HTTPS / TLS Fundamentals

A Complete 12-Chapter Course

Topics covered:

HTTP's weaknesses & the three guarantees · Crypto primitives · Key exchange & forward secrecy

Certificates & X.509 · CAs & the chain of trust · TLS 1.2 & 1.3 handshakes · Cipher suites

Let's Encrypt & ACME · Server config (HSTS, OCSP) · Attacks & defences · mTLS & Certificate Transparency

Exercises: 36 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · openssl / curl / nmap throughout

Table of Contents

01 Why HTTPS? HTTP's Problems

02 Cryptography Building Blocks

03 Public-Key Crypto & Key Exchange

04 Certificates & the X.509 Format

05 Certificate Authorities & the Chain of Trust

06 The TLS 1.2 Handshake

07 TLS 1.3 — What Changed

08 Cipher Suites & Protocol Versions

09 Getting a Certificate — Let's Encrypt & ACME

10 Configuring HTTPS on a Server

11 Common Problems & Attacks

12 Beyond the Basics

CHAPTER 1 OF 12

Why HTTPS? HTTP's Problems

CHAPTER 1

Why HTTPS? HTTP's Problems

What plain HTTP exposes, and the three guarantees HTTPS adds

Before learning *how* HTTPS works, it's worth being precise about *what problem* it solves. HTTPS is just **HTTP carried over TLS** (Transport Layer Security) — the application protocol is unchanged; what changes is that everything travels through an encrypted, authenticated channel instead of as open text. This chapter sets up the threat model the rest of the course answers.

Plain HTTP Travels in the Clear

An HTTP request is plain text sent over TCP. Every device between your computer and the server — your router, your ISP, every network hop, the coffee-shop Wi-Fi access point — handles those bytes, and with plain HTTP they can read every one of them. A login request looks literally like this on the wire:

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

username=philip&password=hunter2
```

There is no encryption, no signature, no proof of who sent it or who received it. Anyone positioned on the path sees the URL, the headers, the cookies, and the body — including that password in the clear. This is the default behaviour of HTTP, and it is the baseline HTTPS exists to fix.

"IT'S ONLY A SMALL SITE, WHO WOULD BOTHER?"

The risk isn't a targeted hacker picking your site — it's automated, passive collection. Open Wi-Fi networks, compromised routers, and ISP-level logging capture traffic in bulk. And it's not only passwords: session cookies sent in the clear can be copied and replayed to hijack a logged-in session without ever needing the password. Plain HTTP exposes *everything* in the conversation, not just the obviously sensitive fields.

Three Things Can Go Wrong

The weaknesses of plain HTTP fall into three distinct categories. Keeping them separate matters, because HTTPS addresses each one with a different mechanism (which later chapters cover):

THREAT	WHAT THE ATTACKER DOES	EXAMPLE
`Eavesdropping`	Passively *reads* traffic as it passes	Capturing a password or cookie on open Wi-Fi
`Tampering`	Actively *modifies* traffic in transit	Injecting ads/malware into a page; altering a bank transfer amount
`Impersonation`	Pretends to *be* the server (or client)	A fake "example.com" capturing your credentials

When all three are combined by an attacker sitting between the two parties — reading, modifying, and impersonating at once — it's called a **man-in-the-middle (MITM)** attack. The MITM is the central adversary the whole TLS design is built to defeat.

The Three Guarantees HTTPS Provides

Each threat is answered by a corresponding guarantee. These three words are the spine of the entire course — every mechanism you'll learn exists to deliver one of them:

\U0001F512

Confidentiality

Traffic is **encrypted**, so an eavesdropper sees only scrambled bytes — defeats eavesdropping.

\U0001F9FE

Integrity

Each message carries a check that **detects any modification** in transit — defeats tampering.

\U0001FAAA

Authentication

A **certificate** proves the server really is who it claims — defeats impersonation.

Notice the mapping is one-to-one: confidentiality ↔ eavesdropping, integrity ↔ tampering, authentication ↔ impersonation. A common beginner mistake is to think "HTTPS = encryption" and stop there — but encryption alone (confidentiality) would still let you have a perfectly private conversation *with an impostor*. Authentication is what makes the encryption meaningful, by guaranteeing *who* you've established the private channel with.

> **HTTPS, TLS, SSL — THE TERMINOLOGY**
>
> **SSL** (Secure Sockets Layer) was the original 1990s protocol; it's obsolete and insecure, but the name stuck culturally — people still say "SSL certificate." **TLS** (Transport Layer Security) is its modern successor (the current versions are TLS 1.2 and 1.3) and what's actually used today. **HTTPS** is simply "HTTP over TLS." So when someone says SSL, they almost always mean TLS — the course uses the accurate term *TLS* throughout.

What HTTPS Does and Doesn't Protect

A precise mental model also means knowing the limits. HTTPS protects data *in transit between the two endpoints* — and only that:

- **Protected:** the request/response bodies, headers, cookies, and the specific path & query string — all unreadable and unmodifiable to anyone on the network path.
- **Still visible to the network:** the *domain name* you're connecting to (via DNS and the TLS handshake's SNI field) and the rough size/timing of traffic. Observers know you visited `example.com`, just not *what* you did there.
- **Not HTTPS's job at all:** security *at* the endpoints — a hacked server, malware on your machine, a phishing site with its own valid certificate, or a weak password. HTTPS secures the pipe, not what's at either end of it.

> **A PADLOCK MEANS "PRIVATE CHANNEL," NOT "TRUSTWORTHY SITE"**
>
> The browser padlock confirms the connection is encrypted and the certificate is valid for that domain — nothing more. A phishing site at `examp1e.com` can obtain a perfectly legitimate certificate and show a padlock. HTTPS guarantees you're talking privately to *whoever owns that exact domain*; it does **not** guarantee that owner is honest. Don't read the padlock as a safety endorsement of the site's content.

The Cost Is Now Negligible

Historically HTTPS was seen as slow and expensive (CPU cost, paid certificates), which is why plain HTTP lingered. That's no longer true: modern CPUs handle TLS with minimal overhead, TLS 1.3 cut the handshake cost dramatically (Chapter 7), and certificates are now free and automatable via Let's Encrypt (Chapter 9). Today HTTPS is the default expectation — browsers mark plain HTTP pages as "Not Secure," and many web features (HTTP/2, service workers, geolocation) simply refuse to run without it. There's no longer a real argument for plaintext HTTP on the public web.

Hands-On Exercises

EXERCISE 1

Using curl's verbose mode, compare a plain HTTP request and an HTTPS request to the same kind of endpoint, and identify in the output where TLS is negotiated for the HTTPS one. (Hint: `curl -v http://example.com` vs `curl -v https://example.com`.)

View solution

EXERCISE 2

Open your browser's DevTools Network tab, load any HTTPS site, click a request, and locate (a) the protocol/security info confirming the connection is encrypted and (b) the request headers — noting that you can read them locally even though the network cannot. Then map what you see to the three guarantees.

📄 View solution

EXERCISE 3

For each of these scenarios, name which of the three guarantees (confidentiality / integrity / authentication) is the one being violated: (a) an ISP injects an ad banner into a web page; (b) someone on your Wi-Fi reads your session cookie; (c) you connect to a rogue access point posing as your bank's site. Write a one-line justification for each.

📄 View solution

CHAPTER 1 QUICK REFERENCE

- **HTTPS = HTTP over TLS** — same HTTP, carried through an encrypted & authenticated channel
- Plain HTTP is **plaintext**: every hop can read URLs, headers, cookies, and bodies
- Three threats: **eavesdropping** (read), **tampering** (modify), **impersonation** (pretend to be the server)
- A **man-in-the-middle (MITM)** combines all three — the core adversary TLS defeats
- Three guarantees: **confidentiality** (encryption), **integrity** (tamper-detection), **authentication** (certificates)
- HTTPS ≠ "encryption only" — without authentication you'd just have a private chat with an impostor
- **SSL** is the obsolete ancestor of **TLS**; "SSL certificate" colloquially means a TLS certificate
- Protects data **in transit**, not the endpoints; the padlock means "private," not "trustworthy"
- **Next chapter:** the cryptographic building blocks — symmetric vs asymmetric encryption, hashing, MACs

CHAPTER 2 OF 12

Cryptography Building Blocks

CHAPTER 2

Cryptography Building Blocks

Symmetric & asymmetric encryption, hashing, and MACs — the primitives TLS combines

TLS doesn't invent its own cryptography — it assembles a handful of well-studied primitives into a protocol. This chapter introduces those primitives in plain terms, mapping each to the guarantee it provides (Chapter 1). You don't need the maths; you need to know what each tool *does*, what key it uses, and what it can't do alone.

Symmetric Encryption — One Shared Key

Symmetric encryption uses a *single secret key* to both encrypt and decrypt. The same key locks and unlocks — like a physical door key that both parties hold a copy of. The dominant algorithm is **AES** (Advanced Encryption Standard).

```
# the same key both scrambles and unscrambles
plaintext  --[ encrypt with KEY ]-->  ciphertext
ciphertext --[ decrypt with KEY ]-->  plaintext
```

Symmetric encryption is **fast** — easily fast enough to encrypt every byte of a web page or video stream — and it delivers **confidentiality**. Its one hard problem: both sides must already share the same secret key. How do two strangers on the internet agree on a shared secret without an eavesdropper learning it? They can't, with symmetric crypto alone. That gap is exactly what asymmetric crypto solves.

Asymmetric Encryption — A Key Pair

Asymmetric (or **public-key**) encryption uses a *pair* of mathematically linked keys: a **public key** that can be shared with anyone, and a **private key** kept secret by its owner. What one key locks, only the other can unlock. The common algorithms are **RSA** and the elliptic-curve family (**ECDH/ECDSA**).

USE THE…	TO…	GIVING YOU
`recipient's PUBLIC key to encrypt`	send a secret only the holder of the matching private key can read	Confidentiality (anyone can encrypt *to* you)

USE THE...	TO...	GIVING YOU
`your own PRIVATE key to sign`	prove a message came from you (only you have that key)	Authentication (a digital signature)

This neatly solves the "strangers sharing a secret" problem: you can hand your public key to the whole world, and anyone can use it to send you something only your private key can open. The catch is that asymmetric operations are **slow** and computationally heavy — far too slow to encrypt an entire data stream.

> **THE HYBRID INSIGHT THAT MAKES TLS WORK**
>
> Symmetric is fast but needs a pre-shared key; asymmetric solves key sharing but is slow. TLS combines them: it uses **asymmetric** crypto briefly, during the handshake, only to *agree on a fresh symmetric key* — then switches to fast **symmetric** encryption (AES) for all the actual data. Slow-but-clever to set up, fast-and-simple to run. This hybrid model is the single most important idea in the whole course; Chapters 3 and 6 build directly on it.

Hashing — A One-Way Fingerprint

A **cryptographic hash function** (e.g. **SHA-256**) takes any input and produces a fixed-size "fingerprint" — the *digest*. It has three defining properties:

- **Deterministic:** the same input always yields the same digest.
- **One-way:** you cannot reverse a digest back into the original input.
- **Collision-resistant & avalanche:** two different inputs (practically) never share a digest, and changing a single bit of input scrambles the entire output.

```
echo "hello" | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03

# change ONE character -> a completely different digest (avalanche)
echo "hellp" | sha256sum
b40711a88c7039756fb8a73827eabe2c0fe5a0346ca7e0a104adc0fc764f528d
```

Hashing on its own provides **integrity checking**: if you know the expected digest of some data, you can re-hash what you received and confirm it wasn't altered. But note what hashing alone does *not* give you — see the next section.

> **A PLAIN HASH DOESN'T PROVE WHO SENT THE DATA**
>
> A hash detects accidental or malicious *changes* — but an attacker who modifies the data can simply recompute the hash to match their tampered version, since hash functions are public and keyless. So a bare hash protects integrity only if the digest itself arrives through some trusted, untampered channel. To bind integrity to a *sender*, you need a key in the mix — that's a MAC.

MACs — Integrity WITH a Key

A **MAC** (Message Authentication Code) is like a hash, but it also folds in a *secret key*. Only someone holding the key can produce a valid MAC for a message, and only someone with the key can verify it. The common construction is **HMAC** (e.g. HMAC-SHA256).

```
# sender, holding KEY:
tag = HMAC(KEY, message)        # send message + tag

# receiver, also holding KEY:
recompute HMAC(KEY, message) and compare to tag
  match    -> message is intact AND came from a key-holder
  mismatch -> tampered, or forged by a non-key-holder -> reject
```

Because forging a valid tag requires the secret key, a MAC delivers integrity *and* a form of authentication at once: it proves the message wasn't altered **and** that it came from someone who shares the key. This is how TLS protects each record of application data after the handshake — using the symmetric key it just negotiated.

> **MODERN TLS USES AEAD — ENCRYPTION AND INTEGRITY IN ONE STEP**
>
> Rather than encrypt and then separately MAC, modern TLS (1.2's GCM suites and all of 1.3) uses **AEAD** ciphers — *Authenticated Encryption with Associated Data*, such as **AES-GCM** and **ChaCha20-Poly1305**. A single AEAD operation provides confidentiality *and* integrity together, producing both ciphertext and an authentication tag. You'll see names like `TLS_AES_256_GCM_SHA384` in Chapter 8 — the "GCM" part is the AEAD mode. Conceptually it's still "symmetric encryption + a MAC," just fused into one safer primitive.

Putting the Primitives Against the Guarantees

PRIMITIVE	KEY MODEL	SPEED	PROVIDES
Symmetric (AES)	one shared secret key	fast	Confidentiality
Asymmetric (RSA, ECDH)	public + private key pair	slow	Key exchange & Authentication
Hash (SHA-256)	no key	fast	Integrity (keyless)
MAC / HMAC	one shared secret key	fast	Integrity + sender authenticity
AEAD (AES-GCM)	one shared secret key	fast	Confidentiality + Integrity together

No single primitive provides all three guarantees — TLS is essentially the recipe for combining them: asymmetric crypto to *establish* a shared symmetric key and to *authenticate* the server (via certificates, Chapter

4), then AEAD symmetric crypto to *protect* every byte of data with confidentiality and integrity. The next chapter zooms in on the asymmetric half: how two parties actually agree on that shared key.

Hands-On Exercises

EXERCISE 1

Use sha256sum (or `openssl dgst -sha256`) to hash a short string, then hash it again after changing a single character. Confirm the digest length is identical but the value is completely different (the avalanche effect), and explain why this property matters for integrity checking.

View solution

EXERCISE 2

Using openssl, time a symmetric operation vs an asymmetric one to feel the speed gap: generate an RSA key and do a public-key operation, and separately encrypt a chunk of data with AES. Then explain in your own words why TLS uses the slow asymmetric step only to set up the fast symmetric one.

View solution

EXERCISE 3

For each item, name the primitive that fits and the guarantee it provides: (a) encrypting a 4 GB video stream efficiently; (b) letting strangers send you a secret with nothing pre-shared; (c) proving a downloaded file wasn't corrupted, given a trusted digest; (d) proving a message both is intact and came from someone sharing your key.

View solution

CHAPTER 2 QUICK REFERENCE

- **Symmetric (AES)** — one shared key encrypts & decrypts; fast; gives confidentiality; problem = sharing the key
- **Asymmetric (RSA, ECDH)** — public + private key pair; slow; solves key sharing & enables signatures/authentication
- **Hybrid model** — asymmetric to agree a fresh symmetric key, then symmetric for the bulk data (the core TLS idea)
- **Hash (SHA-256)** — keyless one-way fingerprint; deterministic, irreversible, avalanche; gives keyless integrity
- A plain hash **can't prove who** sent data — an attacker recomputes it; you need a key
- **MAC / HMAC** — hash + secret key; integrity AND sender authenticity
- **AEAD (AES-GCM, ChaCha20-Poly1305)** — fuses encryption + integrity in one step; used by modern TLS

- **Next chapter:** public-key crypto & key exchange — RSA vs Diffie–Hellman, and forward secrecy

CHAPTER 3 OF 12

Public-Key Crypto & Key Exchange

CHAPTER 3

Public-Key Crypto & Key Exchange

RSA key transport vs Diffie–Hellman, and the idea of forward secrecy

Chapter 2 established the hybrid model: use asymmetric crypto to agree on a shared symmetric key, then switch to fast symmetric encryption. This chapter answers the obvious follow-up — *how exactly* do the two parties agree on that key over a network an attacker is watching? There are two historical approaches, and the difference between them turns out to matter enormously for security.

Approach 1: RSA Key Transport (the old way)

The original TLS method used **RSA key transport**. The logic is simple:

1. The server has an RSA key pair; its public key is in its certificate (Chapter 4).
2. The client invents a random secret (the "pre-master secret"), **encrypts it with the server's public key**, and sends it.
3. Only the server's private key can decrypt it, so now both sides share that secret — from which the symmetric key is derived.

It works and it's easy to picture: the client effectively puts the secret in a box only the server can open. But it has a serious, subtle weakness that took the industry years to fully act on.

RSA KEY TRANSPORT HAS NO FORWARD SECRECY

With this method, every session's secret is protected by *the same long-lived server private key*. Imagine an attacker records years of encrypted traffic today, then later steals (or legally compels) the server's private key. They can now go back and decrypt **all** of that recorded traffic — every past session — because each one's pre-master secret was encrypted to that one key. One key compromise retroactively unlocks the entire archive. This "harvest now, decrypt later" exposure is why RSA key transport was **removed entirely in TLS 1.3**.

Approach 2: Diffie–Hellman Key Exchange (the modern way)

Diffie–Hellman (DH) solves the problem differently — and almost magically. It lets two parties derive a *shared secret* by exchanging only *public* values, such that an eavesdropper who sees everything sent still

cannot compute the secret. The classic intuition is mixing paint:

```
# Paint-mixing analogy for Diffie-Hellman
1. Both agree publicly on a common base colour   (yellow)
2. Each secretly picks a private colour           (client: red,  server: blue)
3. Each mixes base + their secret, sends the mix  (client sends orange, server sends green)
   # an eavesdropper sees yellow, orange, green
4. Each adds their OWN secret to the other's mix:
     client: green  + red    = brown
     server: orange + blue   = brown   # same shared secret!
# the eavesdropper cannot make brown: "un-mixing" a colour is infeasible
```

The shared secret (brown) is *never transmitted* — each side computes it locally by combining their own private value with the other's public value. The "un-mixing is hard" property is, in real DH, the difficulty of certain mathematical problems (discrete logarithms). Critically, the secret was never put in a box tied to a long-lived key — it emerged from values that can be thrown away after the session.

Ephemeral Diffie–Hellman = Forward Secrecy

The security payoff comes when the DH private values are **ephemeral** — freshly generated for each session and discarded immediately afterward. This is signalled by an **E** in cipher names: **DHE** (ephemeral DH) and **ECDHE** (the elliptic-curve, faster variant used almost universally today).

> **WHAT "FORWARD SECRECY" ACTUALLY BUYS YOU**
>
> With ephemeral DH, each session's key is derived from one-time values that are deleted when the session ends. So even if the server's long-term private key is stolen *later*, there's nothing left to decrypt past sessions with — the ephemeral secrets are long gone, and the long-term key was never what protected the session data in the first place (it was only used to *sign*, not to encrypt the secret). The archive of recorded traffic stays safe. This property is called **Forward Secrecy** (or Perfect Forward Secrecy, PFS), and it's the headline reason TLS moved to ephemeral DH.

But Wait — Where Does Authentication Fit?

Plain Diffie–Hellman has a gap: it establishes a shared secret with *somebody*, but it doesn't prove *who*. An active man-in-the-middle could run separate DH exchanges with each side and sit in the middle. DH alone gives confidentiality, not authentication — exactly the Chapter 1 warning that encryption without authentication is a private chat with a possible impostor.

TLS closes the gap by combining DH with a **signature**: the server signs its DH public value with the long-term private key from its certificate. So the two asymmetric tools play distinct roles, and it's worth keeping them separate in your mind:

ASYMMETRIC ROLE	TOOL	PROVIDES	KEY USED
Key agreement	(EC)DHE	a shared secret + forward secrecy	ephemeral, per-session, discarded
Authentication	signature (RSA or ECDSA)	proof of server identity	long-term key from the certificate

This is a key clarification: in modern TLS the certificate's long-term key is used to **sign** (authenticate), *not* to encrypt the session secret. The session secret comes from ephemeral DH. That separation is precisely what delivers both authentication *and* forward secrecy at once — the old RSA-transport method conflated the two into one key and lost forward secrecy as a result.

RSA Transport vs Ephemeral DH — Side by Side

	RSA KEY TRANSPORT (LEGACY)	EPHEMERAL DH — (EC)DHE (MODERN)
How secret is shared	client encrypts it to server's public key	both derive it from exchanged public values
Long-term key's job	decrypts the session secret	only signs (authenticates)
Forward secrecy	No — key theft decrypts all past traffic	Yes — past sessions stay safe
Status	removed in TLS 1.3	the only option in TLS 1.3

The progression is the whole point of this chapter: TLS 1.2 supported both, and TLS 1.3 (Chapter 7) dropped RSA key transport altogether, making forward secrecy mandatory. Every modern HTTPS connection you make uses ephemeral elliptic-curve Diffie–Hellman (ECDHE) for key agreement, authenticated by a certificate signature. The next chapter examines that certificate itself.

Hands-On Exercises

EXERCISE 1

Connect to a real site with `openssl s_client -connect example.com:443` and find the line reporting the negotiated key exchange / cipher (look for "Server Temp Key" and the cipher name). Identify whether ECDHE is in use, and explain what the "E" guarantees.

View solution

EXERCISE 2

Walk through the paint-mixing analogy with actual small numbers using the real DH formula (pick a small prime p and base g, choose two private exponents, compute the public values and the shared secret both ways). Confirm both sides reach the same secret, and state what an eavesdropper would and wouldn't know.

📄 View solution

EXERCISE 3

Explain the "harvest now, decrypt later" attack in one paragraph, then state precisely why RSA key transport is vulnerable to it but ephemeral ECDHE is not. Be specific about which key is stolen and what it can (or can't) unlock in each case.

📄 View solution

CHAPTER 3 QUICK REFERENCE

- **Key exchange** = how two parties agree on the shared symmetric key over a watched network
- **RSA key transport (legacy)** — client encrypts the secret to the server's public key; simple but NO forward secrecy
- **Diffie–Hellman** — both derive a shared secret from exchanged PUBLIC values; the secret is never transmitted
- **Ephemeral DH (DHE / ECDHE)** — fresh per-session values, discarded after = **forward secrecy**
- **Forward secrecy** — stealing the long-term key later can't decrypt past recorded sessions
- Plain DH gives confidentiality but not authentication — TLS **signs** the DH value with the certificate key
- Modern split: **(EC)DHE** agrees the key (ephemeral), the **certificate key** only signs/authenticates
- TLS 1.3 **removed RSA key transport** — ECDHE forward secrecy is now mandatory
- **Next chapter:** certificates & the X.509 format — what proves the server's identity

CHAPTER 4 OF 12

Certificates & the X.509 Format

CHAPTER 4

Certificates & the X.509 Format

What a certificate actually contains, and how to read one with openssl

Chapter 3 left one piece dangling: the server authenticates itself by *signing* the handshake with its long-term private key — but how does the client know that key belongs to the right server? The answer is the **certificate**: a signed document binding a public key to an identity (a domain name). This chapter opens one up and names every part.

What Problem a Certificate Solves

A raw public key is just a number — it carries no identity. If a server simply handed you a public key, an impostor could hand you *their* public key just as easily; you'd have no way to tell which one truly belongs to `example.com`. A certificate fixes this by **binding a public key to a name** and having a trusted third party (a Certificate Authority, Chapter 5) vouch for that binding with a signature.

A CERTIFICATE IS "A PUBLIC KEY + AN IDENTITY + A VOUCHING SIGNATURE"

Strip away the jargon and a TLS certificate is three things glued together: (1) a **public key**, (2) the **identity** it belongs to (the domain name), and (3) a **signature** from a Certificate Authority asserting "I verified that this key really does belong to this domain." Everything else in the X.509 format is supporting detail around those three essentials.

X.509 — The Standard Certificate Format

TLS certificates follow the **X.509** standard, which defines the fields a certificate contains. Here are the ones that matter, as you'll see them in real output:

FIELD	WHAT IT HOLDS	WHY IT MATTERS
`Subject`	who the cert identifies — incl. Common Name (CN)	the identity being vouched for
`Subject Alternative Name (SAN)`	the list of domain names the cert is valid for	**this** is what browsers actually check today
`Issuer`	which CA issued & signed this cert	the next link up the trust chain (Ch. 5)

FIELD	WHAT IT HOLDS	WHY IT MATTERS
`Validity (Not Before / Not After)`	the date range the cert is valid	expired or not-yet-valid = rejected
`Public Key`	the server's public key + algorithm (RSA/EC)	the key being bound to the identity
`Signature`	the issuer's signature over all the above	makes the cert tamper-evident
`Serial / Extensions`	unique ID, key-usage flags, CRL/OCSP URLs	revocation & usage constraints (Ch. 10–11)

Reading a Real Certificate

You can fetch and decode any site's certificate with `openssl`. The `-text` option prints the X.509 fields in human-readable form:

```
# fetch the cert and print its fields
openssl s_client -connect example.com:443 -servername example.com </dev/null \
  | openssl x509 -text -noout
```

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0a:1b:2c: ...
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=DigiCert Inc, CN=DigiCert Global G2 TLS RSA SHA256 2020 CA1
        Validity
            Not Before: Jan  1 00:00:00 2025 GMT
            Not After : Apr  1 23:59:59 2025 GMT
        Subject: CN=example.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
        X509v3 extensions:
            X509v3 Subject Alternative Name:
                DNS:example.com, DNS:www.example.com
```

Every concept so far appears here: the **Subject** identity, the **Public Key** (the one used in Chapter 3's signature), the **Issuer** (the CA that vouched for it), the **Validity** window, and the **SAN** list of covered domains.

SAN, NOT COMMON NAME, IS WHAT'S CHECKED TODAY

Historically the domain lived in the **Common Name (CN)** inside the Subject. Modern browsers *ignore CN for hostname matching* and require the domain to appear in the **Subject Alternative Name** extension — a cert whose name is only in CN and not in SAN will be rejected as invalid. SAN also allows *multiple* names (and wildcards like

`*.example.com`) in one cert, which CN couldn't. When you check "does this cert match the site I'm visiting?", you're checking the SAN list.

How the Signature Makes It Tamper-Proof

The certificate's integrity rests on the primitives from Chapter 2. The CA computes a **hash** of all the certificate's data fields, then **signs** that hash with the CA's *private* key. Anyone can verify it using the CA's *public* key:

```
# conceptually, certificate verification:
1. hash the cert's data fields            -> digest_now
2. decrypt the signature w/ CA public key -> digest_signed_by_CA
3. compare:
     digest_now == digest_signed_by_CA ?  valid, untampered
                                  else  ?  altered or forged -> reject
```

Because only the CA's private key can produce a signature that verifies against its public key, nobody can alter a single field (say, swap in their own public key or change the domain) without invalidating the signature. This is the same hash-then-sign pattern from Chapter 2, applied to the certificate itself — and it's why a certificate can be transmitted in the clear yet still be trustworthy.

PEM vs DER — Encodings, Not Different Certificates

Certificates come in two file encodings, which trip up beginners because they look totally different but hold the same data:

ENCODING	LOOKS LIKE	TYPICAL EXTENSIONS
PEM	Base64 text between `-----BEGIN CERTIFICATE-----` markers	.pem .crt .cer
DER	raw binary (not human-readable)	.der .cer

PEM is the common one on Linux servers (and what you paste into config files); DER is binary. They're interconvertible with `openssl x509 -inform/-outform`, and `-text` decodes either into the readable field listing above. Don't mistake the encoding for the content — a `.pem` and a `.der` of the same cert are the same certificate.

VALIDITY WINDOWS ARE SHORT NOW — AND THAT'S DELIBERATE

Notice the example cert is valid for only ~90 days. Public TLS certificates have moved to short lifetimes (Let's Encrypt issues 90-day certs, and the industry is moving shorter still) because short windows limit the damage of an undetected

key compromise and make revocation (Chapter 11) less critical. Short lifetimes are only practical because issuance is automated — which is exactly what ACME/Let's Encrypt (Chapter 9) enables.

Hands-On Exercises

EXERCISE 1

Fetch a real site's certificate with openssl and print it with `x509 -text -noout`. Locate and write down its Subject, Issuer, Validity dates, public key algorithm/size, and the Subject Alternative Name list.

View solution

EXERCISE 2

Use targeted openssl flags to extract just specific fields: `-subject`, `-issuer`, `-dates`, and `-ext subjectAltName`. Then explain why a browser visiting www.example.com checks the SAN list rather than the Common Name.

View solution

EXERCISE 3

Inspect a certificate in your browser's certificate viewer (click the padlock) and find the same fields. Then reason about three rejection scenarios: an expired cert, a cert whose SAN doesn't include the visited domain, and a cert with one data field altered after signing — state which check fails in each.

View solution

CHAPTER 4 QUICK REFERENCE

- A **certificate** binds a **public key** to an **identity** (domain), vouched for by a CA's **signature**
- It solves the problem that a raw public key carries no identity — an impostor's key looks the same
- **X.509** is the format; key fields: Subject, SAN, Issuer, Validity, Public Key, Signature
- **SAN (Subject Alternative Name)** — the domain list browsers actually check; CN is ignored for matching
- Read one with `openssl x509 -text -noout` (decodes both PEM and DER)
- The CA **hashes then signs** the cert data — altering any field breaks the signature (tamper-evident)
- **PEM** (Base64 text) vs **DER** (binary) are encodings of the *same* certificate
- Validity windows are short (~90 days) by design — automation (ACME) makes that practical

- **Next chapter:** Certificate Authorities & the chain of trust — who signs the signers

CHAPTER 5 OF 12

Certificate Authorities & the Chain of Trust

CHAPTER 5

Certificate Authorities & the Chain of Trust

Root, intermediate, and leaf certificates — how a browser decides to trust

Chapter 4 said a certificate is trustworthy because a Certificate Authority signed it. But that just moves the question: *why trust the CA?* This chapter answers it by following the chain all the way up to its anchor — and explains the one thing your device actually has to trust to begin with.

The Bootstrapping Problem

A certificate's signature is only as good as your trust in whoever signed it. If you have to verify the signer with *another* certificate, and that one with another, the chain has to stop somewhere — otherwise it's turtles all the way down. The chain stops at a **root certificate** that your device trusts *inherently*, not because something else vouched for it.

THE TRUST STORE IS THE REAL ROOT OF TRUST

Your operating system and browser ship with a pre-installed **trust store** (also called the root store): a curated set of a few hundred **root CA certificates** that are trusted by default. Everything in HTTPS ultimately rests on this list. When a chain leads up to a root that's in your trust store, the certificate is trusted; if it leads to a root that *isn't*, it's rejected. Trust isn't magic — it's membership in that pre-shipped list.

Three Tiers: Root, Intermediate, Leaf

Real-world certificates form a chain of (usually) three levels. Each certificate is signed by the one above it:

ROOT CA — IN YOUR TRUST STORE
`CN=DigiCert Global Root G2`
self-signed · private key kept offline · Subject = Issuer

▲ signs ▲

INTERMEDIATE CA
`CN=DigiCert Global G2 TLS RSA SHA256 2020 CA1`
signed by the root · does the day-to-day issuing

▲ signs ▲

LEAF (END-ENTITY) — THE SERVER'S CERT
`CN=example.com`

signed by the intermediate · what the website presents

TIER	SIGNED BY	ROLE
`Root CA`	itself (self-signed)	the trust anchor; lives in the trust store; key guarded offline
`Intermediate CA`	the root (or another intermediate)	issues leaf certs daily, shielding the root key from exposure
`Leaf / end-entity`	an intermediate	the actual certificate for a website's domain

WHY INTERMEDIATES EXIST: PROTECTING THE ROOT KEY

The root's private key is enormously valuable — if it leaked, every device on earth trusts it, and it can't be quietly swapped out (it's baked into trust stores that update slowly). So roots are kept **offline**, often literally in air-gapped hardware in a vault, and used only rarely to sign intermediates. The intermediate does the high-volume daily issuing. If an intermediate key is compromised, it can be revoked and replaced *without* touching the root or re-shipping trust stores. The two-tier split is damage control by design.

How Verification Walks the Chain

When your browser receives the server's certificate, it builds and verifies the chain link by link. The server is expected to send its leaf cert *plus* the intermediate(s) — but **not** the root, which the client already has:

```
# for each link, verify the signature using the issuer's public key
1. leaf "example.com"  -- signed by -->  intermediate
     verify leaf's signature with the INTERMEDIATE's public key   ok
2. intermediate        -- signed by -->  root
     verify intermediate's signature with the ROOT's public key   ok
3. root: is it in my trust store?                                  yes -> TRUSTED
   (if the root is NOT in the trust store)                         -> REJECTED
```

At each step the browser *also* re-checks everything from Chapter 4: validity dates, that the issuer's certificate is allowed to act as a CA, and (at the leaf) that the SAN matches the site. The chain is only trusted if **every** link verifies *and* it terminates at a root in the trust store.

THE #1 REAL-WORLD TLS MISCONFIGURATION: MISSING INTERMEDIATE

A server that sends only its leaf certificate — forgetting to include the intermediate(s) — produces an incomplete chain. Some clients happen to cache the intermediate and succeed, while others fail with "unable to get local issuer certificate," giving the maddening "works in my browser but not on her phone" bug. The fix is to install the *full chain* (leaf + intermediates) on the server. This is the single most common HTTPS setup mistake, and Chapter 10 returns to it.

What a CA Actually Verifies (and Doesn't)

Before issuing a leaf certificate, a CA validates the request — but how much it checks defines the **validation level**:

LEVEL	WHAT THE CA VERIFIES	USED FOR
`DV (Domain Validation)`	only that you control the domain (Chapter 9)	the vast majority of sites; free via Let's Encrypt
`OV (Organization Validation)`	domain control + the organization's real existence	business sites wanting vetted identity
`EV (Extended Validation)`	a rigorous legal-identity check	once gave the "green bar"; now visually de-emphasized

Crucially, even the strictest level only verifies *identity*, never *honesty*. A DV certificate proves "the holder controls this domain" — nothing about whether the site is safe. This is the Chapter 1 point made concrete: the chain of trust authenticates *who* you're talking to, not whether they deserve your trust.

> **"TRUSTED" CA MEANS "TRUSTED TO VERIFY DOMAIN CONTROL," NOT "TRUSTWORTHY SITE"**
>
> A CA being in your trust store means the browser trusts it to *do its verification job correctly* — to not issue a cert for `yourbank.com` to someone who doesn't control that domain. It is not an endorsement of any site the CA issues to. This is also why a single misbehaving CA is dangerous: a CA that wrongly issues a cert for a domain it shouldn't can enable impersonation of that site — which is what Certificate Transparency (Chapter 12) exists to catch.

Self-Signed Certificates

A **self-signed certificate** is its own issuer (Subject = Issuer) with no chain to a trusted root — exactly what a root CA is, except nobody pre-trusts *yours*. Browsers reject it with a warning because it provides encryption but no third-party-verified authentication. They're fine for local development and internal testing (where you control both ends), but never for the public web. You can create one in a single command, which the exercises explore.

Hands-On Exercises

EXERCISE 1

Use `openssl s_client -connect ... -showcerts` to dump the full chain a server sends, and identify each certificate's Subject and Issuer. Confirm the leaf's Issuer equals the intermediate's Subject (the links match up), and note whether the root is included in what the server sent.

View solution

EXERCISE 2

Generate a self-signed certificate with a single `openssl req -x509` command, inspect it to confirm Subject == Issuer, and explain in your own words exactly why a browser refuses to trust it even though the connection it provides is fully encrypted.

View solution

EXERCISE 3

Find your system/browser trust store (list the installed root CAs), and reason through two scenarios: (a) a company installs its own root CA on employee laptops to inspect traffic — why does that work and what does it imply; (b) a server omits its intermediate cert — why do some clients succeed while others fail?

View solution

CHAPTER 5 QUICK REFERENCE

- Trust must bottom out somewhere — at a **root certificate** trusted *inherently*, not via another signature
- The **trust store** (root store) is the pre-shipped list of root CAs your OS/browser trusts by default
- Three tiers: **Root** (self-signed, offline) → **Intermediate** (daily issuing) → **Leaf** (the site's cert)
- Intermediates exist to keep the **root key offline** and make compromise recoverable without re-shipping trust stores
- Verification walks the chain: each cert's signature checked with the issuer's key, ending at a **trusted root**
- Server must send **leaf + intermediates** (not the root) — a **missing intermediate** is the #1 setup bug
- Validation levels: **DV** (domain control) / **OV** (org) / **EV** (legal identity) — all verify identity, not honesty
- **Self-signed** cert = its own issuer, no trusted chain → browser warning; fine for local dev only
- **Next chapter:** the TLS 1.2 handshake — putting keys, certs, and the chain together step by step

CHAPTER 6 OF 12

The TLS 1.2 Handshake

CHAPTER 6

The TLS 1.2 Handshake

Putting keys, certificates, and the chain together — message by message

Everything so far — symmetric/asymmetric crypto, key exchange, certificates, the trust chain — comes together in the **handshake**: the short negotiation at the start of every HTTPS connection that authenticates the server and establishes the shared symmetric key. This chapter walks the TLS 1.2 handshake message by message. (Chapter 7 shows how 1.3 streamlines it; understanding 1.2 first makes that contrast clear.)

What the Handshake Must Accomplish

Before the first byte of HTTP is sent, the two parties must agree on four things. Keep these goals in mind — every message below serves one of them:

- **Agree the protocol & cipher** — which TLS version and cipher suite both support (Chapter 8).
- **Authenticate the server** — verify its certificate chains to a trusted root (Chapters 4–5).
- **Establish a shared secret** — via key exchange, ideally ephemeral for forward secrecy (Chapter 3).
- **Confirm both sides match** — verify nothing was tampered with before switching to encryption.

The Handshake, Message by Message (ECDHE)

1. ClientHello

The client opens with its capabilities: the highest **TLS version** it supports, a list of **cipher suites** it can use, a freshly generated **client random** (random bytes), and extensions — notably **SNI** (the hostname it wants, so a server hosting many sites returns the right certificate) and the supported elliptic curves.

2. ServerHello

The server replies with its *choices* from what the client offered: the agreed TLS version, the single **cipher suite** it selected, and its own **server random**. These two randoms aren't the secret — but they feed into deriving the final keys, ensuring every session is unique even with the same long-term key.

3. Certificate

The server sends its **certificate plus the intermediate chain** (Chapter 5). The client verifies it now: walks the chain to a trusted root, checks validity dates, and confirms the SAN matches the SNI hostname. If any check fails, the handshake aborts here with a warning — this is the **authentication** goal.

4. ServerKeyExchange (signed — the crucial step)

For ephemeral key exchange (ECDHE), the server generates its **ephemeral DH public value** and sends it — *signed with the private key from its certificate*. This single message is where Chapter 3's two roles fuse:

```
# the server's DH value is signed over both randoms + the DH params
signature = Sign(server_private_key,
                 client_random + server_random + server_DH_public)

# the client verifies with the public key FROM THE CERTIFICATE it just checked
Verify(cert_public_key, signature) ?  ok -> this DH value really came from the authenticated server
                                  else  abort
```

The signature binds the ephemeral key to the authenticated identity, defeating the man-in-the-middle: an attacker can't substitute their own DH value because they can't produce a valid signature without the server's private key. **This is the heart of the handshake** — authentication and key agreement joined in one step.

5. ServerHelloDone

A short marker: "I'm done with my side of the hello." The client now has everything it needs to contribute its half of the key exchange.

6. ClientKeyExchange

The client generates *its* ephemeral DH public value and sends it. Now both sides hold the other's DH public value, so each independently computes the same shared **pre-master secret** (the Diffie–Hellman magic from Chapter 3 — the secret itself is never transmitted).

FROM PRE-MASTER SECRET TO SESSION KEYS

Both sides now combine the **pre-master secret** with the **client random** and **server random** through a key-derivation function (the PRF) to produce the **master secret**, and from it the actual symmetric keys used for the rest of the session. Because the two randoms feed in, two sessions never reuse the same keys even if every other input repeats. This is the moment the connection transitions from the slow asymmetric setup to the fast symmetric phase — the hybrid model (Chapter 2) in action.

7. ChangeCipherSpec + Finished

Each side sends **ChangeCipherSpec** ("everything I send after this is now encrypted with the new keys") followed immediately by an encrypted **Finished** message. Finished contains a **hash of the entire handshake transcript** so far. Each side checks the other's Finished against its own view of the transcript:

FINISHED IS THE ANTI-TAMPERING SEAL ON THE WHOLE HANDSHAKE

The early handshake messages (ClientHello, ServerHello, cipher list) were sent *in the clear*. A man-in-the-middle could try to tamper with them — for instance, stripping the strong cipher suites to force a weak one (a **downgrade attack**). The Finished message defeats this: it's a hash of *every* handshake message exchanged. If an attacker altered anything, the two sides' transcripts differ, the Finished hashes won't match, and the connection aborts before any real data flows. The handshake validates its own integrity retroactively.

8. Application Data

Both Finished messages verified, the tunnel is live. All HTTP now flows as encrypted, integrity-protected TLS records using the negotiated symmetric AEAD cipher (Chapter 2). The handshake is over.

Why It's "2-RTT" — and Why That Motivated 1.3

Count the round trips: ClientHello → (server's flight) → ClientKeyExchange/Finished → (server's Finished) → data. The client must wait **two full round trips** before sending application data. On a high-latency mobile link that's a visible delay on every new connection.

HANDSHAKE GOAL	WHICH MESSAGE(S) ACHIEVE IT
`Negotiate version + cipher`	ClientHello / ServerHello
`Authenticate the server`	Certificate + signed ServerKeyExchange
`Establish shared secret`	ServerKeyExchange + ClientKeyExchange (ECDHE)
`Confirm integrity / no tampering`	ChangeCipherSpec + Finished (both sides)

That 2-RTT cost — plus the lingering option of non-forward-secret RSA key transport and a menu of legacy ciphers — is exactly what TLS 1.3 set out to fix. The next chapter shows how it collapses this dance into a single round trip while *removing* the insecure options entirely.

Hands-On Exercises

EXERCISE 1

Run `openssl s_client -connect example.com:443 -tls1_2 -msg` and match the messages it prints (ClientHello, ServerHello, Certificate, ServerKeyExchange, etc.) to the eight steps in this chapter. Note which messages flow before encryption begins.

View solution

EXERCISE 2

Explain, step by step, how the signed ServerKeyExchange stops a man-in-the-middle from substituting their own Diffie–Hellman value. Identify exactly which earlier step the signature depends on, and what the attacker would need to forge it.

View solution

EXERCISE 3

A downgrade attacker tampers with the ClientHello in transit to delete the strong cipher suites, hoping to force a weak one. Trace what happens and identify precisely which handshake message causes the connection to abort, and why the attacker can't cover their tracks.

View solution

CHAPTER 6 QUICK REFERENCE

- The **handshake** negotiates cipher, authenticates the server, establishes the shared key, and confirms integrity — before any HTTP
- **ClientHello** — client's versions, cipher list, client random, SNI (hostname)
- **ServerHello** — server's chosen version + cipher, server random
- **Certificate** — leaf + chain; client verifies to a trusted root & checks SAN vs SNI
- **ServerKeyExchange** — ephemeral DH value *signed* with the cert key (auth + key agreement fused; the MITM defence)
- **ClientKeyExchange** — client's DH value; both derive the pre-master → master secret using the two randoms
- **ChangeCipherSpec + Finished** — switch to encryption; Finished hashes the whole transcript (defeats downgrade/tampering)
- TLS 1.2 needs **2 round trips** before data — the cost that motivated TLS 1.3
- **Next chapter:** TLS 1.3 — 1-RTT handshakes, 0-RTT resumption, and removed legacy crypto

CHAPTER 7 OF 12

TLS 1.3 — What Changed

CHAPTER 7

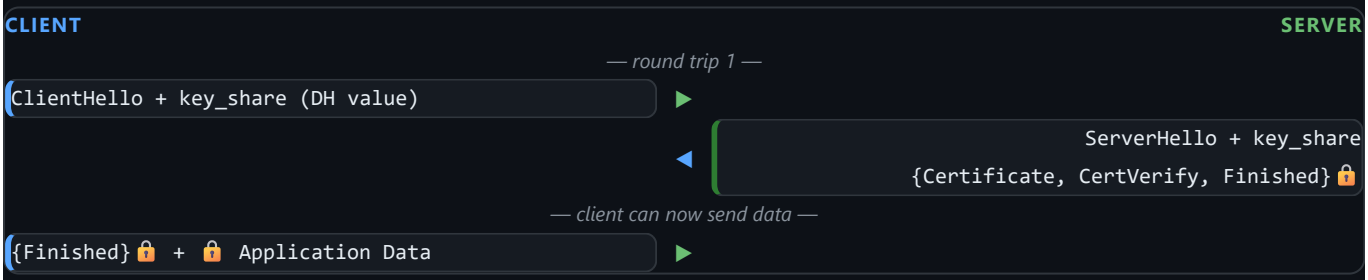
TLS 1.3 — What Changed

A 1-RTT handshake, 0-RTT resumption, and the great purge of legacy crypto

TLS 1.3 (standardized 2018) is the version your browser uses for almost every modern HTTPS connection. It isn't a tweak of 1.2 — it's a redesign driven by two goals: make the handshake **faster** and make it **safe by default** by deleting every legacy option that caused trouble. This chapter contrasts it directly with the 1.2 handshake from Chapter 6.

Change 1: The Handshake Is Now 1-RTT

The big insight: in 1.2, the client waited for the server to choose parameters before it could contribute its key-exchange share — costing a round trip. TLS 1.3 removes the suspense. Since modern key exchange is *always* ephemeral (EC)DHE, the client **guesses** the key-exchange group and sends its DH share *immediately, in the ClientHello*:



By the time the server responds *once*, both sides already share the secret — so the server can send its Certificate, signature, and Finished **already encrypted** in that same flight, and the client can send application data right after its own Finished. One round trip instead of two. (The `{ }` braces mark messages that are now encrypted; note the certificate itself is encrypted in 1.3, unlike 1.2 where it was sent in the clear.)

"GUESSING" THE GROUP COSTS NOTHING IN THE COMMON CASE

The client guesses the most common group (usually X25519) and sends a share for it. Almost always the server supports it and the guess is right → 1-RTT. If the server wants a different group, it replies with a **HelloRetryRequest** asking the client to resend with the right group — falling back to 2-RTT for that one connection. Because the popular groups are near-universal, the retry is rare, so the common case is a clean single round trip.

Change 2: 0-RTT Resumption (with a caveat)

For a server you've connected to *before*, TLS 1.3 can do even better. Using a pre-shared key (a session ticket) from the previous connection, the client can send encrypted application data in its **very first message** — zero round trips of waiting. This "0-RTT early data" makes repeat visits feel instant. But it comes with a sharp security trade-off:

> **0-RTT EARLY DATA IS REPLAYABLE — NOT FOR EVERYTHING**
>
> 0-RTT data is *not* protected against **replay attacks**: an attacker who captures the encrypted early-data packet can resend it, and the server may process it again, because there's no fresh server randomness involved yet. That's harmless for an idempotent `GET /article`, but dangerous for a non-idempotent action like "transfer $100" or "place order," which must never run twice. So 0-RTT is restricted to *safe, idempotent* requests, and forward secrecy doesn't fully apply to that early-data portion. It's an opt-in performance feature with real caveats — use deliberately, not blindly.

Change 3: Legacy Crypto Was Removed, Not Just Discouraged

TLS 1.2's flexibility was also its weakness: it still *permitted* weak options, and many real attacks (downgrade attacks, padding-oracle bugs) exploited them. TLS 1.3's most consequential security move was to **delete the dangerous options entirely**, so they can't be negotiated even by mistake:

REMOVED IN TLS 1.3	WHY IT WAS DANGEROUS
`RSA key transport`	no forward secrecy (Chapter 3's "harvest now, decrypt later")
`Static (non-ephemeral) DH`	also lacked forward secrecy
`CBC-mode & RC4 ciphers`	padding-oracle & keystream-bias attacks (BEAST, Lucky13, etc.)
`Plain (non-AEAD) cipher modes`	encryption without built-in integrity
`MD5 / SHA-1 signatures`	broken / collision-prone hash functions
`Renegotiation, compression`	enabled attacks like CRIME

What's left is a short, curated list: **only AEAD ciphers** (AES-GCM, ChaCha20-Poly1305), **only ephemeral (EC)DHE** key exchange, and modern hashes. There's no weak option to downgrade *to*. This is the deepest lesson of the chapter: **1.3 is safer not because it added cleverness, but because it removed choices**. Forward secrecy went from "available if configured" (1.2) to "mandatory and unavoidable" (1.3).

Change 4: The Cipher Suite Got Simpler

In 1.2, a cipher suite string bundled four choices together: key exchange + authentication + bulk cipher + hash (e.g. `ECDHE-RSA-AES128-GCM-SHA256`). In 1.3, key exchange and authentication are negotiated separately, so a "cipher suite" now names only the bulk AEAD cipher and its hash:

```
# TLS 1.2 suite — bundles everything
ECDHE-RSA-AES128-GCM-SHA256
 \___/ \_/ \_________/ \____/
  kex  auth   cipher    hash

# TLS 1.3 suite — just the AEAD cipher + hash (kex/auth handled separately)
TLS_AES_128_GCM_SHA256
```

This is why TLS 1.3 has only a handful of cipher suites instead of hundreds — Chapter 8 reads these strings in detail.

TLS 1.2 vs 1.3 at a Glance

	TLS 1.2	TLS 1.3
Handshake latency	2-RTT	1-RTT (0-RTT on resume)
Forward secrecy	optional	mandatory (always ephemeral)
RSA key transport	allowed	removed
Cipher modes	AEAD + legacy CBC/RC4	AEAD only
Certificate sent	in the clear	encrypted
Cipher suite count	hundreds	five

In practice you rarely configure any of this — modern servers and browsers negotiate TLS 1.3 automatically and fall back to 1.2 only for older peers. But understanding *what* changed explains why "just use the defaults" is now genuinely safe advice, which it wasn't a decade ago. The next chapter zooms into reading and choosing the cipher suites and protocol versions themselves.

Hands-On Exercises

EXERCISE 1

Connect to the same server twice — once with `-tls1_3` and once with `-tls1_2` using `openssl s_client ... -msg` — and compare the message flows. Identify which messages disappear or move in 1.3, and confirm the certificate is encrypted in 1.3 but not 1.2.

View solution

EXERCISE 2

Explain why TLS 1.3 can achieve 1-RTT when 1.2 needed 2-RTT. Be specific about what the client sends in the ClientHello that it couldn't before, and what assumption makes that possible (and what happens via HelloRetryRequest when the assumption is wrong).

View solution

EXERCISE 3

For each request, decide whether it would be safe to send as 0-RTT early data and justify it: (a) `GET /news/today`; (b) `POST /transfer?amount=500`; (c) `GET /account/balance`; (d) `POST /comments` adding a comment. State the general rule you're applying.

View solution

CHAPTER 7 QUICK REFERENCE

- **TLS 1.3** (2018) — faster *and* safe-by-default; a redesign, not a tweak of 1.2
- **1-RTT handshake** — client sends its (EC)DHE key_share *in the ClientHello*, so the secret exists after one round trip
- **HelloRetryRequest** — fallback to 2-RTT when the client guessed the wrong key-exchange group (rare)
- The **certificate is encrypted** in 1.3 (it was in the clear in 1.2)
- **0-RTT resumption** — repeat visits send early data with zero wait, but it's **replayable** → only for idempotent requests
- **Legacy crypto removed**: RSA key transport, static DH, CBC/RC4, non-AEAD, MD5/SHA-1, renegotiation, compression
- Safer because it **removed choices** — only AEAD ciphers + ephemeral (EC)DHE remain; forward secrecy is mandatory
- 1.3 cipher suites name only **AEAD cipher + hash** (e.g. `TLS_AES_128_GCM_SHA256`) — kex/auth negotiated separately
- **Next chapter:** cipher suites & protocol versions — reading the strings and what to enable/disable

CHAPTER 8 OF 12

Cipher Suites & Protocol Versions

CHAPTER 8

Cipher Suites & Protocol Versions

Decoding the strings, and deciding what to enable or disable

A **cipher suite** is the specific combination of algorithms a TLS connection agrees to use. The intimidating strings you see in `openssl` output or an SSL Labs report are just those algorithm choices spelled out. Once you can read them, configuring a server's security becomes a matter of knowing which to allow — this chapter teaches both.

Anatomy of a TLS 1.2 Cipher Suite

A 1.2 suite name bundles **four** decisions, read left to right:

`ECDHE` - `RSA` - `AES128_GCM` - `SHA256`

KEY EXCHANGE
ECDHE — how the shared secret is agreed. The *E* = ephemeral → forward secrecy (Ch. 3).

AUTHENTICATION
RSA — the algorithm of the certificate's key, used to sign and prove identity (Ch. 4).

BULK CIPHER
AES128-GCM — the symmetric AEAD cipher encrypting the actual data (Ch. 2).

HASH / PRF
SHA256 — the hash used for key derivation and the handshake MAC.

So `ECDHE-RSA-AES128-GCM-SHA256` reads as: "agree the key with ephemeral elliptic-curve Diffie–Hellman, authenticate the server with its RSA certificate, encrypt data with 128-bit AES in GCM mode, and use SHA-256 for hashing." Every piece is a concept from earlier chapters, now named in one line.

READ THE FIRST TWO SEGMENTS TO JUDGE SECURITY AT A GLANCE

The two things to check first are the **key exchange** and the **cipher mode**. Want forward secrecy? The key exchange must be **ECDHE** or **DHE** (has the ephemeral "E") — a suite starting with plain `RSA` (RSA key transport) or `AES…-CBC` instead of GCM is a red flag. If you see `ECDHE` at the front and `GCM` (or `CHACHA20-POLY1305`) for the cipher, it's a modern, safe suite.

TLS 1.3 Suites Are Shorter — On Purpose

As Chapter 7 introduced, TLS 1.3 negotiates key exchange and authentication *separately*, so a 1.3 cipher suite names only the **bulk AEAD cipher + hash**:

```
# the entire list of TLS 1.3 cipher suites — just five
TLS_AES_128_GCM_SHA256
TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_128_CCM_SHA256
TLS_AES_128_CCM_8_SHA256
```

There's no weak choice to make — key exchange is *always* ephemeral (EC)DHE and the cipher is *always* AEAD, so those decisions aren't even in the string. This is why the giant menu of TLS 1.2 suites (hundreds, many insecure) shrinks to five solid options in 1.3. In practice the top three are what you'll see; **ChaCha20-Poly1305** is preferred on devices without AES hardware acceleration (most phones) because it's fast in pure software.

AES-GCM vs ChaCha20-Poly1305

Both are modern AEAD ciphers (confidentiality + integrity in one, Chapter 2) and both are secure. The choice is about performance, not safety:

CIPHER	BEST WHEN	NOTES
`AES-GCM`	CPU has AES hardware (AES-NI) — most desktops/servers	extremely fast with hardware acceleration
`ChaCha20-Poly1305`	no AES hardware — many mobile/low-power devices	fast and constant-time in pure software

Protocol Versions: What to Enable and Disable

Separate from cipher suites is the **protocol version**. The guidance today is simple and worth memorizing:

VERSION	STATUS	ACTION
`SSL 2.0 / 3.0`	badly broken (POODLE)	disable — never enable
`TLS 1.0 / 1.1`	deprecated (2021); weak crypto	disable
`TLS 1.2`	secure if well configured	enable (for compatibility)
`TLS 1.3`	current best	enable (preferred)

The modern baseline is: **support TLS 1.2 and 1.3, disable everything older.** TLS 1.0/1.1 were officially deprecated in 2021 and browsers show warnings for them. You keep 1.2 only because some older clients can't do 1.3 yet; everything below 1.2 is a liability with no upside.

> **ON TLS 1.2, CIPHER ORDER AND SELECTION STILL MATTER — ON 1.3, BARELY**

Because TLS 1.2 still *permits* weaker suites, a misconfigured 1.2 server can negotiate something bad. So on 1.2 you must curate the suite list: prefer ECDHE + GCM/ChaCha20, and explicitly exclude RSA key transport, CBC-mode, RC4, 3DES, and anything "EXPORT" or "NULL". On TLS 1.3 this worry largely evaporates — all five suites are safe, so there's little to misconfigure. This is the practical reward of 1.3's "removed the choices" design (Chapter 7): the surface for getting cipher config wrong is tiny.

Inspecting What a Server Actually Offers

You don't have to guess a server's configuration — you can enumerate it. `nmap`'s ssl-enum-ciphers script is the most readable tool, grading each suite:

```
nmap --script ssl-enum-ciphers -p 443 example.com

| ssl-enum-ciphers:
|   TLSv1.2:
|     ciphers:
|       TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (secp256r1) - A
|       TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (secp256r1) - A
|   TLSv1.3:
|     ciphers:
|       TLS_AKE_WITH_AES_128_GCM_SHA256 - A
|   least strength: A
```

Each suite gets a letter grade; you want all A's with no C/F entries and no TLS 1.0/1.1 sections at all. The same information underlies the SSL Labs report you'll use in Chapter 10. Reading this output is now entirely within reach — every field maps to a concept from the last six chapters.

Hands-On Exercises

EXERCISE 1

Take the suite `ECDHE-ECDSA-AES256-GCM-SHA384` and decode all four parts (key exchange, authentication, bulk cipher, hash). State which guarantee each part contributes, and whether this suite provides forward secrecy.

View solution

EXERCISE 2

Use `openssl ciphers -v 'ECDHE+AESGCM'` to list matching suites, then run `nmap --script ssl-enum-ciphers` against a real site. Identify the protocol versions it supports and whether any weak suite or deprecated version is present.

View solution

EXERCISE 3

Given these four suites, classify each as MODERN-SAFE or AVOID and justify: (a) `TLS_AES_128_GCM_SHA256`; (b) `ECDHE-RSA-AES128-GCM-SHA256`; (c) `AES256-SHA` (i.e. RSA kex, CBC); (d) `ECDHE-RSA-RC4128-SHA`. State the single biggest problem with each "avoid" one.

View solution

CHAPTER 8 QUICK REFERENCE

- A **cipher suite** = the agreed combination of algorithms for a connection
- **TLS 1.2 suite** = *key exchange – authentication – bulk cipher – hash* (e.g. ECDHE-RSA-AES128-GCM-SHA256)
- Check the first two segments: **ECDHE/DHE** = forward secrecy; **GCM/ChaCha20** = modern AEAD
- **TLS 1.3 suite** = just *AEAD cipher + hash* (e.g. TLS_AES_128_GCM_SHA256) — only five exist, all safe
- **AES-GCM** (fast with AES hardware) vs **ChaCha20-Poly1305** (fast in software, mobile) — both secure
- Versions: **enable TLS 1.2 + 1.3**, disable SSL 2/3 and TLS 1.0/1.1 (deprecated 2021)
- On **1.2** curate the suite list (exclude RSA-kex, CBC, RC4, 3DES, EXPORT, NULL); on **1.3** there's little to misconfigure
- Audit with `nmap --script ssl-enum-ciphers` or `openssl ciphers -v` — aim for all A's, no weak entries
- **Next chapter:** getting a certificate — CSRs, domain validation, Let's Encrypt & ACME

CHAPTER 9 OF 12

Getting a Certificate — Let's Encrypt & ACME

CHAPTER 9

Getting a Certificate — Let's Encrypt & ACME

CSRs, domain validation, the ACME protocol, and automatic renewal

The theory is done — now the practical question: how do you actually *obtain* a certificate for a domain you own? This chapter covers the request itself (the CSR), how a CA proves you control the domain (validation), and how **Let's Encrypt** automated the whole thing into a single command via the **ACME** protocol — the reason HTTPS is now free and ubiquitous.

Step Zero: The Key Pair Never Leaves Your Server

Before anything else, one principle that surprises beginners: **you generate your own private key, and it never leaves your server**. The CA never sees it. You only send the CA your *public* key (inside a CSR) plus proof you control the domain; the CA signs a certificate binding that public key to your domain and sends the certificate back. The secret half stays with you the entire time.

The CSR — Certificate Signing Request

A **CSR** is the formal application you send to a CA. It bundles your *public* key with the identity you're requesting (the domain), and is signed by your *private* key to prove you hold it. You can create one with openssl:

```
# generate a private key, then a CSR for example.com
openssl genpkey -algorithm RSA -out example.key

openssl req -new -key example.key -out example.csr \
  -subj "/CN=example.com" \
  -addext "subjectAltName=DNS:example.com,DNS:www.example.com"
```

THE CSR CONTAINS	THE CSR DOES NOT CONTAIN
your public key	`your private key (never!)`
the requested domain(s) — CN & SAN	`the validity dates (the CA sets those)`

THE CSR CONTAINS	THE CSR DOES NOT CONTAIN
a self-signature proving key ownership	**the CA's signature (added on issuance)**

Inspect a CSR with `openssl req -in example.csr -text -noout` — you'll see the same fields as a certificate (Chapter 4) *minus* the issuer and validity, because those are filled in only when a CA signs it.

Domain Validation: Proving You Control the Domain

A CA won't sign a cert for `example.com` until you prove you actually control that domain (Chapter 5's DV level). The proof is a challenge: the CA gives you something to place where only the domain's controller could put it, then checks for it.

CHALLENGE TYPE	YOU PROVE CONTROL BY…	BEST FOR
HTTP-01	serving a specific token file at `http://domain/.well-known/acme-challenge/<token>`	a single host you run a web server on
DNS-01	creating a specific TXT record in the domain's DNS	wildcards (`*.example.com`) & servers not publicly reachable
TLS-ALPN-01	presenting a special cert on a TLS connection	environments where only port 443 is usable

The logic is identical across all three: *only someone who genuinely controls the domain (its web root, its DNS, or its TLS endpoint) could satisfy the challenge.* Wildcard certificates specifically **require DNS-01**, because proving control of one web page doesn't prove control of every possible subdomain — but control of the DNS zone does.

ACME — Automating the Whole Exchange

ACME (Automatic Certificate Management Environment) is the protocol Let's Encrypt introduced to turn that manual back-and-forth into an automated machine-to-machine conversation. A client (like **certbot**) runs the entire flow:

1. **Account & order**
   The ACME client registers a key with the CA and requests a cert for your domain(s).
2. **Receive challenge**
   The CA returns a challenge (e.g. HTTP-01 token) for each domain.
3. **Satisfy challenge**
   The client places the token file (or DNS TXT record) automatically.
4. **CA validates**
   The CA fetches the token / checks DNS to confirm domain control.
5. **Submit CSR & receive cert**

The client sends the CSR; the CA signs and returns the certificate + chain.

In practice this is one command. Certbot can even configure your web server for you:

```
# obtain AND install a cert for nginx, fully automated
sudo certbot --nginx -d example.com -d www.example.com

# or just obtain the cert files, configure the server yourself
sudo certbot certonly --webroot -w /var/www/html -d example.com
```

LET'S ENCRYPT CHANGED THE WEB — FREE, AUTOMATED, UBIQUITOUS

Before Let's Encrypt (2015), certificates cost money and were issued through a slow, manual process, so most small sites stayed on plain HTTP. Let's Encrypt is a free, non-profit CA that issues **only DV certificates**, fully automated via ACME. That combination — zero cost + one command + auto-renewal — is the single biggest reason HTTPS went from "~30% of web traffic" to nearly universal. The short 90-day lifetime (Chapter 4) is only practical *because* ACME makes renewal automatic.

Renewal: Set It and Forget It

Because Let's Encrypt certs last only 90 days, renewal must be automatic — and certbot installs a timer that does it for you. The renewal re-runs the same validation; nothing manual is required as long as the challenge can still be satisfied:

```
# certbot installs a systemd timer / cron job that runs this regularly:
certbot renew --quiet
# it only actually renews certs within ~30 days of expiry; otherwise no-op

# test your renewal works WITHOUT using rate limits:
certbot renew --dry-run
```

TWO REAL-WORLD GOTCHAS: RATE LIMITS AND RENEWAL HOOKS

First, Let's Encrypt enforces **rate limits** (e.g. a cap on certificates per domain per week). While developing, use the **staging environment** (`--test-cert` / `--staging`) so failed experiments don't burn your weekly quota — staging certs aren't publicly trusted but exercise the full flow. Second, after a renewal your server is often still using the *old* cert in memory until it reloads — so a renewal **deploy hook** (e.g. `--deploy-hook "systemctl reload nginx"`) is needed to pick up the new cert. A silently-not-reloaded server is a classic "why did my cert expire when certbot said it renewed?" bug.

When You'd Use a Commercial CA Instead

Let's Encrypt covers the overwhelming majority of needs, but not all. You'd pay a commercial CA when you need **OV/EV validation** (vetted organization identity, Chapter 5), longer support contracts, warranties, or certificate types Let's Encrypt doesn't issue. For ordinary "encrypt my website" purposes, though, free DV via ACME is the default choice — and the rest of this course assumes it.

Hands-On Exercises

EXERCISE 1

Generate a private key and a CSR for a domain with both a CN and a SAN list, then inspect the CSR with `openssl req -text -noout`. Confirm it contains your public key and requested domains but NOT a private key, issuer, or validity dates — and explain why those three are absent.

View solution

EXERCISE 2

Explain how the HTTP-01 and DNS-01 challenges each prove domain control, then state which one you MUST use to obtain a wildcard certificate for `*.example.com` and exactly why the other one can't work for a wildcard.

View solution

EXERCISE 3

A site's certificate expired even though the admin "set up certbot." List the most likely causes (renewal timer not running, server never reloaded the renewed cert, validation challenge now failing) and describe how `certbot renew --dry-run` and a deploy hook would have prevented it.

View solution

CHAPTER 9 QUICK REFERENCE

- You generate your **private key**; it *never leaves your server* — the CA only ever sees your public key
- **CSR** = public key + requested domains (CN/SAN), self-signed to prove key ownership; no issuer/validity yet
- **Domain validation (DV)** proves control via a challenge: **HTTP-01** (token file), **DNS-01** (TXT record), or **TLS-ALPN-01**
- **Wildcards require DNS-01** — only DNS-zone control proves authority over every subdomain
- **ACME** automates the order → challenge → validate → CSR → issue flow; **certbot** is the common client
- **Let's Encrypt** — free, automated, DV-only CA; the reason HTTPS became ubiquitous
- 90-day certs ⇒ **auto-renewal** (certbot timer); test with `certbot renew --dry-run`
- Gotchas: **rate limits** → use **staging** while testing; reload the server via a **deploy hook** after renewal

- **Next chapter:** configuring HTTPS on a server — nginx/Apache, redirects, HSTS, OCSP stapling, SSL Labs

CHAPTER 10 OF 12

Configuring HTTPS on a Server

CHAPTER 10

Configuring HTTPS on a Server

nginx/Apache TLS config, redirects, HSTS, OCSP stapling, and grading with SSL Labs

You have a certificate (Chapter 9). Now you wire it into a web server *correctly* — not just "it works," but configured so the grading tools give you an A. This chapter is the practical payoff: the handful of directives that matter, the mistakes that cost a grade, and how to verify the result.

The Minimum: Certificate, Key, and Protocols

A basic TLS server block needs three things: the certificate chain, the private key, and which protocols/ciphers to allow. Here's an nginx example carrying everything from the last two chapters:

```
server {
    listen 443 ssl;
    server_name example.com www.example.com;

    # full chain = leaf + intermediates (Chapter 5!), NOT just the leaf
    ssl_certificate     /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;

    # only modern protocols (Chapter 8)
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers off;  # let 1.3 pick; fine for modern suites
}
```

USE FULLCHAIN.PEM, NOT CERT.PEM — THE MISSING-INTERMEDIATE TRAP

Certbot writes several files; the two that matter are `fullchain.pem` (leaf *plus* intermediates) and `privkey.pem`. Point `ssl_certificate` at **fullchain.pem**. If you accidentally use `cert.pem` (the leaf alone), you recreate exactly the Chapter 5 "missing intermediate" bug: it works in some browsers (which cache the intermediate) and fails on other clients with "unable to get local issuer certificate." This is the single most common server-config TLS mistake — get this one line right.

Redirect HTTP → HTTPS

Serving HTTPS isn't enough if visitors can still reach plain HTTP. Add a second server block on port 80 that redirects everything to HTTPS, so no one accidentally uses the insecure version:

```
server {
    listen 80;
    server_name example.com www.example.com;
    return 301 https://$host$request_uri;    # permanent redirect to HTTPS
}
```

A `301` (permanent) redirect tells browsers and search engines the HTTPS URL is canonical. But a redirect alone has a subtle gap — which is exactly what HSTS closes.

HSTS — Don't Even Try HTTP Next Time

A plain redirect still involves *one* insecure HTTP request first (the one that gets redirected) — and an active attacker could intercept that initial request before the redirect happens (an "SSL stripping" attack, Chapter 11). **HSTS** (HTTP Strict Transport Security) fixes this with a response header that tells the browser: *for the next N seconds, never use HTTP for this domain at all — go straight to HTTPS yourself.*

```
# sent over HTTPS; browser remembers it and auto-upgrades future requests
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
```

After the first secure visit, the browser refuses to make *any* plaintext request to the domain for a year (`max-age=31536000`), eliminating the vulnerable first hop on every subsequent visit. `includeSubDomains` extends the rule to all subdomains.

> **HSTS IS A COMMITMENT — TEST BEFORE YOU SET A LONG MAX-AGE**
>
> Once a browser has seen the HSTS header, it will **refuse** to connect over HTTP (or to a cert it can't validate) for the full `max-age` — there's no clickthrough. If your HTTPS later breaks (expired cert, misconfiguration), users are *locked out*, not merely warned, until you fix it or the max-age elapses. So roll it out gradually: start with a short max-age (e.g. `300`), confirm everything's solid, then raise it to a year. The **preload list** (hardcoded into browsers) is even more permanent — only submit once you're certain.

OCSP Stapling — Faster, More Private Revocation Checks

Browsers want to know a cert hasn't been *revoked* (Chapter 11). The old way: the browser separately contacts the CA's OCSP responder — which is slow and leaks to the CA which sites you visit. **OCSP stapling** flips this: the *server* periodically fetches a fresh, CA-signed "this cert is still valid" proof and **staples** it to the TLS handshake, so the browser gets it instantly without contacting the CA.

```
ssl_stapling on;
ssl_stapling_verify on;
ssl_trusted_certificate /etc/letsencrypt/live/example.com/chain.pem;
```

It's a win on every axis: faster handshakes (no extra round trip to the CA), better privacy (the CA doesn't see each visitor), and less load on the CA. It's a standard part of a well-tuned config.

The Apache Equivalent

PURPOSE	NGINX	APACHE
Cert (full chain)	ssl_certificate fullchain.pem	SSLCertificateFile fullchain.pem
Private key	ssl_certificate_key privkey.pem	SSLCertificateKeyFile privkey.pem
Protocols	ssl_protocols TLSv1.2 TLSv1.3	SSLProtocol -all +TLSv1.2 +TLSv1.3
HSTS	add_header Strict-Transport-Security ...	Header always set Strict-Transport-Security ...
OCSP stapling	ssl_stapling on;	SSLUseStapling on

The concepts are identical — only the directive names differ. Certbot's `--nginx` / `--apache` plugins generate much of this for you, but knowing what each line does is what lets you debug and harden it.

Verify: Grade It with SSL Labs

Don't guess whether your config is good — measure it. **Qualys SSL Labs** (ssllabs.com/ssltest) runs a deep audit and assigns a letter grade, checking everything from this course: protocol versions, cipher suites, chain completeness, HSTS, key strength, and known vulnerabilities.

WHAT AN A/A+ REQUIRES — A CHECKLIST OF THIS WHOLE COURSE

To score **A+**: serve the *full chain* (Ch. 5), support only **TLS 1.2 + 1.3** with strong AEAD/ECDHE suites (Ch. 8), use a **2048-bit+ RSA or 256-bit EC key**, enable **HSTS** with a long max-age, and have no known-vulnerable settings. The A+ specifically rewards HSTS. Notice the grade is essentially a checklist of everything you've learned — SSL Labs is a great way to see the theory reflected in a real score. (For automation/CI, `testssl.sh` gives similar results from the command line.)

Hands-On Exercises

EXERCISE 1

Write a complete nginx configuration for example.com that: serves the full chain + key, supports only TLS 1.2/1.3, redirects HTTP→HTTPS with a 301, and sets a one-year HSTS header with includeSubDomains. Annotate which line prevents the missing-intermediate bug and why.

View solution

EXERCISE 2

Explain the gap that a plain HTTP→HTTPS redirect leaves open, then explain precisely how HSTS closes it. Include why HSTS only protects *after* the first secure visit, and what the preload list does about that residual first-visit gap.

View solution

EXERCISE 3

Describe what OCSP stapling is and the three problems it solves versus a browser doing its own OCSP lookup. Then list what an SSL Labs A+ grade requires, mapping each requirement back to the chapter that introduced it.

View solution

CHAPTER 10 QUICK REFERENCE

- Minimum config: **ssl_certificate** (full chain) + **ssl_certificate_key** + **ssl_protocols TLSv1.2 TLSv1.3**
- Use **fullchain.pem**, never `cert.pem` — the leaf-only file recreates the missing-intermediate bug
- **Redirect HTTP→HTTPS** with a `301` on a port-80 server block
- **HSTS** (`Strict-Transport-Security`) — browser auto-upgrades to HTTPS, closing the redirect's insecure first hop (SSL-stripping defence)
- HSTS is a **commitment**: long max-age locks users out if HTTPS breaks — start short, then raise; preload is near-permanent
- **OCSP stapling** — server staples a fresh CA-signed validity proof: faster handshake, better privacy, less CA load
- Apache uses the same concepts with `SSLCertificateFile` , `SSLProtocol` , `SSLUseStapling` , etc.
- Verify with **SSL Labs** (or `testssl.sh`) — an A+ is a checklist of this whole course
- **Next chapter:** common problems & attacks — expired/mismatched certs, mixed content, downgrade & SSL-stripping, revocation

CHAPTER 11 OF 12

Common Problems & Attacks

CHAPTER 11

Common Problems & Attacks

Certificate errors, mixed content, downgrade/stripping, revocation, and pinning

You now understand how TLS is *supposed* to work. This chapter is the troubleshooting and threat catalogue: the certificate errors you'll actually hit, the attacks TLS defends against (and how), and the failure modes that still bite well-meaning admins. Each one connects back to a guarantee or mechanism from earlier chapters.

Certificate Errors — What the Browser Is Really Telling You

Most "your connection is not private" warnings come from one of a few failed checks — all from Chapters 4–5. Knowing which check failed tells you exactly what to fix:

ERROR	FAILED CHECK	USUAL CAUSE / FIX
`CERT_DATE_INVALID`	validity window (Ch. 4)	cert expired — renew it (auto-renewal, Ch. 9)
`COMMON_NAME_INVALID`	SAN ≠ hostname (Ch. 4)	cert doesn't list the domain visited — reissue with correct SAN
`AUTHORITY_INVALID`	chain to trusted root (Ch. 5)	self-signed, or missing intermediate — install full chain
`unable to get local issuer`	incomplete chain (Ch. 5)	server sent leaf only — use fullchain.pem (Ch. 10)

Mixed Content — HTTPS Page, HTTP Resources

A page served over HTTPS that pulls in *sub-resources* over plain HTTP is **mixed content**. It quietly defeats the page's security: those HTTP requests are unencrypted and tamperable, so the "secure" page isn't actually secure end-to-end.

```
<!-- page loaded over https://example.com but... -->
<script src="http://example.com/app.js"></script>   # insecure! tamperable
<script src="https://example.com/app.js"></script>  # or just //example.com/app.js
```

Browsers split this into two severities: **active** mixed content (scripts, stylesheets, iframes — things that can alter the page) is *blocked outright*, while **passive** mixed content (images, media) triggers a downgraded padlock warning. The fix is to load every resource over HTTPS (use `https://` or protocol-relative URLs), and a `Content-Security-Policy: upgrade-insecure-requests` header can auto-rewrite stragglers.

SSL Stripping & Downgrade Attacks

Two related man-in-the-middle attacks, both already half-covered in earlier chapters — here's how they're defeated:

- **SSL stripping** — the attacker intercepts the victim's initial *plaintext* request and keeps them on HTTP, proxying to the real HTTPS site so the user never sees a warning. **Defeated by HSTS** (Chapter 10): the browser refuses HTTP for the domain, so there's no plaintext request to strip. Preload closes even the first-visit gap.
- **Protocol/cipher downgrade** — the attacker tampers with the cleartext ClientHello to force a weak protocol or cipher. **Defeated by the Finished hash** (Chapter 6), which covers the whole transcript, plus TLS 1.3 simply *removing* the weak options to downgrade to (Chapter 7).

> **NOTICE THE PATTERN: EVERY ATTACK MAPS TO A DEFENCE YOU'VE ALREADY LEARNED**
>
> This is the satisfying part of reaching Chapter 11 — you're not learning new magic, you're seeing the mechanisms pay off. Stripping → HSTS. Downgrade → Finished + 1.3's removed options. Eavesdropping → encryption. Tampering → AEAD/MAC. Impersonation → certificates + chain. Forward-secrecy harvesting → ephemeral DH. TLS is a layered system where each defence neutralizes a specific attack class.

Implementation Bugs: Heartbleed and Friends

The protocol can be sound while the *code* implementing it is flawed. The famous example is **Heartbleed** (2014): a buffer over-read bug in OpenSSL's TLS heartbeat extension let an attacker read up to 64 KB of server memory per request — potentially leaking private keys, session cookies, and passwords, with no trace in logs.

> **HEARTBLEED'S DEEPER LESSON: A LEAKED KEY MEANS REVOKE AND ROTATE**
>
> Heartbleed mattered so much because it could expose the server's **private key** — and remember from Chapter 3 that without forward secrecy, a stolen key decrypts past *recorded* traffic too. The full remediation was a chain: patch OpenSSL → **generate a new key** → reissue the certificate → **revoke the old certificate** → and force-reset potentially exposed user credentials. Patching alone was not enough; any key that *might* have leaked must be treated as compromised. This is also a strong argument for forward secrecy (ephemeral DH), which limits how much a future key leak can retroactively expose.

Revocation: Cancelling a Certificate Before It Expires

When a key is compromised (as in Heartbleed) or a cert is mis-issued, you need to invalidate it *before* its natural expiry. That's **revocation**, and it has historically been the weakest link in the PKI:

MECHANISM	HOW IT WORKS	WEAKNESS
`CRL`	CA publishes a big list of revoked serial numbers	large, slow to download, cached stale
`OCSP`	browser asks the CA about one cert in real time	slow, privacy-leaking, often "soft-fail"
`OCSP stapling`	server staples a fresh CA-signed status (Ch. 10)	needs server support

THE DIRTY SECRET: REVOCATION CHECKING OFTEN "SOFT-FAILS"

If a browser can't reach the CA to check revocation (responder down, network blocked), most browsers **proceed anyway** rather than block the page — otherwise a CA outage would break the web. But that means an attacker who can block the OCSP request can also suppress the revocation check. This unreliability is the real reason the industry shifted to **short-lived certificates** (Chapter 4's 90-day certs): if a cert is only valid for weeks, the damage window from a compromise is small even if revocation never reaches the client. Short lifetimes are revocation that *actually works*.

Certificate Pinning — Powerful and Dangerous

Pinning hard-codes which specific certificate or public key an app expects, so even a validly-issued cert from a *different* (possibly fraudulent) CA is rejected. It defends against the Chapter 5 risk of a misbehaving CA. But it's a sharp tool:

PINNING CAN BRICK YOUR OWN SITE — IT'S LARGELY DEPRECATED FOR THE WEB

If you pin a key and then have to rotate it (or your CA changes), every client still holding the old pin **refuses to connect** — you can lock out your entire user base with no easy recovery. Browser-based *HTTP Public Key Pinning (HPKP)* was so footgun-prone that browsers **removed it**. Pinning survives mainly in **mobile apps** (where you control both client and server and can ship updates) and high-security contexts. For the general web, Certificate Transparency (Chapter 12) is the preferred, safer answer to the rogue-CA problem.

Hands-On Exercises

EXERCISE 1

Use badssl.com (or openssl against it) to trigger several deliberate certificate errors — expired, wrong-host, self-signed, untrusted-root. For each, name the exact check that failed and which chapter's concept it maps to.

View solution

EXERCISE 2

For each attack — (a) SSL stripping, (b) protocol downgrade, (c) passive eavesdropping, (d) a stolen server key used to decrypt old recorded traffic — name the specific TLS mechanism that defends against it and the chapter that introduced that defence.

View solution

EXERCISE 3

A server running a vulnerable OpenSSL version is found to be exposed to Heartbleed. Write the full remediation checklist in the correct order, and explain why simply patching OpenSSL is insufficient. Then explain why short-lived certificates reduce reliance on revocation.

View solution

CHAPTER 11 QUICK REFERENCE

- **Cert errors** map to failed checks: DATE→validity, COMMON_NAME→SAN, AUTHORITY/local-issuer→chain (Ch. 4–5)
- **Mixed content** — HTTP sub-resources on an HTTPS page; active (scripts) blocked, passive (images) warned; load all over HTTPS
- **SSL stripping** → defeated by **HSTS** (Ch. 10); **downgrade** → defeated by the **Finished hash** + TLS 1.3 removing weak options
- **Heartbleed** — an *implementation* bug (OpenSSL) leaking memory incl. private keys; protocol-sound ≠ code-safe
- Key compromise = **patch** → **new key** → **reissue** → **revoke old** → **reset credentials**; patching alone is not enough
- **Revocation**: CRL / OCSP / OCSP stapling — but checks often **soft-fail**, so **short-lived certs** are the real mitigation
- **Pinning** defends against rogue CAs but can brick your site; browser HPKP was removed — survives mainly in mobile apps
- Every attack maps to a defence already learned — TLS is layered, each mechanism neutralizing one attack class
- **Next chapter:** beyond the basics — mTLS, Certificate Transparency, and the modern PKI ecosystem

CHAPTER 12 OF 12

Beyond the Basics

CHAPTER 12

Beyond the Basics

mTLS, Certificate Transparency, and the modern PKI ecosystem

You've covered the full path from "why HTTPS" to running a hardened server. This final chapter surveys the frontier — the mechanisms that secure machine-to-machine systems, keep the CA ecosystem honest, and define where TLS is heading — so you know what exists and where to go next.

Mutual TLS (mTLS) — Both Sides Present Certificates

In ordinary HTTPS, only the *server* proves its identity; the client stays anonymous (it logs in separately with a password). **Mutual TLS** adds the symmetric half: the **client also presents a certificate**, and the server verifies it the same way the client verified the server. Both ends are cryptographically authenticated before any data flows.

	NORMAL TLS	MUTUAL TLS (MTLS)
`Server authenticated?`	yes (certificate)	yes (certificate)
`Client authenticated?`	no (anonymous)	yes (client certificate)
`Typical use`	the public web	service-to-service, APIs, zero-trust networks

mTLS isn't used for the public web (you can't issue a cert to every visitor), but it's foundational for **internal systems**: microservices authenticating each other, API clients, VPNs, and "zero-trust" architectures where every connection — even inside the network — must prove identity. Service meshes like Istio automate mTLS between every pod. It's the same handshake you learned in Chapter 6, with one extra step: the server requests and verifies a client certificate.

Certificate Transparency — Keeping CAs Honest

Chapters 5 and 11 raised the deep risk: a trusted CA could *mis-issue* a certificate for a domain — say, an attacker (or a compromised/coerced CA) obtaining a valid cert for `yourbank.com` they don't own. Chain verification wouldn't catch it; the cert is genuinely trusted. **Certificate Transparency (CT)** is the ecosystem's answer.

CT requires every issued certificate to be published to public, append-only, cryptographically-verifiable **logs**. Browsers refuse certificates that aren't accompanied by proof of CT logging. The effect:

- **Nothing issues in secret** — every cert for your domain becomes publicly visible, whoever requested it.
- **Domain owners can monitor** — you can watch the logs (or use a service) and get alerted if *any* cert is issued for your domain that you didn't request.
- **Mis-issuance gets caught** — several CAs have been distrusted after CT logs exposed bad behaviour.

> **CT IS DETECTION, NOT PREVENTION — AND THAT'S THE POINT**
>
> CT doesn't *stop* a CA from mis-issuing a cert; it makes mis-issuance **impossible to hide**. That shift from prevention to *guaranteed detection* is what makes it powerful: a CA that knows every cert is publicly logged and monitored has overwhelming incentive not to misbehave, and the community can react fast when one does. You can search the logs yourself right now at `crt.sh` — type any domain and see every certificate ever issued for it. It's the safer successor to certificate pinning (Chapter 11) for the rogue-CA problem.

The CA/Browser Ecosystem

The rules that hold all this together aren't ad hoc. The **CA/Browser Forum** — browser vendors and CAs together — sets the *Baseline Requirements* every public CA must follow: validation standards, maximum certificate lifetimes, mandatory CT logging, key-strength minimums. Browser **root programs** (Mozilla, Apple, Microsoft, Chrome) decide which CAs are trusted and can **distrust** a CA that breaks the rules — a commercial death sentence that keeps CAs disciplined. The trust store you inspected in Chapter 5 is the output of these programs.

Where TLS Is Heading

TREND	WHAT IT DOES
`Encrypted Client Hello (ECH)`	encrypts the SNI hostname in the handshake — closing the last big metadata leak (Chapter 1's "domain is still visible")
`Ever-shorter cert lifetimes`	industry moving toward ~47-day (and shorter) certs — making revocation matter even less (Chapter 11)
`Post-quantum cryptography`	new key-exchange/signature algorithms resistant to quantum attacks; hybrid PQ key exchange already shipping in browsers
`Automation everywhere`	ACME for internal CAs, shorter renewals, less manual TLS handling overall

> **THE QUANTUM THREAT IS WHY "HARVEST NOW, DECRYPT LATER" STILL MATTERS**
>
> Recall Chapter 3's attack: record encrypted traffic today, decrypt it later. A sufficiently powerful quantum computer could one day break the (EC)DHE key exchange protecting today's sessions. That's why **post-quantum key exchange**

is being deployed now, before such computers exist — traffic recorded today needs to resist tomorrow's attacks. Forward secrecy helps but isn't enough against a quantum break of the key exchange itself, so browsers and servers are already rolling out hybrid classical+post-quantum key exchange. It's the clearest example of TLS evolving ahead of the threat.

The Whole Course in One Mental Model

Step back and the entire course collapses into a single sentence you can now unpack completely: *HTTPS uses asymmetric cryptography and a CA-issued, transparency-logged certificate to authenticate the server and agree — with forward secrecy — on a symmetric key, which then protects every byte with confidentiality and integrity.* Every term in that sentence is a chapter you've worked through.

Where to Go Next

- **Read a real handshake in Wireshark** — capture a TLS 1.3 connection and watch the messages from Chapters 6–7 on the wire.
- **Run your own internal CA** with `step-ca` or `cfssl`, and try issuing client certs for mTLS.
- **The standards themselves** — RFC 8446 (TLS 1.3) is surprisingly readable now that you have the concepts.
- **Get an A+ on a real domain** — apply Chapters 9–10 end to end and verify with SSL Labs.
- **Explore crt.sh** for your own domains and set up CT monitoring alerts.

Hands-On Exercises

EXERCISE 1

Explain how mTLS differs from normal TLS in terms of who authenticates whom, and give two concrete scenarios where mTLS is appropriate and one where it is not. Tie the mechanism back to the Chapter 6 handshake — what one extra step does mTLS add?

View solution

EXERCISE 2

Search `crt.sh` for a domain you control (or a well-known one) and examine the issued certificates. Explain what problem Certificate Transparency solves, why it's described as "detection not prevention," and how it improves on certificate pinning for the rogue-CA problem.

View solution

EXERCISE 3

Explain why post-quantum cryptography is being deployed *now* rather than waiting for quantum computers to exist, explicitly connecting it to the "harvest now, decrypt later" attack from Chapter 3. Then write the one-sentence summary of the whole course and label which chapter each key term came from.

View solution

CHAPTER 12 QUICK REFERENCE

- **mTLS** — both client AND server present certificates; for service-to-service, APIs, zero-trust (not the public web)
- mTLS = the Chapter 6 handshake + one extra step: server requests & verifies a client certificate
- **Certificate Transparency (CT)** — every cert published to public append-only logs; browsers require it
- CT is **detection, not prevention** — mis-issuance can't hide; monitor your domains via `crt.sh`
- CT is the safer successor to **pinning** for the rogue-CA problem (Chapter 11)
- **CA/Browser Forum** + browser **root programs** set the rules and can distrust misbehaving CAs
- Frontier: **ECH** (encrypts SNI), **ever-shorter certs**, **post-quantum** key exchange (vs harvest-now-decrypt-later)
- Next steps: Wireshark a handshake, run an internal CA + mTLS, read RFC 8446, get an A+ on a real site

★ HTTPS / TLS Fundamentals Complete — 12 / 12 chapters

From plaintext HTTP's three weaknesses, through the cryptographic primitives, key exchange, certificates and the chain of trust, both handshakes, cipher suites, obtaining and configuring certificates, the attack catalogue, and the modern PKI frontier. You can now read a TLS handshake, configure HTTPS to an A+, diagnose certificate errors, and reason about the guarantees — confidentiality, integrity, authentication — that every secure connection rests on.