Securing DevOps

Security in the cloud

Julien Vehent

Sample Chapter

MANNING

Securing DevOps

by Julien Vehent

Chapter 5

brief contents

5 Security layer 3: securing communications

This chapter covers

- Understanding the concepts and vocabulary of Transport Layer Security
- Establishing a secure connection between a web browser and a server
- Obtaining certificates from AWS and Let's Encrypt
- Configuring HTTPS on the application's public endpoint
- Modernizing HTTPS using Mozilla's guidelines

The application controls added in chapter 3 and infrastructure controls added in chapter 4 are all critical to guaranteeing that customer data is stored safely and protected against theft and integrity loss. We have, so far, focused our efforts on the hosting environment and ignored a large security hole: data transiting between the user and the service is left unprotected and can be stolen or modified by anyone in the pathway. In this chapter, I explain how to bring confidentiality and integrity to network communications using HTTPS.

HTTPS is composed of HTTP, the application protocol of the web, and Transport Layer Security, or TLS, the most widely used cryptographic protocol on the internet. Most of the security controls provided by HTTPS come from TLS, and we'll logically spend most of this chapter exploring how to use this protocol correctly. What isn't covered by TLS directly requires enabling controls at the HTTP level, so we'll discuss HTTP Strict Transport Security (HSTS) and HTTP Public Key Pinning (HPKP) near the end of the chapter.

If you've never worked with TLS or cryptographic protocols, you may find a lot of its jargon foreign to you. Terms like "certificate authorities," "public key infrastructure," and "perfect forward secrecy" are part of the vocabulary of security engineers, and understanding them is an important goal of the chapter. We'll start this chapter by discussing these terms, where they come from, and how they relate to HTTPS.

5.1 What does it mean to secure communications?

The security of a communication channel depends on three core properties, illustrated in figure 5.1:

- *Confidentiality*—Only the legitimate participants of the discussion must be able to access the information.
- *Integrity*—Messages exchanged between the participants must not be modified in transit.
- *Authenticity*—Participants of the discussion must be able to prove their identity to each other.

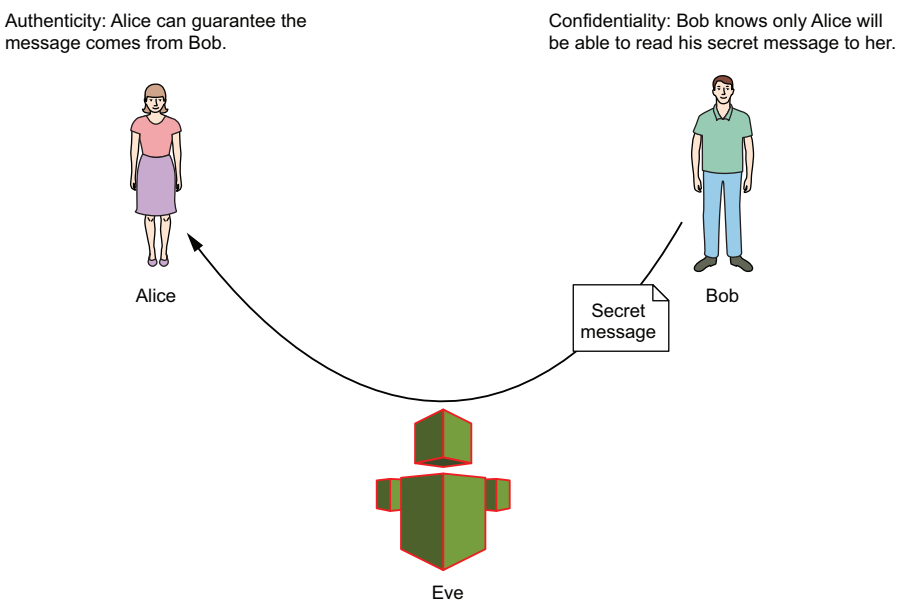


Figure 5.1 Confidentiality, authenticity, and integrity are the core security properties that allow Alice and Bob to communicate safely and prevent Eve from interfering.

TLS provides all three properties, which is no small feat. To explain how TLS achieves this, we need to go back in time and discuss the origins of cryptography. The sophistication we reached today comes from solving increasingly complex security problems over centuries of scientific progress. For those who already have a security background, feel free to skip ahead to section 5.2 "Understanding SSL/TLS."

5.1.1 Early symmetric cryptography

In the early days, not all three properties were guaranteed, and early security protocols focused primarily on confidentiality. Caesar's substitution cipher is an example of an early cryptographic protocol used by the Roman general in his private correspondence. Caesar's cipher required participants to share a number to shift their alphabet by and encrypt or decrypt messages with it. The following listing shows a simple example of substitution cipher that uses an alphabet shifted by seven letters.

Listing 5.1 Encrypting and decrypting using a simple substitution cipher

```
key: 7
alphabet: abcdefghijklmnopqrstuvwxyz
shifted : hijklmnopqrstuvwxyzabcdefg
cleartext:  attack the southern gate at dawn
ciphertext: haahjr aol zvbaolyu nhal ha khdu
```

The recipient of the `ciphertext` must first possess the key to decrypt the message, which could be agreed on in person before exchanging messages. Because the same key is used to encrypt and decrypt a message, we call it a *symmetric encryption protocol*. Besides having an impractical key-management process, this protocol also lacks integrity and authenticity protection:

- *The* ciphertext *can be modified in transit even by an attacker who isn't able to decrypt it.* This would likely lead to making the clear text unintelligible, but the recipient has no way to differentiate between message tampering and author inebriation.
- *There's no proof that the message originates from the expected author.* Someone else could crack the key and issue fraudulent messages, which would be a great way to mislead an adversary, as shown in figure 5.2.

Both problems led cryptographers to protect messages with seals, initially made of beeswax and later colored red. The author of a message would apply their own seal to close a letter, and the recipient could verify the seal was intact upon reception. As long as an attacker was unable to reproduce a seal, the protocol was safe, and confidentiality, integrity, and authenticity were provided. Even today, sealing messages is an important part of the TLS protocol.

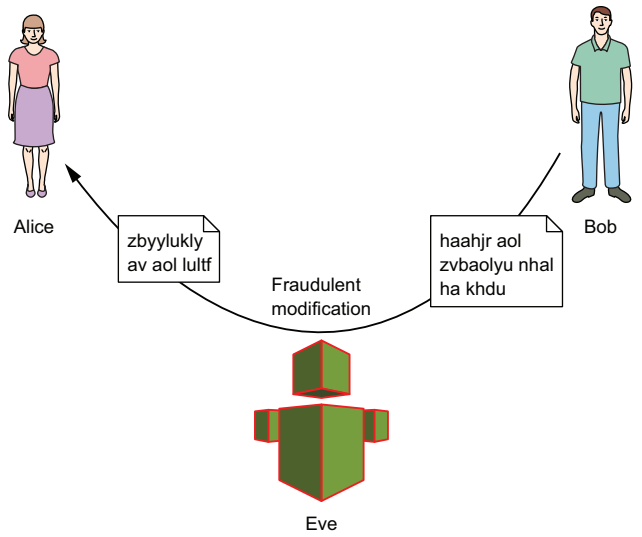


Figure 5.2 The lack of authentication and integrity in Caesar's cipher allows Eve to replace Bob's secret message with her own. Can you decrypt it?

5.1.2 Diffie-Hellman and RSA

Centuries of progress and hundreds of cryptosystems have improved on Caesar's cipher and produced algorithms that were harder and harder to crack, but the problem of securely sharing cryptographic keys between participants remained a weakness in any communication system.

Exchanging keys in person has always been the safest way to guarantee a key belongs to the right person, and no one modified it in transit (OpenPGP key signing still uses this method in its web of trust), but isn't a protocol that works across continents when people can't meet directly. After World War II, scientists and engineers spent more time and effort than ever perfecting cryptographic protocols to protect the fast-growing communication networks that would soon become the internet. With more and more participants in distant locations, the pressure on the shared-encryption-key problem increased rapidly.

A breakthrough happened when Whitfield Diffie and Martin Hellman (with the help of Ralph Merkle) published the Diffie-Hellman (DH) key-exchange algorithm in 1976. Using Diffie-Hellman exchange (DHE), two people can start a communication channel by first performing a key-exchange protocol that produces an encryption key. The encryption key itself never transits on the wire, and the only values exchanged publicly can't be used to deduce the encryption key. In effect, DH is a way to securely agree on an encryption key over a public network, while preventing eavesdroppers from learning anything useful about the key. The exchanged key can then be used to encrypt and decrypt messages.

The Diffie-Hellman key exchange

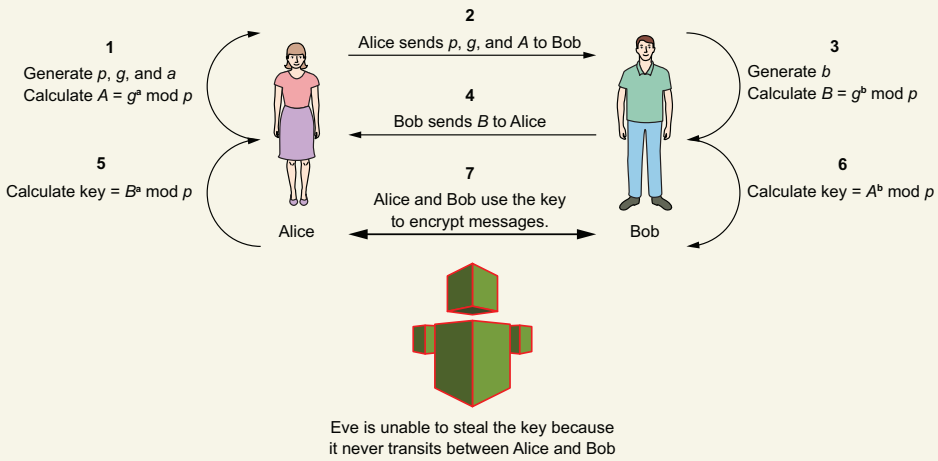
The mathematics behind the Diffie-Hellman algorithm can be understood with only high school math. Alice and Bob want to agree on an encryption key to exchange messages securely.

1. Alice picks a prime number, p, a generator, g, and a random secret, a. Alice calculates the value of $A=g^a \bmod p$, and sends p, g, and A to Bob.
2. Upon reception, Bob generates a random secret, b, calculates $B=g^b \bmod p$, and sends B to Alice.

Both Alice and Bob now share enough information to calculate the encryption key. Alice calculates key=$B^a \bmod p$, and Bob calculates key=$A^b \bmod p$. They both end up with the same value for the key, without that value ever crossing the wire.

Diffie-Hellman key exchange with small values

```
Alice generates prime p=23, generator g=5 and random secret a=6
Alice calculates A = g^a mod p = 56 mod 23 = 8
Alice sends p=23, g=5 and A=8 to Bob
         Bob generates secret b=15
         Bob calculates B = g^b mod p = 515 mod 23 = 19
         Bob sends B=19 to Alice
Alice calculates the key = B^a mod p = 196 mod 23 = 2
         Bob calculates the key = A^b mod p = 815 mod 23 = 2
Alice and Bob have negotiated key=2
```



The Diffie-Hellman key exchange allows Alice and Bob to exchange a key without Eve being able to steal it.

Diffie-Hellman created a tidal wave in the cryptographic world. Because the algorithm uses public and private values (a and b are private, A and B are public), it's said that Diffie-Hellman invented asymmetric public-key encryption.

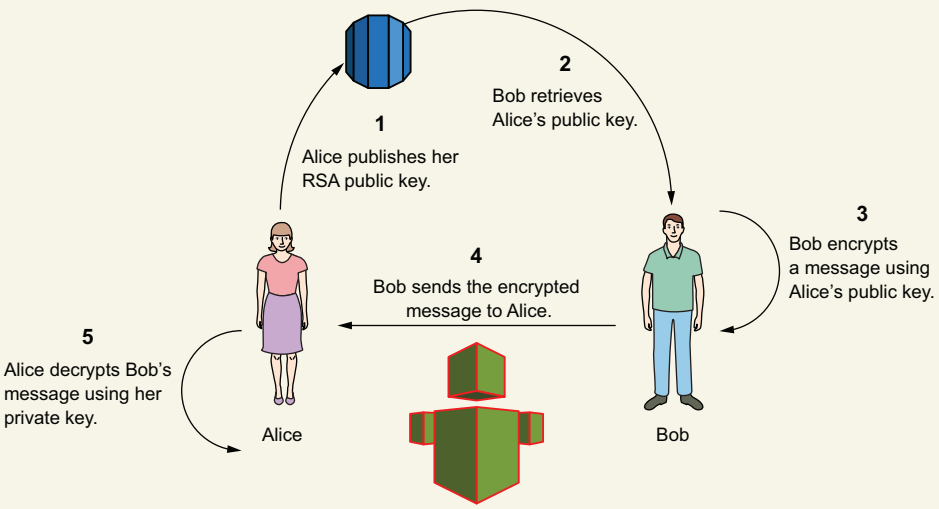
A year after the publication of DH, Ron Rivest, Adi Shamir, and Leonard Adleman published RSA, a public-key cryptosystem that built on top of the DH algorithm and introduced the public and private keys we still use today. RSA provides a way for individuals to create their own pair of keys: one public key to share with the world, and one private key to keep private. RSA provides two important security features—encryption and signature:

- *Encryption*—Messages encrypted with one key can only be decrypted by the other, allowing people to send each other messages using their respective public keys for encryption and private keys for decryption.
- *Signature*—Messages encrypted by someone's private key can only be decrypted by the corresponding public key, proving the holder of the private key issued the message and effectively providing a digital signature.

Take a moment to understand these concepts. They're complex but foundational to how TLS secures communications today. DH are RSA are the security building blocks that allowed the internet to prosper as a marketplace.

The RSA algorithm

The RSA algorithm enables participants of a communication to exchange secret messages using two keys. When one key encrypts a message, the other key can decrypt it, but the key that encrypted can't decrypt. Imagine two participants, Alice and Bob, who want to communicate securely. Alice creates a key-pair and puts her public key on the internet. Bob takes Alice's public key and encrypts a message with it. No one else can decrypt that message but Alice, who securely keeps the private key that can decrypt the message. The following figure illustrates the RSA workflow.



The RSA cryptosystem allows Bob to send a message to Alice encrypted with her public key. Eve can't decrypt the message because she doesn't know Alice's private key.

(continued)

This two-key system is revolutionary because one of the keys can be published without reducing the security of the protocol. If you're curious about the mathematics of RSA, a simple example is shown here:

1. Select two random prime numbers, p and q, and calculate $n = p*q$.

   $p = 17, q = 13$

   $n = p \times q = 221$

2. Calculate $\varphi(n)$, the great common divisor of $(p\text{-}1)(q\text{-}1)$.

   $\varphi(n) = (p - 1) \times (q - 1) = 16 \times 12 = 192$

3. Pick any public exponent, e, that's prime with $\varphi(n)$. Here, we take e=5, but a common value is e=65537. The value of e and n together forms the public key.
4. Using e, select a value for d that satisfies the formula: $d*e \bmod \varphi(n)=1$. For example, d=77.

   $d \times 5 \bmod 192 = 1$

   $77 \times 5 \bmod 192 = 1$

5. The value of d and n together form the private key.

Take a message, m, which is the number 123. To encrypt m with the public key (n, e), we use $c(m)=m^e \bmod n=123^5 \bmod 221=106$. The encrypted text $c(m)$ is the value 106.

Now, to decrypt $c(m)$ with the private key (d, n) and get back the original message, we calculate $\text{cleartext}=c(m)^d \bmod n=106^{77} \bmod 221=123$.

5.1.3 Public-key infrastructures

RSA provides almost all the security necessary to secure a communication, but one problem remains. Imagine you're communicating with Bob for the first time. Bob tells you his public key is *29931229*. You haven't established a secure channel yet, so how can you be sure that someone isn't tampering with this information via a *man-in-the-middle* (MITM)? You have no proof, unless someone else can confirm that this is indeed Bob's public key.

In the real world, this problem is similar to how we trust passports and driver's licenses: possessing the document itself isn't enough. It must come from a trusted authority, like a local government agency (for a driver's license) or a foreign government (for a passport). In the digital world, we took this exact same notion and created *public-key infrastructures* (PKI) to link keys to identities.

The PKI works by first trusting a set of authorities, or more specifically trusting their public keys. In web browsers, you encounter those authorities under the name *certificates*

authorities (CA) that are kept in *root stores*, or *trust stores*. The concept of the PKI is simple: the public key of Bob must be cryptographically signed by the private key of a CA you trust to be considered valid. When Bob sends you his public key, he also sends you the signature of his public key performed by the CA. By verifying the signature using the CA's public key, which you trust, you obtain the assurance that Bob's key is trustworthy and not replaced by some man in the middle. The CA must make sure to only sign keys that belong to the right people, but that's their job, not yours. In concept, this is identical to Alice's passport being signed (or rather, issued) by a trusted government that first verified her identity: because we trust the authority keys in the PKI, we carry that trust to keys signed by them.

5.1.4 SSL and TLS

It's likely that military agencies started using RSA and PKIs in the 1970s and '80s, but it took nearly two decades for the web to be built and start using these techniques. In 1995, Netscape released Navigator 1.0, which added support for the Secure Socket Layer protocol. SSL, then in version 2 (v1 was never released), uses RSA and PKIs to secure communication between a browser and a server.

SSL uses a PKI to decide if a server's public key is trustworthy by requiring servers to use a security certificate signed by a trusted CA. When Navigator 1.0 was released, it trusted a single CA operated by the RSA Data Security corporation. The server's public RSA key is stored inside the security certificate, which can then be used by the browser to establish a secure communication channel. The security certificates we use today still rely on the same standard (named X.509) that Netscape Navigator 1.0 used back then.

Netscape's intent was to train users to differentiate secure communications from insecure ones, so they put a lock icon next to the address bar. When the lock is open, the communication is insecure. A closed lock means communication has been secured with SSL, which required the server to provide a signed certificate. You're obviously familiar with this icon as it's been in every browser ever since. The engineers at Netscape truly created a standard for secure internet communications.

A year after releasing SSL 2.0, Netscape fixed several security issues and released SSL 3.0, a protocol that, albeit being officially deprecated since June 2015, remains in use in certain parts of the world more than 20 years after its introduction. In an effort to standardize SSL, the Internet Engineering Task Force (IETF) created a slightly modified SSL 3.0 and, in 1999, unveiled it as Transport Layer Security (TLS) 1.0. The name change between SSL and TLS continues to confuse people today. Officially, TLS is the new SSL, but in practice, people use SSL and TLS interchangeably to talk about any version of the protocol.

TLS continues to evolve under the supervision of the IETF: version 1.1 was released in 2006 and 1.2 in 2008. The next version of TLS, logically numbered 1.3, was released in 2018. Each new version fixes security issues and brings cryptographic innovations that we won't cover here.

TLS has become the standard for securing any kind of network communication, from serving web pages to protecting video-conference systems to establishing VPN tunnels. The amount of work devoted to securing (and breaking) its cryptographic primitives makes TLS the most reliable security protocol ever built. It also makes TLS a complex protocol that few people can grasp in its entirety.

Thankfully, you don't need a complete understanding of the inner workings of TLS to properly secure a web service. In the rest of this chapter, I give an overview of the way TLS works, and quickly move on to securing the HTTP endpoint of the invoicer.

5.2 Understanding SSL/TLS

Establishing a TLS connection is easy to do using a web browser and an HTTPS address, but to get more information about the connection establishment, you need to use the command line of OpenSSL. The following listing shows some of the TLS parameters of a connection to google.com, truncated for readability. It's a mouthful, so we'll discuss it section by section.

Listing 5.2 TLS connection to google.com obtained via the `openssl` tool

```
$ openssl s_client -connect google.com:443
---
Certificate chain
 0 s:/C=US/ST=California/L=Mountain View/O=Google Inc/CN=*.google.com
   i:/C=US/O=Google Inc/CN=Google Internet Authority G2
 1 s:/C=US/O=Google Inc/CN=Google Internet Authority G2
   i:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
 2 s:/C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
   i:/C=US/O=Equifax/OU=Equifax Secure Certificate Authority
---

SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES128-GCM-SHA256
    Session-ID: 0871E6F1A35AE705A…
    Session-ID-ctx:
    Master-Key: 01F2462FD1D61...
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 100800 (seconds)
    TLS session ticket:
    0000 - d7 2a 55 df .. .. .. ..
```

The chain of trust of Google's certificate points to the Equifax Certificate Authority.

TLSI.2 is the latest version of the protocol.

Cipher suite negotiated

Unique ID of the session

Cryptographic master key

Encrypted master key in session tickets

5.2.1 The certificate chain

The first part of the output of the OpenSSL command shows three certificates numbered 0, 1, and 2. Each certificate has a subject, *s*, and an issuer, *i*. The first certificate, number 0, is called the *end-entity certificate*. The subject line tells us it's valid for any subdomain of google.com because its subject is set to *.google.com. The issuer line indicates it's issued by Google Internet Authority G2, which also happens to be the subject of the second certificate, number 1. Number 1 is itself signed by GeoTrust Global CA, which we find in number 2. You can see where this is going: each certificate is issued by the certificate that follows it—except for number 2, whose issuer, Equifax Secure Certificate Authority, is nowhere to be found.

What the OpenSSL command line doesn't show here is the trust store that contains the list of CA certificates trusted by the system OpenSSL runs on. The public certificate of Equifax Secure Certificate Authority must be present in the system's trust store to close the verification chain. This is called a *chain of trust*, and figure 5.3 summarizes its behavior at a high level.

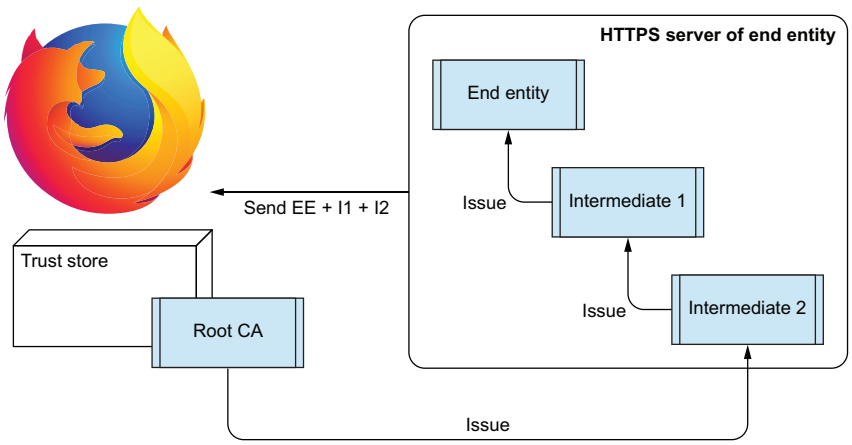


Figure 5.3 High-level view of the concept of chain of trust applied to verifying the authenticity of a website. The Root CA in the Firefox trust store provides the initial trust to verify the entire chain and trust the end-entity certificate.

In practice, verifying the chain of trust is vastly more complex than just verifying the issuers, but I'll leave finding out these details as an exercise for the reader. What matters here is the fact that OpenSSL verified the identity of the Google server and is thus certain it's communicating with the proper entity. Authenticity being established, the handshake moves on to negotiating the cryptographic details of the communication channel.

5.2.2 The TLS handshake

TLS is designed to allow a client and a server to agree on a suite of cryptographic algorithms to use for a connection, called a *cipher suite*. Each version of TLS, from the original SSLv2 to the current TLSv1.3, comes with its own set of cipher suites, and more-modern versions of the protocol use higher security ciphers.

In the output of the OpenSSL command line from listing 5.2, the client and server agreed to use TLSv1.2 with the ECDHE-RSA-AES128-GCM-SHA256 cipher suite. This cryptic string has a specific meaning:

- ECDHE is an algorithm known as the Elliptic Curve Diffie-Hellman Exchange. It's a mathematical construct that allows the client and server to negotiate a master key securely. We'll discuss what "ephemeral" means in a little bit; for now, know that ECDHE is used to perform the key exchange.
- RSA is the public-key algorithm of the certificate provided by the server. The public key in the server certificate isn't directly used for encryption (because RSA requires multiplication of large numbers, which is too slow for fast encryption), but instead is used to sign messages during the handshake and thus provides authentication.
- AES128-GCM is a symmetric encryption algorithm, like Caesar's cipher, but vastly superior. It's a fast cipher designed to quickly encrypt and decrypt large amounts of data transiting through the communication. As such, AES128-GCM is used for confidentiality.
- SHA256 is a hashing algorithm used to calculate fixed-length checksums of the data that transits through the connection. SHA256 is used to guarantee integrity.

The full TLS handshake would take pages to describe (the RFC of TLS1.2 is 100 pages long; see http://mng.bz/jGFT). Figure 5.4 shows a simplified version of the handshake, as described here:

1. The client sends a HELLO message to the server with a list of protocols and algorithms it supports.
2. The server says HELLO back and sends its chain of certificates. Based on the capabilities of the client, the server picks a cipher suite.
3. If the cipher suite supports ephemeral key exchange, like ECDHE does, the server and the client negotiate a premaster key with the Diffie-Hellman algorithm. The premaster key is never sent over the wire.
4. The client and server create a session key that will be used to encrypt the data transiting through the connection.

At the end of the handshake, both parties possess a secret session key used to encrypt data for the rest of the connection. This is what OpenSSL refers to as `Master-Key` in the output from listing 5.2.

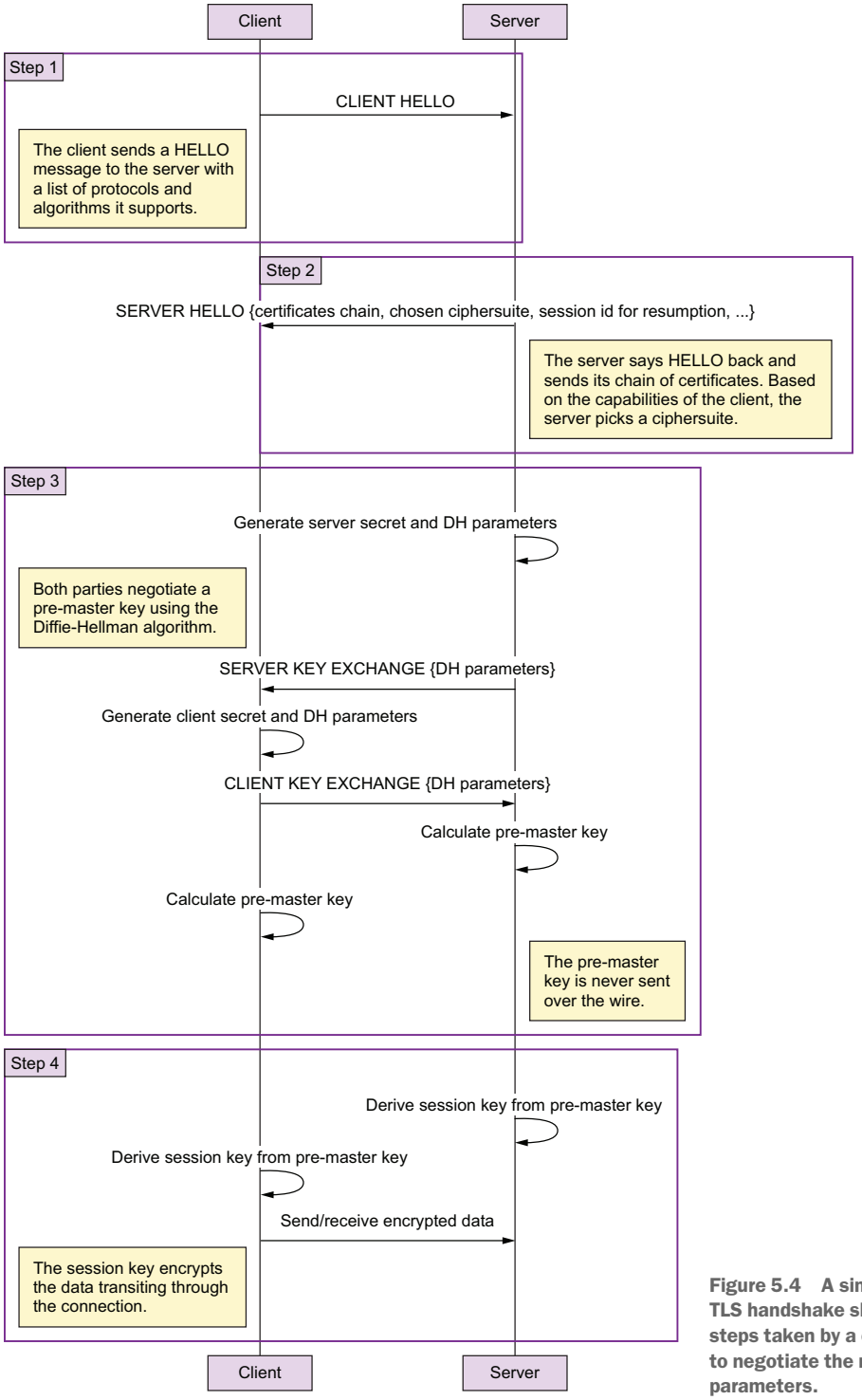


Figure 5.4 A simplified view of the TLS handshake shows the four main steps taken by a client and a server to negotiate the necessary security parameters.

5.2.3 Perfect forward secrecy

The term "ephemeral" in the key exchange provides an important security feature called *perfect forward secrecy* (PFS).

In a non-ephemeral key exchange, the client sends the pre-master key to the server by encrypting it with the server's public key. The server then decrypts the pre-master key with its private key. If, at a later point in time, the private key of the server is compromised, an attacker can go back to this handshake, decrypt the pre-master key, obtain the session key, and decrypt the entire traffic. Non-ephemeral key exchanges are vulnerable to attacks that may happen in the future on recorded traffic. And because people seldom change their password, decrypting data from the past may still be valuable for an attacker.

An ephemeral key exchange like DHE, or its variant on elliptic curve, ECDHE, solves this problem by not transmitting the pre-master key over the wire. Instead, the pre-master key is computed by both the client and the server in isolation, using nonsensitive information exchanged publicly. Because the pre-master key can't be decrypted later by an attacker, the session key is safe from future attacks: hence, the term *perfect forward secrecy*.

The downside to PFS is that all those extra computational steps induce latency on the handshake and slow the user down. To avoid repeating this expensive work at every connection, both sides cache the session key for future use via a technique called *session resumption*. This is what the session-ID and TLS ticket are for: they allow a client and server that share a session ID to skip over the negotiation of a session key, because they already agreed on one previously, and go directly to exchanging data securely.

This is the end of the overview of TLS. I introduced a lot of new concepts and covered a huge amount of information, which can be overwhelming if this is your first dive into the fascinating world of cryptography. You should expect that mastering TLS takes time and patience, but the core concepts introduced in the last few pages are sufficient to secure an online service, which you'll do right away by enabling HTTPS on the invoicer.

More information about TLS

I could spend an entire book talking only about TLS. And as it happens, someone did: Ivan Ristic, the creator of SSL Labs, wrote a comprehensive study of TLS, PKI, and server configurations in his book *Bulletproof SSL and TLS* (Feisty Duck, 2017). A must-read if this short chapter doesn't satisfy your curiosity on this fantastic protocol.

5.3 Getting applications to use HTTPS

Enabling HTTPS on the application is processed in three phases:

1. Obtain a domain name you control that points to the invoicer's public endpoint.
2. Get an X.509 certificate for that domain issued by a trusted CA.
3. Update your configuration to enable HTTPS with that certificate.

Until now, you've used the AWS-generated address of the ELB of the invoicer, but for a real application, you obviously want a real domain name, like invoicer.securing-devops.com. I'll skip over the details of purchasing a domain and creating the necessary CNAME record to point to the invoicer's ELB. Once created, the record should be similar to the following listing.

Listing 5.3 CNAME record points invoicer.securing-devops.com to the invoicer's ELB

```
$ dig invoicer.securing-devops.com
;; ANSWER SECTION:
invoicer.securing-devops.com. 10788 IN CNAME
                  invoicer-api.3pjw7ca4hi.us-east-1.elasticbeanstalk.com.
invoicer-api.3pjw7ca4hi.us-east-1.elasticbeanstalk.com. 48 IN A
                  52.70.99.109
invoicer-api.3pjw7ca4hi.us-east-1.elasticbeanstalk.com. 48 IN A
                  52.87.136.111
```

Requesting a certificate used to be a complex process that required hours of online reading to learn obscure options from tools like OpenSSL, to generate a certificate signing request for a CA, and to install a signed certificate on a web server. You may be familiar with this procedure if you manage traditional infrastructure, but recent initiatives from certificate authorities have made this process a lot less painful:

- Let's Encrypt provides a fully automated—and free—process to obtain certificates via the ACME verification protocol.
- AWS issues certificates for free, but which can only be used inside AWS (private keys can't be exported).
- Traditional CAs, including free ones, are progressively adopting the ACME protocol.

Let's first look at the CA from AWS, and then we'll discuss using Let's Encrypt.

5.3.1 Obtaining certificates from AWS

If you only care about running your application in AWS, obtaining a certificate via the Certificate Manager service is as simple as running the command from the following listing.

Listing 5.4 Requesting a certificate for the invoicer from AWS Certificate Manager

```
$ aws acm request-certificate --domain-name invoicer.securing-devops.com

{
    "CertificateArn": "arn:aws:acm:us-east-1:93:certificate/6d-7c-4a-bd-09"
}
```

The preceding command tells Amazon to generate a private key and certificate in the AWS account (the operator can't extract the private key from the account). Before signing the certificate with its own PKI, Amazon must verify the operator controls the

domain they're requesting a certificate for, which is done by emailing the operator at predefined addresses, such as postmaster@securing-devops.com, with a verification code. The operator must click the link with the verification code to confirm the issuance of the certificate, making it immediately available to use within the AWS account. The AWS Certificate Manager service provides the easiest way to obtain a certificate for a service hosted on Amazon's infrastructure, but if you want control over the private key, Let's Encrypt provides an excellent alternative.

5.3.2 Obtaining certificates from Let's Encrypt

From the point of view of a CA, one of the most complex tasks when issuing certificates is verifying that the user making the request is the legitimate owner of the domain. As discussed, AWS does so by emailing the domain owner at a predefined address. Let's Encrypt uses a more sophisticated approach that goes through a set of challenges defined in the ACME specification.[1]

The most common challenge involves HTTP, where the operator requesting the certificate is provided a random string by the CA, which must be placed at a predefined location of the target website for the CA to verify ownership. For example, when requesting a certificate for invoicer.securing-devops.com, the CA will look for a challenge at http://invoicer.securing-devops.com/.well-known/acme-challenge/evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA.

The HTTP challenge method works well for traditional web servers, but your invoicer infrastructure doesn't have a web server you could easily configure to serve this challenge. Instead, you'll use the DNS challenge, which requests an ACME challenge under the _acme-challenge.invoicer.securing-devops.com TXT record. For this challenge to work, you need two components:

- An ACME client that can perform the handshake with Let's Encrypt, configure the DNS, and request the certificate
- A registrar that can be configured to serve the TXT ACME challenge

For the client, use lego,[2] a Go client for Let's Encrypt that supports DNS (and more) challenges. My registrar of choice is Gandi.net, but lego supports several DNS providers that would work just as well. Requesting a certificate for your domain can be done with a single command.

Listing 5.5 Requesting a certificate from Let's Encrypt using a DNS challenge

```
$ GANDI_API_KEY=8aewloliqa80AOD10alsd lego
--email="julien@securing-devops.com"
--domains="invoicer.securing-devops.com"
--dns="gandi"
--key-type ec256
run
```

[1] ACME is currently an IETF draft, accessible at https://tools.ietf.org/wg/acme/.

[2] lego can be installed with the `$ go get -u github.com/xenolf/lego` command.

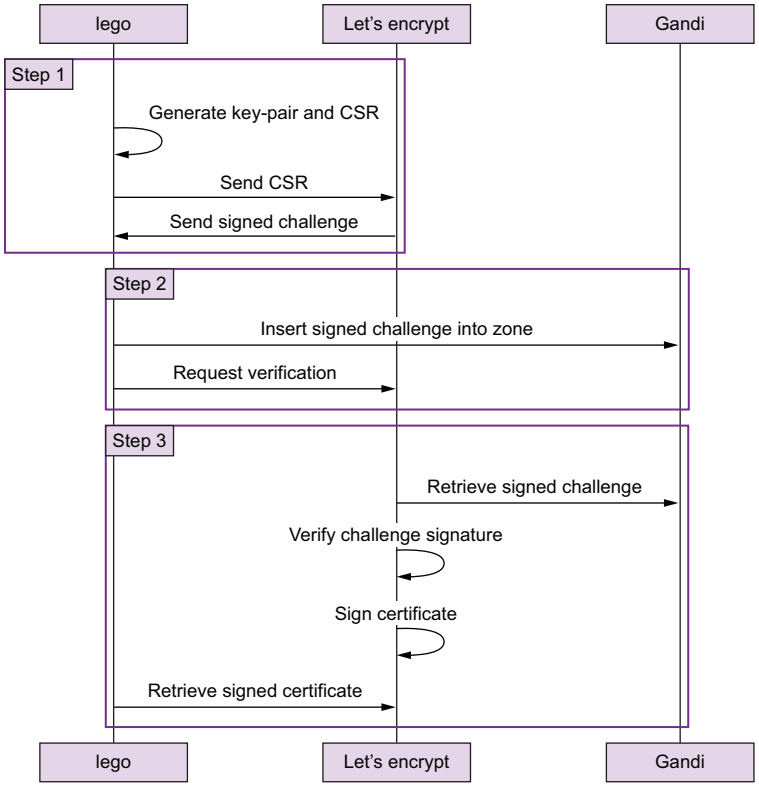


Figure 5.5 The ACME protocol between the client (lego), the CA (Let's Encrypt), and the registrar (Gandi) automates the issuance of a signed certificate for the invoicer.

The Gandi API key is obtained from the account preferences. Figure 5.5 details the conversation between lego, Let's Encrypt, and Gandi. lego first generates a private key and a CSR. The CSR is sent to Let's Encrypt, which replies with a signed challenge. lego inserts the challenge into the DNS of securing-devops.com and asks Let's Encrypt to perform the verification.

Let's Encrypt verifies the challenge and signs the CSR with its intermediate key. lego can then retrieve the signed certificate.

Note that the private key type is set to `ec256`, indicating you want an ECDSA P-256 key, not an RSA one.

ECDSA keys

ECDSA is an alternative algorithm to RSA, which provides a digital signature using elliptic curves. The benefit of ECDSA keys is their reduced size compared to RSA: a 256-bit ECDSA key provides security equivalent to a 3072-bit RSA key. Smaller keys mean faster computation, and the performance gain of ECDSA is increasingly pushing site operators to use this algorithm instead of RSA.

The command can take several minutes to complete because DNS records can take some time to propagate. Once finished, a certificate chain and a private key are written to ~/.lego/certificates.

Listing 5.6 The private key and certificate chain issued by Let's Encrypt

```
$ tree ~/.lego/certificates/
├── invoicer.securing-devops.com.crt
└── invoicer.securing-devops.com.key
```

Following Let's Encrypt's issuance policy, the certificate is valid for 90 days. Automating the renewal of this certificate at regular intervals is left as an exercise for the reader (and could easily be done via a script executed by the deployer). For now, you need to upload this information to AWS for the invoicer's ELB to use.

5.3.3 Enabling HTTPS on AWS ELB

Considering the invoicer.securing-devops.com.crt file, you'll notice two `CERTIFICATE` blocks that follow each other. The first block contains the server certificate (also called *end entity*, or EE) for invoicer.securing-devops.com, and the second block contains the intermediate certificate that signed the EE. AWS requires you to upload the EE and intermediate certificates separately, not as a single file, so you split them into two files using a text editor and upload them as follows.

Listing 5.7 Uploading the private key as well as EE and intermediate certificates to AWS

```
$ aws iam upload-server-certificate
--server-certificate-name "invoicer.securing-devops.com-20160813"
--private-key
     file://$HOME/.lego/certificates/invoicer.securing-devops.com.key
--certificate-body
     file://$HOME/.lego/certificates/invoicer.securing-devops.com.EE.crt
--certificate-chain
     file://$HOME/.lego/certificates/letsencrypt-intermediate.crt

{
    "ServerCertificateMetadata": {
        "Path": "/",
        "Expiration": "2016-11-11T13:31:00Z",
        "Arn": "arn:aws:iam::973:server-certificate/invoicer.securing-
     devops.com-20160813",
        "ServerCertificateName": "invoicer.securing-devops.com-20160813",
        "UploadDate": "2016-08-13T15:37:30.334Z",
        "ServerCertificateId": "ASCAJJ5ZF2467KDBETALA"
    }
}
```

The command returns the metadata of the uploaded certificate. Next, you attach the certificate to the ELB of the invoicer. This is a two-step process, as you need to retrieve

the internal name of the ELB, and then enable an HTTPS listener using the certificate you obtained.

Retrieving the name of the ELB is done by extracting the details of the Elastic Beanstalk environment. You know the environment ID from your work in chapter 2, so retrieving the ELB name is just one command away.

Listing 5.8 Retrieving the ELB name by extracting resources from Elastic Beanstalk

```
$ aws elasticbeanstalk describe-environment-resources
--environment-id e-curu6awket |
jq -r '.EnvironmentResources.LoadBalancers[0].Name'

awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5
```

You can now create a new listener on the ELB. Note that the argument to the listener syntax that can seem a little obscure at first:

- `Protocol` and `LoadBalancerPort` indicate the public-facing configuration; here, HTTPS on port 443.
- `InstanceProtocol` and `InstancePort` indicate where the traffic should be sent to; here, to the invoicer's application.
- `SSLCertificateId` is the ARN (Amazon Resource Name) of the certificate as returned by the certificate upload command run previously.

Listing 5.9 Creating the HTTPS listener on the invoicer's ELB

```
$ aws elb create-load-balancer-listeners
--load-balancer-name awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5
--listeners "Protocol=HTTPS,LoadBalancerPort=443,
InstanceProtocol=HTTP,InstancePort=80,
SSLCertificateId=arn:aws:iam::973:server-certificate/invoicer.securing-
     devops.com-20160813"
```

You can verify the configuration using the `aws elb describe-load-balancers` command. The output, shown in the following listing, indicates that both the HTTP and HTTPS listeners are configured. It also indicates the HTTPS load balancer uses a policy named `ELBSecurityPolicy-2015-05`, which we'll discuss and tweak later.

Listing 5.10 Describing the active listeners on the invoicer's ELB

```
$ aws elb describe-load-balancers
--load-balancer-names awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5 |
jq -r '.LoadBalancerDescriptions[0].ListenerDescriptions'
[
  {
    "Listener": {
      "InstancePort": 80,
      "InstanceProtocol": "HTTP",
      "Protocol": "HTTP",
      "LoadBalancerPort": 80
    },
```

```
    "PolicyNames": []
  },
  {
    "Listener": {
      "InstancePort": 80,
      "InstanceProtocol": "HTTP",
      "Protocol": "HTTPS",
      "LoadBalancerPort": 443,
      "SSLCertificateId": "arn:aws:acm:us-east-1:93:certificate/6d-7c-4a-
     bd-09"
    },
    "PolicyNames": [
      "ELBSecurityPolicy-2015-05"
    ]
  }
]
```

Although the ELB is now configured, it's not yet functional. The security group that fronts it doesn't allow connections to port 443. You fix this by allowing the entire internet, 0.0.0.0/0, to connect to port 443.

Listing 5.11 Retrieving the ELB's security group and opening port 443

```
$ aws elb describe-load-balancers
--load-balancer-names awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5 |
jq -r '.LoadBalancerDescriptions[0].SecurityGroups[0]'
sg-9ec96ee5

$ aws ec2 authorize-security-group-ingress
--group-id sg-9ec96ee5
--cidr 0.0.0.0/0
--protocol tcp
--port 443
```

The HTTPS endpoint of the invoicer is now fully functional and accessible at https://invoicer.securing-devops.com. As you can see in figure 5.6, Firefox shows a green lock indicating the connection was secured using a certificate issued by Let's Encrypt.

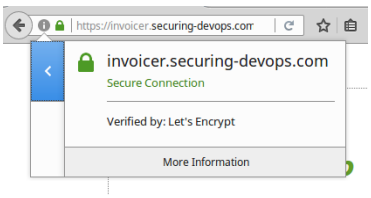


Figure 5.6 Firefox indicates the connection to the invoicer's web UI is secure by displaying a green lock in the address bar.

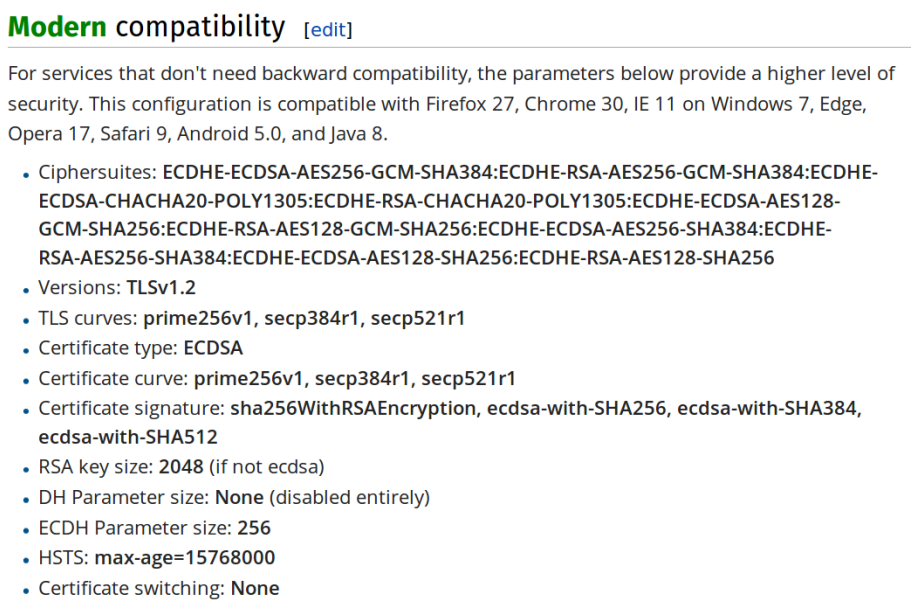
Following the concept first introduced by Netscape, the closed green lock tells you the connection is secure, but it doesn't tell you anything about *how* secure it is. Over half of the web relies on the TLS protocol to protect the integrity, authenticity, and confidentiality of HTTP traffic (see http://mng.bz/e9w9), but a significant portion does so using bad and sometimes dangerously insecure configurations, leaving data transiting through insecure channels at risk of tampering or leaking. Although web browsers try

to identify these bad configurations and alert users, you still need to audit this configuration yourself, and take steps to modernize it.

5.4 Modernizing HTTPS

Several guides exist to provide operators with modern TLS configurations. In this section, we'll discuss the guide maintained by Mozilla, which provides three levels of configuration (see http://mng.bz/6K5k).

- The Modern level is designed to support only the latest, most secure, cryptographic algorithms at the cost of supporting only modern web browsers. Figure 5.7 shows a screenshot of the modern configuration guidelines.
- The Intermediate level strikes a balance between security and backward compatibility to support most clients at a reasonable security level. When the population of clients that needs to access a site is large, the Intermediate level is recommended, as it provides reasonable security without removing algorithms needed by older clients.
- The Old level is designed to continue supporting ancient clients, like Windows XP pre-service pack 3. This level should only be used when support of very old clients is an absolute necessity, because it enables algorithms that are known to be insecure.



Modern compatibility [edit]

For services that don't need backward compatibility, the parameters below provide a higher level of security. This configuration is compatible with Firefox 27, Chrome 30, IE 11 on Windows 7, Edge, Opera 17, Safari 9, Android 5.0, and Java 8.

- Ciphersuites: **ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES128-SHA256**
- Versions: **TLSv1.2**
- TLS curves: **prime256v1, secp384r1, secp521r1**
- Certificate type: **ECDSA**
- Certificate curve: **prime256v1, secp384r1, secp521r1**
- Certificate signature: **sha256WithRSAEncryption, ecdsa-with-SHA256, ecdsa-with-SHA384, ecdsa-with-SHA512**
- RSA key size: **2048** (if not ecdsa)
- DH Parameter size: **None** (disabled entirely)
- ECDH Parameter size: **256**
- HSTS: **max-age=15768000**
- Certificate switching: **None**

Figure 5.7 Recommendations for the Modern TLS configuration level on the wiki of Mozilla

Figure 5.7 shows all the parameters that an operator can tweak when configuring TLS on a web server (depending on the web server or service operating TLS, some parameters may not be tweakable). You should recognize most of them by now: cipher suites, versions, certificate signature, and so on. Some may still be obscure, but it's safe to ignore them for now.

Had you read this recommendation without having an explanation of the protocol, you probably would have been overwhelmed by its complexity. TLS is a complex protocol, and unless you're ready to invest the time and energy to understand its details and build your own configuration, I strongly recommend you follow the guidelines proposed by Mozilla and other trustworthy resources almost blindly. The guidelines are updated when the state of the art of cryptography changes, and when algorithms once considered safe become massive security holes overnight.

I also recommend that you don't trust the default settings that come with web servers and libraries, as those are generally too permissive, to accommodate older clients. You should regularly test your TLS configuration, and particularly the enabled cipher suites. Cipher suites are the core of the TLS protocol. A cipher suite is a set of cryptographic algorithms designed to provide a given level of security. Four versions of SSL/TLS have brought us over three hundred cipher suites, most of which shouldn't be used when targeting high security.

Before explaining how you can tweak your HTTPS configuration, let's first discuss ways to test it and evaluate its current state.

5.4.1 Testing TLS

The flexibility of the TLS protocol allows a client and a server to negotiate connection parameters based on what they both support. In an ideal situation, both parties would agree to use the most secure set of parameters common to them. As a site operator, it's your responsibility to ensure your services are configured to prefer strong ciphers and discard unsafe ones.

Many tools can help you test your TLS configuration. Most of them probe a server to test every possible configuration supported. Tools like Cipherscan (https://github.com/jvehent/cipherscan), written by the author of this book, and testssl.sh (https://testssl.sh/) will give you such reports. A few advanced tools will also make recommendations and highlight major issues. The most popular and comprehensive of them is certainly SSLLabs.com, an online TLS scanner that outputs a letter grade from A through F to represent the security of a configuration. An open source alternative is Mozilla's TLS Observatory (https://observatory.mozilla.org), available as a command-line tool and a web interface. The following listing shows the output of the `tlsobs` command line against the invoicer.

Listing 5.12 Installing and using the TLS Observatory client on the ELB invoicer

```
$ go get -u github.com/mozilla/tls-observatory/tlsobs

$ $GOPATH/bin/tlsobs -r invoicer.securing-devops.com
```

```
Scanning invoicer.securing-devops.com (id 12323098)

--- Certificate ---
Subject  CN=invoicer.securing-devops.com
SubjectAlternativeName
- invoicer.securing-devops.com
Validity 2016-08-13T13:31:00Z to 2016-11-11T13:31:00Z
CA       false
SHA1     5648102550BDC4EFC65529ACD21CCF79658B79E1
SigAlg   SHA256WithRSA
Key      ECDSA 256bits P-256

--- Trust ---
Mozilla Microsoft Apple
   ✓        ✓       ✓

--- Ciphers Evaluation ---
pri cipher                          protocols              pfs        curves
1   ECDHE-ECDSA-AES128-GCM-SHA256 TLSv1.2                ECDH,P-256 prime256
2   ECDHE-ECDSA-AES128-SHA256     TLSv1.2                ECDH,P-256 prime256
3   ECDHE-ECDSA-AES128-SHA        TLSv1,TLSv1.1,TLSv1.2 ECDH,P-256 prime256
4   ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2                ECDH,P-256 prime256
5   ECDHE-ECDSA-AES256-SHA384     TLSv1.2                ECDH,P-256 prime256
6   ECDHE-ECDSA-AES256-SHA        TLSv1,TLSv1.1,TLSv1.2 ECDH,P-256 prime256
OCSP Stapling        false
Server Side Ordering true
Curves Fallback      false

--- Analyzers ---
Measured level "non compliant" does not match target level "modern"
* Mozilla evaluation: non compliant
  - for modern level: remove ciphersuites ECDHE-ECDSA-AES128-SHA, ECDHE-
    ECDSA-AES256-SHA
  - for modern level: consider adding ciphers ECDHE-ECDSA-CHACHA20-POLY1305
  - for modern level: remove protocols TLSv1, TLSv1.1
  - for modern level: consider enabling OCSP stapling
```

The Certificate section displays details about the site's certificate.

The Trust section tells you the EE certificate chains to a CA trusted by Mozilla, Microsoft and Apple.

The Ciphers Evaluation section lists the cipher suites accepted by the server by order of preference.

In the Analyzers section, the tool provides recommendations on what should be changed to match Mozilla's Modern configuration level.

Each of the four sections carries important information to your configuration:

- The Certificate section displays details about the end entity. You see that it's valid for your domain and only for a period of three months.
- The Trust section tells you the EE certificate chains to a CA trusted by Mozilla, Microsoft and Apple. Most certificates obtained through common CAs are trusted everywhere, but it's possible to find certificates issued by obscure CAs that are trusted by one browser and not another.
- The Ciphers Evaluation section lists the cipher suites accepted by the server by order of preference. This list is small and, had you used an RSA certificate, it would be significantly larger, but ECDSA certificates are more recent, and fewer cipher suites support them. Notice the `Server Side Ordering` flag set to `true` at the end of the output, which indicates the server will force its own preferred ordering over the client's. The evaluation also tells you which ciphers support perfect forward secrecy in the pfs column.

- In the Analyzers section, the tool provides recommendations on what should be changed to match Mozilla's Modern configuration level. You see that a few cipher suites should be removed, and missing ones should be added. TLSv1 and TLSv1.1 aren't recommended, and only TLSv1.2 should be kept. Overall, the evaluation tool considers your current setup to be noncompliant with Mozilla's guidelines.

It's possible, and preferable, to perform the evaluation of the invoicer's endpoint against Mozilla's guidelines automatically by calling the `tlsobs` client as a deployment test. To do so, you wrap it into a bash script placed under the deploymentTests directory of the deployer you configured in chapter 4. The `tlsobs` client supports an option called `-targetLevel` that evaluates a target against one of Mozilla's configuration levels. By setting this option to `Modern`, you instruct `tlsobs` to verify the target is configured per the Modern configuration level.

Listing 5.13 Test executed by the deployer to evaluate HTTPS quality

```
#!/usr/bin/env bash
go get -u github.com/mozilla/tls-observatory/tlsobs
$GOPATH/bin/tlsobs -r -targetLevel modern invoicer.securing-devops.com
```

As expected, this test will fail until you modernize the configuration of your endpoint, and the logs of the deployer contain the full output from `tlsobs` in listing 5.12. You can verify this by triggering a build of the invoicer in CircleCI and looking at the logs of the deployer.

Listing 5.14 Test exits with an error because HTTPS isn't supported

```
2016/08/14 15:35:17 Received webhook notification
2016/08/14 15:35:17 Verified notification authenticity
2016/08/14 15:35:17 Executing test /app/deploymentTests/2-ModernTLS.sh
2016/08/14 15:35:32 Test /app/deploymentTests/ModernTLS.sh failed:
exit status 1
[...]
--- Analyzers ---
Measured level "non compliant" does not match target level "modern"
* Mozilla evaluation: non compliant
```

With your testing infrastructure now ready, let's move on to modernizing your endpoint.

5.4.2 Implementing Mozilla's Modern guidelines

Enabling HTTPS on the invoicer took you 90% of the way to having a secure endpoint. Tweaking it to match Mozilla's Modern level requires creating a new configuration that only enables selected parameters, instead of using the defaults automatically provided by AWS: only TLS version 1.2 must be activated, and the list of activated cipher suites must be reduced to a minimum. AWS ELB only supports a limited set of parameters, which you need to choose from (see http://mng.bz/V96x).

> **NOTE** The configuration presented here is current at the time of writing, but will likely change over time as Mozilla evolves its guidelines and AWS supports more ciphers. Make sure to refer to the links provided and always use the latest version of the recommendations when configuring your endpoints.

Call this new configuration `MozillaModernV4`. The following listing shows how to create it using the AWS command line.

Listing 5.15 Creating a custom load-balancer policy mapping Mozilla's Modern level

```
$ aws elb create-load-balancer-policy
--load-balancer-name awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5
--policy-name MozillaModernV4
--policy-type-name SSLNegotiationPolicyType
--policy-attributes AttributeName=Protocol-TLSv1.2,AttributeValue=true
AttributeName=ECDHE-ECDSA-AES256-GCM-SHA384,AttributeValue=true
AttributeName=ECDHE-ECDSA-AES128-GCM-SHA256,AttributeValue=true
AttributeName=ECDHE-ECDSA-AES256-SHA384,AttributeValue=true
AttributeName=ECDHE-ECDSA-AES128-SHA256,AttributeValue=true
AttributeName=Server-Defined-Cipher-Order,AttributeValue=true
```

The next step is to assign the newly created policy to your ELB, by switching the ELB from using the ELBSecurityPolicy-2015-05 AWS default policy over to MozillaModernV4.

Listing 5.16 Assigning the MozillaModernV4 policy to the invoicer's ELB

```
$ aws elb set-load-balancer-policies-of-listener
--load-balancer-name awseb-e-c-AWSEBLoa-1VXVTQLSGGMG5
--load-balancer-port 443
--policy-names MozillaModernV4
```

With this change in place, you'll kick off a rebuild of the invoicer to verify the ELB passes the compliance test in the deployer logs. The configuration level is now being measured as Modern, so the deployer continues its work by triggering an update of the invoicer's infrastructure.

Listing 5.17 Logs showing the invoicer's ELB passes the Modern TLS configuration test

```
2016/08/14 16:42:46 Received webhook notification
2016/08/14 16:42:46 Verified notification authenticity
2016/08/14 16:42:46 Executing test /app/deploymentTests/2-ModernTLS.sh
2016/08/14 16:42:49 Test /app/deploymentTests/ModernTLS.sh succeeded:
     Scanning invoicer.securing-devops.com (id 12123107)
[…]
--- Analyzers ---
* Mozilla evaluation: modern

2016/08/14 16:42:51 Deploying EBS application: {
  ApplicationName: "invoicer201605211320",
  EnvironmentId: "e-curu6awket",
  VersionLabel: "invoicer-api"
}
```

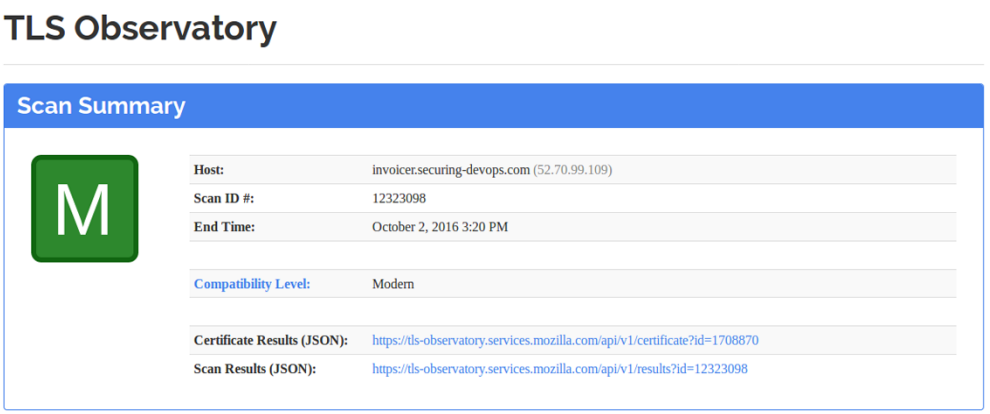


Figure 5.8 The scan summary from https://observatory.mozilla.org shows the invoicer's TLS endpoint being measured as compliant with Mozilla's Modern guidelines.

You can also use the web interface of the Observatory to check the quality of your configuration. Figure 5.8 shows https://invoicer.securing-devops.com being measured as Modern by the scanner.

Configuring the protocol layer of TLS is the biggest part of enabling HTTPS on a service, but I mentioned at the beginning of this chapter that some controls must be placed at the HTTP layer to increase the security of HTTPS. These controls are Strict Transport Security (HSTS) and Public Key Pinning (HPKP). In the following sections, I'll introduce both and discuss how to implement them on the invoicer.

5.4.3 HSTS: Strict Transport Security

Once a service is fully configured to use HTTPS, there shouldn't be any reason to fall back to the insecure HTTP. Knowing that a site should always be accessed through HTTPS is useful information for web browsers to prevent downgrade attacks (forcing a user through an insecure version of the site to steal cookies or inject fraudulent traffic). HTTP Strict Transport Security (HSTS) is an HTTP header that a service can send to the browser to enforce the use of HTTPS at all times. Browsers cache the HSTS information locally for a period of time during which all connections to the site will use HTTPS.

HSTS also has the interesting property of forcing browsers to use HTTPS even if not explicitly asked to, like when the user doesn't specify the https:// handler when entering the site's address. This little benefit replaces the need for an HTTP listener that would redirect users to HTTPS, but only for users who have already visited the site.

The HSTS header consists of three parameters, shown in listing 5.18.

- `max-age`—Indicates the lifetime in seconds of the information in the browser cache.

- `includeSubDomains`—Tells the browser to force HTTPS for the current domain and all its subdomains.
- `preload`—Indicates the operator's intention to add their sites to the HSTS preload list. When set, an operator can request the addition of a domain to the list of sites Firefox, Chrome, Internet Explorer, Opera, and Safari will connect to via HTTPS only. The Google Chrome team operates the form to make this request (https://hstspreload.appspot.com/). A site must meet several requirements prior to joining the preload list, such as serving HSTS for the entire domain (not just subdomains), or having a `max-age` value of at least 18 weeks.

Listing 5.18 An example HSTS header with `max-age` set to one year

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

The simple syntax of the HSTS header makes it easy to add to new applications. For legacy sites with dozens of resources and subdomains, operators should use this header carefully, and start implementing it without `includeSubDomains` and with `max-age` set to a few seconds. Only after evaluating the impact of HSTS on a site should an operator use the preceding header. Once the header is out, and users cache it in their browsers, there's no going back. You're committed to HTTPS!

Testing for HSTS is simple: because the header is a static value, you can compare it during deployment. The script in the following listing does this comparison in the deployer.

Listing 5.19 Test script to verify the value of the HSTS header on the invoicer

```
#!/bin/bash
EXPECTEDHSTS="Strict-Transport-Security: max-age=31536000; includeSubDomains;
     preload"
SITEHSTS="$(curl -si https://invoicer.securing-devops.com/ | grep Strict-
     Transport-Security | tr -d '\r\n' )"

if [ "${SITEHSTS}" == "${EXPECTEDHSTS}" ]; then
    echo "HSTS header matches expectation"
    exit 0
else
    echo "Expected HSTS header not found"
    echo "Found:    '${SITEHSTS}'"
    echo "Expected: '${EXPECTEDHSTS}'"
    exit 100
fi
```

5.4.4 HPKP: Public Key Pinning

One of the weaknesses of the PKI ecosystem is the vast number of certificate authorities that can issue trusted certificates for any site on the planet. Imagine living in a country with a repressive regime and trying to use Google or Twitter to communicate with your peers, only to discover your connection is being hijacked by a rogue certificate

authority that issued fraudulent, yet trusted, certificates for Google and Twitter. This situation unfortunately happens and puts real people at risk.

Mozilla, Microsoft, and Apple operate their own root CA programs where they maintain lists of certificate authorities trusted to issue intermediate and end-entity certificates. They all try their best to blacklist misbehaving CAs,[3] or CA victims of breaches, as quickly as possible. But with over 150 CAs in the Firefox trust store, keeping track of everyone's behavior is hard.

Web browsers don't have a way of knowing which CA an operator trusts, and therefore must accept any certificate issued by any CA in their trust stores. The HTTP Public Key Pinning (HPKP) mechanism provides a solution to this problem by allowing operators to indicate which CAs, intermediate or end-entity, they intend to use with a given site.

Like HSTS, HPKP is an HTTP header sent to browsers and cached for a given duration of time. The header contains hashes of certificates permitted to secure the site. Should the user of a site with HPKP enabled be the victim of a fraudulent CA trying to hijack their connection, the browser will use the cached HPKP information to detect that the fraudulent CA isn't authorized to issue certificates for the site and present the user with an error.

The HPKP header takes four parameters, and can be a little tricky to construct:

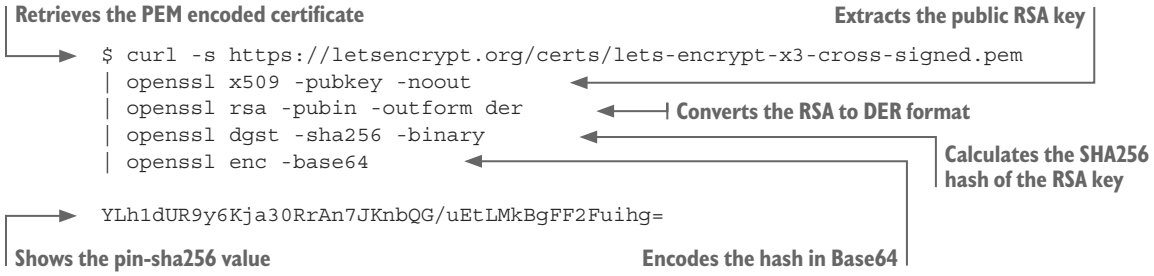
- `max-age` is the time, in seconds, web browsers should remember a site can only be accessed using one of the defined keys.
- `pin-sha256` is the Base64 hash of the public key of a certificate trusted for the current site. There must be a minimum of two `pin-sha256`s defined in the header: one primary and one backup.
- `includeSubDomains` indicates that all children of the current domain should apply the HPKP policy.
- `report-uri` is an optional endpoint where violations of the policy should be sent by web browsers. Not all browsers support this feature.

The core of HPKP is the `pin-sha256` values that indicate which certificates are trusted for a site. For certificates that change relatively often, like the Let's Encrypt one you generated for the invoicer, it's recommended to pin the intermediate CA, not the end entity. You also need to provide a backup pin in case you decide to stop using Let's Encrypt. In this case, you'll set the backup to the AWS CA.

Obtaining the `pin-sha256` value of a certificate is done by extracting the public key from the certificate, hashing it with the SHA256 algorithm, and then encoding it in Base64. The following listing shows how to perform this in one command on the Let's Encrypt intermediate certificate.

[3] In September 2016, Mozilla and Apple both decided to distrust CAs operated by WoSign following evidence of fraudulent behavior in their issuance of certificates.

Listing 5.20 Generating the `pin-sha256` value of the Let's Encrypt intermediates



You can perform the same calculation with the intermediate certificate from AWS (https://amazontrust.com/repository/) and add these two values to an HPKP header in the invoicer application (in middleware.go, see `setResponseHeaders`), the value of which follows.

Listing 5.21 HPKP header that permits certificates from Let's Encrypt and AWS CA

```
Public-Key-Pins: max-age=1296000; includeSubDomains; pin-sha256="YLh1dUR9y6Kj
     a30RrAn7JKnbQG/uEtLMkBgFF2Fuihg="; pin-sha256="++MBgDH5WGvL9Bcn5Be30cRcL
     0f5O+NyoXuWtQdX1aI="
```

Like testing for HSTS, you can use a script that compares the value of the HPKP header in the deployer with a reference you set statically. The script in the following listing performs a simple string comparison to verify the presence of the HPKP value.

Listing 5.22 Test script to verify the value of the HPKP header on the invoicer

```
#!/bin/bash
EXPECTEDHPKP='Public-Key-Pins: max-age=1296000; includeSubDomains; pin-sha256
     ="YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg="; pin-sha256="++MBgDH5WGv
     L9Bcn5Be30cRcL0f5O+NyoXuWtQdX1aI="'
SITEHPKP="$(curl -si https://invoicer.securing-devops.com/ |grep Public-Key-
     Pins | tr -d '\r\n' )"

if [ "${SITEHPKP}" == "${EXPECTEDHPKP}" ]; then
    echo "HSTS header matches expectation"
    exit 0
else
    echo "Expected HSTS header not found"
    echo "Found:    '${SITEHPKP}'"
    echo "Expected: '${EXPECTEDHPKP}'"
    exit 100
fi
```

You can also verify that HSTS and HPKP are active in the developer tools of Firefox, under the security tab of the network section. Figure 5.9 shows HSTS and HPKP both enabled on the invoicer's public site.

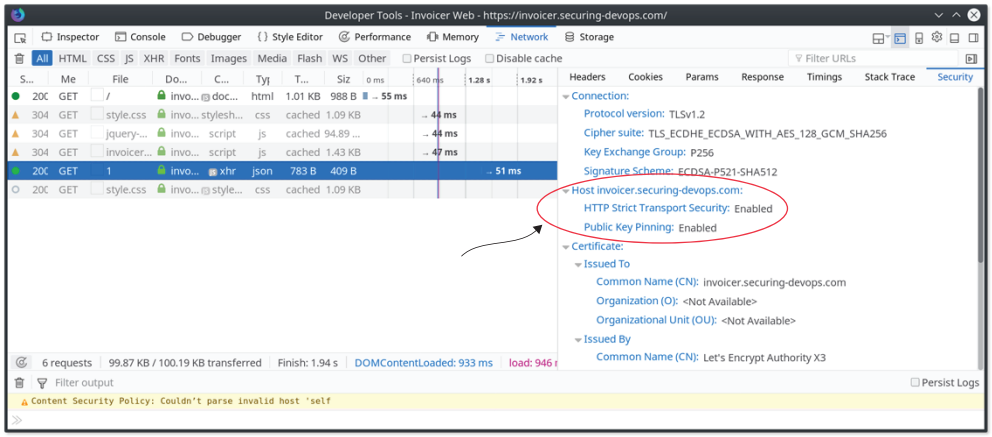


Figure 5.9 HSTS and HPKP show as enabled in Firefox's developer tools, confirming the headers are active on the invoicer's public page.

This concludes our tour of HTTPS. A lot more could be said about the protocol that made the internet a safe place for commerce and communication, and I strongly encourage the reader to stay up to date with improvements to TLS. Whether you're an operator, a developer, or a security expert, you'll have to work with TLS and HTTPS at one point or another. Maintaining an up-to-date understanding of strong communication security will help you run better services for your users.

Summary

- TLS guarantees the confidentiality, integrity, and authenticity of a connection between a client and a server.
- Servers use the X.509 security certificate signed by a certificate authority to prove their identity to clients.
- The TLS communication uses cipher suites negotiated during a handshake to protect the data in transit.
- Obtaining a trusted certificate for a site requires proving the operator owns the domain the site is hosted on.
- Security parameters enabled by default on HTTPS servers may not provide sufficient security, and testing tools must be used to improve a configuration.
- HSTS is an HTTP header that indicates to web browsers that a site must always be reached via HTTPS.
- HPKP is an HTTP header that indicates to web browsers that only white-listed certificates are trusted to issue security certificates for a given site.

SECURITY/OPERATIONS

Securing DevOps

Julien Vehent

An application running in the cloud can benefit from incredible efficiencies, but they come with unique security threats too. A DevOps team's highest priority is understanding those risks and hardening the system against them.

Securing DevOps teaches you the essential techniques to secure your cloud services. Using compelling case studies, it shows you how to build security into automated testing, continuous delivery, and other core DevOps processes. This experience-rich book is filled with mission-critical strategies to protect web applications against attacks, deter fraud attempts, and make your services safer when operating at scale. You'll also learn to identify, assess, and secure the unique vulnerabilities posed by cloud deployments and automation tools commonly used in modern infrastructures.

What's Inside

- An approach to continuous security
- Implementing test-driven security in DevOps
- Security techniques for cloud services
- Watching for fraud and responding to incidents
- Security testing and risk assessment

Readers should be comfortable with Linux and standard DevOps practices like CI, CD, and unit testing.

Julien Vehent is a security architect and DevOps advocate. He leads the Firefox Operations Security team at Mozilla, and is responsible for the security of Firefox's high-traffic cloud services and public websites.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/securing-devops

"Provides both sound ideas and real-world examples. A must-read."
—Adrien Saladin, PeopleDoc

"Makes a complex topic completely approachable. Recommended for DevOps personnel and technology managers alike."
—Adam Montville
Center for Internet Security

"Practical and ready for immediate application."
—Yan Guo, Eventbrite

"An amazing resource for secure software development—a must in this day and age—whether or not you're in DevOps."
—Andrew Bovill, Next Century

ISBN-13: 978-1-61729-413-6
ISBN-10: 1-61729-413-6
54999
9 781617 294136

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]