BULLETPROOF SSL AND TLS

Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications

Ivan Ristić

Feisty Duck

Bulletproof SSL and TLS

Ivan Ristić

Feisty Duck
LONDON

Bulletproof SSL and TLS

by Ivan Ristić

Published in August 2014 (build 515).

ISBN: 978-1-907117-04-6

Feisty Duck Limited
www.feistyduck.com
contact@feistyduck.com

Address:
6 Acantha Court
Montpelier Road
London W5 2QP
United Kingdom

Production editor: Jelena Girić-Ristić

Copyeditor: Melinda Rankin

Table of Contents

Preface

You are about to undertake a journey into the mysterious world of cryptography. I've just completed mine—writing this book—and it's been an amazing experience. Although I'd been a user of SSL since its beginnings, I developed a deep interest in it around 2004, when I started to work on my first book, *Apache Security*. About five years later, in 2009, I was looking for something new to do; I decided to spend more time on SSL, and I've been focusing on it ever since. The result is this book.

My main reason to go back to SSL was the thought that I could improve things. I saw an important technology hampered by a lack of tools and documentation. Cryptography is a fascinating subject: it's a field in which when you know more, you actually know less. Or, in other words, the more you know, the more you discover how much you don't know. I can't count how many times I've had the experience of reaching a new level of understanding of a complex topic only to have yet another layer of complexity open up to me; that's what makes the subject amazing.

I spent about two years writing this book. At first, I thought I'd be able to spread the effort so that I wouldn't have to dedicate my life to it, but that wouldn't work. At some point, I realized that things are changing so quickly that I constantly need to go back and rewrite the "finished" chapters. Towards the end, about six months ago, I started to spend every spare moment writing to keep up.

I wrote this book to save you time. I spent the large part of the last five years learning everything I could about SSL/TLS and PKI, and I knew that only a few can afford to do the same. I thought that if I put the most important parts of what I know into a book others might be able to achieve a similar level of understanding in a fraction of the time—and here we are.

This book has the word "bulletproof" in the title, but that doesn't mean that TLS is unbreakable. It does mean that if you follow the advice from this book you'll be able to get the most out of TLS and deploy it as securely as anyone else in the world. It's not always going to be easy—especially with web applications—but if you persist, you'll have better security than 99.99% of servers out there. In fact, even with little effort, you can actually have better security than 99% of the servers on the Internet.

Broadly speaking, there are two paths you can take to read this book. One is to take it easy and start from the beginning. If you have time, this is going to be the more enjoyable approach. But if you want answers quickly, jump straight to chapters 8 and 9. They're going to tell you everything you need to know about deploying secure servers while achieving good performance. After that, use chapters 1 through 7 as a reference and chapters 10 through 16 for practical advice as needed.

Scope and Audience

This book exists to document everything you need to know about SSL/TLS and PKI for practical, daily work. I aimed for just the right mix of theory, protocol detail, vulnerability and weakness information, and deployment advice to help you get your job done.

As I was writing the book, I imagined representatives of three diverse groups looking over my shoulder and asking me questions:

System administrators
: Always pressed for time and forced to deal with an ever-increasing number of security issues on their systems, system administrators need reliable advice about TLS so that they can deal with its configuration quickly and efficiently. Turning to the Web for information on this subject is counterproductive, because there's so much incorrect and obsolete documentation out there.

Developers
: Although SSL initially promised to provide security transparently for any TCP-based protocol, in reality developers play a significant part in ensuring that applications remain secure. This is particularly true for web applications, which evolved around SSL and TLS and incorporated features that can subvert them. In theory, you "just enable encryption"; in practice, you enable encryption but also pay attention to a dozen or so issues, ranging from small to big, that can break your security. In this book, I made a special effort to document every single one of those issues.

Managers
: Last but not least, I wrote the book for managers who, even though not necessarily involved with the implementation, still have to understand what's going on and make decisions. The security space is getting increasingly complicated, so understanding the attacks and threats is often a job in itself. Often, there isn't any one way to deal with the situation, and the best way often depends on the context.

Overall, you will find very good coverage of HTTP and web applications here but little to no mention of other protocols. This is largely because HTTP is unique in the way it uses encryption, powered by browsers, which have become the most popular application-delivery platform we've ever had. With that power come many problems, which is why there is so much space dedicated to HTTP.

But don't let that deceive you; if you take away the HTTP chapters, the remaining content (about two-thirds of the book) provides generic advice that can be applied to any protocol that uses TLS. The OpenSSL, Java, and Microsoft chapters provide protocol-generic information for their respective platforms.

That said, if you're looking for configuration examples for products other than web servers you won't find them in this book. The main reason is that—unlike with web servers, for which the market is largely split among a few major platforms—there are a great many products of other types. It was quite a challenge to keep the web server advice up-to-date, being faced with nearly constant changes. I wouldn't be able to handle a larger scope. Therefore, my intent is to publish additional configuration examples online and hopefully provide the initial spark for a community to form to keep the advice up-to-date.

Contents

This book has 16 chapters, which can be grouped into several parts. The parts build on one another to provide a complete picture, starting with theory and ending with practical advice.

The first part, chapters 1 through 3, is the foundation of the book and discusses cryptography, SSL, TLS, and PKI:

- Chapter 1, *SSL, TLS, and Cryptography*, begins with an introduction to SSL and TLS and discusses where these secure protocols fit in the Internet infrastructure. The remainder of the chapter provides an introduction to cryptography and discusses the classic threat model of the active network attacker.
- Chapter 2, *Protocol*, discusses the details of the TLS protocol. I cover TLS 1.2, which is the most recent version. Information about earlier protocol revisions is provided where appropriate. An overview of the protocol evolution from SSL 3 onwards is included at the end for reference.
- Chapter 3, *Public-Key Infrastructure*, is an introduction to Internet PKI, which is the predominant trust model used on the Internet today. The focus is on the standards and organizations as well as governance, ecosystem weaknesses and possible future improvements.

The second part, chapters 4 through 7, details the various problems with trust infrastructure, our security protocols, and their implementations in libraries and programs:

- Chapter 4, *Attacks against PKI*, deals with attacks on the trust ecosystem. It covers all the major CA compromises, detailing the weaknesses, attacks, and consequences. This chapter gives a thorough historical perspective on the security of the PKI ecosystem, which is important for understanding its evolution.

- Chapter 5, *HTTP and Browser Issues*, is all about the relationship between HTTP and TLS, the problems arising from the organic growth of the Web, and the messy interactions between different pieces of the web ecosystem.
- Chapter 6, *Implementation Issues*, deals with issues arising from design and programming mistakes related to random number generation, certificate validation, and other key TLS and PKI functionality. In addition, it discusses voluntary protocol downgrade and truncation attacks and also covers Heartbleed.
- Chapter 7, *Protocol Attacks*, is the longest chapter in the book. It covers all the major protocol flaws discovered in recent years: insecure renegotiation, BEAST, CRIME, Lucky 13, RC4, TIME and BREACH, and Triple Handshake Attack. A brief discussion of Bullrun and its impact on the security of TLS is also included.

The third part, chapters 8 through 10, provides comprehensive advice about deploying TLS in a secure and efficient fashion:

- Chapter 8, *Deployment*, is the map for the entire book and provides step-by-step instructions on how to deploy secure and well-performing TLS servers and web applications.
- Chapter 9, *Performance Optimization*, focuses on the speed of TLS, going into great detail about various performance improvement techniques for those who want to squeeze every bit of speed out of their servers.
- Chapter 10, *HSTS, CSP, and Pinning*, covers some advanced topics that strengthen web applications, such as HTTP Strict Transport Security and Content Security Policy. It also covers pinning, which is an effective way of reducing the large attack surface imposed by our current PKI model.

The fourth and final part consists of chapters 11 through 16, which give practical advice about how to use and configure TLS on major deployment platforms and web servers and how to use OpenSSL to probe server configuration:

- Chapter 11, *OpenSSL Cookbook*, describes the most frequently used OpenSSL functionality, with a focus on installation, configuration, and key and certificate management. The last section in this chapter provides instructions on how to construct and manage a private certification authority.
- Chapter 12, *Testing with OpenSSL*, continues with OpenSSL and explains how to use its command-line tools to test server configuration. Even though it's often much easier to use an automated tool for testing, OpenSSL remains the tool you turn to when you want to be sure about what's going on.
- Chapter 13, *Configuring Apache*, discusses the TLS configuration of the popular Apache `httpd` web server. This is the first in a series of chapters that provide practical advice to match the theory from the earlier chapters. Each chapter is dedicated to one major technology segment.

- Chapter 14, *Configuring Java and Tomcat*, covers Java (versions 7 and 8) and the Tomcat web server. In addition to configuration information, this chapter includes advice about securing web applications.
- Chapter 15, *Configuring Microsoft Windows and IIS*, discusses the deployment of TLS on the Microsoft Windows platform and the Internet Information Server. This chapter also gives advice about the use of TLS in web applications running under ASP.NET.
- Chapter 16, *Configuring Nginx*, discusses the Nginx web server, covering the features of the recent stable versions as well as some glimpses into the improvements in the development branch.

SSL versus TLS

It is unfortunate that we have two names for essentially the same protocol. In my experience, most people are familiar with the name SSL and use it in the context of transport layer encryption. Some people, usually those who spend more time with the protocols, use or try to make themselves use the correct name, whichever is right in the given context. It's probably a lost cause. Despite that, I tried to do the same. It was a bit cumbersome at times, but I think I managed it by (1) avoiding either name where possible, (2) mentioning both where advice applies to all versions, and (3) using TLS in all other cases. You probably won't notice, and that's fine.

SSL Labs

SSL Labs (*www.ssllabs.com*) is a research project I started in 2009 to focus on the practical aspects of SSL/TLS and PKI. I joined Qualys in 2010, taking the project with me. Initially, my main duties were elsewhere, but, as of 2014, SSL Labs has my full attention.

The project largely came out of my realization that the lack of good documentation and tools is a large part of why TLS servers are generally badly configured. (Poor default settings being the other major reason.) Without visibility—I thought—we can't begin to work to solve the problem. Over the years, SSL Labs expanded into four key projects:

Server test
: The main feature of SSL Labs is the server test, which enables site visitors to check the configuration of any public web server. The test includes dozens of important checks not available elsewhere and gives a comprehensive view of server configuration. The grading system is easy to understand and helps those who are not security experts differentiate between small and big issues. One of the most useful parts of the test is the handshake simulator, which predicts negotiated protocols and cipher suites with about 40 of the most widely used programs and devices. This feature effectively takes the guesswork out of TLS configuration. In my opinion, it's indispensable.

Client test
: As a fairly recent addition, the client test is not as well known, but it's nevertheless very useful. Its primary purpose is to help us understand client capabilities across a large number of devices. The results obtained in the tests are used to power the handshake simulator in the server test.

Best practices
: *SSL/TLS Deployment Best Practices* is a concise and reasonably comprehensive guide that gives definitive advice on TLS server configuration. It's a short document (about 11 pages) that can be absorbed in a small amount of time and used as a server test companion.

SSL Pulse
: Finally, SSL Pulse is designed to monitor the entire ecosystem and keep us informed about how we're doing as a whole. It started in 2012 by focusing on a core group of TLS-enabled sites selected from Alexa's top 1 million web sites. Since then, SSL Pulse has been providing a monthly snapshot of key ecosystem statistics.

There are also several other smaller projects; you can find out more about them on the SSL Labs web site.

Online Resources

This book doesn't have an online companion (although you can think of SSL Labs as one), but it does have an online file repository that contains the files referenced in the text. The repository is available at *github.com/ivanr/bulletproof-tls*. In time, I hope to expand this repository to include other useful content that will complement the book.

To be notified of events and news as they happen, follow `@ivanristic` on Twitter. TLS is all I do these days, and I try to highlight everything that's relevant. There's hardly any noise. In addition, my Twitter account is where I will mention improvements to the book as they happen.

My blog is available at *blog.ivanristic.com*. This is where I'll react to important ecosystem news and discoveries, announce SSL Labs improvements, and publish my research.

If you bought this book in digital form, then you can always log back into your account on the Feisty Duck web site and download the most recent release. A purchase includes unlimited access to the updates of the same edition. Unless you modified your email subscription settings, you'll get an email about book updates whenever there's something sufficiently interesting, but I generally try to keep the numbers of emails to a minimum (and never use the list for any other purpose).

Feedback

I am fortunate that I can update this book whenever I want to. It's not a coincidence; I made it that way. If I make a change today, it will be available to you tomorrow, after an automated daily build takes place. It's a tad more difficult to update paper books, but, with print on demand, we're able to publish a revision every quarter or so.

Therefore, unlike with many other books that might never see a new edition, your feedback matters. If you find an error, it will be fixed in a few days. The same is true for minor improvements, such as language changes or clarifications. If one of the platforms changes in some way or there's a new development, I can cover it. My aim with this book is to keep it up-to-date for as long as there's interest in it.

Please write to me at `ivanr@webkreator.com`.

About the Author

In this section, I get to write about myself in third person; this is my "official" biography:

> *Ivan Ristić is a security researcher, engineer, and author, known especially for his contributions to the web application firewall field and development of ModSecurity, an open source web application firewall, and for his SSL/TLS and PKI research, tools, and guides published on the SSL Labs web site.*
>
> *He is the author of two books,* Apache Security *and* ModSecurity Handbook*, which he publishes via Feisty Duck, his own platform for continuous writing and publishing. Ivan is an active participant in the security community, and you'll often find him speaking at security conferences such as Black Hat, RSA, OWASP AppSec, and others. He's currently Director of Application Security Research at Qualys.*

I should probably also mention *OpenSSL Cookbook*, which is a short and free ebook that combines Chapter 11 from this book and the *SSL/TLS Deployment Best Practices guide* in one package.

Acknowledgments

Although I wrote all of the words in this book, I am not the sole author. My words build on an incredible wealth of information about cryptography and computer security scattered among books, standards documents, research papers, conference talks, and blog posts—and even tweets. There are hundreds of people whose work made this book what it is.

Over the years, I have been fortunate to correspond about computer security with many people who have enriched my own knowledge of this subject. Many of them lent me a hand

by reviewing parts of the manuscript. I am grateful for their help. It's been particularly comforting to have the key parts of the book reviewed by those who either designed the standards or broke them and by those who wrote the programs I talk about.

Kenny Paterson was tremendously helpful with his thorough review of the protocol attacks chapter, which is easily the longest and the most complicated part of the book. I suspect he gave me the same treatment his students get, and my writing is much better because of it. It took me an entire week to update the chapter in response to Kenny's comments.

Benne de Weger reviewed the chapters about cryptography and the PKI attacks. Nasko Oskov reviewed the key chapters about the protocol and Microsoft's implementation. Rick Andrews and his colleagues from Symantec helped with the chapters on PKI attacks and browser issues, as did Adam Langley. Marc Stevens wrote to me about PKI attacks and especially about chosen-prefix attacks against MD5 and SHA1. Nadhem AlFardan, Thai Duong, and Juliano Rizzo reviewed the protocol attacks chapter and were very helpful answering my questions about their work. Ilya Grigorik's review of the performance chapter was thorough and his comments very useful. Jakob Schlyter reviewed the chapter about advanced topics (HSTS and CSP), with a special focus on DANE. Rich Bowen and Jeff Trawick reviewed the Apache chapter; Jeff even fixed some things in Apache related to TLS and made me work harder to keep up with the changes. Xuelei Fan and Erik Costlow from Oracle reviewed the Java chapter, as did Mark Thomas, William Sargent, and Jim Manico. Andrei Popov and Ryan Hurst reviewed the Microsoft chapter. Maxim Dounin was always quick to respond to my questions about Nginx and reviewed the chapter on it.

Vincent Bernat's microbenchmarking tool was very useful to write the performance chapter.

Also, a big thanks to my readers who responded to the early versions of this book: Pascal Cuoq, Joost van Dijk, Daniël van Eeden, Brian Howson, Brian King, Colm MacCárthaigh, Pascal Messerli, and Christian Sage.

My special thanks goes to my copyeditor, Melinda Rankin, who was always quick to respond with her edits and adapted to my DocBook-based workflow. I'd be amiss not to mention my employer, Qualys, for supporting my writing and my work on SSL Labs.

1 SSL, TLS, and Cryptography

We live in an increasingly connected world. During the last decade of the 20th century the Internet rose to popularity and forever changed how we live our lives. Today we rely on our phones and computers to communicate, buy goods, pay bills, travel, work, and so on. Many of us, with *always-on* devices in our pockets, don't connect to the Internet, we *are* the Internet. There are already more phones than people. The number of smart phones is measured in billions and increases at a fast pace. In the meantime, plans are under way to connect all sorts of devices to the same network. Clearly, we're just getting started.

All the devices connected to the Internet have one thing in common—they rely on the protocols called SSL (*Secure Socket Layer*) and TLS (*Transport Layer Security*) to protect the information in transit.

Transport Layer Security

When the Internet was originally designed, little thought was given to security. As a result, the core communication protocols are inherently insecure and rely on the honest behavior of all involved parties. That might have worked back in the day, when the Internet consisted of a small number of nodes—mostly universities—but falls apart completely today when everyone is online.

SSL and TLS are cryptographic protocols designed to provide secure communication over insecure infrastructure. What this means is that, if these protocols are properly deployed, you can open a communication channel to an arbitrary service on the Internet, be reasonably sure that you're talking to the correct server, and exchange information safe in knowing that your data won't fall into someone else's hands and that it will be received intact. These protocols protect the communication link or *transport layer*, which is where the name TLS comes from.

Security is not the only goal of TLS. It actually has four main goals, listed here in the order of priority:

Cryptographic security
: This is the main issue: enable secure communication between any two parties who wish to exchange information.

Interoperability
: Independent programmers should be able to develop programs and libraries that are able to communicate with one another using common cryptographic parameters.

Extensibility
: As you will soon see, TLS is effectively a framework for the development and deployment of actual cryptographic protocols. Its important goal is to be independent of the actual cryptographic primitives used, allowing migration from one primitive to another without needing to create new protocols.

Efficiency
: The final goal is to achieve all of the previous goals at an acceptable performance cost, reducing costly cryptographic operations down to the minimum and providing a session caching scheme to avoid them on subsequent connections.

Networking Layers

At its core, the Internet is built on top of IP and TCP protocols, which are used to package data into small packets for transport. As these packets travel thousands of miles across the world, they cross many computer systems (called *hops*) in many countries. Because the core protocols don't provide any security by themselves, anyone with access to the communication links can gain full access to the data as well as change the traffic without detection.

IP and TCP aren't the only vulnerable protocols. There's a range of other protocols that are used for *routing*—helping computers find other computers on the network. DNS and BGP are two such protocols. They, too, are insecure and can be hijacked in a variety of ways. If that happens, a connection intended for one computer might be answered by the attacker instead.

When encryption is deployed, the attacker might be able to gain access to the encrypted data, but she wouldn't be able to decrypt it or modify it. To prevent impersonation attacks, SSL and TLS rely on another important technology called PKI (*public-key infrastructure*), which ensures that the traffic is sent to the correct recipient.

To understand where SSL and TLS fit, we're going to take a look at the *Open Systems Interconnection* (OSI) model, which is a conceptional model that can be used to discuss network communication. In short, all functionality is mapped into seven layers. The bottom layer is the closest to the physical communication link; subsequent layers build on top of one another and provide higher levels of abstraction. At the top is the application layer, which carries application data.

> **Note**
>
> It's not always possible to neatly organize real-life protocols into the OSI model. For example, SPDY and HTTP/2 could go into the session layer because they deal with connection management, but they operate after encryption. Layers from five onwards are often fuzzy.

Table 1.1. OSI model layers

#	OSI Layer	Description	Example protocols
7	Application	Application data	HTTP, SMTP, IMAP
6	Presentation	Data representation, conversion, encryption	SSL/TLS
5	Session	Management of multiple connections	-
4	Transport	Reliable delivery of packets and streams	TCP, UDP
3	Network	Routing and delivery of datagrams between network nodes	IP, IPSec
2	Data link	Reliable local data connection (LAN)	Ethernet
1	Physical	Direct physical data connection (cables)	CAT5

Arranging communication in this way provides clean separation of concerns; protocols don't need to worry about the functionality implemented by lower layers. Further, protocols at different layers can be added and removed; a protocol at a lower layer can be used for many protocols from higher levels.

SSL and TLS are a great example of how this principle works in practice. They sit above TCP but below higher-level protocols such as HTTP. When encryption is not necessary, we can remove TLS from our model, but that doesn't affect the higher-level protocols, which continue to work directly with TCP. When you do want encryption, you can use it to encrypt HTTP, but also any other TCP protocol, for example SMTP, IMAP and so on.

Protocol History

SSL protocol was developed at Netscape, back when Netscape Navigator ruled the Internet.[1] The first version of the protocol never saw the light of day, but the next—version 2—was released in November 1994. The first deployment was in Netscape Navigator 1.1, which was released in March 1995.

Developed with little to no consultation with security experts outside Netscape, SSL 2 ended up being a poor protocol with serious weaknesses. This forced Netscape to work on SSL 3, which was released in late 1995. Despite sharing the name with earlier protocol versions, SSL 3 was a brand new protocol design that established the design we know today.

[1] For a much more detailed history of the early years of the SSL protocol, I recommend Eric Rescorla's book *SSL and TLS: Designing and Building Secure Systems* (Addison-Wesley, 2001), pages 47–51.

In May 1996, the TLS working group was formed to migrate SSL from Netscape to IETF.[2] The process was painfully slow because of the political fights between Microsoft and Netscape, a consequence of the larger fight to dominate the Web. TLS 1.0 was finally released in January 1999, as RFC 2246. The new protocol was different from SSL 3 only in some minor details, but that was enough for the two revisions to be incompatible. The name was changed to please Microsoft.[3]

The next version, TLS 1.1, wasn't released until April 2006 and contained essentially only security fixes. However, a major change to the protocol was incorporation of *TLS extensions*, which were released a couple of years earlier, in June 2003.

TLS 1.2 was released in August 2008. It added support for authenticated encryption and generally removed all hard-coded security primitives from the specification, making the protocol fully flexible.

The next protocol version, which is currently in development, is shaping to be a major revision aimed at simplifying the design, removing many of the weaker and less desirable features, and improving performance. You can follow the discussions on the TLS working group mailing list.[4]

Cryptography

Cryptography is the science and art of secure communication. Although we associate encryption with the modern age, we've actually been using cryptography for thousands of years. The first mention of a *scytale*, an encryption tool, dates to the seventh century BC.[5] Cryptography as we know it today was largely born in the twentieth century and for military use. Now it's part of our everyday lives.

When cryptography is correctly deployed, it addresses the three core requirements of security: keeping secrets (*confidentiality*), verifying identities (*authenticity*), and ensuring safe transport (*integrity*).

In the rest of this chapter, I will discuss the basic building blocks of cryptography, with the goal of showing where additional security comes from. I will also discuss how cryptography is commonly attacked. Cryptography is a very diverse field and has a strong basis in mathematics, but I will keep my overview at a high level, with the aim of giving you a foundation that will enable you to follow the discussion in the rest of the text. Elsewhere in the book, where the topic demands, I will discuss some parts of cryptography in more detail.

---

[2] TLS Working Group (IETF, retrieved 23 June 2014)

[3] Security Standards and Name Changes in the Browser Wars (Tim Dierks, 23 May 2014)

[4] TLS working group mailing list archives (IETF, retrieved 19 July 2014)

[5] Scytale (Wikipedia, retrieved 5 June 2014)

> **Note**
>
> If you want to spend more time learning about cryptography, there's plenty of good literature available. My favorite book on this topic is *Understanding Cryptography*, written by Christof Paar and Jan Pelzl and published by Springer in 2010.

Building Blocks

At the lowest level, cryptography relies on various *cryptographic primitives*. Each primitive is designed with a particular useful functionality in mind. For example, we might use one primitive for encryption and another for integrity checking. The primitives alone are not very useful, but we can combine them into *schemes* and *protocols* to provide robust security.

Who Are Alice and Bob?

Alice and *Bob* are names commonly used for convenience when discussing cryptography.[6] They make the otherwise often dry subject matter more interesting. Ron Rivest is credited for the first use of these names in the 1977 paper that introduced the RSA cryptosystem.[7] Since then, a number of other names have entered cryptographic literature. In this chapter, I use the name *Eve* for an attacker with an eavesdropping ability and *Mallory* for an active attacker who can interfere with network traffic.

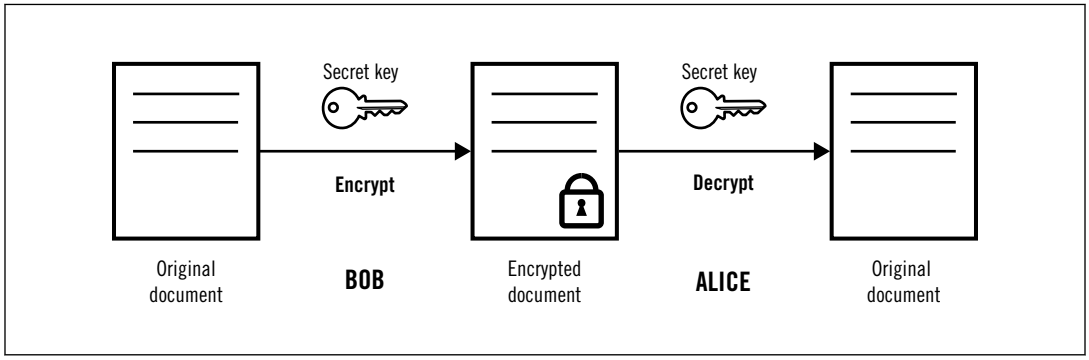
Symmetric Encryption

Symmetric encryption (or *private-key cryptography*) is a method for obfuscation that enables secure transport of data over insecure communication channels. To communicate securely, Alice and Bob first agree on the encryption algorithm and a secret key. Later on, when Alice wants to send some data to Bob, she uses the secret key to encrypt the data. Bob uses the same key to decrypt it. Eve, who has access to the communication channel and can see the encrypted data, doesn't have the key and thus can't access the original data. Alice and Bob can continue to communicate securely for as long as they keep the secret key safe.

[6] Alice and Bob (Wikipedia, retrieved 5 June 2014)

[7] Security's inseparable couple (Network World, 2005)

Figure 1.1. Symmetric encryption



> **Note**
>
> Three terms are commonly used when discussing encryption: *plaintext* is the data in its original form, *cipher* is the algorithm used for encryption, and *ciphertext* is the data after encryption.

Symmetric encryption goes back thousands of years. For example, to encrypt with a *substitution cipher*, you replace each letter in the alphabet with some other letter; to decrypt, you reverse the process. In this case, there is no key; the security depends on keeping the method itself secret. That was the case with most early ciphers. Over time, we adopted a different approach, following the observation of a nineteenth-century cryptographer named *Auguste Kerckhoffs*:[8]

> *A cryptosystem should be secure even if the attacker knows everything about the system, except the secret key.*

Although it might seem strange at first, Kerckhoffs's principle—as it has come to be known—makes sense if you consider the following:

- For an encryption algorithm to be useful, it must be shared with others. As the number of people with access to the algorithm increases, the likelihood that the algorithm will fall into the enemy's hands increases too.
- A single algorithm without a key is very inconvenient to use in large groups; everyone can decrypt everyone's communication.
- It's very difficult to design good encryption algorithms. The more exposure and scrutiny an algorithm gets, the more secure it can be. Cryptographers recommend a conservative approach when adopting new algorithms; it usually takes years of breaking attempts until a cipher is considered secure.

[8] la cryptographie militaire (Fabien Petitcolas, retrieved 1 June 2014)

A good encryption algorithm is one that produces seemingly random ciphertext, which can't be analyzed by the attacker to reveal any information about plaintext. For example, the substitution cipher is not a good algorithm, because the attacker could determine the frequency of each letter of ciphertext and compare it with the frequency of the letters in the English language. Because some letters appear more often than others, the attacker could use his observations to recover the plaintext. If a cipher is good, the only option for the attacker should be to try all possible decryption keys, otherwise known as an *exhaustive key search*.

At this point, the security of ciphertext depends entirely on the key. If the key is selected from a large *keyspace* and breaking the encryption requires iterating through a prohibitively large number of possible keys, then we say that a cipher is *computationally secure*.

> **Note**
>
> The common way to measure encryption strength is via key length; the assumption is that keys are essentially random, which means that the keyspace is defined by the number of bits in a key. As an example, a 128-bit key (which is considered very secure) is one of 340 billion billion billlion billion possible combinations.

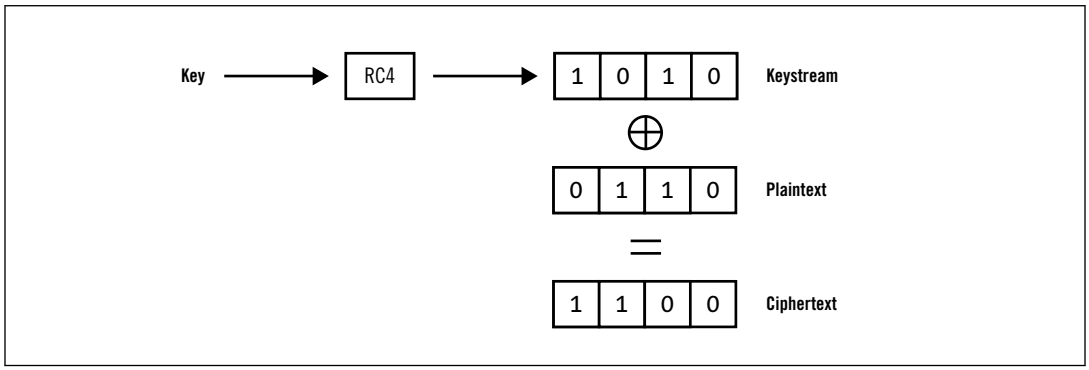
Ciphers can be divided into two groups: stream and block ciphers.

Stream Ciphers

Conceptually, *stream ciphers* operate in a way that matches how we tend to imagine encryption. You feed one byte of plaintext to the encryption algorithm, and out comes one byte of ciphertext. The reverse happens at the other end. The process is repeated for as long as there is data to process.

At its core, a stream cipher produces an infinite stream of seemingly random data called a *keystream*. To perform encryption, one byte of keystream is combined with one byte of plaintext using the XOR logical operation. Because XOR is reversible, to decrypt you perform XOR of ciphertext with the same keystream byte. This process is illustrated in Figure 1.2, "RC4 encryption".

Figure 1.2. RC4 encryption



An encryption process is considered secure if the attacker can't predict which keystream bytes are at which positions. For this reason, it is vital that stream ciphers are never used with the same key more than once. This is because, in practice, attackers know or can predict plaintext at certain locations (think of HTTP requests being encrypted; things such as request method, protocol version, and header names are the same across many requests). When you know the plaintext and can observe the corresponding ciphertext, you uncover parts of the keystream. You can use that information to uncover the same parts of future ciphertexts if the same key is used. To work around this problem, stream algorithms are used with one-time keys derived from long-term keys.

RC4 is the best-known stream cipher.[9] It became popular due to its speed and simplicity, but it's no longer considered secure. I discuss its weaknesses at some length in the section called "RC4 Weaknesses". Other modern and secure stream ciphers are promoted by the *ECRYPT Stream Cipher Project*.[10]

Block Ciphers

Block ciphers encrypt entire blocks of data at a time; modern block ciphers tend to use a block size of 128 bits (16 bytes). A block cipher is a transformation function: it takes some input and produces seemingly random output from it. For every possible input combination, there is exactly one output, as long as the key stays the same. A key property of block ciphers is that a small variation in input (e.g., a change of one bit anywhere) produces a large variation (e.g., most bits in the output change).

On their own, block ciphers are not very useful because of several limitations. First, you can only use them to encrypt data lengths equal to the size of the encryption block. To use a block cipher in practice, you need a scheme to handle data of arbitrary length. Another

[9] RC4 (Wikipedia, retrieved 1 June 2014)

[10] eSTREAM: the ECRYPT Stream Cipher Project (European Network of Excellence in Cryptology II, retrieved 1 June 2014)

problem is that block ciphers are *deterministic*; they always produce the same output for the same input. This property opens up a number of attacks and needs to be dealt with.

In practice, block ciphers are used via encryption schemes called *block cipher modes*, which smooth over the limitations and sometimes add authentication to the mix. Block ciphers can also be used as the basis for other cryptographic primitives, such as hash functions, message authentication codes, pseudorandom generators, and even stream ciphers.

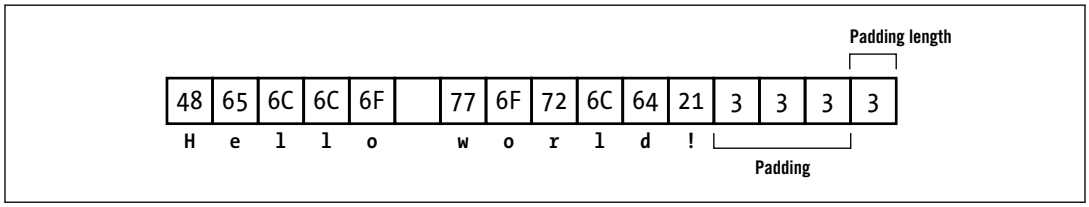
The world's most popular block cipher is AES (short for *Advanced Encryption Standard*), which is available in strengths of 128, 192, and 256 bits.[11]

Padding

One of the challenges with block ciphers is figuring out how to handle encryption of data lengths smaller than the encryption block size. For example, 128-bit AES requires 16 bytes of input data and produces the same amount as output. This is fine if you have all of your data in 16-byte blocks, but what do you do when you have less than that? One approach is to append some extra data to the end of your plaintext. This extra data is known as *padding*.

The padding can't consist of just any random data. It must follow some format that allows the receiver to see the padding for what it is and know exactly how many bytes to discard. In TLS, the last byte of an encryption block contains padding length, which indicates how many bytes of padding (excluding the padding length byte) there are. All padding bytes are set to the same value as the padding length byte. This approach enables the receiver to check that the padding is correct.

Figure 1.3. Example of TLS padding



To discard the padding after decryption, the receiver examines the last byte in the data block and removes it. After that, he removes the indicated number of bytes while checking that they all have the same value.

Hash Functions

A *hash function* is an algorithm that converts input of arbitrary length into fixed-size output. The result of a hash function is often called simply a *hash*. Hash functions are common-

[11] Advanced Encryption Standard (Wikipedia, retrieved 1 June 2014)

ly used in programming, but not all hash functions are suitable for use in cryptography. *Cryptographic hash functions* are hash functions that have several additional properties:

Preimage resistance
: Given a hash, it's computationally unfeasible to find or construct a message that produces it.

Second preimage resistance
: Given a message and its hash, it's computationally unfeasible to find a different message with the same hash.

Collision resistance
: It's computationally unfeasible to find two messages that have the same hash.

Hash functions are most commonly used as a compact way to represent and compare large amounts of data. For example, rather than compare two files directly (which might be difficult, for example, if they are stored in different parts of the world), you can compare their hashes. Hash functions are often called *fingerprints*, *message digests*, or simply *digests*.

The most commonly used hash function today is SHA1, which has output of 160 bits. Because SHA1 is considered weak, upgrading to its stronger variant, SHA256, is recommended. Unlike with ciphers, the strength of a hash function doesn't equal the hash length. Because of the *birthday paradox* (a well-known problem in probability theory),[12] the strength of a hash function is at most one half of the hash length.

Message Authentication Codes

A hash function could be used to verify data integrity, but only if the hash of the data is transported separately from the data itself. Otherwise, an attacker could modify both the message and the hash, easily avoiding detection. A *message authentication code* (MAC) or a *keyed-hash* is a cryptographic function that extends hashing with authentication. Only those in possession of the *hashing key* can produce a valid MAC.

MACs are commonly used in combination with encryption. Even though Mallory can't decrypt ciphertext, she can modify it in transit if there is no MAC; *encryption provides confidentiality but not integrity*. If Mallory is smart about how she's modifying ciphertext, she could trick Bob into accepting a forged message as authentic. When a MAC is sent along with ciphertext, Bob (who shares the hashing key with Alice) can be sure that the message has not been tampered with.

Any hash function can be used as the basis for a MAC using a construction known as HMAC (short for *hash-based message authentication code*).[13] In essence, HMAC works by interleaving the hashing key with the message in a secure way.

[12] Birthday problem (Wikipedia, retrieved 6 June 2014)

[13] RFC 2104: HMAC: Keyed-Hashing for Message Authentication (Krawczyk et al., February 1997)

Block Cipher Modes

Block cipher modes are cryptographic schemes designed to extend block ciphers to encrypt data of arbitrary length. All block cipher modes support confidentiality, but some combine it with authentication. Some modes transform block ciphers to produce stream ciphers.

There are many output modes, and they are usually referred to by their acronyms: ECB, CBC, CFB, OFB, CTR, GCM, and so forth. (Don't worry about what the acronyms stand for.) I will cover only ECB and CBC here: ECB as an example of how not to design a block cipher mode and CBC because it's still the main mode in SSL and TLS. GCM is a relatively new addition to TLS, available starting with version 1.2; it provides confidentiality and integrity, and it's currently the best mode available.

Electronic Codebook Mode

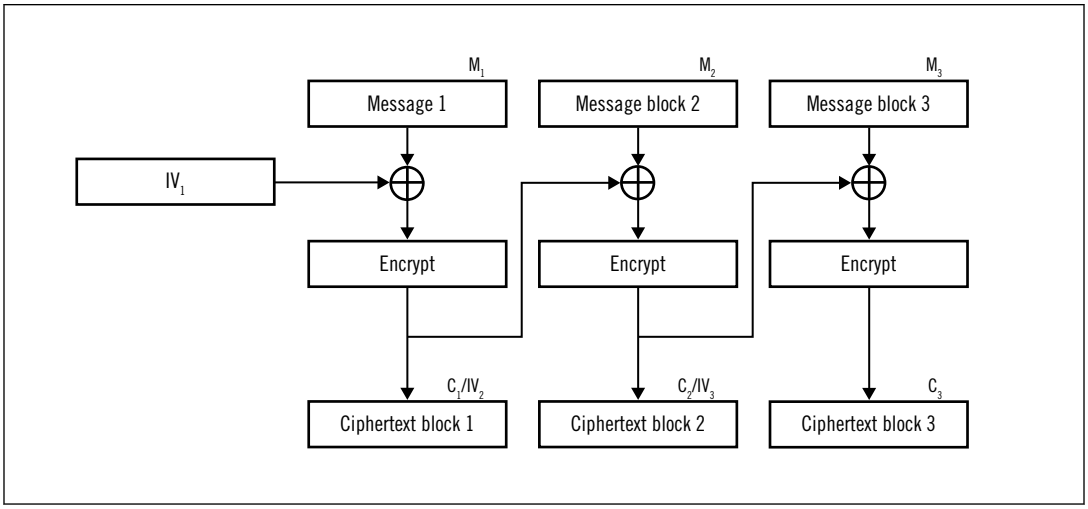
Electronic Codebook (ECB) mode is the simplest possible block cipher mode. It supports only data lengths that are the exact multiples of the block size; if you have data of different length, then you need to apply padding beforehand. To perform encryption, you split the data into chunks that match the block size and encrypt each block individually.

The simplicity of ECB is its downside. Because block ciphers are deterministic (i.e., they always produce the same result when the input is the same), so is ECB. This has serious consequences: (1) patterns in ciphertext will appear that match patterns in plaintext; (2) the attacker can detect when a message is repeated; and (3) an attacker who can observe ciphertext and submit arbitrary plaintext for encryption (commonly possible with HTTP and in many other situations) can, given enough attempts, *guess* the plaintext. This is what the BEAST attack against TLS was about; I discuss it in the section called "BEAST" in Chapter 7.

Cipher Block Chaining Mode

Cipher Block Chaining (CBC) mode is the next step up from ECB. To address the deterministic nature of ECB, CBC introduces the concept of the *initialization vector* (IV), which makes output different every time, even when input is the same.

Figure 1.4. CBC mode



The process starts by generating a random (and thus unpredictable) IV, which is the same length as the encryption block size. Before encryption, the first block of plaintext is combined with the IV using XOR. This masks the plaintext and ensures that the ciphertext is always different. For the next encryption block, the ciphertext of the previous block is used as the IV, and so forth. As a result, all of the individual encryption operations are part of the same *chain*, which is where the mode name comes from. Crucially, the IV is transmitted on the wire to the receiving party, who needs it to perform decryption successfully.

Asymmetric Encryption

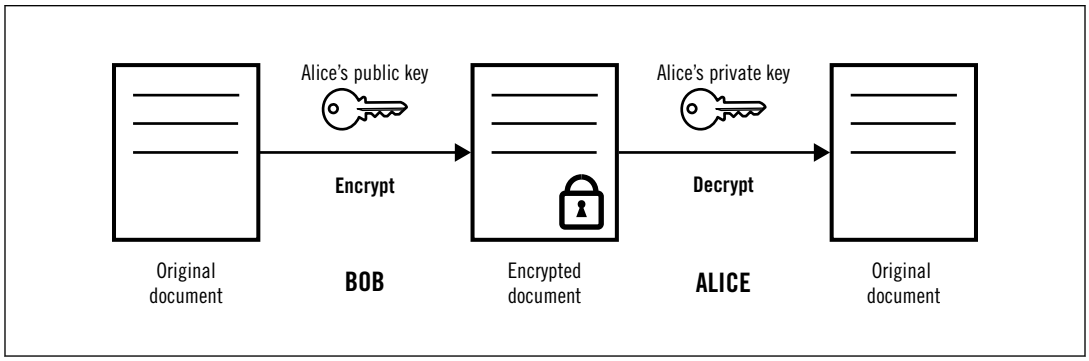
Symmetric encryption does a great job at handling large amounts of data at great speeds, but it leaves a lot to be desired as soon as the number of parties involved increases:

- Members of the same group must share the same key. The more people join a group, the more exposed the group becomes to the key compromise.
- For better security, you could use a different key for every two people, but this approach doesn't scale. Although three people need only three keys, ten people would need 45 (9 + 8 + . . . + 1) keys. A thousand people would need 499,550 keys!
- Symmetric encryption can't be used on unattended systems to secure data. Because the process can be reversed by using the same key, a compromise of such a system leads to the compromise of all data stored in the system.

Asymmetric encryption (also known as *public-key cryptography*) is a different approach to encryption that uses two keys instead of one. One of the keys is *private*; the other is *public*. As the names suggest, one of these keys is intended to be private, and the other is intended

to be shared with everyone. There's a special mathematical relationship between these keys that enables some useful features. If you encrypt data using someone's public key, only their corresponding private key can decrypt it. On the other hand, if data is encrypted with the private key anyone can use the public key to unlock the message. The latter operation doesn't provide confidentiality, but it does function as a digital signature.

Figure 1.5. Asymmetric encryption



Asymmetric encryption makes secure communication in large groups much easier. Assuming that you can securely share your public key widely (a job for PKI, which I discuss in Chapter 3, *Public-Key Infrastructure*), anyone can send you a message that only you can read. If they also sign that message using their private key, you know exactly whom it is from.

Despite its interesting properties, public-key cryptography is rather slow and unsuitable for use with large quantities of data. For this reason, it's usually deployed for authentication and negotiation of shared secrets, which are then used for fast symmetric encryption.

RSA (named from the initials of Ron Rivest, Adi Shamir, and Leonard Adleman) is by far the most popular asymmetric encryption method deployed today.[14] The recommended strength for RSA today is 2,048 bits, which is equivalent to about 112 symmetric bits. I'll discuss the strength of cryptography in more detail later in this chapter.

Digital Signatures

A *digital signature* is a cryptographic scheme that makes it possible to verify the authenticity of a digital message or document. The MAC, which I described earlier, is a type of digital signature; it can be used to verify authenticity provided that the secret hashing key is securely exchanged ahead of time. Although this type of verification is very useful, it's limited because it still relies on a private secret key.

[14] RSA (Wikipedia, retrieved 2 June 2014)

Digital signatures similar to the real-life handwritten ones are possible with the help of public-key cryptography; we can exploit its asymmetric nature to devise an algorithm that allows a message signed by a private key to be verified with the corresponding public key.

The exact approach depends on the selected public-key cryptosystem. For example, RSA can be used for encryption and decryption. If something is encrypted with a private RSA key, only the corresponding public key can decrypt it. We can use this property for digital signing if we combine it with hash functions:

1. Calculate a hash of the document you wish to sign; no matter the size of the input document, the output will always be fixed, for example, 256 bits for SHA256.
2. Encode the resulting hash and some additional metadata. For example, the receiver will need to know the hashing algorithm you used before she can process the signature.
3. Encrypt the encoded hash using the private key; the result will be the signature, which you can append to the document as proof of authenticity.

To verify the signature, the receiver takes the document and calculates the hash independently using the same algorithm. Then, she uses your public key to decrypt the message and recover the hash, confirm that the correct algorithms were used, and compare with the decrypted hash with the one she calculated. The strength of this signature scheme depends on the individual strengths of the encryption, hashing, and encoding components.

> **Note**
>
> Not all digital signature algorithms function in the same way as RSA. In fact, RSA is an exception, because it can be used for both encryption and digital signing. Other popular public key algorithms, such as DSA and ECDSA, can't be used for encryption and rely on different approaches for signing.

Random Number Generation

In cryptography, all security depends on the quality of random number generation. You've already seen in this chapter that security relies on known encryption algorithms and secret keys. Those keys are simply very long random numbers.

The problem with random numbers is that computers tend to be very predictable. They follow instructions to the letter. If you tell them to generate a random number, they probably won't do a very good job.[15] This is because truly random numbers can be obtained only by observing certain physical processes. In absence of that, computers focus on collecting small

[15] Some newer processors have built-in random number generators that are suitable for use in cryptography. There are also specialized external devices (e.g., in the form of USB sticks) that can be added to feed additional entropy to the operating system.

amounts of *entropy*. This usually means monitoring keystrokes and mouse movement and the interaction with various peripheral devices, such as hard disks.

Entropy collected in this way is a type of *true random number generator* (TRNG), but the approach is not reliable enough to use directly. For example, you might need to generate a 4,096-bit key, but the system might have only a couple of hundreds of bits of entropy available. If there are no reliable external events to collect enough entropy, the system might stall.

For this reason, in practice we rely on *pseudorandom number generators* (PRNGs), which use small amounts of true random data to get them going. This process is known as *seeding*. From the seed, PRNGs produce unlimited amounts of pseudorandom data on demand. General-purpose PRNGs are often used in programming, but they are not appropriate for cryptography, even if their output is statistically seemingly random. *Cryptographic pseudorandom number generators* (CPRNGs) are PRNGs that are also unpredictable. This attribute is crucial for security; an adversary mustn't be able to reverse-engineer the internal state of a CPRNG by observing its output.

Protocols

Cryptographic primitives such as encryption and hashing algorithms are seldom useful by themselves. We combine them into *schemes* and *protocols* so that we can satisfy complex security requirements. To illustrate how we might do that, let's consider a simplistic cryptographic protocol that allows Alice and Bob to communicate securely. We'll aim for all three main requirements: confidentiality, integrity, and authentication.

Let's assume that our protocol allows exchange of an arbitrary number of messages. Because symmetric encryption is very good at encrypting bulk data, we might select our favorite algorithm to use for this purpose, say, AES. With AES, Alice and Bob can exchange secure messages, and Mallory won't be able to recover the contents. But that's not quite enough, because Mallory can do other things, for example, modify the messages without being detected. To fix this problem, we can calculate a MAC of each message using a hashing key known only to Alice and Bob. When we send a message, we send along the MAC as well.

Now, Mallory can't modify the messages any longer. However, she could still drop or replay arbitrary messages. To deal with this, we extend our protocol to assign a sequence number to each message; crucially, we make the sequences part of the MAC calculation. If we see a gap in the sequence numbers, then we know that there's a message missing. If we see a sequence number duplicate, we detect a replay attack. For best results, we should also use a special message to mark the end of the conversation. Without such a message, Mallory would be able to end (truncate) the conversation undetected.

With all of these measures in place, the best Mallory can do is prevent Alice and Bob from talking to one another. There's nothing we can do about that.

So far, so good, but we're still missing a big piece: how are Alice and Bob going to negotiate the two needed keys (one for encryption and the other for integrity validation) in the presence of Mallory? We can solve this problem by adding two additional steps to the protocol.

First, we use public-key cryptography to authenticate each party at the beginning of the conversation. For example, Alice could generate a random number and ask Bob to sign it to prove that it's really him. Bob could ask Alice to do the same.

With authentication out of the way, we can use a *key-exchange scheme* to negotiate encryption keys securely. For example, Alice could generate all the keys and send them to Bob by encrypting them with his public key; this is how the RSA key exchange works. Alternatively, we could have also used a protocol known as *Diffie-Hellman* (DH) key exchange for this purpose. The latter is slower, but it has better security properties.

In the end, we ended up with a protocol that (1) starts with a handshake phase that includes authentication and key exchange, (2) follows with the data exchange phase with confidentiality and integrity, and (3) ends with a shutdown sequence. At a high level, our protocol is similar to the work done by SSL and TLS.

Attacking Cryptography

Complex systems can usually be attacked in a variety of ways, and cryptography is no exception. First, you can attack the cryptographic primitives themselves. If a key is small, the adversary can use brute force to recover it. Such attacks usually require a lot of processing power as well as time. It's easier (for the attacker) if the used primitive has known vulnerabilities, in which case he can use analytic attacks to achieve the goal faster.

Cryptographic primitives are generally very well understood, because they are relatively straightforward and do only one thing. Schemes are often easier to attack because they introduce additional complexity. In some cases, even cryptographers argue about the right way to perform certain operations. But both are relatively safe compared to protocols, which tend to introduce far more complexity and have a much larger attack surface.

Then, there are attacks against protocol *implementation*; in other words, exploitation of software bugs. For example, most cryptographic libraries are written in low-level languages such as C (and even assembly, for performance reasons), which make it very easy to introduce catastrophic programming errors. Even in the absence of bugs, sometimes great skill is needed to implement the primitives, schemes, and protocols in such a way that they can't be abused. For example, naïve implementations of certain algorithms can be exploited in *timing attacks*, in which the attacker breaks encryption by observing how long certain operations take.

It is also common that programmers with little experience in cryptography nevertheless attempt to implement—and even design—cryptographic protocols and schemes, with predictably insecure results.

For this reason, it is often said that cryptography is bypassed, not attacked. What this means is that the primitives are solid, but the rest of the software ecosystem isn't. Further, the keys are an attractive target: why spend months to brute-force a key when it might be much easier to break into a server to obtain it? Many cryptographic failures can be prevented by following simple rules such as these: (1) use well-established protocols and never design your own schemes; (2) use high-level libraries and never write code that deals with cryptography directly; and (3) use well-established primitives with sufficiently strong key sizes.

Measuring Strength

We measure the strength of cryptography using the number of operations that need to be performed to break a particular primitive, presented as *bits* of security. Deploying with strong key sizes is the easiest thing to get right, and the rules are simple: 128 bits of security (2^{128} operations) is sufficient for most deployments; use 256 bits if you need very long-term security or a big safety margin.

> **Note**
>
> The strength of symmetric cryptographic operations increases exponentially as more bits are added. This means that increasing key size by one bit makes it twice as strong.

In practice, the situation is somewhat more complicated, because not all operations are equivalent in terms of security. As a result, different bit values are used for symmetric operations, asymmetric operations, elliptic curve cryptography, and so on. You can use the information in Table 1.2, "Security levels and equivalent strength in bits, adapted from ECRYPT2 (2012)" to convert from one size to another.

Table 1.2. Security levels and equivalent strength in bits, adapted from ECRYPT2 (2012)

#	Protection	Symmetric	Asymmetric	DH	Elliptic Curve	Hash
1	Attacks in real time by individuals	32	-	-	-	-
2	Very short-term protection against small organizations	64	816	816	128	128
3	Short-term protection against medium organizations	72	1,008	1,008	144	144
4	Very short-term protection against agencies	80	1,248	1,248	160	160
5	Short-term protection (10 years)	96	1,776	1,776	192	192
6	Medium-term protection (20 years)	112	2,432	2,432	224	224
7	Long-term protection (30 years)	128	3,248	3,248	256	256
8	Long-term protection and increased defense from quantum computers	256	15,424	15,424	512	512

The data, which I adapted from a 2012 report on key and algorithm strength,[16] shows rough mappings from bits of one type to bits of another, but it also defines strength in relation to attacker capabilities and time. Although we tend to discuss whether an asset is secure (assuming *now*), in reality security is a function of time. The strength of encryption changes, because as time goes by computers get faster and cheaper. Security is also a function of resources. A key of a small size might be impossible for an individual to break, but doing so could be within the reach of an agency. For this reason, when discussing security it's more useful to ask questions such as "secure against whom?" and "secure for how long?"

> **Note**
>
> The strength of cryptography can't be measured accurately, which is why you will find many different recommendations. Most of them are very similar, with small differences. In my experience, ENISA (the *European Union Agency for Network and Information Security*) provides useful high-level documents that offer clear guidance[17] at various levels.[18] To view and compare other recommendations, visit *keylength.com*.[19]

Although the previous table provides a lot of useful information, you might find it difficult to use because the values don't correspond to commonly used key sizes. In practice, you'll find the following table more useful to convert from one set of bits to another:[20]

Table 1.3. Encryption strength mapping for commonly used key sizes

Symmetric	RSA / DSA / DH	Elliptic curve crypto	Hash
80	1,024	160	160
112	2,048	224	224
128	3,072	256	256
256	15,360	512	512

Man-in-the-Middle Attack

Most attacks against transport-layer security come in the form of a *man-in-the-middle* (MITM) attack. What this means is that in addition to the two parties involved in a conversation there is a malicious party. If the attacker is just listening in on the conversation, we're talking about a *passive network attack*. If the attacker is actively modifying the traffic or influencing the conversation in some other way, we're talking about an *active network attack*.

[16] ECRYPT2 Yearly Report on Algorithms and Keysizes (European Network of Excellence for Cryptology II, 30 September 2012)

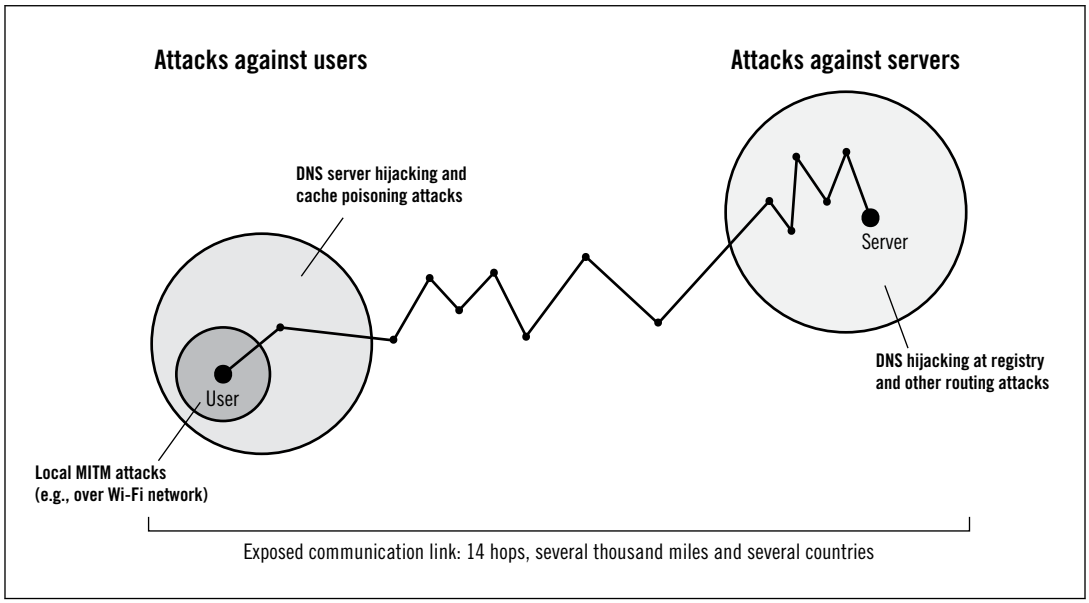
[17] Algorithms, Key Sizes and Parameters Report (ENISA, 29 October 2013)

[18] Recommended cryptographic measures - Securing personal data (ENISA, 4 November 2013)

[19] BlueKrypt: Cryptographic Key Length Recommendation (BlueKrypt, retrieved 4 June 2014)

[20] NIST Special Publication 800-57: Recommendation for Key Management – Part 1: General, Revision 3 (NIST, July 2012)

Figure 1.6. Conceptual SSL/TLS threat model



Gaining Access

In many cases, attacks require proximity to the victim or the server or access to the communication infrastructure. Whoever has access to the cables and intermediary communication nodes (e.g., routers) can see the packets as they travel across the wire and interfere with them. Access can be obtained by tapping the cables,[21] in collaboration with telecoms,[22] or by hacking the equipment.[23]

Conceptually, the easiest way to execute a MITM attack is by joining a network and rerouting the victims' traffic through a malicious node. Wireless networks without authentication, which so many people use these days, are particularly vulnerable, because anyone can join.

Other ways to attack include interfering with the routing infrastructure for domain name resolution, IP address routing, and so on.

ARP spoofing

Address Resolution Protocol (ARP) is used on local networks to associate network MAC addresses[24] with IP addresses. An attacker with access to the network can claim any IP address and effectively reroute traffic.

[21] The Creepy, Long-Standing Practice of Undersea Cable Tapping (The Atlantic, 16 July 2013)

[22] New Details About NSA's Collaborative Relationships With America's Biggest Telecom Companies From Snowden Docs (Washington Post, 30 August 2013)

[23] Photos of an NSA "upgrade" factory show Cisco router getting implant (Ars Technica, 14 May 2014)

[24] In this case, MAC stands for *media access control*. It's a unique identifier assigned to networking cards during manufacture.

WPAD hijacking

Web Proxy Auto-Discovery Protocol (WPAD) is used by browsers to automatically retrieve HTTP proxy configuration. WPAD uses several methods, including DHCP and DNS. To attack WPAD, an attacker starts a proxy on the local network and announces it to the local clients who look for it.

DNS hijacking

By hijacking a domain name with the registrar or changing the DNS configuration, an attacker can hijack all traffic intended for that domain name.

DNS cache poisoning

DNS cache poisoning is a type of attack that exploits weaknesses in caching DNS servers and enables the attacker to inject invalid domain name information into the cache. After a successful attack, all users of the affected DNS server will be given invalid information.

BGP route hijacking

Border Gateway Protocol (BGP) is a routing protocol used by the core internet routers to discover where exactly IP address blocks are located. If an invalid route is accepted by one or more routers, all traffic for a particular IP address block can be redirected elsewhere, that is, to the attacker.

Passive Attacks

Passive attacks are most useful against unencrypted traffic. During 2013, it became apparent that government agencies around the world routinely monitor and store large amounts of internet traffic. For example, it is alleged that GCHQ, the British spy agency, records all UK internet traffic and keeps it for three days.[25] Your email messages, photos, internet chats, and other data could be sitting in a database somewhere, waiting to be cross-referenced and correlated for whatever purpose. If bulk traffic is handled like this, it's reasonable to expect that specific traffic is stored for much longer and perhaps indefinitely. In response to this and similar discoveries, the IETF declared that "pervasive monitoring is an attack" and should be defended against by using encryption whenever possible.[26]

Even against encrypted traffic, passive attacks can be useful as an element in the overall strategy. For example, you could store captured encrypted traffic until such a time when you can break the encryption. Just because some things are difficult to do today doesn't mean that they'll be difficult ten years from now, as computers get more powerful and cheaper and as weaknesses in cryptographic primitives are discovered.

To make things worse, computer systems often contain a critical configuration weakness that allows for retroactive decryption of recorded traffic. The most common key-exchange

[25] GCHQ taps fibre-optic cables for secret access to world's communications (The Guardian, 21 June 2013)

[26] RFC 7258: Pervasive Monitoring Is an Attack (S. Farrell and H. Tschofenig, May 2014)

mechanism in TLS is based on the RSA algorithm; on the systems that use this approach, the RSA key used for the key exchange can also be used to decrypt all previous conversations. Other key-exchange mechanisms don't suffer from this problem and are said to support *forward secrecy*. Unfortunately, most stay with the RSA algorithm. For example, Lavabit, the encrypted email service famously used by Edward Snowden, didn't support forward secrecy. Using a court order, the FBI compelled Lavabit to disclose their encryption key.[27] With the key in their possession, the FBI could decrypt any recorded traffic (if they had any, of course).

Passive attacks work very well, because there is still so much unencrypted traffic and because when collecting in bulk the process can be fully automated. As an illustration, in July 2014 only 58% of email arriving to Gmail was encrypted.[28]

Active Attacks

When someone talks about MITM attacks, they most commonly refer to active network attacks in which Mallory interferes with the traffic in some way. Traditionally, MITM attacks target authentication to trick Alice into thinking she's talking to Bob. If the attack is successful, Mallory receives messages from Alice and forwards them to Bob. The messages are encrypted when Alice sends them, but that's not a problem, because she's sending them to Mallory, who can decrypt them using the keys she negotiated with Alice.

When it comes to TLS, the ideal case for Mallory is when she can present a certificate that Alice will accept as valid. In that case, the attack is seamless and almost impossible to detect.[29] A valid certificate could be obtained by playing the public key infrastructure ecosystem. There have been many such attacks over the years; in Chapter 4, *Attacks against PKI* I document the ones that are publicly known. A certificate that *seems* valid could be constructed if there are bugs in the validation code that could be exploited. Historically, this is an area in which bugs are common. I discuss several examples in Chapter 6, *Implementation Issues*. Finally, if everything else fails, Mallory could present an invalid certificate and hope that Alice overrides the certificate warning. This happened in Syria a couple of years ago.[30]

The rise of browsers as a powerful application-delivery platform created additional attack vectors that can be exploited in active network attacks. In this case, authentication is not attacked, but the victims' browsers are instrumented by the attacker to submit specially crafted requests that are used to subvert encryption. These attack vectors have been exploited in recent years to attack TLS in novel ways; you can find more information about them in Chapter 7, *Protocol Attacks*.

---

[27] Lavabit (Wikipedia, retrieved 4 June 2014)

[28] Transparency Report: Email encryption in transit (Google Gmail, retrieved 27 July 2014)

[29] Unless you're very, very paranoid, and keep track of all the certificates previously encountered. There are some browser add-ons that do this (e.g., Certificate Patrol for Firefox).

[30] A Syrian Man-In-The-Middle Attack against Facebook (The Electronic Frontier Foundation, 5 May 2011)

Active attacks can be very powerful, but they're more difficult to scale. Whereas passive attacks only need to make copies of observed packets (which is a simple operation), active attacks require much more processing and effort to track individual connections. As a result, they require much more software and hardware. Rerouting large amounts of traffic is difficult to do without being noticed. Similarly, fraudulent certificates are difficult to use successfully for large-scale attacks because there are so many individuals and organizations who are keeping track of certificates used by various web sites. The approach with the best chance of success is exploitation of implementation bugs that can be used to bypass authentication, but such bugs, devastating as they are, are relatively rare.

For these reasons, active attacks are most likely to be used against individual, high-value targets. Such attacks can't be automated, which means that they require extra work, cost a lot, and are thus more difficult to justify.

There are some indications that the NSA deployed extensive infrastructure that enables them to attack almost arbitrary computers on the Internet, under the program called *QuantumInsert*.[31]

This program, which is a variation on the MITM theme, doesn't appear to target encryption; instead, it's used to deliver browser exploits against selected individuals. By placing special packet-injection nodes at important points in the communication infrastructure, the NSA is able to respond to connection requests faster than the real servers and redirect some traffic to the exploitation servers instead.

[31] Attacking Tor: How the NSA Targets Users' Online Anonymity (Bruce Schneier, 4 October 2013)