is sent. The first message each side transmits in a handshake has a value of zero, and 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, however, that from the DTLS record layer's perspective, the retransmission of a message requires a new record. So the sequence number of the DTLS record (as introduced and discussed above) will have a new value.
- A 24-bit *fragment offset* field that contains a value that refers to the number of bytes contained in previous fragments.
- A 24-bit *fragment length* field that contains a value that refers to the length of the fragment.

So the first handshake message that is sent has a message sequence field value of zero, a fragment offset field value of zero, and an appropriate fragment length field value. If this value is n_1, then the second handshake message has a message sequence field value of one, a fragment offset field value of n_1, and a fragment length field value of n_2. The third handshake message has a message sequence field value of two, a fragment offset field value of $n_1 + n_2$, and a fragment length field value of n_3. This continues until the last handshake message is sent.

4.2.2.2 Message Retransmission

Due to the fact that the DTLS protocol is layered on top of UDP, handshake messages may get lost. So the DTLS protocol must be able to handle this situation. The usual approach to handle message loss is the use of acknowledgments of receipt. This means that the sender starts a timer when it sends out a message and that it then waits for an acknowledgment of receipt before a timeout occurs. The DTLS protocol also envisions this approach: When a client sends an initial CLIENTHELLO message to the server, it starts a timer and it expects to receive a HELLOVERIFYREQUEST message (type 3) back from the server within a reasonable amount of time.[2] Note that the HELLOVERIFYREQUEST message type is new and DTLS-specific and that it has not been used in SSL/TLS so far (because these protocols don't have to handle message loss). If the client does not receive such a HELLOVERIFYREQUEST message before a timeout occurs, then it knows that either the CLIENTHELLO or the HELLOVERIFYREQUEST message was lost. It then retransmits the CLIENTHELLO message to the server. The server also maintains a timer and retransmits the message when its timer expires. The DTLS protocol specification recommends a one-second timer (to improve latency for real-time applications).

4.2.2.3 Cookie Exchange

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 at the victim.
- 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.

To mitigate these attacks, the DTLS protocol uses a *cookie exchange* that has also been used in other network security protocols, such as the Photuris [9], which is actually a predecessor of the IKE protocol. Before the proper handshake begins, the server must provide a stateless cookie in the HELLOVERIFYREQUEST message, and the client must replay it in the CLIENTHELLO message in order to demonstrate

2 Note that the HELLOVERIFYREQUEST message is not to be included in the MAC computation for the CERTIFICATEVERIFY and FINISHED messages.

that it is capable of receiving packets at its 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, this maximum cookie size is even enlarged in DTLS 1.2 to 255 bytes.

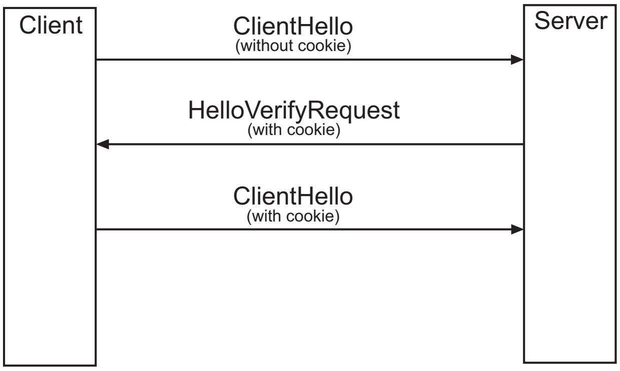

Figure 4.4 The cookie exchange mechanism used by the DTLS handshake protocol.

The cookie exchange mechanism used by the DTLS handshake protocol is illustrated in Figure 4.4. First, the client sends a DTLS CLIENTHELLO message without a cookie to the server. The server then generates a cookie for this particular client and sends it to the client in the HELLOVERIFYREQUEST message (see above). 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 and is identical to the SSL/TLS protocols (up to version 1.2). This means that the protocol continues with a SERVERHELLO message and that also 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. Only if it receives a valid cookie (under this address) can it actually 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. So an adversary can still mount a DoS attack from his or her 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 generally very simple to mount in practice.

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) and that clients must be prepared to do a cookie exchange with every DTLS handshake.

4.3 SECURITY ANALYSIS

Given the fact that the DTLS protocol was specifically designed to be as similar as possible to the TLS protocol, one can reasonably argue that the security analyses of TLS, at least in principle, also apply to DTLS. So people generally have a good feeling about the security of the DTLS protocol. (This is particularly true if an AEAD cipher suite is in use.) It is, however, still a feeling, and fairly little is known about the real security of the DTLS protocol.

From all attacks that can be mounted against the SSL/TLS protocols (cf. Sections 2.4 and 3.8), there are some attacks that also 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 in DTLS. With regard to padding oracle attacks, for example, the protection mechanisms that have been built into TLS 1.2 are also present in DTLS 1.2, and hence one can reasonably expect DTLS 1.2 to be resistant against them. Unfortunately, this is not always the case, and there are still a few specifically crafted padding oracle attacks that can be used to recover plaintext. For example, in 2012, some padding oracle attacks against two DTLS implementations that are in widespread use (i.e., OpenSSL and GnuTLS) were demonstrated [10].[3] 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 error message is generated and sent back to the sender. This means that an adversary can exploit neither the type of error message nor its timing behavior, and this, in turn, suggests that the padding oracle attacks that have been mounted against SSL/TLS in the past do not automatically also work against DTLS. However, in the work mentioned above, it was shown that

3 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.

the lack of error messages can be compensated with Heartbeat messages. As argued in Section 3.4.1.13, 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 error message (to evaluate the 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 respective Heartbeat response message reveals some information about the amount of computation that is done on the server side. If the padding is invalid, then the server has comparably little to do (in particular, because MAC verification is not performed in this case). If, however, the padding is valid, then the record is decrypted and the MAC is verified. This takes some time that is noticeable in many cases. If, instead of sending only one record, multiple identical records are sent in sequence,[4] then the amount of work to be done by the server is multiplied. Needless to say, the use of multiple identical records amplifies the timing difference, and hence this makes the attack more feasible. In either case, the fact that an invalid padding does not terminate a connection in the case of DTLS simplifies the attack considerably. 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. It will be interesting to see how padding oracle attacks against DTLS are going to evolve and improve over time. The use of Heartbeat messages is just one possibility to implement and take advantage of a side channel. Many more side channels are likely to exist, literally waiting for their exploitation.

Maybe the biggest worry regarding 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 and that there may be entirely new attacks that exploit this fact. In network security, it is well known that UDP-based applications are harder 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 very likely that research will address this question, and it will be interesting to learn the answers. In the meantime, we live with a good feeling and assume that the DTLS protocol provides a reasonable level of security. Keep in mind that this feeling is not really justified and that the DTLS protocol still needs to be thoroughly analyzed in terms of security.

4 The paper in which the attack is described uses the term *train of packets* to refer to these multiple records. For the purpose of this book, however, we don't use this terminology.

4.4 FINAL REMARKS

This chapter elaborates on the DTLS protocol that represents the UDP version of the SSL/TLS protocols. The differences are minor and are mainly due to the characteristics of UDP, being a connectionless best-effort datagram delivery protocol that operates at the transport layer. This requires some minor modifications in the DTLS record and handshake protocols. The modifications 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 4.3, this argument should be taken with a grain of salt. It may be the case that some entirely new security weaknesses and problems will be revealed in the future. This is always possible, but in the case of the DTLS protocol it is more probable, because the security of the DTLS protocol hasn't 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 new and have not been thoroughly analyzed and tested so far. 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, for example, to determine the optimal values and backoff strategies. This is another research area that deserves some study (in addition to security). As of this writing, it is absolutely not clear what values and backoff strategy values are optimal for the deployment of DTLS. The same is true for the firewall traversal of the DTLS protocol. As we will see in Chapter 5, many firewall technologies are well-suited for TCP-based applications and application protocols, but they are less well-suited for UDP-based applications and application protocols. Consequently, the secure firewall traversal of the DTLS protocol is another research area that holds some interesting challenges for the future. We scratch the surface in Chapter 5.

References

[1] Kohler, E., M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," Standards Track RFC 4340, March 2006.

[2] Stewart, R. (ed.), "Stream Control Transmission Protocol," Standards Track 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),” Standards Track RFC 5238, May 2008.

[6] Tuexen, M., R. Seggelmann, and E. Rescorla, “Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP),” Standards Track RFC 6083, January 2011.

[7] Rescorla, E., and N. Modadugu, “Datagram Transport Layer Security,” Standards Track RFC 4347, April 2006.

[8] Rescorla, E., and N. Modadugu, “Datagram Transport Layer Security Version 1.2,” Standards Track RFC 6347, January 2012.

[9] Karn, P., and W. Simpson, “Photuris: Session-Key Management Protocol,” Experimental RFC 2522, March 1999.

[10] 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.

Chapter 5

Firewall Traversal

In spite of the fact that firewalls are omnipresent today, their use and interplay with the SSL/TLS protocols remains tricky and somehow contradictory. On one hand, the SSL/TLS protocols are used to provide end-to-end security services and hence to establish secure end-to-end connectivity, but on the other hand, firewalls are used to restrict or at least control this end-to-end connectivity. It is, therefore, not obvious if and how the SSL/TLS protocols can effectively traverse a firewall. 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, and concludes with some final remarks in Section 5.4. 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 messaging and streaming media protocols. (See, for example, [1] for a respective overview.) This chapter is not meant to be comprehensive and only briefly scratches the surface of the topic. 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* for short. For example, according to RFC 4949 [2], 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 very precise.

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 [3]:

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 grades 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 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 security policy is pointless in practice (because it tends to get holey 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 applicaton layer. In some literature, the former are called *circuit-level gateways*, whereas the latter are called *application-level gateways* [3]. 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 decentrally, typically at the desktop level. 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., [4–6]). 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. Such a firewall is 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

1 Dynamic packet filtering is also known as *stateful inspection*.

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.

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 troublesome if the existence of the proxy server is not visible to the user.

In general, different application protocols have different requirements for proxy servers. On a high level of abstraction, an application protocol can either be proxied or tunneled through a proxy server, described as follows:

- 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 many "normal" network applications, such as Telnet, FTP, SMTP, and—most importantly for the purpose of this book—HTTP.
- Contrary to that, we say that an application protocol is *tunneled* when we mean that the corresponding proxy server (which is basically acting 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 between the client and the 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 is possible for a full-fledged proxy server. Examples of protocols that are sometimes tunneled by proxy servers or circuit-level gateways include proprietary protocols, protocols for which a proxy server is currently not available, as well as protocols that are protected by cryptographic security protocols like SSL/TLS. This includes, for example, HTTPS.

With regard to the SSL/TLS protocols, the two possibilities itemized above are illustrated in Figure 5.1. Figure 5.1(a) shows SSL/TLS tunneling, whereas Figure 5.1(b) shows SSL/TLS 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.

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

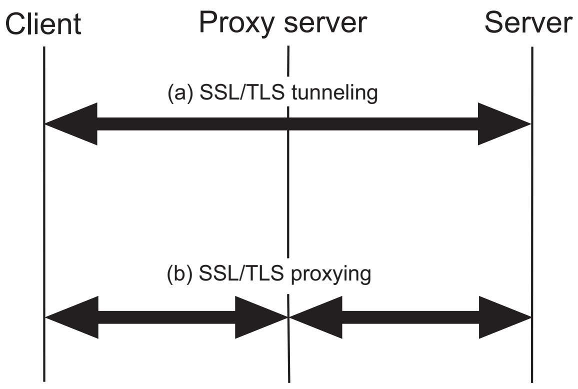

Figure 5.1 SSL/TLS (a) tunneling and (b) proxying.

two SSL/TLS connections—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. In SSL/TLS tunneling, the proxy server is passive in the sense that it 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 the data transmission. Needless to say, this has some severe security and privacy implications.

In the past, it has been been common practice for companies and organizations to tunnel outbound SSL/TLS connections and proxy inbound SSL/TLS connections. This practice, however, is about to change as the deployment settings are getting more involved. Let us now more thoroughly address SSL/TLS tunneling and SSL/TLS proxying.

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. (See, for example, [7] for an early reference.) 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 proxies—is part of the HTTP specification and hence documented in RFC 2817 [8]. So additional specification is no longer required.

In short, SSL/TLS tunneling allows a client to open a secure tunnel through an HTTP proxy server that resides on a firewall. 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 for them to service and to provide proper error notification. As such, SSL/TLS tunneling is not really SSL/TLS-specific. Instead, it is a general mechanism to have a third party establish a connection between two end points, 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 specifiy the hostname 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
```

This example requires an SSL/TLS-enabled web server running at port 443 (default value) of `www.esecurity.ch`. This server does not exist, but the example may still give you an idea.

The message is sent out 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 the server that represents `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 actually perform an SSL/TLS handshake.

SSL/TLS tunneling can also be combined with the “normal” authentication and authorization mechanisms employed by an HTTP proxy server. 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 request user credentials. The corresponding HTTP response message may begin with the following two lines:

```
HTTP/1.0 407 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 above). 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 (containing the authentication information). The corresponding HTTP request message may begin with the following two lines:

```
CONNECT www.esecurity.ch:443 HTTP/1.0
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 this server respectively.

Note that the CONNECT method provides a lower level function than many other HTTP methods. Think of it as some kind of an "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 hostname 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 the proxy server configuration should therefore explicitly limit allowed (tunneled) connections to well-known SSL/TLS ports, such as 443 for HTTPS (or any other port number assigned by the IANA).

SSL/TLS tunneling is supported by almost all commercially or freely available HTTP clients and proxy servers (as they support HTTP and the respective CONNECT method by default). However, the end-to-end characteristic of SSL/TLS tunneling also comes along with a few disadvantages. First and foremost, if an

HTTP proxy server supports SSL/TLS tunneling, then the end-to-end characteristic of SSL/TLS tunneling prohibits the proxy server from doing content screening and caching in some meaningful way. Similarly, the HTTP 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 (especially 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 HTTPS, 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 actually transmitted.

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 many 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 that acts as an SSL/TLS wrapper and does the SSL/TLS processing on the original server's behalf. An exemplary software that implements this idea and is widely deployed is *stunnel*.[3]

5.3 SSL/TLS PROXYING

As its name suggests, a server proxies SSL/TLS in SSL/TLS proxying (instead of establishing a tunnel). This means that the proxy server must understand the SSL/TLS protocols and terminate each connection that passes the proxy server. So if a user is not employing SSL/TLS tunneling but invokes SSL/TLS proxying instead, then the following four-step procedure must be performed:

- First, the user client establishes 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).

3 http://stunnel.mirt.net.

- Fourth, the proxy server proxies 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 can occur. This makes everything transparent to the proxy server and firewall, but it also means that end-to-end security cannot be achieved and that the proxy server represents a MITM. Due to its ability to intercept data traffic, an SSL/TLS proxy server is sometimes also called an *iterception proxy*. It can serve good or bad purposes, depending on who is actually running the proxy. Needless to say, an iterception proxy has a huge potential for misuse. It is therefore ultimately important to trust the operator of the proxy.

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[4] for the SSL/TLS connection.

To the best of our knowledge, the first SSL/TLS proxy server or gateway was developed by a group of researchers at the DEC Systems Research Center in 1998. They basically used a combination of SSL client authentication (at the inbound proxy) and URL rewriting techniques in a technology called *secure web tunneling*[5] [9]. A similar technology to access internal web servers was developed and complemented with a one-time password system by a group of researchers at AT&T Laboratories [10]. Since these early days of SSL/TLS proxying, many companies and organizations have developed SSL/TLS proxy servers and iterception proxies sometimes with additional features. A proxy server that is widely deployed is Squid.[6]

An interesting situation occurs in the realm of *content delivery networks* (CDNs), such as the ones provided by companies like Akamai or CloudFlare: If an origin server is operated inside a CDN, then it make 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. 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 (i.e., from attacks that originate from the Internet). So a CDN

4 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).

5 Note that, in spite of its name, the technology refers to SSL/TLS proxying and not tunneling.

6 http://www.squid-cache.org.

must always enforce SSL/TLS proxying, and 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 them. For obvious reasons, the first possibility is simpler and more straightforward to deploy. However, it poses the origin servers' private keys at higher risks. This is why most CDNs have started to 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 in 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 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 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 realm of CloudFlare, for example, such a feature has been known as *Keyless SSL*. Other CDN providers use other terms to refer to the same idea and architecture.

5.4 FINAL REMARKS

This chapter addresses 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 that is sent back and forth. In fact, there are many *web application firewalls* (WAFs) that basically represent SSL/TLS proxy servers with many additional functionalities and features. The design and implementation of WAFs that are resistant to contemporary attacks is a timely and fast evolving research area.

Today, most companies and organizations use SSL/TLS tunneling for oubound connections and SSL/TLS proxying for inbound connections. However, due to the huge amount of content-driven attacks (e.g., malware attacks), this general approach is about to change. In fact, many security professionals opt for proxying outbound SSL/TLS connections, as well. But there is a caveat to mention here: If all SSL/TLS connections are proxied, then this also applies to employees running inherently private applications, such as Internet banking. Such applications typically require clients to connect to trustworthy servers on an end-to-end basis. It is simply not tolerable to have any proxy server interfere with the communications. As there are only a few applications and servers of this type, it is perfectly fine to white-list them and to support SSL/TLS tunneling only for them.

Due to the connectionless and best-effort nature of UDP, making the DTLS protocol traverse a firewall is conceptually more challenging than SSL/TLS. In particular, proxy-based firewalls do not natively work for UDP, and hence it is not at all 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 (which can be expected), then these techniques are likely to become more important in the future.

References

[1] Johnston, A.B., and D.M. Piscitello, *Understanding Voice over IP Security*. Artech House Publishers, Norwood, MA, 2006.

[2] Shirey, R., "Internet Security Glossary, Version 2" Informational RFC 4949 (FYI 36), August 2007.

[3] Cheswick, W.R., and S.M. Bellovin, "Network Firewalls," *IEEE Communications Magazine,* September 1994, pp. 50–57.

[4] Zwicky, E.D., S. Cooper, and D.B. Chapman, *Building Internet Firewalls*, 2nd edition. O'Reilly, Sebastopol, CA, 2000.

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

[6] 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.

[7] Luotonen, A., and K. Altis, "World-Wide Web Proxies," *Computer Networks and ISDN Systems,* Vol. 27, No. 2, 1994, pp. 147–154.

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

[9] Abadi, M., et al., "Secure Web Tunneling," *Proceedings of 7th International World Wide Web Conference,* Elsevier Science Publishers B.V., Amsterdam, the Netherlands, 1998, pp. 531–539.

[10] 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.

Chapter 6

Public Key Certificates and Internet Public Key Infrastructure (PKI)

Previous chapters have emphasized that the SSL/TLS protocols yet require public key certificates but that the management of these certificates is not addressed in the respective protocol specifications. This basically means that the management of these certficates must be handled outside the scope of SSL/TLS, and that a PKI for the Internet is needed for this purpose. 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 several new approaches in Section 6.6, and conclude with some final remarks in Section 6.7. We already mentioned in the preface that the topic is important and that the respective chapter is comprehensive and long. There are even more things to say, and readers who want to get more information are referred to [1–3] or the many papers and articles that have been written in the past.

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 are called *public key certificates*. They are used by many cryptographic security protocols, including SSL/TLS and DTLS. Again referring to RFC 4949, 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 (i.e., 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. This more general class of certificates is commonly referred to as *attribute certificates* (ACs). In the realm of SSL/TLS, we have already come across ACs when we discussed the authorization extensions of TLS in Section 3.4.1.8. Here, we want to add a few things related to ACs that are independent from SSL/TLS. According to RFC 4949, 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 recognized and trusted by a community of users. These authorities are described as follows.

- In the case of public key certificates, the authorities are called *certification authorities* (CAs[1]) 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 the assessment of a CA or CSP.
- In the case of attribute certificates, the authorities are called *attribute authorities* (AAs).

It goes without saying that an CA and an AA may in fact be 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 the principals that become certificate owners. In addition, the RA may also be involved in tasks like token distribution, certificate revocation

1 In the past, CAs were sometimes called TTPs. This is particularly true for CAs that are operated by government bodies.

reporting, key generation, and key archival. In fact, a CA can delegate some of its authorities (apart from certificate signing) to an RA. Consequently, RAs are optional components that are transparent to the users. Also, the certificates that are generated by the CAs may be made available in online directories and certificate repositories.

In short, a PKI consists of one (or several) CA(s). According to RFC 4949, 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, a PKI 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.

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, most of these service providers have not been commercially successful and have gone out of business. The ones that have survived make their living from other businesses and subsidize their PKI activities.

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. (The corresponding certificates are further addressed in 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.[2]

In 1995, the IETF recognized the importance of public key certificates, and chartered a now concluded IETF Public-Key Infrastructure X.509 (PKIX[3]) 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. (In fact, the output of this WG has been tremendously large.) It has been closely aligned with the respective activities within the ITU-T. In spite of the practical importance of the specifications of the IETF PKIX WG (especially

2 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.

3 http://datatracker.ietf.org/wg/pkix/charter/.

regarding CP/CPS), we do not delve into the details in this book. The work is documented in a series of respective RFC documents.

As illustrated in Figure 6.1, 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 the public key in the first place. The public key, in turn, can be from any public key cryptosystem, such as RSA, Elgamal, Diffie-Hellman, DSA, ECDSA, or anything else. The format (and hence also the size) of the public key depends on the cryptosystem in use.

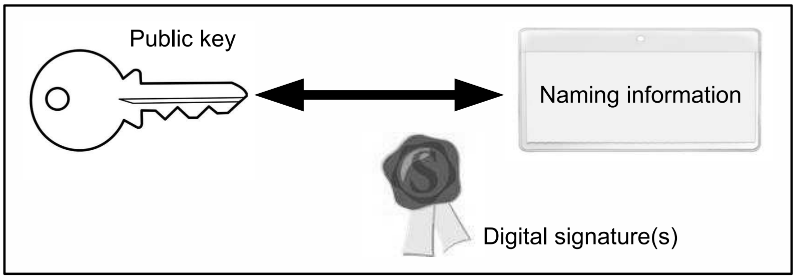

Figure 6.1 A public key certificate comprising three pieces of information.

The *naming information* is used to identify the owner of the public key and 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 the 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 *distinguished name* (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] sprang 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 on January 29, 1997, to produce a certificate infrastructure and operating procedure to meet the needs of the Internet community for trust management in a way 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 abandoned in 2001.[4] 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. They hardly play a role in today's discussions about the management of public key certificates. However, the underlying argument that globally unique namespaces are not easily available 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. In Figure 6.1, this is illustrated by the two arrowheads that bind the two pieces 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.

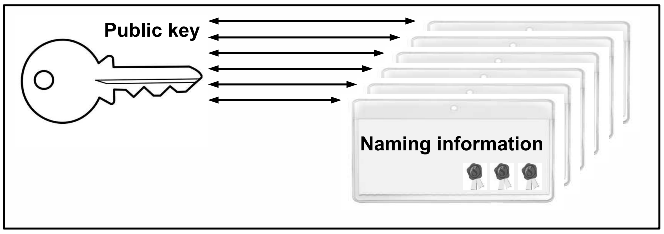

Figure 6.2 The general format of an OpenPGP public key certificate.

4 The WG was formally concluded on February 10, 2001, only four years after it was chartered.

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, described as follows.

- 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 this binding. This more general format of an OpenPGP public key certificate is illustrated in Figure 6.2 (without further explanation).
- With regard to the trust models, we first mention that such a 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. In addition to the direct trust model, there is a cumulative trust model employed by OpenPGP and a hierarchical trust model employed by ITU-T X.509.

In practice, X.509 certificates clearly dominate the field, especially in the realm of SSL/TLS. Keep in mind, however, that there is a specific TLS extension (i.e., the `cert_type` extension) that allows a client and a server to negotiate the use of OpenPGP certificates (cf. Section 3.4.1.9). But due to their dominance in the field, we focus on X.509 certificates here.

6.2 X.509 CERTIFICATES

As mentioned above (and as their name suggests), 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. It specifies both a certificate format and a certificate distribution scheme. The specification language used in the recommendation is ASN.1. (See, for example, the appendix of [13] for a brief summary of 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 Internet. The resulting X.509 version 3 (X.509v3) specification has since then 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 have a closer look at the X.509 certificate format and the 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 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 any additional extension fields are possible):[5]

- *Version:* This field is used to specify the X.509 version in use (i.e., version 1, 2, or 3).
- *Serial number:* This field is used to specify a serial number for the certificate. The serial number is 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 object identifier (OID) of the algorithm that is used to digitally sign the certificate. For example, the OID 1.2.840.113549.1.1.5 refers to `sha1RSA`, which stands for 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).

5 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 e-mail 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 [i.e., 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 certificate revocation lists (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.[6]

6 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 looks like at first sight.

> 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 basic purposes indicated in the key usage extension field.

The last three fields make X.509v3 certificates very flexible, but also very difficult to deploy in an interoperable manner. Anyway, the certificate must come along with a digital signature that conforms to the digital signature algorithm specified in the Algorithm ID field.

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 7.3, 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 is self-signed, meaning that the issuer and subject fields are the same and that they must be trusted by default. Note that 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 6.3, 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 particular piece of software (which may be an operating system or some application software). For example, all major vendors of operating systems have programs to officially include a CA in the list of trusted root CAs shipped with the software. (This point is further addressed in Section 6.3.) The same is true for many application software vendors, such as Adobe. The respective software can then be configured to use the operating system’s list of root CAs, its own list, or even both. So there is a lot of flexibility that is sometimes beyond the capabilities of software users.

In the hierarchy of trusted CAs, each root CA may issue certificates for other CAs that are then called *intermediate CAs*. The 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 issue certificates for end users or other entities, such as web servers. These certificates are sometimes called *leaf certificates*. They have a parameter setting (with regard

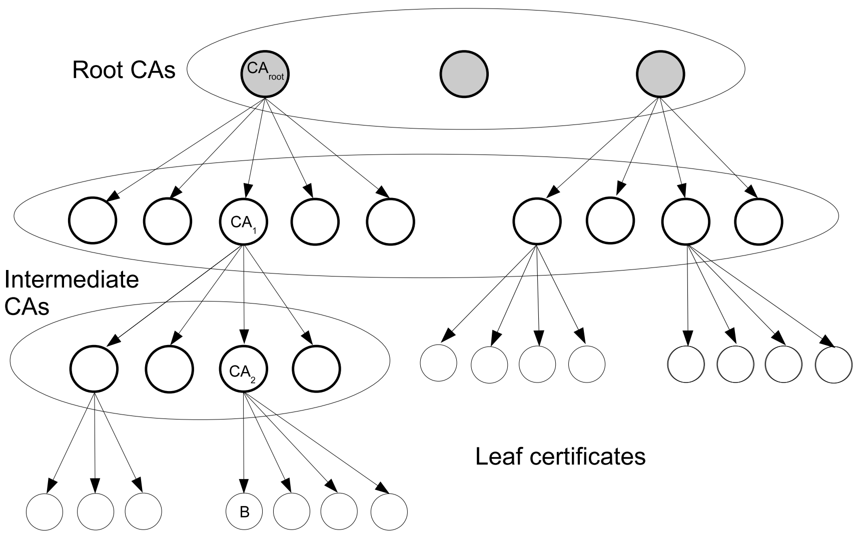

Figure 6.3 A hierarchy of trusted root and intermediate CAs that issue leaf certificates.

to the basic constraints extension) that makes sure that they cannot be used to issue other certificates.

In a typical setting, a commercial CSP operates a root CA and several subordinate CAs that represent intermediate CAs. Note, however, that not all client software makes a clear distinction between a root CA and an intermediate CA. In the case of web browsers, for example, Microsoft Internet Explorer and all other browsers that rely on the certificate management functions of the Windows operating system (e.g., Google Chrome) clearly distinguish between a root CA and an intermediate CA, and this distinction is also reflected in the user interface. Some other browsers, such as Mozilla Firefox, do not make this distinction and only accept root CAs to bootstrap trust.

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 *certification chain*) from one of the root certificates (or a certificate from an intermediate CA if such a CA exists) to the leaf certificate. Formally speaking, a certification path or chain is defined as a sequence of certificates that leads from a trusted certificate (of a root or intermediate CA) to a leaf certificate. Each certificate certifies the public key of its successor. Finally, the leaf certificate is typically issued for a person or end system. 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 be represented as follows:

$$\begin{aligned}
&CA_{root} \ll CA_1 \gg \\
&CA_1 \ll CA_2 \gg \\
&CA_2 \ll CA_3 \gg \\
&\ldots \\
&CA_{n-1} \ll CA_n \gg \\
&CA_n \ll B \gg
\end{aligned}$$

In this notation, $X \ll Y \gg$ is a certificate that has been issued by X for the public key of Y. There are many subtleties that are omitted here, so the focus is entirely on the chain. Figure 6.3 illustrates a certification path with two intermediate CAs. The path consists of $CA_{root} \ll CA_1 \gg$, $CA_1 \ll CA_2 \gg$, and $CA_2 \ll B \gg$. If a client supports intermediate CAs, then it may be sufficient to find a sequence of certificates that lead from a trusted intermediate CA's certificate to the leaf certificate. This may shorten certification chains considerably. In our example, it may be the case that CA_2 is an intermediate CA that is already trusted. In this case, the leaf certificate $CA_2 \ll B \gg$ would be sufficient to verify the legitimacy of B's public key.

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).[7] In such a general structure (or mesh), a certification path may not be unique and multiple certification paths may coexist. In this situation, it is required to have authentication metrics in place that allow one to handle multiple certification paths. The design and analysis of such metrics is an interesting research topic that is not further addressed here.

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

7 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).

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 *certificate revocation list* (CRL). In essence, a CRL is a black list that enumerates all certificates (by their serial numbers) that have been revoked. 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 grow 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 large and 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* as specified in RFC 2560 [15]. The OCSP is conceptually very simple: an entity using a certificate issues a request to the certificate-issuing entity (or its OCSP responder, respectively) to find out whether the certificate is still valid. In the positive case, the responder sends back a response that basically says "yes." In the negative case, however, things are more involved, and there are a few security problems that then pop up (as further addressed in Section 6.5). So OCSP works fine in theory, but it has a few problems and subtleties to consider in practice. Some of these problems and subtleties can be addressed with OCSP stapling, which is briefly introduced in Section 3.4.1.6. Remember that there is even an explicit TLS extension dedicated to OCSP stapling.

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 contruction 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. This is partly because OCSP stapling can be used in these cases. SCVP is therefore not further addressed in this book.

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 since SSL 3.0—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 that has been issued to the server. If the verification succeeds, then the server certificate is directly accepted by the client. Otherwise (i.e., if the verification fails), then the server certificate cannot be directly accepted by the client. In this case, the client must either refuse the connection or inform the user and ask him or her whether he or she wants to accept the certificate (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.

In order to avoid user dialogs and corresponding inconveniences, SSL/TLS-enabled web servers are usually equipped and configured with a certificate that is issued by a commonly trusted root CA or a subordinate CA. Subordinate CAs are not root CAs, and hence their certificates are not self-signed. Instead, they are part of a certification hierarchy and obtain their certificates either directly from a root CA or from another subordinate CA that is then called intermediate CA. So while subordinate CAs are issuing CAs, meaning that they issue certificates to end entities, intermediate CAs issue certificates only to other (subordinate) CAs.

As mentioned above, most browser manufacturers have a program for root CAs (and sometimes also for intermediate CAs) to include their certificates in their browsers' lists of trusted CAs (sometimes called "certificate stores" or "trust stores"). The respective programs are called *root certificate programs* or something similar. All big software vendors have root certificate programs, including, for example, Microsoft,[8] Mozilla,[9] and Apple.[10] While Apple's certificate store is relatively closed, the certificate stores of Microsoft and Mozilla are open to other vendors. So these vendors sometimes rely on these certificate stores. Software provided by Adobe can, for example, usually be configured to either use the certificate stores of Microsoft or Mozilla (depending on the platform that is in use), or use an entirely different (separate) certificate store. It goes without saying that this choice is difficult to make for a casual user.

The root certificate programs that are available are 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 the European Telecommunications Standards Institute (ETSI) [17, 18] and the ISO [19]. There is an audit program called WebTrust that specifically

8 https://technet.microsoft.com/en-us/library/cc751157.aspx.
9 https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/policy.
10 https://www.apple.com/certificateauthority/ca_program.html.

targets the high-end market.[11] As is usually the case, a CA that is accepted by one root certificate program will also be accepted by other root certificate programs.

In spite of the fact that certificate stores nowadays comprise dozens or hundreds of trusted CAs, the market for server certificates is still dominated by only a few internationally operating CSPs. Most importantly, there is Comodo,[12] Symantec (formerly known as VeriSign[13]), Thawte,[14] and GeoTrust[15] with its subsidiary RapidSSL.[16] 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 affilated 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 equally valid.
- *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 make 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.[17] As its name suggests, the CA/Browser Forum consists of commercial CA operators, 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. EV certificates tend to be a little bit more expensive than their DV and OV counterparts, but except for that there is hardly any disadvantage for the service providers that want to show their high security standards to their customers. They are therefore widely deployed, and

11 http://www.webtrust.org.
12 https://www.comodo.com.
13 https://www.verisign.com.
14 https://www.thawte.com.
15 https://www.geotrust.com.
16 http://rapidssl.com.
17 http://www.cabforum.org.

all major browsers have specific indicators in their GUI to make sure that users are aware that an EV certificate is put in place.

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 prohibitive.

In addition to DV, OV, and EV certificates, there are two other types of certificates that need to be mentioned here:

- Section 2.2.2.4 introduces the notion of *International Step-Up* and *SGC certificates* as employed by Netscape and Microsoft 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 International Step-Up and SGC 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. Nevertheless, there are still a few CAs that continue to sell SGC certificates. Consequently, there seems to be a market for these types of certificates, and hence not everybody is using a browser that natively supports strong cryptography. This is surprising, especially in times in which we have to consider the FREAK attack (cf. Section 3.8.5).

- 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 and every FQDN). In addition to higher costs, this means that many certificates must be managed simultaneously. To improve 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 very widely used. 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[18]). In 2015, the ISRG launched an initiative[19] that was prominently named "Let's Encrypt." The aim of the initiative is to enable a more straightforward transition from HTTP to HTTPS, and hence to provide an automated certificate management environment (ACME) that allows a web service provider to receive and automatically install a DV certificate for free. As of this writing, the initiative has just started, and it is too early to tell whether the initiative has been well received in the marketplace and will be successful in the long term. For many nonprofit organizations, ISRG-issued certificates provide a viable alternative for the server certificates they need.

6.4 CLIENT CERTIFICATES

From a theoretical perspective, there is no major difference between a server certificate and a client certificate. While the former is issued to a FQDN or a domain (in the case of a wildcard certificate), the latter is typically issued to a user. The user then employs the client certificate when he or she uses SSL/TLS to connect to a server that mandates certificate-based client authentication. (Remember from our previous discussions that client authentication is optional in SSL/TLS.) So the fields of a server certificate and a client certificate are essentially the same—only the contents differ.

18 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.

19 https://letsencrypt.org.

The major difference between a server certificate and a client certificate is the way they are issued: While the former is typically issued by an internationally operating trusted CA (as discussed above), the latter can be issued by any locally operating CA that is trusted by the server(s). This makes the issuance of client certificates conceptually simple and straightforward. However, there is also a scalability issue to consider: While there are usually many clients that need to be equipped with a certificate, there is usually only one or a few servers that need a certificate. For example, if we consider an Internet bank that wants to employ certificate-based client authentication in SSL/TLS, then literally all customers of the bank need to be equipped with a certificate. If the bank is large, then we are talking about a couple hundred thousand or even millions of certificates that need to be rolled out. This is not a simple task (to say the least).

There are many CSPs that sell client certificates. The companies mentioned above are clearly in this business, but there are also a few other companies that try to compete. Most of these companies are local and only serve a state or country.

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 by specific attacks. For example, in 2009, Moxie Marlinspike showed that the ASN.1 encoding of X.509 is ambiguous in the sense that it supports multiple string representations and formats (including PASCAL and C) and that this ambiguity can be exploited in an attack to gain valid certificates for arbitrary domains issued by a trusted CA.[20] Later in 2009, Marlinspike revealed another vulnerability related to OCSP: When a client requests the revocation status of a particular certificate, then the OCSP server sends back a response status value for the certificate. Ideally, this value suggests that the certificate is either valid or not. However, for operational reasons, OCSP supports many other response status values, including for example, a value that essentially says "try later" (value 3). When a client receives such a response, the typical reaction is to accept the certificate. It goes without saying that this reaction is not in line with the reaction one would expect from a security viewpoint. So the large-scale use of OCSP in the field may raise some security concerns.[21] In a similar line of research, a group of researchers have found 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

20 http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf.

21 http://www.thoughtcrime.org/papers/ocsp-attack.pdf.

validation process [20].[22] This type of automated adversarial testing has turned out to be successful as well.

More recently (and maybe most importantly), many commercially operating CAs have come under attack [21]. 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 have shocked the community and have been widely discussed in the media:

- In March 2011, at least two Italian Comodo resellers[23] 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 shown by an interim report[24] and an independent final report.[25]

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 [22], but there was little evidence that such attacks really occured in practice. This changed fundamentally, and, more recently, at least one solution provider for a data loss prevention (DLP) system has received a CA certificate from Trustwave that allows it to issue valid certificates for any server of its choice.[26] To make things worse, the same attack vectors that work for

22 The authors of [20] coined the term *frankencerts* to refer to these certificates.

23 GlobalTrust (http://www.globaltrust.it) and InstantSSL (http://www.instantssl.it).

24 http://cryptome.org/0005/diginotar-insec.pdf.

25 http://www.rijksoverheid.nl/documenten-en-publicaties/rapporten/2012/08/13/black-tulip-update.html.

26 Trustwave has since revoked the certificate and vowed to refrain from issuing such certificates in the future.

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. Unfortunately, there are only a few technologies that can be used to protect applications against MITM attacks, such as mutually authenticated SSL/TLS sessions (using client-side certificates) or SSL/TLS session-aware user authentication [23]. More research and development is required here. Fraudulently issued certificates 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 [24] 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 almost endless.

In the sequel, we neither address the specifics of these attacks nor do we discuss the appropriateness of the term *APT*. Instead, we use a probability-theoretic observation to make clear that similar attacks will occur again and again (unless one changes the underlying paradigms) and that respective countermeasures must be designed, implemented, and put in place. These countermeasures must go beyond browser add-ons to improve SSL/TLS certificate scrutiny, like Certificate Patrol[27] or Certlock[28] for Mozilla Firefox.

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, 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,

$$\Pr[CA_i \text{ is not compromised}] = 1 - p_i,$$

27 https://addons.mozilla.org/de/firefox/addon/certificate-patrol.
28 http://code.google.com/p/certlock.

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:

$$\begin{aligned}
\Pr[\text{no CA is compromised}] &= \Pr[CA_1 \text{ is not compromised}] \cdot \\
&\quad \Pr[CA_2 \text{ is not compromised}] \cdot \\
&\quad \ldots \cdot \\
&\quad \Pr[CA_n \text{ is not compromised}] \\
&= (1-p_1)(1-p_2)\ldots(1-p_n) \\
&= \prod_{i=1}^{n}(1-p_i)
\end{aligned}$$

Hence, the probability that at least one CA is compromised is the complement of this expression:

$$\begin{aligned}
\Pr[\text{at least one CA is compromised}] &= 1 - \Pr[\text{no CA is compromised}] \\
&= 1 - \prod_{i=1}^{n}(1-p_i)
\end{aligned}$$

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 percent. 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.) Some recent incidents related to another Dutch CA named KPN[29] and the Malaysian CA Digicert[30] bear witness to this claim [although the Digicert case is slightly different, because one has found certificates for key lengths that are simply too short (i.e., 512 bits)]. 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

29 http://www.kpn.com.
30 http://www.digicert.com.my.

subordinate or intermediate (trusted) CAs. The overall situation is dangerous and worrisome. Consider the following:

- First, all properly managed certificate revocation mechanisms, like CRLs or OCSP requests and responses, are capable of addressing the problem of a fraudulently issued certificate, once the fraud is discovered. They implement a “black list” approach, meaning that they aim at identifying certificates that have been black listed in the first place. However, a certificate that is fraudulently issued does not automatically appear on a black list. Instead, it looks perfectly fine and valid, and hence any questions of whether the certificate has been revoked need 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 dealt with so far, and we think that this omission needs be repaired in future certificate revocation mechanisms.
- 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 considerably 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 any 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 term “certificate legitimation” [21]. Certificate legitimation is about who is legitimately authorized to issue and revoke certificates for a given entity (e.g., web site 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 widely used (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 occur, certificate legitimation is getting more and more important and key to the security of the Internet PKI and the applications that depend on it. There are a few new approaches that can be used for certificate legitimation. These approaches are overviewed and briefly discussed in Section 6.6.

6.6 NEW APPROACHES

As mentioned above, it is more appropriate to follow a “white list” approach to address the legitimacy of a certificate. In such an approach, any certificate that has been properly ordered and delivered and for which the appropriate certification fees have been paid is included in a white list (i.e., a list of legitimate certificates). If somebody were trying to compromise a CA and fraudulently issue 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 a “white list” approach works better and provides more security than a “black list” approach. In the real world, for example, one can implement border control with either approach (or a combination of the two):

- In a “black list” approach, the agency in charge of border control has a list of identities of persons who are not allowed to pass the border (and enter the country, respectively). Everybody else is allowed to pass.
- In a “white list” approach, the agency has a list of identities of persons who are allowed to pass the border. Everybody else is not allowed to pass.

It goes without saying that the “black list” approach 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 the “white list” approach. In a real setting, the two approaches are typically combined: a black list for known terrorists and criminals and a white list 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 the 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 validated and checked. This is not the case with certificates.

Similarly, in the realm of firewall design, the "black list" approach refers to a default-permit stance (i.e., everything is allowed unless it is explicitly forbidden), whereas the "white list" approach refers to a default-deny stance (i.e., everything is forbidden unless it is explicitly allowed). Here, it is undisputed and commonly agreed that a default-deny stance is more secure, but 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 white lists in many settings, the PKI community has traditionally believed that the disadvantages of a "white list" approach outweigh their advantages, and hence that a "black list" approach is more appropriate. This belief is, for example, evident in the discussions on the IETF PKIX WG mailing list. In the aftermath of the Comodo and DigiNotar attacks, however, this design decision was revisited, and people nowadays have different priorities. In late 2011, for example, Google launched a respective project[31] named *Certificate Transparency* and acronymed CT [25, 26].[32] The preliminary result of the CT project was a framework for auditing and monitoring public key certificates that is further refined within the public notary transparency (trans) working group of the IETF security area. The aim is to come up with an RFC document that can be submitted to the Internet standards track (note that [26] is only an experimental RFC). The basic idea of CT is to provide a log of legitimately issued certificates that can be verified by everybody. The log itself is based on an append-only data structure known as a binary Merkle hash tree [27]. Every legitimately issued certificate (or the SHA-256 hash value thereof, respectively) is appended as a leaf to the hash tree, where it is automatically authenticated. The Merkle-tree logs are stored on a relatively small number of servers. Every time a CA issues a new certificate, it sends a copy to all log servers, which then return a cryptographic proof that the certificate has been added to the log. Browsers can be preconfigured with a list of log servers. (This does not make the lists of trusted CAs obsolete, but rather complements them.) Transparency is only useful if it is periodically checked whether the participants play

31 http://www.links.org/files/CertificateAuthorityTransparencyandAuditability.pdf.

32 http://www.certificate-transparency.org.

by the rules. This is where CT gets involved. In the original outline of the project [25] it was suggested that periodically, maybe every hour or so, a set of "monitor" servers contact the log servers and ask for a list of all new certificates for which they have issued proofs. These "monitor" servers may be operated by third parties, such as companies, nonprofit organizations, individuals, or service providers, and their job it is to discover unauthorized certificates. Also, to make sure that the log servers behave honestly, the outline suggested another set of servers, namely audit servers (auditors) to which a browser occasionally sends all the proofs it has received so far. If there happens to occur a proof signed by a log server that does not appear in the respective log, then the auditor knows that something illegitimate is going on and that something is wrong. This way rogue CAs and log servers may be revealed and removed from the browsers' lists. The more parties that participate in CT, the faster the blocking and cleaning up processes will be. Initially, browsers that adopt CT cannot block connections for which no proofs are offered. However, after a certain amount of time, it is hoped that browsers can at least warn users before establishing a secure connection without a proof or even stop connecting to a site without one. It goes without saying that it is up to the browser manufacturers to define an appropriate behavior.

CT is one possibility for implementing a white list, which is currently only supported by Google Chrome for EV certificates. There may be other possibilities, or there may be variants of CT proposed in the future. In either case, it is important to note that a "white list" approach also has its problems,[33] that a black list for revoked certificates and a white list for legitimate certificates are not mutually exclusive, and that it makes a lot of sense to combine the two approaches in a practical setting.

As mentioned above, the second problem area is related to certificate authorization, meaning that it is important to define who is authorized to issue certificates and declare them as valid for any given entity. The X.509 name constraints extension may be used here, but it has its own problems and shortcomings: Some operating systems ignore and do not process it, it is not in line with some business models, and hence it is not widely deployed in the field. More fundamentally, one may argue whether an inband mechanism to specify the scope of a certificate is meaningful in the first place. If somebody is able to fraudulently issue a certificate, then he or she can also specify its scope at will. It may be more appropriate to have an outband mechanism to specify the scope of a certificate.

To deal with the lack of a viable solution for certificate authorization, some software vendors have taken the responsibility and defined lists of commonly trusted

33 For example, an attacker may cause certificates to be issued with serial numbers already assigned to legitimate certificates. This makes it difficult for the CA to detect that there are fraudulently issued certificates in circulation, and when it becomes necessary to revoke them, the process may be delayed, because legitimate certificate holders will be adversely affected as well.

CAs. These lists are either included in the operating system (as in the case of Microsoft Windows) or are part of application software (as in the case of Mozilla Firefox or Adobe Acrobat). It is even possible to combine the two approaches and to inherit the operating system CAs in the application software. The point is that the respective CAs are universally valid and that the Comodo and DigiNotar incidents have shown that this is inherently dangerous.

Against this background, it is sometimes advisable to customize and streamline the list of trusted root CAs. This is feasible in a corporate setting where it is possible to have the list of trusted root CAs only include a few CAs. However, the more users need to communicate with outside partners, the more difficult it is to work with small lists and the more flexibility is typically required. Against this background, Google developed and pioneered a technology known as *public key pinning* that has been implemented in the Chrome browser since version 13. Initially, public key pinning allowed Google to specify a set of public keys that are authorized to authenticate Google sites. (Public keys are used instead of certificates to enable operators to generate new certificates containing old public keys.) Fraudulently issued certificates refer to other public keys and are therefore not accepted by such a browser—at least not to authenticate Google sites. Because it may make sense to generalize public key pinning to other (non-Google) sites, Google has crafted an RFC document that it has submitted to the Internet standards track [28]. The resulting technology is HPKP (HTTP public key pinning). The basic idea of HPKP is that whenever a browser connects to a site using SSL/TLS, the site can use a new HTTP response header, `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 time frame (specified in seconds using the `max-age` directive).

By its very nature, public key pinning is a trust-on-first-use (TOFU) mechanism. It has advantages and disadvantages. The advantages are related to simplicity and easy deployment. 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 or loses control of its private key. In this case, the site cannot properly authenticate itself during the execution of the SSL/TLS handshake protocol. 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, however, 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 the site is permanently locked out (unless he or she can still use the backup key).

Another term that frequently pops up in the context of public key pinning is *trust assurances for certificate keys* (TACK). It refers to a pinning variant that was proposed as a TLS extension by Trevor Perrin and Moxie Marlinspike in 2012. Instead of pinning to a CA-provided signing key, the idea of TACK was to pin to a server-provided signing key. The obvious advantage is that TACK is independent from any CA or Internet PKI (because the servers provide the signing keys themselves). Also, because TACK was envisioned as a TLS extension, it was thought to be independent from HTTP and to be useful for any protocol protected by SSL/TLS. This is in contrast to HPKP, which is bound to HTTPS. In spite of the fact that the authors of the TACK proposal provided some proof-of-concept implementations for some popular platforms, there is still no official support in any client. The proposal is therefore silently sinking into oblivion. It is reasonable to assume that TACK will disappear as a pinning variant in the future. (It is not even included in the list of official TLS extensions outlined in Section 3.4.1.)

If one wants to more properly generalize public key pinning to arbitrary domains, then it is reasonable to link and make use of the domain name system (DNS) in one way or another. In fact, the *Sovereign Keys* project[34] that was launched by the EFF in 2011 pursues this idea. In paricular, a domain name can be claimed using a sovereign key that is recorded in some publicly verifiable form, such as a DNS entry. Once a name is claimed, its certificates are valid only if they are signed by the sovereign key. In spite of its early announcement in 2011, the Sovereign Keys project hasn't progressed past the idea stage. In fact, many problems remain unsolved, such as, for example, what happens if the sovereign key gets lost. The lack of a recovery mechanism makes this proposal very risky.

A conceptually similar but more mature approach was specified by the IETF DANE (DNS-based authentication of named entities) working group.[35] The approach is also named DANE, and it is specified in two related RFCs [29, 30].[36] The basic idea is to use special types of DNS resource records (RRs), namely TLSA records, to specify what public keys or respective certificates can be used for certificate validation. This is similar in spirit to public key pinning, but it overcomes its disadvantages (in particular, the bootstrapping problem). Also, 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. DANE is therefore also in line with the remark mentioned above that an outband mechanism to specify the scope of a certificate may be more appropriate. The major disadvantage

34 https://www.eff.org/sovereign-keys.

35 http://datatracker.ietf.org/wg/dane/.

36 Note that the first RFC is only informational, whereas the second RFC has been submitted to the Internet standards track.

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. Anyway, technologies like Sovereign Keys and—even more importantly—DANE improve security and strengthen resistance against MITM attacks. It is reasonable to expect DANE support to grow in the future.

If one can go one step further and change the trust model in use, then one can design and come up with entirely new approaches and solutions. Such an approach was, for example, proposed in the *Perspectives* project[37] [31]. In this approach, 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 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—in spite of its comparably long lifetime—it is still up and running.

Based on the work that has 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.

Distributed approaches like Perspectives or Convergence are useful when making collective decisions about the trustworthiness of a particular certificate. They work, if only a few CAs (or notaries) are compromised. They do not work if many CAs (or notaries) are simultaneously compromised. 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

37 http://perspectives-project.org.

also introduce problems related to performance, availability, and—maybe most importantly—running costs. Also, large web sites 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 will be created. This is because a view from one notary might not be the only correct one. Due to all of these disadvantages, distributed approaches still have a shadowy existence.

In an attempt to overcome this situation and to make a distributed approach more convenient to the clients, people have come up with the notion of a *mutually endorsing CA infrastructure* (MECAI).[38] It is basically a variant of a notary system in which the participating CAs run the infrastructure, meaning that they do the work and obtain freshness vouchers that are delivered to the clients. This means that most of the process happens behind the scenes, and this improves privacy and performance (at least from the clients' perspective). Similar to many other ideas mentioned here, MECAI was first published in 2011 and was updated in 2012, but it does not seem to have progressed past the idea stage since then.

6.7 FINAL REMARKS

This chapter addresses 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 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 web site operators. 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 the reasons discussed, for example, in [32]) or have changed their business model fundamentally. The bottom line is that we are far away from having a full-fledged PKI that we can use for SSL/TLS support on the client side and that web application providers must therefore use other client or user authentication technologies. Ideally, these technologies are part of the TLS protocol specification and implemented accordingly. Unfortunately, this is not yet 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 has the problem that MITM attacks become feasible (e.g., [23]).

When it comes to using public key certificates, trust is a major issue. Each browser comes along with a preconfigured set of trusted CAs (i.e., root CAs and 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.

38 http://kuix.de/mecai.

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 CAs 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, implement and put in place appropriate countermeasures. Certificate legitimation is going to be important in the future, and approaches like DANE or Sovereign Keys (ideally in combination with DNSSEC) as well as Perspectives and Convergence look promising. They either bind the legitimacy of a CA to the DNS or follow a distributed approach to find out whether a particular public key or certificate is the same as the one provided by a set of (randomly chosen) notaries. Both approaches do not solve all security problems, but they make the resulting system more resilient against specific attacks. Interestingly, the two approaches are not mutually exclusive and can be combined in one way or another. We expect more proposals (and proposals on how to combine them) to be developed and published in the future. For example, it may be possible to use Bitcoin technologies to come up with a commonly agreed on white list of valid certificates (similar to a Bitcoin ledger of valid transactions). Anyway, there are quite a few research challenges, and the Internet PKI(s) we use in the future may look fundamentally different from what we know today.

References

[1] NIST SP 800-32, *Introduction to Public Key Technology and the Federal PKI Infrastructure*, National Institute of Standards and Technology (NIST), 2001, http://csrc.nist.gov/publications/nistpubs/800-32/sp800-32.pdf.

[2] Adams, C., and S. Lloyd, *Understanding PKI: Concepts, Standards, and Deployment Considerations*, 2nd edition. Addison-Wesley, Reading, MA, 2002.

[3] Buchmann, J.A., E. Karatsiolis, and A. Wiesmaier, *Introduction to Public Key Infrastructures*, Springer-Verlag, Berlin, 2014.

[4] Shirey, R., "Internet Security Glossary, Version 2," Informational 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,” Experimental RFC 2692, September 1999.

[12] Ellison, C., et al., “SPKI Certificate Theory,” Experimental RFC 2693, September 1999.

[13] Oppliger, R., *Secure Messaging on the Internet.* Artech House Publishers, Norwood, MA, 2014.

[14] Cooper, D., et al., “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” Standards Track RFC 5280, May 2008.

[15] Myers, M., et al., “X.509 Internet Public Key Infrastructure Online Certificate Status Protocol—OCSP,” Standards Track RFC 2560, June 1999.

[16] Freeman, T., et al., “Server-Based Certificate Validation Protocol (SCVP),” Standards Track RFC 5055, December 2007.

[17] ETSI TS 101 456, *Electronic Signatures and Infrastructures (ESI); Policy Requirements for Certification Authorities Issuing Qualified Certificates,* 2007.

[18] ETSI TS 102 042, *Policy Requirements for Certification Authorities Issuing Public Key Certificates,* 2004.

[19] ISO 21188, *Public Key Infrastructure for Financial Services—Practices and Policy Framework,* 2006.

[20] 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.

[21] Oppliger, R., “Certification Authorities Under Attack: A Plea for Certificate Legitimation,” *IEEE Internet Computing,* Vol. 18, No. 1, January/February 2014, pp. 40–47.

[22] 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.

[23] Oppliger, R., R. Hauser, and D. Basin, "SSL/TLS Session-Aware User Authentication," *IEEE Computer,* Vol. 41, No. 3, March 2008, pp. 59–65.

[24] Langner, R., "Stuxnet: Dissecting a Cyberwarfare Weapon," *IEEE Security & Privacy,* Vol. 9, No. 3, 2011, pp. 49–51.

[25] Laurie, B., and C. Doctorow, "Secure the Internet," *Nature,* Vol. 491, November 2012, pp. 325–326.

[26] Laurie, B., A. Langley, and E. Kasper, "Certificate Transparency," Experimental RFC 6962, June 2013.

[27] Merkle, R., "Secrecy, Authentication, and Public Key Systems," Ph.D. Dissertation, Stanford University, 1979.

[28] Evans, C., C. Palmer, and R. Sleevi, "Public Key Pinning Extension for HTTP," Standards Track RFC 7469, April 2015.

[29] Barnes, R., "Use Cases and Requirements for DNS-Based Authentication of Named Entities (DANE)," Informational RFC 6394, October 2011.

[30] Hoffman, P., and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA," Standards Track RFC 6698, August 2012.

[31] 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.

[32] 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 are now ready to finish the book with the following four concluding remarks:

- First, the book does not provide any statistics about the use of the protocols in the field, and there are mainly two reasons for this willful absence: On one hand, 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 the trade press are more appropriate here.) On the other hand, there are many web sites that provide this type of information, particularly Qualys' SSL Labs.[1] Most statistics that are available reveal the facts that the use of the SSL/TLS protocols is steadily increasing and that SSL/TLS is likely going to be the predominant security technology for the Internet in the future. It is nowadays even used in areas where people would have argued a few years ago that the computational power of the hardware in place is not sufficient. This applies, for example, to the entire field of mobile computing. Even moderately successful applications nowadays have built-in SSL/TLS implementations that are used in the field. Taking into account HSTS, this development is likely to continue and eventually also expand itself to the DTLS protocol. The bottom line is that we can be optimistic with regard to the deployment and use of the protocols in the future.
- 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

1 https://www.ssllabs.com.

and their implementations. Many problems have been found both at the protocol and implementation levels. Throughout this book, we have come across many vulnerabilities and respective attacks with sometimes fancy acronyms [1]. 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 once again that security is a tricky and overly involved topic and that the entire security of a system can fall down if only a tiny detail is implemented incorrectly. Against this background, any initiative that is aimed at coming up with implementations that have formally verified security is highly appreciated. One such example is miTLS that is written in F#[2] and verified in F7.[3] More projects like this should be pursued in practice.

- Third, there are a few 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. As mentioned in the preface, [2] can serve this purpose (and a summary regarding SSL/TLS deployment best practices is freely available on the Internet [3]). In addition, there are several governmental and standardization bodies that have also issued (and continue to issue) recommendations for the secure use of these protocols. Most importantly, the German Federal Office for Information Security (BSI) and the IETF have released such documents [4, 5]. The IETF document [5] is particularly interesting, because it does not only make recommendations from a theoretical and security-centric viewpoint but also takes into account what is realistic and can actually be deployed in the field.[4] The cipher suites that are recommended combine DHE or ECDHE with RSA-based authentication (to provide PFS), 128-bit or 256-bit AES operated in GCM (to provide authenticated encryption), and two hash functions from the SHA-2 family (i.e., SHA-256 and SHA-384). In fact, the document advocates the following four cipher suites:
  - TLS_DHE_RSA_WITH_AES_128_GCM_SHA256;
  - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256;
  - TLS_DHE_RSA_WITH_AES_256_GCM_SHA384;

2 http://fsharp.org.
3 http://research.microsoft.com/en-us/projects/f7.
4 Similar to informational RFC 7457 referenced in Section 3.8, this document, which is also a BCP document, is provided by the IETF UTA WG (cf. https://datatracker.ietf.org/wg/uta/).

  - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384.

  From what we know today, these cipher suites are cryptographically sound and can be implemented efficiently. Note that these cipher suites are similar but use a different signature algorithm (RSA instead of ECDSA) than the ones that are in line with suite B cryptography (cf. Section 3.9).

- 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 provide PFS). This is uncompromising and somehow breaks with the tradition of protocol design in standardization committees, but it is going to have a deep impact on the security of TLS in the future. All known attacks against the previous versions of the SSL/TLS protocols no longer work against TLS 1.3. In practice, however, deployed systems are only going to profit from this advanced level of security if they are not configured in a backward-compatible way. Unfortunately, this is not likely to be the case in many situations, and hence the security of the SSL/TLS and DTLS protocols is still going to bother us in the foreseeable future. So, this book has by no means become obsolete.

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 for you to not leave you stranded, and we wish you good luck for your future endeavors in this area.

References

[1] Sheffer, Y., R. Holz, and P. Saint-Andre, "Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)," Informational RFC 7457, February 2015.

[2] 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.

[3] Ristić, I., *SSL/TLS Deployment Best Practices,* Qualsys SSL Labs, 2014, `https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practi-ces.pdf`.

[4] German Federal Office for Information Security (BSI), "Mindeststandard des BSI für den Einsatz des SSL/TLS-Protokolls durch Bundesbehörden," 2014.

[5] Sheffer, Y., R. Holz, and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 7525 (BCP 195), May 2015.

Appendix A

Registered 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] 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), DTLS 1.0 (RFC 4347), and DTLS 1.2 (RFC 6347) are omitted.

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
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

1 http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml.

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 FF	TLS_EMPTY_RENEGOTIATION_INFO_SCSV	X	5746
56 00	TLS_FALLBACK_SCSV	X	
C0 01	TLS_ECDH_ECDSA_WITH_NULL_SHA	X	4492
C0 02	TLS_ECDH_ECDSA_WITH_RC4_128_SHA		4492
C0 03	TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	X	4492
C0 04	TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	X	4492

C0 05	TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	X	4492
C0 06	TLS_ECDHE_ECDSA_WITH_NULL_SHA	X	4492
C0 07	TLS_ECDHE_ECDSA_WITH_RC4_128_SHA		4492
C0 08	TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	X	4492
C0 09	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	X	4492
C0 0A	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	X	4492
C0 0B	TLS_ECDH_RSA_WITH_NULL_SHA	X	4492
C0 0C	TLS_ECDH_RSA_WITH_RC4_128_SHA		4492
C0 0D	TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	X	4492
C0 0E	TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	X	4492
C0 0F	TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	X	4492
C0 10	TLS_ECDHE_RSA_WITH_NULL_SHA	X	4492
C0 11	TLS_ECDHE_RSA_WITH_RC4_128_SHA		4492
C0 12	TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	X	4492
C0 13	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	X	4492
C0 14	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	X	4492
C0 15	TLS_ECDH_anon_WITH_NULL_SHA	X	4492
C0 16	TLS_ECDH_anon_WITH_RC4_128_SHA		4492
C0 17	TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	X	4492
C0 18	TLS_ECDH_anon_WITH_AES_128_CBC_SHA	X	4492
C0 19	TLS_ECDH_anon_WITH_AES_256_CBC_SHA	X	4492
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	X	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

Appendix B

Padding Oracle Attacks

As its name suggests and as is illustrated in Figure B.1, a *padding oracle attack* refers to an attack in which an adversary (left side) has access to a padding oracle (right side). A padding oracle, in turn, is best seen as a black box with a specific input-output behavior. In particular, it takes as input a ciphertext and produces 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 is not returned to the adversary. It is obvious that the notion of a padding oracle only makes sense if the encryption in use employs some padding, such as, for example, in the case of a block cipher operated in CBC mode. However, there are many (other) situations in which padding is also required—be it in the case of symmetric encryption or in the case of asymmetric encryption. If a padding oracle is available, then it can usually be invoked arbitrarily many times.[1]

Since the adversary can feed a padding oracle with ciphertext of his or her choice, a padding oracle attack represents a *chosen ciphertext attack* (CCA) or—if the adversary can dynamically adapt his or her choices while the attack is going on—an *adaptive CCA* (CCA2). As its name suggests, a CCA or CCA2 is characterized by the fact that the adversary is free to choose any ciphertext that he or she wants to have decrypted by the oracle. In general, the oracle returns the resulting plaintext. In a padding oracle attack, however, the oracle only returns one bit of information, namely whether the resulting plaintext is properly padded or not. Hence, a padding oracle attack can also be seen as a very limited form of a CCA(2).

1 For technical reasons, it is sometimes required that the number of times the padding oracle can be invoked is polynomially bounded by the length of the input.

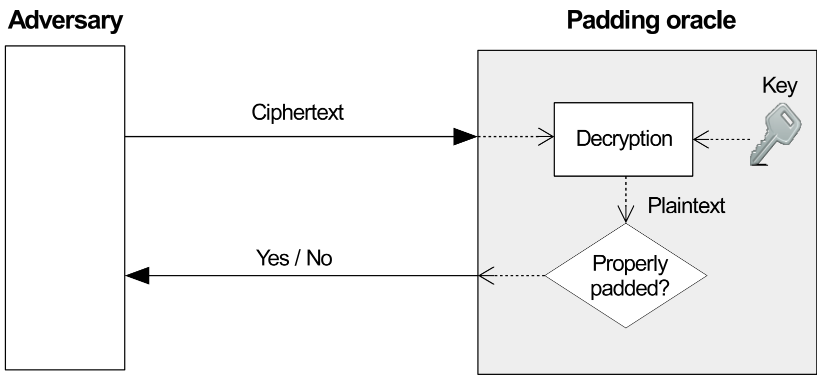

Figure B.1 A padding oracle attack.

In a general setting, it is not obvious how to actually mount a CCA(2). If the adversary had access to a decryption oracle, then he or she could also use this oracle to decrypt any ciphertext of his or her choice. Hence, a CCA(2) is a somehow artificial construction: The adversary can use the decryption oracle, but he or she is not allowed to directly input a ciphertext for which he or she wants to learn the underlying plaintext message. Hence, the feasibility of a CCA(2) had been discussed controversially in the community until Daniel Bleichenbacher and Serge Vaudenay published their findings in 1998 and 2002, respectively. Their attacks were not yet called padding oracle attacks, but they still represented prototypes of such attacks. Since then, it has come to be taken for granted that CCA(2)s are feasible in many situations, and hence that they pose a real threat. Consequently, research has started to look more seriously into possibilities to protect encryption systems against padding oracle attacks.

This appendix introduces, discusses, and puts into perspective the two padding oracle attacks mentioned above: the Bleichenbacher attack that affects the RSA encryption system and the Vaudenay attack that affects any block cipher operated in CBC mode (as used, for example, in many SSL/TLS cipher suites). Keep in mind, though, that both attacks have been known for more than a decade and that many other padding oracle attacks have been proposed in the meantime. In fact, many of the more recent attacks against SSL/TLS are also (sometimes very sophisticated) padding oracle attacks. So this appendix is not meant to be be comprehensive. Its sole purpose is to give an introduction and to provide some technical background information about the two most important padding oracle attacks. As such, it should help the reader to understand and put into perspective any of the more recent padding oracle attacks that have been published and that are mentioned in the book.

B.1 BLEICHENBACHER ATTACK

As mentioned in Section 2.4 and in the preceding section, in 1998 Bleichenbacher published a CCA2 against PKCS #1 version 1.5 [1] that is used in SSL 3.0 to pad messages before they are RSA encrypted [2]. Historically speaking, the *Bleichenbacher attack* was the first padding oracle attack that was ever published.[2] It is based on the two following well-known facts about RSA when used for encryption:

- It has been known since the early 1980s that RSA encryption (at least in its native form) is susceptible to a very simple and straightforward CCA [3]: If an adversary has access to a decryption oracle and is tasked to decrypt a ciphertext c (i.e., compute $m \equiv c^d \pmod{n}$ without feeding c directly into the oracle), then he or she can choose a random integer r and feed the innocent-looking ciphertext $c' \equiv r^e c \pmod{n}$ into the oracle. If the oracle computes $m' \equiv (c')^d \pmod{n}$ and returns this value to the adversary, then he or she can recover the original plaintext message m by simply computing $m \equiv m' r^{-1} \pmod{n}$. This CCA is feasible because RSA is multiplicatively homomorphic.
- Similarly, it has been known since the end of the 1980s that the least significant bit (LSB) of an RSA encryption is as secure as the whole message [4], meaning that somebody being able to illegitimately decrypt the LSB is also able to decrypt the entire message. In 2004, it was proven that all individual RSA bits—including, for example, the most significant bit (MSB)—are secure in this sense [5]. This means that RSA encryption protects all bits of a plaintext message, and this property is known as the bit security property of RSA. From a security viewpoint, the bit security property is a dual-edged sword: On the one hand, it is good because it ensures that all bits of a plaintext message are equally well protected. However, on the other hand, it also means that

2 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 requirement to make encryption systems resistant against them.

an implementation that leaks a single bit can be used to decrypt an entire message.

The Bleichenbacher attack employs a padding oracle, meaning that the adversary can feed arbitrary ciphertexts into the oracle. For each of these ciphertexts the oracle returns one bit of information, namely whether the plaintext that results from decrypting the ciphertext is formed according to PKCS #1 (block type 2). In the positive case, the ciphertext is said to be *PKCS #1 conforming*, or *PKCS conforming* in short. The PKCS #1 block format is illustrated in Figure B.2. A respective data block must comprise a zero byte (0x00), a byte referring to block type 2 (0x02), a variable length[3] padding string, a zero byte (0x00), and the actual data block that is encrypted. The padding string must not include a zero string, otherwise the PKCS #1 encoding could not be decoded unambiguously.

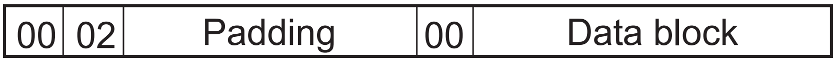

Figure B.2 PKCS #1 block format for encryption (block type 2).

The Bleichenbacher attack exploits the fact that a PKCS conforming data block must always start with the two bytes 0x00 and 0x02. This means that the adversary can choose an arbitrary ciphertext c and submit it to the padding oracle. The oracle, in turn, decrypts c [i.e., computes $m = c^d \pmod{n}$], and checks whether the resulting plaintext message m is PKCS conforming. The result of this check is then communicated to the adversary, but the adversary does not learn anything else about the plaintext message than this fact. In particular, he or she does not get to see the plaintext message that is decrypted.

Bleichenbacher showed that if an adversary can invoke a padding oracle sufficiently many times, then he or she is able to perform one operation with the private key d—without actually knowing d. This means that the adversary can either decrypt a ciphertext (that has been encrypted with d) or digitally sign a message of his or her choice (using the same key d). Maybe most importantly, the adversary can decrypt an SSL/TLS CLIENTKEYEXCHANGE message in an RSA-based key exchange that he or she has recorded previously. This message includes the premaster secret that is encrypted with the recipient's public key. If the adversary is able to mount a Bleichenbacher attack against this message using the recipient as a padding oracle, then he or she is able to decrypt the CLIENTKEYEXCHANGE message and retrieve the premaster secret accordingly. Equipped with this secret, the adversary is then able to decrypt the entire session (if a recording of the session is

3 The padding string must be at least 8 bytes long.

available in the first place). This possibility is worrisome (to say the least), and the Bleichenbacher attack is therefore highly relevant for the security of the SSL/TLS protocols and many other protocols that employ PKCS #1 for message encoding.

In theory, the existence of the Bleichenbacher attack is not surprising, as the adversary could also use the algorithm given in the reduction proof of [5] to decrypt the ciphertext. However, this algorithm is so inefficient that it cannot be used in practice. Instead, Bleichenbacher proposed another algorithm that reduces the number of chosen ciphertexts to something that can be managed. The algorithm starts from the observation that if an adversary wants to compute $m \equiv c^d \pmod{n}$ for any given ciphertext c, then he or she may choose an integer s, compute $c' \equiv s^e c \pmod{n}$, and send c' to the padding oracle. If the padding oracle responds in the affirmative, then the adversary knows that c' is PKCS conforming. This, in turn, means that the first two bytes of $ms \pmod{n}$ are 00 and 02, and hence that

$$2B \leq ms \pmod{n} < 3B$$

for $B = 2^{8(k-2)}$ and k referring to the byte length of n. The adversary now has an interval for $ms \pmod{n}$, and he or she can try to iteratively reduce the size of the interval by cleverly feeding chosen ciphertexts into the padding oracle and analyzing the respective results. The aim is to find a sequence of nested intervals that terminates if the interval comprises a single value. In this case, m can be computed (by dividing the single value by s). Typically (and according to an analysis given in [2]), 2^{20}—which is slightly more than one million—chosen ciphertexts are needed. This is why the Bleichenbacher attack is colloquially 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 used in this book.

After having overviewed the general idea of the Bleichenbacher attack, it is important to specify an algorithm that implements the attack. Such an algorithm is given in Algorithm B.1. The algorithm 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). At various places the algorithm needs access to a padding oracle that can decide whether a given ciphertext is PKCS conforming. The algorithm basically comprises four steps (which are partially iterated):

- In step 1, c is blinded so that the resulting ciphertext c_0 is PKCS conforming. Blinding is performed by multiplying c with $(s_0)^e$ modulo n, where s_0 is a randomly chosen blinding factor. If c is already PKCS conforming, then this step can be skipped. Technically, this can be achieved by setting s_0 to 1. According to what has been said above, we know that $m_0 s_0$ is in the interval $[2B, 3B - 1]$ if c_0 is PKCS conforming. This interval is assigned to M_0. In

Algorithm B.1 Algorithm that implements the Bleichenbacher attack.

```
$(e, n), c$
----------------------------------------------------------------
Step 1: Blinding $c$
    Search integer $s_0$ such that $c(s_0)^e \pmod{n}$ is PKCS conforming
    $c_0 \leftarrow c(s_0)^e \pmod{n}$
    $M_0 \leftarrow \{[2B, 3B-1]\}$
    $i \leftarrow 1$
Step 2: Searching more PKCS-conforming messages
    case $(i = 1)$: Search smallest integer $s_1 \geq n/(3B)$ such that $c_0(s_1)^e \pmod{n}$ is
                PKCS conforming (step 2.a)
        $(i > 1) \wedge (|M_{i-1}| > 1)$: Search smallest integer $s_i > s_{i-1}$ such that $c_0(s_i)^e \pmod{n}$ 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 $c_0(s_i)^e \pmod{n}$ 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 $m \equiv a(s_0)^{-1} \pmod{n}$
                else $i \leftarrow i + 1$ and go to step 2
----------------------------------------------------------------
$m$
```

the end of this step, the index i is assigned value 1. In the subsequent steps, the algorithm tries to find a sequence of nested intervals for m_0. These steps are interated until the last interval comprises a single value.

- In step 2, the algorithm searches more PKCS-conforming messages. Three cases must be are distinguished (steps 2.a–2.c):

    - If $i = 1$ (meaning that we are in the first iteration of the algorithm), then the algorithm searches the smallest positive integer $s_1 \geq n/(3B)$, such that $c_0(s_1)^e \pmod{n}$ is PKCS conforming. The threshold $n/(3B)$ applies, because there are no smaller integers that can satisfy the requirement. In Algorithm B.1, this case is referred to step 2.a.
    - If $i > 1$ and $|M_{i-1}| > 1$, then the algorithm searches the smallest positive integer $s_i > s_{i-1}$, such that $c_0(s_i)^e \pmod{n}$ is PKCS conforming. In Algorithm B.1, this case is referred to step 2.b.

  - If $i > 1$ and $|M_{i-1}| = 1$, 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 $c_0(s_i)^e \pmod{n}$ is PKCS conforming. The first condition (regarding r_i) is to make sure that the remaining interval is divided roughly in half in each iteration. The second condition (regarding s_i) is derived from the fact that $2B \leq m_0 s_i - r_i n < 3B$ and $m_0 \in [a, b]$. In Algorithm B.1, this case is referred to step 2.c.

- In step 3, the set of solutions M_i is computed according to s_i that has been found in step 2. More specifically,

$$M_i = \bigcup_{(a,b,r)} \left\{ \left[\max\left(a, \left\lceil \frac{2B + rn}{s_i} \right\rceil \right), \min\left(b, \left\lfloor \frac{3B - 1 + rn}{s_i} \right\rfloor \right) \right] \right\}$$

  for all $[a, b] \in M_{i-1}$ and $\frac{as_i - 3B + 1}{n} \leq r \leq \frac{bs_i - 2B}{n}$.

- 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 $m \equiv c^d \pmod{n}$ is computed as $m \equiv a(s_0)^{-1} \pmod{n}$. Otherwise, i is incremented by 1 and the algorithm returns to step 2.

The consequences and implications of the Bleichenbacher attack are discussed in Section 2.4. Here, we just mention that PKCS #1 was first updated to version 2.0 [6] adapting OAEP [7], that several implementations of PKCS #1 version 2.0 were still vulnerable to modified versions of the Bleichenbacher attack [8, 9], that PKCS #1 was therefore updated again in 2003 to version 2.1 [10], and that the most recent version is 2.2. This is where we stand now, and anybody using RSA for encryption should employ a padding that conforms to PKCS #1 version 2.1 or 2.2.

B.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, as mentioned in Section 3.3.1 and at the beginning of this appendix, Vaudenay published a paper on padding oracle attacks against CBC padding in 2002 [11]—four years after Bleichenbacher published his results. The resulting Vaudenay attack is best seen as the symmetric analog of the Bleichenbacher attack. It had only been theoretically interesting, until it was shown one year later that it can actually be mounted against real-world applications using SSL/TLS [12]. Since then, the feasibility of the Vaudenay attack and many variations thereof has been demonstrated in several application environments (e.g., [13, 14]). The bottom line is that the attacks pose a real threat and that implementations are better shown to be resistant against them.

If a block cipher is operated in CBC mode, then padding is required to make sure that the plaintext 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 as specified in RFC 5652 [15] provides the most widely deployed padding scheme. PKCS #5 employs the same padding scheme but is restricted to a block length of 8 bytes (whereas PKCS #7 supports variable block lengths). The TLS protocol employs PKCS #7 padding with a minor modification. It is required that at least one byte of padding is appended to the plaintext. So if the last plaintext block is equally long as the block length, then an additional full block of padding needs to be appended. In accordance with PKCS #7, the padding consists of a byte representing the length of the padding that is repeatedly written. More specifically, the byte that is repeatedly written refers to the padding length minus one (the minus one is the modification mentioned above). If, for example, the padding were one byte, then the byte that is appended to the plaintext block would be 0x00 (instead of 0x01 as required by PKCS #7 in the strict sense). If the padding were two bytes, then the byte repeated twice would be 0x01. 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 were bigger than 16 bytes, then bigger values would be possible, as well. Because the maximal value of a byte is 256 (referring to 0xFF), 256 bytes is the biggest possible block length that is currently supported. Needless to say, this is sufficiently large for all practical purposes.

So CBC padding in TLS requires the last plaintext block to end with one of the following 16 possible byte sequences:

$$\begin{array}{c} \text{0x00} \\ \text{0x01 0x01} \\ \text{0x02 0x02 0x02} \\ \text{0x03 0x03 0x03 0x03} \\ \cdots \\ \underbrace{\text{0x0F 0x0F} \ldots \text{0x0F 0x0F 0x0F}}_{16} \end{array}$$

This fact is exploited by the Vaudenay attack. Note that a similar attack can 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.

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 very last byte of the padding. All other padding bytes can be randomly chosen and comprise arbitrary values (cf. Figure 2.26). As discussed in Section 2.4, this simplifies padding oracle attacks considerably, and the POODLE attack takes advantage of this. We don't distinguish the two cases here. Instead, we restrict ourselves in this appendix to padding oracle attacks against PKCS #7 (refer to Section 2.4 for a padding oracle attack against SSL padding).

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 [11]. At first sight, this seems to be 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 his colleagues showed how to construct a padding oracle that may work in practice—at least under certain circumstances [12]. As mentioned above, this work was continued by other researchers who have shown that padding oracles exist in many situations and that it is sometimes quite simple and straightforward to exploit them (e.g., [13, 14]). So whether an adversary really has access to a padding oracle mainly depends on the implementation under attack. There is no universally valid answer, and one has to differentiate.

In what follows, we just introduce the basic idea and the working principles of the Vaudenay attack, and we don't delve into the 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 decryption key). So k represents the block length of the block cipher in bytes. Maybe the adversary knows that the block comprises some secret information, such as a user password, login 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 byte j in C_i, where j is an integer between 1 and k. This means that ciphertext block C_i can be written as follows:

$$C_i = C_i[1]C_i[2]C_i[3]\ldots C_i[k-1]C_i[k]$$

The same notation applies to the underlying plaintext block P_i (which, by the way, is the target of the attack):

$$P_i = P_i[1]P_i[2]P_i[3]\ldots P_i[k-1]P_i[k]$$

Referring to the beginning of this appendix (and Figure B.1), a padding oracle attack is a CCA 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 really 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:

$$P_i = D_K(C_i) \oplus C_{i-1} \qquad \text{(B.1)}$$

The decryption starts with $i = k$ and ends with $i = 1$ (using $C_0 = IV$, where IV is the initialization vector). Equation (B.1) basically means that C_i is decrypted by first decrypting it (using the decryption function $D(\cdot)$ and key K) and then adding the result bitwise modulo 2 to the predecessor ciphertext block C_{i-1}. This is illustrated in Figure B.3, the figure that should be kept in mind when going through the attack step by step.

To mount the Vaudenay attack and decrypt the ciphertext block C_i, the adversary can sequentially process each byte of C_i individually (i.e., $C_i[1], C_i[2], \ldots, C_i[k]$). 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

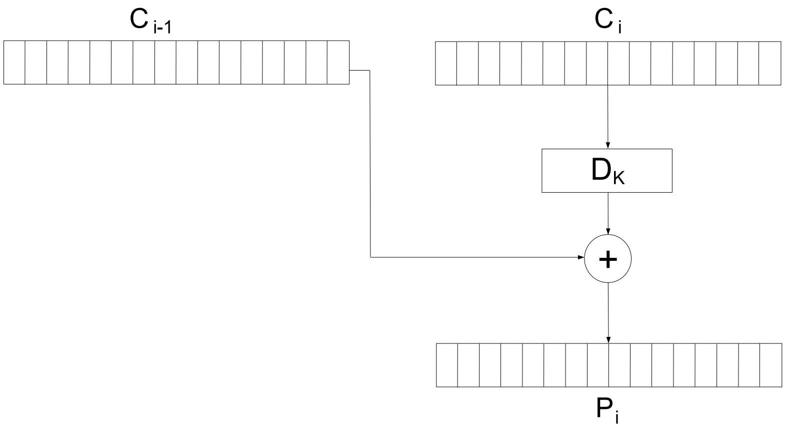

Figure B.3 CBC decryption of C_i and C_{i-1}.

the 16 bytes of the block can be attacked individually). In contrast to 2^{128}, 2^{12} is a complexity that is perfectly manageable, and hence the respective attack is feasible.

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 sent to the padding oracle, where it is decrypted according to (B.1). The result of the decryption process 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 used at all.

The adversary knows two pieces of information: On one hand, he or she knows that

$$P_2' \quad = \quad D_K(C_i) \oplus C'$$

On the other hand, he or she also knows 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), the adversary can conclude 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 but that it must also hold at the byte level (because the only operation that is used is the addition modulo 2). This means that

$$P'_2[j] = P_i[j] \oplus C_{i-1}[j] \oplus C'[j] \qquad \text{(B.2)}$$

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 better situation, because he or she may know something about the padding used in $P'_2[j]$. This knowledge can then be turned into an efficient algorithm to find $P_i[j]$, and hence to decrypt $C_i[j]$. This applies to all k bytes of C_i.

Let us now delve more deeply into the attack and its functioning. We assume an adversary who faces a ciphertext block C_i that is CBC encrypted with a block cipher with a block length of 16 bytes. Referring to what we said above, the adversary starts with the rightmost byte $C_i[k]$, which is $C_i[16]$ in this example. The respective attack scenario is illustrated in Figure B.4. Note that the adversary also knows C_{i-1} (because this ciphertext block has been transmitted on the same network) but that this block is not illustrated in Figure B.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 each and every message. In fact, these blocks differ only in their rightmost byte, i.e., $C'[16]$. All other bytes of C' are 0x00. So C' looks as follows:

$$C' = \underbrace{00\ 00\ \ldots\ 00\ 00\ 00}_{15}\ C'[16]$$

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 enough.

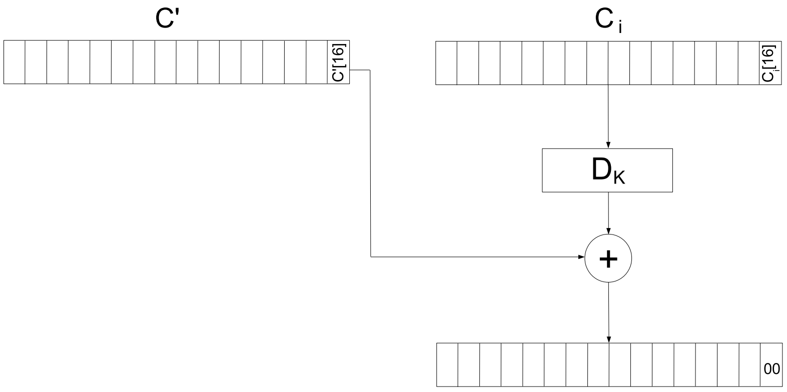

Figure B.4 CBC padding attack against $C_i[16]$.

Referring to (B.2), the adversary 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 hence to 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 the next step, the adversary goes for $C_i[15]$. The respective situation is overviewed in Figure B.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

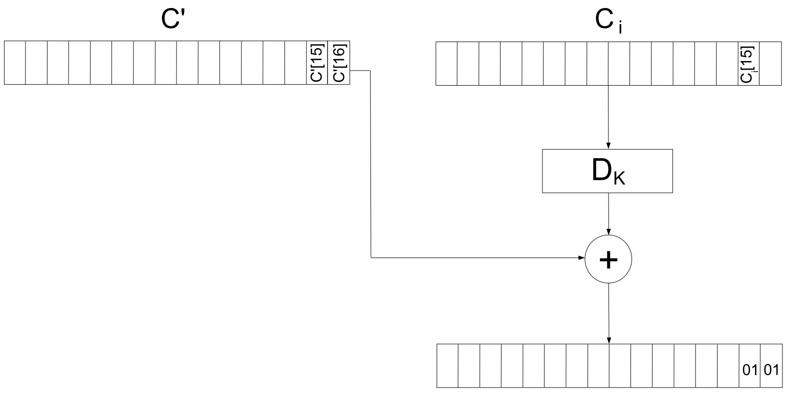

Figure B.5 CBC padding attack against $C_i[15]$.

$C'[16]$) 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 (i.e., 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 B.4. 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 value of $C'[16]$:

$$C' = \underbrace{00\ 00\ \ldots\ 00\ 00\ 00}_{14}\ C'[15]\ C'[16]$$

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 decryption key. This, in turn, means that he or she has now already decrypted two bytes of C_i, namely $C_i[16]$ and $C_i[15]$.

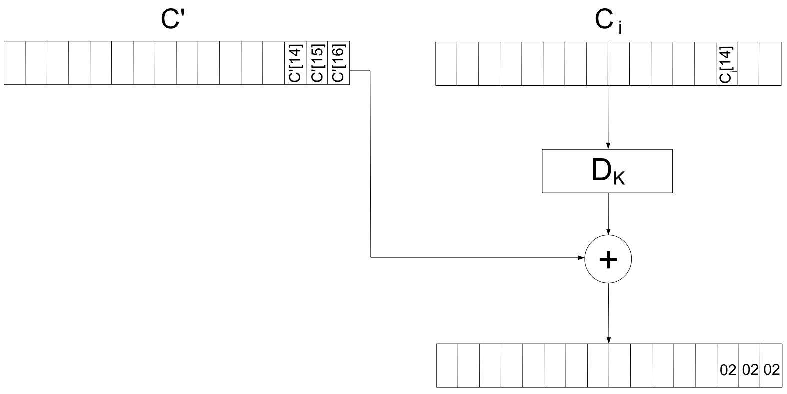

Figure B.6 CBC padding attack against $C_i[14]$.

To further continue the attack, the adversary now goes for $C_i[14]$. This is illustrated in Figure B.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 values of $C'[15]$ and $C'[16]$:

$$C' = \underbrace{00\ 00\ \ldots\ 00\ 00\ 00}_{13}\ C'[14]\ C'[15]\ C'[16]$$

Using the ciphertext block C', the adversary can now 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 $C_i[14]$ without knowing the proper decryption key.

This attack can be carried on for $C_i[13], C_i[12], \ldots, C_i[1]$ until the entire ciphertext block C_i is decrypted. This continuation of the attack is not repeated here. Feel free to complete it yourself. Also, it is generally a good exercise to go through the attack with a numerical example. This has been done, for example, by Ron Brown, and the respective results are available on his blog.[4]

The bottom line is that the adversary can decrypt the entire ciphertext block C_i without knowing the decryption key. All the adversary must have access to is a padding oracle. If such an oracle exists, then padding oracle attacks seem feasible. In practice, there are many possibilities to instantiate a padding oracle, and hence padding oracle attacks are often possible. This point is further addressed in Section 3.3.1 and at some other places throughout the book. The aim of this appendix is only to provide a prototypical description of the Vaudenay attack. Many real-world attacks are just variations thereof. If one wants to understand them, then one has to fully understand the Vaudenay attack first.

References

[1] Kaliski, B., "PKCS #1: RSA Encryption Version 1.5," Informational 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.

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

[7] Bellare, M., and P. Rogaway, "Optimal Asymmetric Encryption," *Proceedings of EUROCRYPT '94,* Springer-Verlag, LNCS 950, 1994, pp. 92–111.

[8] 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.

4 https://blog.skullsecurity.org/2013/a-padding-oracle-example.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] Housley, R., "Cryptographic Message Syntax (CMS)," Standards Track RFC 5652, September 2009.

Appendix C

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
AEAD	authenticated encryption with additional data
AES	Advanced Encryption Standard
ALPN	application-layer protocol negotiation
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
CBC	cipherblock chaining
CCA	chosen ciphertext attack
CCA2	adaptive CCA

CCM	counter with CBC-MAC mode
CDN	content delivery network
CFB	cipher feedback
CFRG	Crypto Forum Research Group
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
CSP	certification service provider
CSR	certificate signing request
CSRF	cross-site request forgery
CT	certificate transparency
CTR	counter mode encryption
CVE	common vulnerabilities and exposures
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	fixed 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
DoS	denial of service
DSA	digital signature algorithm
DSS	digital signature standard
DTLS	datagram TLS
DV	domain validation

E&A	encrypt-and-authenticate
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
EFF	Electronic Frontier Foundation
EKE	encrypted key exchange
EKM	exported keying material
EtA	encrypt-then-authenticate
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
GUI	graphical user interface
HA	high assurance
HMAC	hashed MAC
HPKP	HTTP public key pinning
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
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
KEA	key exchange algorithm
LAN	local area network
LDAP	lightweight directory access protocol
LRA	local registration authority (or agent)
LSB	least significant bit
MAC	mandatory access control
	message authentication code
MECAI	mutually endorsing CA infrastructure
MIME	multipurpose Internet mail extensions
MITM	man-in-the-middle
MPL	Mozilla Public License
MSB	most significant bit
MTU	maximum transmission unit

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
OMA	Open Mobile Alliance
OSI	open systems interconnection
OV	organization validation
PCT	private communication technology
PGP	pretty good privacy
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
PUB	Publication
RA	registration authority
RB	random byte
RBAC	role-based access control
RC2	Rivest Cipher 2
RC4	Rivest Cipher 4
RFC	request for comments
RSA	Rivest, Shamir, and Adleman
RTT	round-trip time

s2n	signal to noise
SAML	security assertion markup language
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
SHA	secure hash algorithm
SHS	secure hash standard
SHTTP	secure HTTP
S-HTTP	secure HTTP
SIP	session initiation protocol
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
SSH	secure shell
SSL	secure sockets layer
STLP	secure transport layer protocol
S-HTTP	secure HTTP (also known as SHTTP)
TACK	trust assurances for certificate keys
TC11	Technical Committee 11
TCP	transmission control protocol
TIME	timing info-leak made easy
TLS	transport layer security
TLS-SA	SSL/TLS session-aware
TOFU	trust-on-first-use
TTP	trusted third party
UDP	user datagram protocol
URL	uniform resource locator

UTA	using TLS in applications
UTC	coordinated universal time (in French)
VoIP	voice over IP
VPN	virtual private network
W3C	World Wide Web Consortium
WAF	web application firewall
WAP	wireless application protocol
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 received an M.Sc. and a Ph.D. 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) in Berkeley, California, he joined the federal authorities of the Swiss Confederation 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 (http://www.esecurity.ch) to provide scientific and state-of-the-art consulting, education, and engineering services related to IT security and began serving as the editor of Artech House's Information Security and Privacy Series (http://www.esecurity.ch/serieseditor.html). 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 is a member of the editorial board of some prestigious periodicals in the field, such as the *International Journal of Information Security*, *IEEE Computer*, *IEEE Security & Privacy*, and *Security and Communication Networks*. He is a senior member of the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE), as well as a member of the IEEE Computer Society and the International Association for Cryptologic Research (IACR). Besides, he has 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.

Index