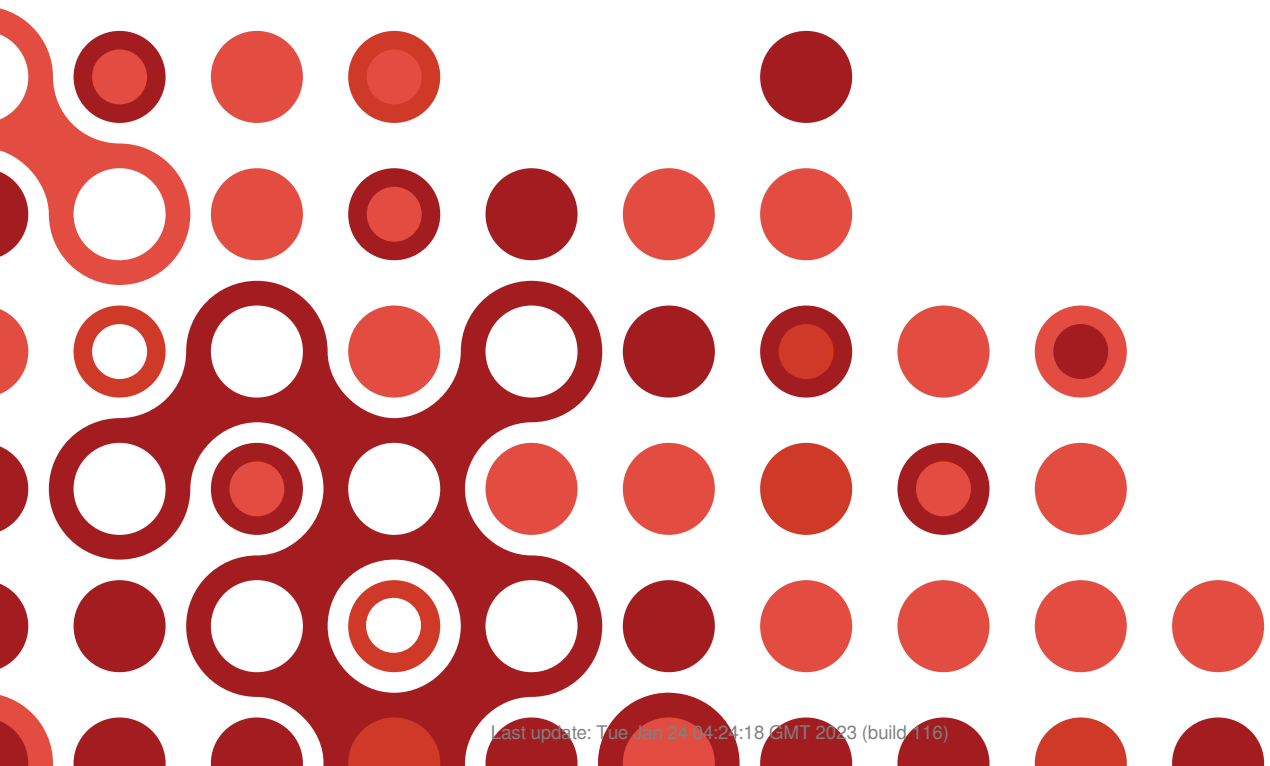
Feisty Duck

Bulletproof TLS Guide



Last update: Tue Jan 24 04:24:18 GMT 2023 (build 116)

Bulletproof TLS Guide

Ivan Ristić

Feisty Duck
LONDON

Bulletproof TLS Guide

by Ivan Ristić

Version 2022-draft1 (build 116), published in January 2023.

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

Production editor: Jelena Girić-Ristić

Copyeditor: Melinda Rankin

Table of Contents

Preface

Our journey to achieving good network transport security has been long and fraught with difficulties. In the 1990s—when this story began with the early versions of SSL and the Netscape browser—the main challenges were lack of good encryption standards, restrictions on the export of cryptography, and insufficiently powerful computer chips. It took us a good three decades to work through these problems. During that period, a few things improved. The export restrictions went away and computers became faster. A few other things became worse, chiefly because the web platform continued to evolve organically without sufficient thought given to security.

But we collectively kept chipping away at the problems, eventually figuring out what secure network protocols should look like and what kind of security we'd like to have. We figured out that we don't have to configure each and every server individually and that we can instead rely on the secure and sane defaults now available. We also figured out that we don't need to manually rotate every single certificate and that automation can achieve much better results with far less time and effort.

At some point, the threads of our progress started to converge, and there is now a feeling that transport security is within our reach. *Your* reach. Things are significantly better, but we're not quite there just yet. The field remains complex and filled with many moving parts that need to be accounted for. Some assembly is required.

The guide that's in front of you has been designed to get you over the finish line. If you follow the assembly instructions specified herein, you will be able to deploy TLS and PKI with confidence. Yes, there will still be things you'll need to figure out, but the path should at least be predictable and easy to follow.

I have been involved with SSL/TLS and PKI in some form since the early days, but especially in the last two decades, focusing my efforts on what I sometimes like to call the *last mile* of transport security. My work consisted chiefly of researching the field and communicating my findings in various forms.

For example, SSL Labs, one of my earlier projects, provided free assessments of SSL/TLS and PKI configuration and ended up being fairly successful. It happened to be right there when

the world decided to start caring about such things, roughly around the time of Heartbleed. (Look it up.) In the end, SSL Labs helped improve the security of millions of web sites, and I'm very happy with that. These days, you're probably better off taking a look at my follow-up project, Hardenize, which helps with problems related to a wider range of network and security standards.

Most of what I learned has been recorded in my book *Bulletproof TLS and PKI*, which I've been continuously writing and publishing for about a decade now. You should definitely read it if you're involved with computer security, software development, or system administration. But even if you don't have time for that, this guide will tell you everything you need to know. In fact, this guide has been taken directly from my book and published stand-alone for the very purpose of being easily available to a large audience.

1 Configuration Guide

Private Keys and Certificates

Private keys are the cornerstone of TLS security, but also the easiest thing to get right. After all, CAs won't be willing to issue certificates against weak keys. But despite our focus on key sizes, the weakest link is usually key management, or the job of keeping the private keys private. We'll touch upon that in this section. Equally important are certificates, which build upon the keys with important metadata, such as the permission to associate a certificate with a particular domain name.

Use Strong Private Keys

For the certificate private key, you have a choice of RSA or ECDSA algorithms. The easy option is to use RSA keys because they are universally supported. But at 2,048 bits, which is the current minimum, RSA keys offer less security and worse performance than ECDSA keys.

At the same time, ECDSA is the algorithm of the future and RSA is slowly being left behind. A 256-bit ECDSA key provides 128 bits of security versus only 112 bits for a 2,048-bit RSA key. At these sizes, in addition to providing better security, ECDSA is also significantly faster.

By now, ECDSA is very widely supported and the devices that don't support it are very rare and probably support only obsolete security protocols. If you're still concerned about interoperability, it's possible to deploy services with dual certificates, thus supporting RSA and ECDSA keys simultaneously. The only disadvantage of this setup is the increased maintenance overhead. Some managed providers do this automatically and thus make it trivial.

Secure Your Private Keys

Although we spend the most time obsessing about key size, issues surrounding key management are more likely to have a real impact on your security. There is ample evidence to sug-

gest that the most successful attacks bypass encryption rather than break it. If someone can break into your server and steal your private key or otherwise compel you to disclose the key, why would they bother with brute-force attacks against cryptography?

Keep your private keys private

Treat your private keys as an important asset, restricting access to the smallest possible group of employees while still keeping the arrangements practical. Some CAs offer to generate private keys for you, but they should know better. The hint is in the name: private keys should stay private, without exception.

Think about random number generation

The security of encryption keys depends on the quality of the random number generator (RNG) of the computer on which the keys are generated. Keys are often created on servers right after installation and rebooting, but at that point the server might not have sufficient entropy to generate a strong key. It's better to generate all your keys in one (offline) location, where you can ensure that a strong RNG is in place.

Password protect your keys

Your keys should have a passphrase on them from the moment they are created. This helps reduce the attack surface if your backup system is compromised. It also helps prevent leakage of the key material when copying keys from one computer to another (directly or using USB sticks); it's getting increasingly difficult to safely delete data from modern file systems.

Don't share keys among unrelated servers and applications

Sharing keys is dangerous: if one system is broken into, its compromised key could be used to attack other systems that use the same key, even if they use different certificates. Different keys allow you to establish strong internal access controls, giving access to the keys only to those who need them.

Change keys regularly

Treat private keys as a liability. Keep track of when the keys were created to ensure they don't remain in use for too long. You *must* change them after a security incident and when a key member of your staff leaves, and you should change them when obtaining a new certificate. When you generate a new key, you remove the old key as an attack vector. This is especially true for systems that do not use or support forward secrecy. In this case, your key can be used to decrypt all previous communications, if your adversary has recorded them. By deleting the key safely, you ensure that it can't be used against you.

Your default should be to generate a new private key with every certificate renewal. Systems with valuable assets that do not use forward secrecy (which is not advisable) should have their keys changed at least quarterly.

Store keys safely

Keep a copy of your keys in a safe location. Losing a server key is usually not a big deal because you can always generate a new one, but it's a different story altogether with keys used for intermediate and private CAs and keys that are used for public key pinning.

Generating and keeping private keys in tamper-resistant hardware is the safest approach you can take, if you can afford it. Such tools are known as *Hardware Security Modules* (HSMs). If you use such a device, private keys never leave the HSM and, in fact, can't be extracted. These days, HSMs are even available as a service. They don't provide the same level of security as tools that you might be able to host in your data centers, but they're a great improvement nevertheless.

Choose the Right Certification Authority

For a small site that needs only a simple domain-validated certificate, virtually any CA will suffice. You can do what I do—just buy the cheapest certificate you can find. Or, if you can automate certificate renewal, just get your certificates for free from Let's Encrypt and other similar providers. After all, any public CA can issue a certificate for your web site without asking you; what's the point of paying more if you don't have to? If you have complex requirements, you may want to explore the commercial options, at which point you should take your time and select a CA that meets your requirements.

Features

At a minimum, you will want to work with a CA that supports both RSA and ECDSA certificate keys. If you care about revocation, your chosen CA must support OCSP certificate revocation checking backed by robust network availability and performance.

We now finally have end user standards for automated certificate issuance (*Automatic Certificate Management Environment*, or ACME for short), and you should use them wherever possible. For this, you'll need a CA that supports automation.

Focus and expertise

PKI is a field that requires deep expertise and dedication; it's easy to make a big mistake. If you're going to be relying on a CA for a critical function, you may as well choose an organization that's serious about it. This is not quite easy to quantify, but you should examine the CA's history, staff, and business direction. It's best to work with CAs for which certificate issuance is the core part of their business.

Service

At the end of the day, it's all about the service. The certificate business is getting more complicated by the day. If you don't have experts on your staff, perhaps you should work with a CA on which you can rely. Costs matter, but so do the management in-

terfaces and the quality of the support. Determine what level of support you will require from your CA, and choose an organization that will be able to provide it when you need it.

You should be aware that if you're getting your certificates from only one CA, they are your single point of failure. If your deployments are sufficiently important to justify the additional effort, consider getting your certificates from two different CAs at the same time. With overlapping certificate lifetimes, you will always have a backup certificate to use if the primary fails for whatever reason.

Prevent Certificate Warnings

Certificate warnings are not unusual and happen for a number of reasons, but all of them can be prevented. The world of technology is confusing enough; you shouldn't add to the cognitive load your users are already experiencing. If you don't pay attention, you will confuse them and weaken their confidence in your technical abilities. In addition, for web sites that disable certificate warnings via *HTTP Strict Transport Security* (HSTS), misconfigured certificates lead to immediate breakage.

Getting certificates right is not very difficult, especially when compared to everything else you need to do to ensure security. With correct initial configuration, renewal automation, and monitoring, you will ensure a smooth experience for your users.

You should pay attention to ensure you have valid certificates for all different domain names and subdomains. As a rule of thumb, keep track of every DNS name that points to your properties and get certificates for all of them. For example, if your main web site is at *www.example.com*, the domain name *example.com* should also have a valid certificate, even though this variant will be configured only to redirect your users to the main location. It's easy to use just one certificate for all of this.

Control Key and Certificate Sharing

In PKI, private keys and certificates can be shared among properties. This practice is not necessarily insecure, but only if it's done in a way that's understood. For best results, don't share. Don't use the same certificate on multiple properties; don't even put different hostnames on the same certificate. With this approach, each property will be independently secured.

The main issue with sharing is that if one property is compromised, the other ones in the same group also follow. There are situations in which this is not a problem. For example, if you have a group of properties that are all managed by the same team and are all part of the same system, sharing is not necessarily bad. On the other hand, multiple teams and multiple distinct properties sharing certificates is always bad.

Wildcard certificates have their place. For example, they are best used by a single property when you need to support an arbitrary number of subdomains, usually one per customer. Avoid them otherwise.

Think Chains, Not Certificates

Although we spend a lot of time talking about server certificates, in practice we need complete and valid *certificate chains* to establish secure connections. Because this is something server operators have to configure manually, mistakes are rife. Most commonly, you will see TLS servers with just the leaf certificate or a set of certificates that don't actually form a valid chain.

An invalid certificate chain may render the server certificate invalid, causing a browser warning. To make things worse, this problem is often difficult to diagnose because some browsers try hard to fix it and others don't. This is a good example of a problem that should be diagnosed by an independent assessment tool.

Deploy Certification Authority Authorization

Certification Authority Authorization (CAA) is a relatively recent security standard that enables you to restrict what CAs are allowed to issue certificates for your properties. CAA is delivered via DNS. When a new certificate is requested, the CA must look for a CAA policy on the affected hosts and verify that they have permission to proceed. If they don't, the issuance fails.

```
example.com.  CAA     0 issue "digicert.com"
example.com.  CAA     0 issue "globalsign.com"
example.com.  CAA     0 issue "letsencrypt.org"
example.com.  CAA     0 issue "sectigo.com"
```

CAA is a very useful addition to the defense arsenal. Even a policy that allows many CAs is helpful as a way of reducing the attack surface, compared to the default, which allows all CAs. Deploying CAA may be difficult in complex environments because a policy set on the domain name applies to all subdomains.

Automate Certificate Renewal

When it comes to certificate lifetimes, renew yearly if you're still doing this work by hand. Aim to automate certificate renewal, then switch to quarterly issuance. Because it is currently impossible to revoke compromised certificates reliably, certificates with shorter lifespans are effectively more secure.

Don't leave it until the last moment to initiate the renewal. In fact, it's better if you start much earlier, about a month before the current certificate expires. Doing so will provide you

with a margin of safety should the new issuance fail for whatever reason. Many things can go wrong, among them issues with the CA itself or issues with the CAA configuration.

For best results, deploy new certificates to production about two weeks after they are issued. This practice (1) helps avoid certificate warnings for some users who don't have the correct time on their computers and also (2) avoids failed revocation checks with CAs that need extra time to propagate their new certificates to their OCSP responders.

Use Certificate Transparency Monitoring

Since 2018, all public certificates are recorded via *Certificate Transparency* (CT), a Google-led effort to improve transparency of the PKI. There are specialized monitoring services that observe all recorded certificates and make it possible to find all certificates issued for your properties. CT monitoring is an excellent and very cost-effective way to understand issuance in complex environments (in terms of who is doing what and where), enforce policy, and catch unexpected certificates or misissuance.

Configuration

When it comes to protocol configuration, your choices are likely to be influenced by a combination of security, interoperability, and regulatory requirements. In the ideal world, focusing on security alone, you would allow only TLS 1.3 and disable all other protocol versions. But that works only in well-understood environments; although modern browsers support TLS 1.3, many other products and tools still don't.

Use Secure Protocols

A web site intended for public use needs to support TLS 1.3 and TLS 1.2 at minimum. You can still use TLS 1.1 and TLS 1.0 if you need to reach a wide audience. The remaining protocols, SSL 3 and SSL 2, are both obsolete and insecure. Let's consider each protocol in turn:

- SSL 2 is completely broken and must not be used. This is the first ever protocol version and is so bad that it can be used to attack even well-configured servers that use overlapping certificates or private keys (the so-called DROWN attack).
- SSL 3 is better, but still insecure when used with HTTP because of the POODLE attack. It's also very weak when used with other protocols. It's obsolete and lacks essential security capabilities. Don't use.
- TLS 1.0 is a legacy protocol that lacks essential security capabilities. Modern clients no longer support this protocol version, but there are still old clients out there that need it. TLS 1.0 is vulnerable to the BEAST attack, although most browsers have deployed mitigations as a workaround.

- TLS 1.1 is also a legacy protocol that lacks modern capabilities. It's usually not needed at all, because all clients that support it also have support for TLS 1.2.
- TLS 1.2 is a relatively modern protocol that can provide good security, but it supports both bad and good cryptographic primitives. It can be used securely, but doing so requires more effort. This protocol version is necessary in order to support a wide range of customers.
- TLS 1.3 is a completely reworked revision of TLS. It's secure by default and requires no further configuration effort. This protocol version, which modern browsers support, should be what protects most of your network communication.

If you need to support older clients and wish to enable TLS 1.0, base your decisions on evidence, not fear. These protocols are seen as not secure enough; for example, the PCI DSS standard no longer allows them.

Use Forward Secrecy

Forward secrecy (also known as *perfect forward secrecy*) is a feature of cryptographic protocols that ensures that every communication (connection) to the server uses a different and unique set of encryption keys. These keys are called *ephemeral* because they are discarded when they are no longer needed. Ephemeral connection keys do not depend on any long-term keys—for example, the server key. When there is no forward secrecy, an adversary who can record your network traffic and later obtain the server key can also decrypt all past communications.

SSL and TLS initially used only the RSA key exchange that doesn't support forward secrecy. To fix that, the ephemeral *Diffie-Hellman* (DHE) and *Elliptic Curve Diffie-Hellman* (ECDHE) key exchanges were added over time, along with some protocol improvements in TLS 1.3. Don't be confused by the fact that RSA can be used for key exchange and authentication; there is nothing wrong with the latter. For as long as you continue to use RSA private keys, the string `RSA` will remain in the suite name.

In TLS 1.2 and earlier protocol releases, the key exchange (and thus forward secrecy) is controlled via cipher suite configuration. Therefore, you want to ensure that all enabled suites embed the keywords `DHE` and `ECDHE`. From TLS 1.3, all suites incorporate forward secrecy and the RSA key exchange is no longer supported.

Use a Strong Key Exchange

In recent years, the DHE key exchange fell out of fashion; many modern clients no longer support it. As a result, there is only one widely supported secure option for the key exchange, and that's ECDHE. Although DHE suites do have some issues, they are not likely to

be a problem in practice if used only as fallback. You shouldn't use the RSA key exchange (not to be confused with RSA keys) because in that case you lose forward security.

For key exchange to be secure, ECDHE and DHE have to be used with secure parameters. For ECDHE, the parameters are called *named curves* and only two are practical: `X25519` and `P-256` (also known as `sec256r1`). For DHE (if using), ensure the parameters provide 2,048 bits of security. Some server software provides secure DHE parameters out of the box; with others, you'll have to provide your own.

Prioritize the Best Cipher Suites

In TLS, servers are in the best position to determine the most secure communication option to use with the connecting clients. That's because the first step of the TLS handshake involves a client sending a list of supported features. What remains is for the server to choose what feature to proceed with.

Unfortunately, many platforms don't actively choose the best option, instead resorting to choosing the first one offered by clients. For the best results, check what your platform does and enable server preference wherever possible. In general, avoid platforms that don't support server preference enforcement as it may not be possible to configure them securely in a general case.

Use Secure Cipher Suites

In TLS, cipher suites are the most visible aspect of server configuration. Usually a lot of effort goes into understanding what options are available, secure, and required to achieve secure communication. Determining what cipher suites to use has traditionally been difficult; over time, the TLS protocol accumulated a very large number of suites, most of which are considered insecure or inadequate today.

On the positive side, TLS 1.3, the most recent incarnation of the TLS protocol, supports only a handful of suites, and all of them are secure. If you base your configuration on this protocol version, all connections with clients that also support it will be secure with ease. You should lead with the following three suites (which are usually enabled by default anyway):

```
TLS_AES_128_GCM_SHA256
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_256_GCM_SHA384
```

When it comes to TLS 1.2, you should rely on cipher suites that provide strong key exchange, forward secrecy, and AEAD (*authenticated encryption with associated data*) encryption of 128 bits. Also use AES and ChaCha20 encryption algorithms. Your configuration can continue to utilize non-AEAD suites, but only to support very old clients, if that's necessary. Don't use anything else unless you're a cryptographer and know what you're doing.

The preceding recommendations, translated to specific suites, yields the following:

```
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
```

This configuration is designed with both security and performance in mind. It supports both ECDSA and RSA keys, with priority given to the former, which is faster. It also includes more suites than strictly necessary in order to support a wider range of clients.

If you're using OpenSSL, the following configuration is exactly the same but uses the nonstandard suite names that OpenSSL will understand:

```
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-ECDSA-CHACHA20-POLY1305
ECDHE-ECDSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES128-SHA
ECDHE-ECDSA-AES256-SHA
ECDHE-ECDSA-AES128-SHA256
ECDHE-ECDSA-AES256-SHA384
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-RSA-CHACHA20-POLY1305
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES128-SHA
ECDHE-RSA-AES256-SHA
ECDHE-RSA-AES128-SHA256
ECDHE-RSA-AES256-SHA384
DHE-RSA-AES128-GCM-SHA256
DHE-RSA-CHACHA20-POLY1305
DHE-RSA-AES256-GCM-SHA384
```

```
DHE-RSA-AES128-SHA
DHE-RSA-AES256-SHA
DHE-RSA-AES128-SHA256
DHE-RSA-AES256-SHA256
```

I recommend that you always configure OpenSSL with an explicit list of desired suites, as indicated earlier. This approach is the simplest and provides great visibility into exactly what is enabled. With OpenSSL, it's also possible to use the legacy, keyword-based configuration approach, but that approach leads to opaque configurations that are difficult to understand and often end up doing the wrong thing.

Ensure Ticket Keys Are Rotated

In TLS, session resumption is implemented using one of two approaches. The original approach is to have the server keep the state in persistent storage somewhere. Later, session tickets were added, and they work like HTTP cookies do. The session state is packaged into a binary blob, encrypted, and sent back to the client to keep and send back to the server later.

When session tickets are used, the security of all connections depends on the main ticket key. This key is used to safely encrypt and decrypt session tickets. The security of the ticket key is an area in which current server software doesn't provide adequate controls. Most applications that rely on OpenSSL use implicit ticket keys that are created on server startup and stay the same for the duration of the process. If the process stays up for weeks and months, so does the ticket key. Backdooring applications is easy; you can inject a static, never-changing ticket key to give you the ability to decrypt all communication. The most secure deployments of TLS configure ticket keys explicitly and rotate them on a predetermined schedule—for example, daily.

Session ticket security is very important to get right if you're deploying TLS 1.2. In this situation, knowing the ticket key is all you need to decrypt past communications. TLS 1.3 brought some much-needed improvement in this area. This updated protocol version uses session tickets for authentication, but has an option (enforced by all modern browsers) to perform an ephemeral Diffie-Hellman handshake on all resumed connections. The end result is that knowing the ticket key is no longer sufficient for passive decryption, making it a much smaller attack vector.

Mitigate Known Problems

There was a period of time when it was common to learn about new protocol issues, but that now appears to be behind us. The problems were exhausted and largely fixed. Then TLS 1.3 came along, which made things much better still. Critical issues at the protocol level are not so common today, but it's generally accepted that security deteriorates over time. For

that reason, it's a good practice to be aware of what's going on. At this point in time, the most likely problems you will encounter are implementation issues in libraries and server software. Apply patches promptly when they become available.

Supporting Legacy Platforms

Sometimes you'll find yourself needing to support legacy clients that don't have the latest and greatest security features. In this case, you will need to reach out for, and enable, certain components that are not ideal but are still acceptable for use in exceptional situations. It could be that the risk of the exploitation is low or that the service is not that valuable, or perhaps you have mitigation measures in place. If you are in this situation, this section is for you; I will outline here some of those imperfect but palatable legacy features.

The good news is that it's generally possible to deploy strong and weak elements at the same time, relying on the TLS negotiation features and server configuration to ensure that individual connections use the best commonly supported features. This means that those weak elements in your configuration will be used only as a last resort.

> **Note**
>
> If you have to resort to using some of these weak encryption components in production, consider splitting your systems into those that are well-configured and those that are intended to serve your legacy clients. Structuring your deployments like that will help you minimize the risk.

RSA private keys
: The ECDSA algorithm is gaining in popularity on account of its speed, but you will often find that it's not supported by some old clients. In this case, fall back to the RSA algorithm. If you care about performance, deploy with two certificates, using both ECDSA and RSA at the same time.

TLS 1.1, TLS 1.0, SSL 3, and SSL 2
: Legacy clients won't support TLS 1.2, so you may need to enable TLS 1.0 for them. This is not terrible and you may find that my recommended suite configuration works for you. If you're considering SSL 3, then you should carefully study the weaknesses of this protocol and determine if its use is acceptable in your situation. SSL 2 cannot be used securely—and it's worse than no encryption because this protocol version can be used to exploit secure servers (via DROWN). Nobody cares about TLS 1.1.

Weak Diffie-Hellman key exchange
: There are some old clients (e.g., Java before version 8) that can't use the DH key exchange at 2,048 bits, which is the recommended strength today. You may consider dropping the strength to 1,024 bits to accommodate these clients. If you do, you must

generate a unique set of DH parameters on each server. You must not use any of the predefined well-known groups because they can be exploited via a precomputation attack. Anything below 1,024 bits is very easy to exploit.

You need to be aware that if you reduce the DH strength, it will affect both modern and legacy clients. This is one aspect of TLS configuration that cannot be negotiated on a per connection basis. The best approach is to have separate systems for modern and legacy customers. If you can't do that, preferring the ECDHE key exchange (as in my recommended configuration) will ensure that modern clients all use ECDHE and never attempt DHE.

Weak cipher suites

Very old clients were never capable of great encryption—for example, often not supporting DHE and ECDHE key exchange or the AES encryption algorithm. This is effectively the situation with Windows XP and some early Android platforms. Therefore, to communicate with those platforms you'll have to support the plain old RSA key exchange that doesn't provide forward security. And for Windows XP, you'll need to support 3DES (slow and on the way down, but not entirely out, strictly speaking). Here are some last-resort suites to place at the end of your prioritized list of suites if there is no other way:

```
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
```

HTTP and Application Security

Although SSL and TLS were designed so that they can secure any connection-oriented protocol, the immediate need was to protect HTTP. To this day, web site encryption remains the most common TLS use case. Over the years, the Web has evolved from a simple document distribution system into a complex application delivery platform. This complexity creates additional attack vectors and requires more effort to secure.

Encrypt Everything

There is no longer any excuse not to encrypt everything by default. A long time ago there was—*maybe*—but not any longer. The first barrier fell with the increase of CPU power, which removed encryption as a bottleneck. More recently, several things happened to make encryption widely adopted. First, there was the rise of Let's Encrypt, which started to offer free certificates and automated issuance. Second, browsers started to mark plaintext content as insecure and search engines started to favor encrypted content.

Mixed content is the name we use to refer to web pages that are themselves encrypted but rely on resources that are not. For example, an HTML page could be fetching audio or visual

files without encryption. The original excuse—that heavy content can't be delivered encrypted—no longer applies, and today we need to deal with the legacy. Browsers have been restricting mixed content for a while. The long-term direction is not only that all content within a page must be encrypted, but also that the related actions (e.g., downloads) must be as well.

Secure Cookies

In HTTP, cookies are a weak link and need additional attention. You could have a web site that is 100% encrypted and yet remains insecure because of how its cookies are configured.

Mark cookies secure
: Cookies will by default span HTTP and HTTPS contexts, which is why they need to be explicitly marked as secure for browsers to know to avoid plaintext.

Mark cookies as HttpOnly
: If a web site uses cookies that need not be accessed from the browser itself, they should be marked as HttpOnly. This is a defense-in-depth technique that aims to minimize the attack surface.

Use cookie name prefixes
: Cookie prefixes are a new security measure that is now supported by browsers and being added to the main cookie specification (RFC 6265bis). Cookies with names that start with prefixes `__Host-` and `__Secure-` are given special powers that address a variety of problems that existed for years. All cookies should be transitioned to use these prefixes.

For best results, consider adding cryptographic integrity validation or even encryption to your cookies. These techniques are useful with cookies that include some application data. Encryption can help if the data inadvertently includes something that the user doesn't already know. Integrity validation will prevent tampering. With these kinds of cookies, it's also a good practice to bond the cookies to the context in which they were issued—for example, to the specific user account.

Use Strict Transport Security

For proper security of the transport layer, you must indicate your preference for encrypted content. *HTTP Strict Transport Security* (HSTS) is a standard that allows web sites to request strict handling of encryption. Web sites signal their policies via an HTTP response header for enforcement in compliant browsers. Once HSTS is deployed, compliant browsers will switch to always using TLS when communicating with the web site. This addresses a number of issues that are otherwise difficult to enforce: (1) users who have plaintext bookmarks

and follow plaintext links, (2) insecure cookies, (3) HTTPS stripping attacks, and (4) mixed-content issues within the same site.

In addition, and perhaps more importantly, HSTS fixes handling of invalid certificates. Without HSTS, when browsers encounter invalid certificates, they allow their users to proceed to the site. Many users can't differentiate between attacks and configuration issues and decide to proceed, which makes them susceptible to active network attacks. With HSTS, certificate validation failures are final and can't be bypassed. That brings TLS back to how it should have been implemented in the first place.

All web sites should deploy HSTS to fix legacy browser issues in how encryption is handled. In fact, deploying HSTS is probably the single most important improvement you can make. The following configuration enables HSTS on the current domain and all subdomains, with a policy duration of one full year:

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

For best results, consider adding your properties to the HSTS preload list. With that, browsers and other clients can ship with a full list of encryption-properties, which means that even first access to those sites can enforce encryption.

> **Warning**
>
> Unless you have full control over your infrastructure, it's best to deploy HSTS incrementally, starting with a short policy duration (e.g., 300 seconds) and no preloading. The fact that HSTS has a memory effect, combined with its potential effect on subdomains, can lead to problems in complex environments. With incremental deployments, problems are discovered while they're still easy to fix. Request preloading as the last deployment step, and after you activate sufficiently long policy duration.

HSTS is not the only technology that can help with enforcing encryption. Although much more recent and with a lot of catching up to do, there are also the *HTTPS DNS resource records*, which build on the DNS infrastructure to carry various metadata, including signaling of support for encryption.

Deploy Content Security Policy

Content Security Policy (CSP) is a mechanism that allows web sites to control how resources embedded in HTML pages are retrieved and over what protocols. As with HSTS, web sites signal their policies via an HTTP response header for enforcement in compliant browsers. Although CSP was originally primarily designed as a way of combating XSS, it has an important application for web site encryption; that is, it can be used to prevent third-party mixed content by rejecting plaintext links that might be present in the page via the following command:

```
Content-Security-Policy: upgrade-insecure-requests
```

Disable Caching

Encryption at the network level prevents network attacks, but TLS doesn't provide end-to-end encryption. Each party involved in the communication has access to the plaintext. Caching is commonly used with HTTP to improve performance, so, for example, browsers may choose to store decrypted data in persistent storage. Intermediate proxy services (e.g., content delivery networks) may choose to not only cache sensitive data, but even share it with other users in some situations.

With the increase of cloud-based application delivery platforms and content delivery networks, it's never been more important to very carefully mark all sensitive content as private. The most secure option is to indicate that the content is private and that it must not be cached:

```
Cache-Control: private, no-store
```

With this setting, neither intermediate devices nor browsers will be allowed to cache the served content.

Be Aware of Issues with HTTP Compression

In 2012, the CRIME attack showed how data compression can be used to compromise network encryption, and TLS in particular. This discovery eventually led to the removal of compression from TLS. The following year, TIME and BREACH attack variations focused on retrieving secrets from compressed HTTP response content. Unlike TLS compression, HTTP compression has a huge performance and financial impact and the world decided to leave it on, along with the security issue.

TIME and BREACH attacks can target any sensitive data embedded in a HTML page, which is why there isn't a generic mitigation technique. In practice, most attacks would target CSRF tokens, which would give attackers the ability to carry out some activity on a web site under the identity of the attacked user. For best security, ensure that CSRF tokens are masked. In addition, web sites should generally be looking at adopting same-site cookies, another recent security measure designed to improve cookie security, this time against CSRF attacks.

Understand and Acknowledge Third-Party Trust

When everything else is properly configured and secured, we still can't escape the fact that many web sites have to rely on services provided by third parties. It could be that some JavaScript libraries are hosted on a content delivery network or that ads are supplied by an

ad delivery network or that there are genuine services (e.g., chat widgets) supplied by others.

These third parties are effectively a backdoor that can be used to break your web site. The bigger the service, the more attractive it is. For example, Google Analytics is known to provide its service to half the Internet; what if its code is compromised?

This is not an easy problem to solve. Although it would be ideal to self-host all resources and have full control over everything, in practice that's not quite possible because we don't have infinite budgets to do everything ourselves. What we should do, however, is evaluate every third-party dependency from a security perspective and ask ourselves if keeping it is worth the risk.

A technology called *Subresource Integrity* (SRI) can be used to secure resources that are hosted by third parties and that don't change. SRI works by embedding cryptographic hashes of included references, which browsers check every time the resource is retrieved.

Performance

Everybody worries about security, but they worry about performance even more. What use is a secure service that people can't or don't want to use? Properly configured, TLS is plenty fast. With little effort, the performance will be good enough. In some cases, it's even possible to deploy TLS with virtually no performance overhead. In this section, we'll look at how you can achieve best-in-class performance with some additional effort.

Don't Use Too Much Security

We all like security, but it's possible to have too much of it. If you go overboard and choose cryptographic primitives that are too strong, your security won't be better in any meaningful way, but your services will nevertheless be slower, and sometimes significantly so. Most sites should aim to use elements that provide 128 bits of security. We make an exception for DHE, which, at 2,048 bits, provides 112 bits of security. That's close enough. You will virtually always use ECDHE anyway, which provides a full 128 bits of security.

The next step up is to use primitives that offer 256 bits of security. This is something you might decide to do if you think quantum computing is a realistic threat.

Enable Session Resumption

In TLS terminology, when a client and server have a successful handshake, they establish a session. Handshakes involve a fair amount of computation, which is why cryptographic protocols focused on performance also incorporate so-called session caching that makes it pos-

sible to continue to use the results of one handshake over a period of time, typically for up to a day. This is called *session caching* or *session resumption*.

Session resumption is an essential performance optimization that ensures smooth operation, even for web sites that don't need to scale. Servers that don't use it or don't use it well are going to perform significantly slower.

Optimize Connection Management

In the early days, slow cryptographic operations were the main bottleneck introduced by encryption. Since then, CPU speed has improved, so much so that most sites don't worry about its overhead. The main overhead of TLS now lies with the increased latency of the handshake. The best way to improve TLS performance is to find ways to avoid handshakes.

Keep connections open
: The best approach to avoiding TLS handshakes is to keep existing connections open for an extended period of time and reuse them for subsequent user requests. In HTTP, this feature is known as *keep-alives*, and using it is a simple matter of web server reconfiguration.

Use TLS 1.3
: The complete redesign of TLS in version 1.3 to improve security was also a good opportunity to improve its performance. As a result, this protocol version reduces full handshake latency by half, compared to the standard handshake in earlier protocol revisions. TLS 1.3 also introduces a special 0-RTT (*zero round-trip time*) mode, in which TLS adds no additional latency over TCP. Your servers will fly with this mode enabled, but with the caveat that using it opens you up to replay attacks. As a result, this mode is not appropriate for use with all applications.

Use modern HTTP protocols
: There was a very fast pace of HTTP protocol evolution recently. After HTTP/1.1, there was a long period of no activity, but then we got HTTP/2 and soon thereafter HTTP/3. These two releases didn't really change HTTP itself but focused on connection management and the underlying transport mechanism, including encryption via QUIC.

Use content delivery networks
: *Content delivery networks* (CDNs) can be very effective at improving network performance, provided they are designed to reduce the network latency of new connections to origin servers. Normally, when you open a connection to a remote server, you need to exchange some packets with it for the handshake to complete. At best, you need to send your handshake request and receive the server's response before you can start sending application data. The further the server, the worse the latency. CDNs, by definition, are going to be close to you, which means that the latency between you and

them is going to be small. CDNs that keep connections to origin servers open won't have to open new connections all the time, leading to potentially substantial performance improvements.

Enable Caching of Nonsensitive Content

An earlier section in this guide recommended that you disable HTTP caching by default. Although that's the most secure option, not all properties require the same level of security. HTTPS is commonly used for all web sites today, even when the content on them is not sensitive. In that case, you want to selectively enable caching in order to improve performance.

The first step might be to enable caching at the browser level by indicating that the content is private:

```
Cache-Control: private
```

If you have a content delivery network in place and want to utilize its caching abilities, indicate that the content is public:

```
Cache-Control: public
```

In both situations, you can use other HTTP caching configuration options to control how the caching is to be done.

Use Fast Cryptographic Primitives

Measured server-side, the overhead of cryptography tends to be very low and there aren't many opportunities for performance improvements. In fact, my recommended configuration is also the fastest as it prefers ECDSA, ECDHE, and AES with reasonable key sizes. These days, to deploy fast TLS, you generally (1) deploy with ECDSA keys and (2) double-check that your servers support hardware-accelerated TLS.

However, things change somewhat if we look at the performance from the client perspective. Recently there's been an explosion in the adoption of mobile devices, which use different processors and have different performance characteristics. What's good for your servers may not be what's good for mobile phones. So which do you want to optimize?

In practice, some organizations are choosing to take a performance hit on their servers in order to provide a better end user experience. In practice, this means that they choose to negotiate ChaCha20 suites with mobile devices because they are not only several times faster, but also consume less battery. But how do we know which clients are mobile devices?

The trick is to use something called an *equal preference cipher suites configuration*. Normally, we want our servers to select the best possible cipher suite, so naturally that would be something with AES-GCM. But with mobile devices we want ChaCha20. BoringSSL was the first

to introduce a feature in which the client's list of suites is analyzed to determine if it prefers ChaCha20 over AES. Only if it did would the server select a ChaCha20 suite over an AES-GCM one. This feature is now also available in OpenSSL.

Validate and Monitor

Configuring TLS, especially for use on web sites, has become increasingly complex in recent years. There are so many options to choose that you're virtually guaranteed to get something wrong when you first try. Moreover, things change—sometimes accidentally, sometimes silently through software upgrades. For that reason, we recommend that you find a reliable configuration assessment tool that you trust. Use it periodically to ensure that you stay secure.

Several modern browser technologies come with reporting facilities, which can give you real-time insight into problems that your users are experiencing. CSP supports reporting and even a report-only mode without policy enforcement. A more recent technology, called *Network Error Logging* (NEL), provides reporting for a wide variety of network problems, including TLS and PKI.[1]

[1] There is also the Expect-CT HTTP response header, which was designed to support early opt-in into CT before this technology became de facto required. With Expect-CT reporting, it's possible to get a full certificate chain that's not CT compliant. Although this feature is very useful, it is most likely that Expect-CT will be deprecated, now that pre-CT public certificates have all expired.