TLS and Certificates Course

Flavio Copes

Updated August 3, 2026

Contents

Inspect TLS connections, create a local certificate authority, run HTTPS and mutual TLS labs, and diagnose certificate failures with evidence.

What you'll learn: Create and protect keys, issue local certificates, serve verified HTTPS and mutual TLS, and identify the exact cause of a failed handshake.

Inspect TLS

Inspect live handshakes, certificates, names, chains, and negotiated connection properties.

Connect with openssl s_client

Open a TLS connection, send the intended server name, and separate handshake output from application data.

The `s_client` command is a diagnostic TLS client. It opens a real TLS connection to a server and prints the raw facts about the handshake: which certificates the server sent, which protocol version and cipher were negotiated, and whether verification passed.

You reach for it when a browser or application reports a vague TLS error and you need to see what actually happened on the wire.

Inspect a public HTTPS endpoint:

```
openssl s_client -connect example.org:443 -servername example.org </dev/null
```

Two options matter here. `-connect` gives the TCP target. `-servername` sends **SNI** (Server Name Indication), the hostname the client requests during the handshake. Many servers host several sites on one IP address and use SNI to select the matching certificate. For name-based servers, send SNI with `-servername` so the server selects the intended certificate. Skip it and you may be handed a default certificate that has nothing to do with the site you wanted to test.

The `</dev/null` redirect closes standard input right away, so the command exits after the handshake instead of sitting there waiting for you to type application data.

What to read in the output

The output is long. Scan for these lines:

```
Certificate chain
 0 s:CN = example.org
   i:C = US, O = DigiCert Inc, CN = DigiCert Global G3 TLS ECC SHA384 2020 CA1
...
Protocol  : TLSv1.3
Cipher    : TLS_AES_256_GCM_SHA384
...
Verify return code: 0 (ok)
```

The `Certificate chain` block lists what the server presented, with `s:` for subject and `i:` for issuer. `Protocol` and `Cipher` show what was negotiated. `Verify return code: 0 (ok)` means the chain validated against your local trust store. Any other code names a specific failure: 10 means an expired certificate, 20 means OpenSSL could not find the issuer.

A `Verify return code` of 0 proves a TLS conversation, not application health. The service behind it can still be down or broken, so treat this as a transport-layer check only.

One habit to build now: `s_client` output ends up in tickets and chat logs. Do not paste private keys or real client credentials into diagnostic commands or shared output.

Read a certificate

Decode a PEM certificate and inspect its subject, issuer, validity, names, key, and extensions.

A certificate binds names or identities to a public key under an issuer signature. Everything a client checks during verification — who this certificate is for, who vouched for it, when it stops being valid — lives in fields you can decode and read yourself.

Save a lab certificate as `server.crt`, then inspect it:

```
openssl x509 -in server.crt -noout -subject -issuer -dates -ext subjectAltName
```

`-noout` suppresses the encoded certificate itself, so you only get the decoded fields you asked for:

```
subject=CN = app.lab.test
issuer=CN = Practical TLS Lab CA
notBefore=Aug  1 09:12:00 2026 GMT
notAfter=Aug 15 09:12:00 2026 GMT
X509v3 Subject Alternative Name:
    DNS:app.lab.test, DNS:localhost
```

Read them in order. `subject` is the identity the certificate claims. `issuer` is the authority that signed it. `notBefore` and `notAfter` bound the validity window, always in GMT. The last block lists the names this certificate covers.

Check the **Subject Alternative Name** extension for hostnames. Modern clients validate names there rather than relying on the common name alone. If the hostname you connect to is not in the SAN list, verification fails no matter what the subject line says.

When you need the complete picture, dump the full decode:

```
openssl x509 -in server.crt -noout -text
```

That adds the public key type and size, the signature algorithm, and extensions like Key Usage and Basic Constraints. It is more than you usually need, but it is the view that settles arguments about what a certificate does or does not contain.

A mistake I see often: inspecting the wrong file. Servers ship with leaf certificates, intermediates, and bundles that all look identical from the outside. If the `subject` shows a CA name instead of your hostname, you decoded a CA certificate, not the leaf. Check the file name and the SAN before drawing conclusions.

A certificate is public. You can commit it, mail it, or paste it anywhere. The matching private key is secret. Keep the two files and their permissions distinct.

Understand the certificate chain

Follow a leaf certificate through intermediate authorities to a locally trusted root.

A single certificate rarely stands alone. The certificate for a website is signed by an **intermediate CA**, and that intermediate is signed by a **root CA** that your operating system or browser already trusts. Verification means walking that path: leaf, through intermediates, up to a locally trusted root.

The split exists for damage control. Root CA keys are kept offline and used rarely. Day-to-day signing happens with intermediate keys, which can be replaced if something goes wrong without every device on earth updating its trust store.

Servers normally send the leaf certificate and required intermediate certificates. Clients already hold trusted roots and build a verified path.

Show every certificate returned by a server:

```
openssl s_client -connect example.org:443 -servername example.org -showcerts </dev/null
```

`-showcerts` prints each certificate the server sent as a PEM block, in the order presented. Above each block you get the subject and issuer:

```
 0 s:CN = example.org
   i:C = US, O = DigiCert Inc, CN = DigiCert Global G3 TLS ECC SHA384 2020 CA1
 1 s:C = US, O = DigiCert Inc, CN = DigiCert Global G3 TLS ECC SHA384 2020 CA1
   i:C = US, O = DigiCert Inc, CN = DigiCert Global Root G3
```

Certificate `0` is the **leaf**, the one for the site itself. Certificate `1` is an intermediate: notice its `s:` subject matches the leaf's `i:` issuer exactly. That subject-to-issuer linkage is the chain. The intermediate's own issuer points at a root.

Count the presented certificates and identify leaf versus intermediate. The trusted root may be absent because clients already store it — servers are not expected to send it, and clients ignore it if they do.

This structure explains a whole class of production failures. A server configured with only the leaf works in browsers that cached the intermediate elsewhere, then fails in curl, mobile apps, or CI with an ”unable to get local issuer certificate” error. You will reproduce and fix that failure later in this course.

A chain is not trusted because certificates merely exist. Signatures, constraints, names, time, and local trust must all validate. Any single broken link fails the whole path.

Record the negotiated connection

Capture TLS version, cipher, key exchange, peer identity, ALPN, and verification result as evidence.

A TLS policy becomes real only in the negotiated connection. You can configure a server for TLS 1.3 and strong ciphers, but what a given client actually gets depends on both sides' capabilities, their configuration, and any network intermediaries in between. When someone asks ”is this connection secure?”, the honest answer is a recording of one specific handshake.

The full `s_client` output is too noisy for note-taking. Request a concise summary instead:

```
openssl s_client -brief -connect example.org:443 -servername example.org </dev/null
```

`-brief` reduces the output to the facts worth recording:

```
CONNECTION ESTABLISHED
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_256_GCM_SHA384
Peer certificate: CN = example.org
Hash used: SHA256
Signature type: ECDSA
Verification: OK
Server Temp Key: X25519, 253 bits
```

Record the protocol, cipher suite, peer certificate, and verification status. `Verification: OK` is the line that tells you the chain validated; anything else names the failure.

If the service speaks HTTP/2, you can also check which application protocol gets negotiated by

offering choices with **ALPN**:

```
openssl s_client -brief -alpn h2,http/1.1 -connect example.org:443 -servername example.org </
```

An `ALPN protocol: h2` line in the output means the server picked HTTP/2 during the handshake.

Now compare. Run the same command against a second service, or force an older protocol with `-tls1_2`, and put the two summaries side by side. Differences in protocol version or key exchange jump out immediately, and you have concrete evidence for a ticket instead of "it seems fine".

This habit pays off during incidents. A recorded baseline from last month tells you whether today's `Protocol version: TLSv1.2` is normal for that service or a regression someone introduced.

Do not judge security from the cipher name alone. A strong-sounding cipher suite over a connection whose certificate failed verification protects you from nothing. Identity verification and current protocol policy matter too, which is why the summary records all of them together.

Check your understanding: inspect tls

Check the commands, decisions, safety boundaries, and failure cases from inspect tls.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build a local certificate authority

Create protected keys, a local CA, CSRs, and signed leaf certificates for a controlled lab.

Generate a private key

Create a modern private key, inspect only its public properties, and protect the secret material.

Everything in this module builds on one secret: a private key. The certificate, the signatures, the trust decisions — all of them assume that only you control this file. A private key proves control of an identity. Generate it on the system where it will be protected and avoid copying it through chat, logs, or source control.

We start with the key for our lab certificate authority. Create an encrypted lab CA key:

```
openssl genpkey -algorithm RSA -aes-256-cbc -out lab-ca.key -pkeyopt rsa_keygen_bits:3072
```

OpenSSL prompts for a passphrase and writes the key encrypted with AES-256. `-pkeyopt rsa_keygen_bits:3072` picks a 3072-bit RSA key, a solid choice for a CA key that will sign other certificates. The passphrase means a stolen file is not immediately a stolen identity.

Confirm the file exists and restrict it to its owner:

```
chmod 600 lab-ca.key
ls -l lab-ca.key
# -rw------- 1 flavio flavio 2.6K Aug  3 10:20 lab-ca.key
```

The `-rw-------` permissions mean only the owning user can read or write the file. On a shared machine, that line is the difference between a secret and a leak.

Use `openssl pkey -in lab-ca.key -check -noout` to validate the structure:

```
openssl pkey -in lab-ca.key -check -noout
# Key is valid
```

You will be asked for the passphrase, then OpenSSL checks the key's internal consistency. `Key is valid` is the output you want. This command never prints the key material itself, which is exactly how you should inspect keys: look at properties, not contents.

A realistic failure: you generate the key as root during setup, then the server process running as another user cannot read it. The symptom is a permission-denied error at startup, not a TLS error. Check ownership with `ls -l` before suspecting the key itself.

One note on the passphrase. For unattended server keys, encryption requires a deliberate secure unlock mechanism — someone or something must supply the passphrase at every restart. Never remove protection casually just to make automation easier. Decide where the secret lives first, then choose whether the key file is encrypted.

Create a local CA certificate

Create a self-signed root certificate for an isolated lab and understand what makes it trusted.

A **root CA certificate** is self-signed: the subject and the issuer are the same entity, and the certificate is signed with its own private key. Nothing above it vouches for it. It becomes trusted only when you intentionally add it to a client trust store. That is the whole trick behind every root certificate on your machine — someone decided to trust it, and everything signed by it inherits that decision.

We are building a lab CA, so we get to make that decision ourselves, in a contained way.

Create a short-lived lab root using the key from the previous lesson:

```
openssl req -x509 -new -key lab-ca.key -sha256 -days 30 -out lab-ca.crt -subj '/CN=Practical
```

`-x509` tells `req` to produce a self-signed certificate instead of a signing request. `-days 30` keeps the root short-lived, which is deliberate: a learning CA should expire soon after you stop using it. `-subj` sets the identity inline so OpenSSL does not prompt for country and organization fields we do not need.

Now inspect what you created:

```
openssl x509 -in lab-ca.crt -noout -subject -issuer -dates
# subject=CN = Practical TLS Lab CA
# issuer=CN = Practical TLS Lab CA
# notBefore=Aug  3 10:30:00 2026 GMT
# notAfter=Sep  2 10:30:00 2026 GMT
```

Subject and issuer are identical. That is what self-signed means.

Check the basic constraints too:

```
openssl x509 -in lab-ca.crt -noout -ext basicConstraints
# X509v3 Basic Constraints: critical
#     CA:TRUE
```

`CA:TRUE` marks this certificate as an authority allowed to sign other certificates. Clients reject chains where a non-CA certificate tried to act as a signer, so this flag matters when we issue the server certificate later.

Keep the CA key offline between signing exercises. The certificate file `lab-ca.crt` is public and will be handed to clients; the key stays put.

The security lesson underneath: never install an unknown root certificate. A trusted root can authorize certificates for many names — any name, in fact. When some tool asks you to install its root CA, it is asking for the power to impersonate every website you visit. Our lab root gets trusted per-command only, which you will see in a later lesson.

Create a certificate signing request

Generate a server key and CSR containing the identity and public key to be signed.

A server never sends its private key to a certificate authority. Instead it sends a **certificate signing request** (CSR). The CSR carries a public key, requested identity information, and a signature proving possession of the private key. The CA reads the request, decides whether the identity is allowed, and returns a signed certificate.

This split is what makes public CAs possible: Let's Encrypt signs millions of certificates without ever seeing a private key.

Create a key and CSR for `app.lab.test` in one command:

```
openssl req -new -newkey rsa:2048 -nodes -keyout app.key -out app.csr -subj '/CN=app.lab.test
```

A few options worth understanding. `-newkey rsa:2048` generates a fresh 2048-bit RSA key alongside the request. `-nodes` leaves the server key unencrypted, so the lab server can start without a passphrase prompt — acceptable here because the file stays on your machine with tight permissions. `-addext` puts both hostnames into the **subjectAltName** request, which is where clients will actually look for names.

Inspect what you are about to hand to the CA:

```
openssl req -in app.csr -noout -text -verify
```

The `-verify` flag checks the CSR's self-signature and prints a verify confirmation line before the decode. In the text output, confirm two things: the `Subject` shows `CN = app.lab.test`, and the `Requested Extensions` section lists both DNS names:

```
Requested Extensions:
    X509v3 Subject Alternative Name:
        DNS:app.lab.test, DNS:localhost
```

Confirm both requested names and the signature before signing anything.

A common mistake is skipping `-addext` and requesting only a common name. Some signing setups will not add a SAN for you, and a certificate without SAN entries fails name verification in modern clients even though every field looks reasonable. Catch this at the CSR stage, where fixing it costs one command.

Remember which file is which. The CSR is shareable with the signer — it contains only public information. The generated `app.key` is not. If you ever paste a CSR into a support ticket, double-check you did not paste the key by mistake.

Sign a server certificate

Issue a short-lived leaf certificate with server constraints and verify it against the lab root.

Signing is the moment your lab CA acts as an authority. The CA signs the CSR after deciding that its requested identity is allowed. In a real CA that decision involves domain validation; in our lab, you are the policy, so the decision is yours — but the mechanics are the same.

Sign the lab request while copying its SAN extension:

```
openssl x509 -req -in app.csr -CA lab-ca.crt -CAkey lab-ca.key -CAcreateserial -out app.crt -
```

Walk through the options. `-CA` and `-CAkey` identify the signer; you will be prompted for the CA key passphrase. `-CAcreateserial` generates a serial number file (`lab-ca.srl`) so each issued certificate gets a unique serial. `-days 14` keeps the leaf short-lived. The one that trips people up is `-copy_extensions copy`: without it, OpenSSL drops the requested extensions, and your certificate comes out with no **subjectAltName** at all. A useful leaf certificate also needs suitable extensions, not just a signed name.

Now verify the result against the lab root:

```
openssl verify -CAfile lab-ca.crt app.crt
# app.crt: OK
```

`app.crt: OK` means the chain from this leaf to your root validates: the signature checks out, the dates are current, and the issuer is trusted (because you told `verify` to trust it with `-CAfile`).

Then inspect SAN, validity, issuer, and basic constraints:

```
openssl x509 -in app.crt -noout -issuer -dates -ext subjectAltName
# issuer=CN = Practical TLS Lab CA
# notBefore=Aug  3 10:45:00 2026 GMT
# notAfter=Aug 17 10:45:00 2026 GMT
# X509v3 Subject Alternative Name:
#     DNS:app.lab.test, DNS:localhost
```

If the SAN block is missing, you forgot `-copy_extensions copy`. Re-sign — the CSR is still valid, so this costs one command. This is the single most common issuance mistake, and the symptom appears much later as a hostname mismatch that makes no sense until you decode the certificate.

Keep lab roots and certificates short-lived. Do not reuse this learning CA for production systems. A production internal CA needs revocation, audited issuance policy, and protected key storage — none of which our two files provide.

Check your understanding: build a local certificate authority

Check the commands, decisions, safety boundaries, and failure cases from build a local certificate authority.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Serve local HTTPS

Run a local HTTPS application, trust the lab root narrowly, and verify names from multiple clients.

Run a Node.js HTTPS server

Load the lab certificate and key into a minimal local HTTPS server bound to loopback.

Time to put the lab certificate to work. An HTTPS server combines an application listener with a certificate chain and matching private key. Node.js ships this in the standard library, so we need no dependencies at all.

Create `server.mjs` in the directory that holds `app.key` and `app.crt`:

```
import https from 'node:https'
import fs from 'node:fs'

https.createServer({
  key: fs.readFileSync('app.key'),
  cert: fs.readFileSync('app.crt'),
}, (request, response) => {
  response.end('secure lab\n')
}).listen(8443, '127.0.0.1')
```

The options object is where TLS happens: `key` is the server's private key, `cert` is the certificate we issued. The second argument to `listen` binds the server to `127.0.0.1` only. Keep the first lab on loopback — nothing outside your machine can reach it, which is exactly what you want while practicing.

Run it:

```
node server.mjs
```

No output means it started. Confirm port 8443 is bound only to loopback:

```
lsof -iTCP:8443 -sTCP:LISTEN
# COMMAND   PID   USER   TYPE  NAME
# node    41235 flavio   IPv4  localhost:8443 (LISTEN)
```

The `localhost:8443` in the NAME column confirms the loopback bind. If you saw `*:8443` instead, the server would be listening on every interface.

Now probe it from a second terminal:

```
curl https://127.0.0.1:8443/
# curl: (60) SSL certificate problem: unable to get local issuer certificate
```

That error is correct behavior. curl does not trust your lab CA, so verification fails. The next lessons fix this the right way — by giving curl the hostname and the root — not by turning verification off.

One failure mode worth knowing: if `key` and `cert` do not belong together, server creation stops immediately with a key-values-mismatch error from OpenSSL. Node refuses to start rather than serve a broken identity. When you see it, you loaded a key from one generation and a certificate from another; re-check which files you copied.

Restrict private-key permissions with `chmod 600 app.key` and do not expose the practice service beyond your own computer.

Map the lab hostname

Give the loopback service its certificate hostname without creating a public DNS record.

Our certificate says `app.lab.test`, but the server lives at `127.0.0.1`. TLS name verification uses the URL hostname: the client takes the name from the URL and checks it against the certificate's SAN entries. Connect by IP and the check fails, because `127.0.0.1` is not one of the names we issued.

So the lab needs a way to type `https://app.lab.test:8443/` and land on loopback. For a local lab, map `app.lab.test` to loopback through the hosts file or curl `--resolve`.

The `--resolve` option is the narrowest tool. It tells curl to use a specific address for one hostname and port, for this command only. Test without a global hosts-file change:

```
curl --resolve app.lab.test:8443:127.0.0.1 https://app.lab.test:8443/
# curl: (60) SSL certificate problem: unable to get local issuer certificate
```

The first attempt should fail because the lab root is not trusted yet. That is the correct security behavior. But notice what changed: curl no longer complains about the name. It resolved `app.lab.test` to loopback, sent that name as SNI, matched it against the certificate SAN, and failed only on the missing trust anchor. One problem left instead of two.

If you want the mapping to apply to every tool on the machine — a browser, Node scripts, anything — add a hosts-file entry instead:

```
echo '127.0.0.1 app.lab.test' | sudo tee -a /etc/hosts
```

The hosts file is consulted before DNS, so every process now resolves `app.lab.test` to loopback. Remember to remove the line when the lab is done; stale hosts entries cause confusing "works on my machine" bugs months later, when the same name gets used somewhere real.

The `.test` top-level domain is reserved for exactly this purpose. It will never be registered publicly, so your lab names cannot collide with a real site. That is why this course uses `app.lab.test` and not something ending in `.com`.

Which leads to the rule: do not use a public hostname you do not control for locally issued certificates. If your lab issues a certificate for someone else's domain and the mapping leaks into a real environment, you have built a small impersonation setup. Reserved names keep the whole exercise contained.

Trust the lab CA for one command

Verify the local server with an explicit CA file before changing an operating-system trust store.

The last missing piece is trust. curl rejected our server because `lab-ca.crt` is not among its trusted roots. There are several ways to fix that, and they differ enormously in blast radius. My advice is to always start with the narrowest one.

The narrowest trust change is command-specific. curl can use the lab root for one transfer without trusting it for every application. The `--cacert` option replaces curl's default root store with a file you name, for this command only.

Connect using the explicit root:

```
curl --cacert lab-ca.crt --resolve app.lab.test:8443:127.0.0.1 https://app.lab.test:8443/
# secure lab
```

The request should now pass both chain and hostname verification, and you get the server's response body. Everything the previous lessons built is now working together: the hostname resolves to

loopback, the server presents the certificate we issued, and curl walks the chain to the root we explicitly supplied.

Add `-v` if you want to watch it happen:

```
curl -v --cacert lab-ca.crt --resolve app.lab.test:8443:127.0.0.1 https://app.lab.test:8443/
#  subject: CN=app.lab.test
#  issuer: CN=Practical TLS Lab CA
# * SSL certificate verify ok.
```

The `SSL certificate verify ok` line is the verification result you worked for.

Now break it on purpose, because controlled failure teaches you what each check does. Remove either the CA option or correct hostname to observe a controlled failure. Drop `--cacert` and you get error 60, `unable to get local issuer certificate` — the trust check. Keep `--cacert` but connect to `https://127.0.0.1:8443/` and the name check fails instead, because `127.0.0.1` is not in the SAN. Two different protections, two different errors.

Note what we never typed: `--insecure`. That flag skips verification entirely and teaches you nothing except bad habits. With `--cacert` every check still runs; we just told curl which root to measure against.

System-wide root installation grants broader authority — every application on the machine would accept anything your lab CA signs. Use it only when the development workflow genuinely requires it and document removal, so the lab root does not outlive the lab.

Serve a complete chain

Configure a server to send its leaf and intermediate certificates in the correct order.

So far our chain has been short: leaf, root, done. Real deployments almost always have at least one **intermediate CA** between them. That changes what the server must send. A client may trust the root but lack your intermediate — the server must provide the leaf followed by required intermediates so clients can build the path from the certificate they received up to a root they hold.

The standard fix is a **full-chain file**: the leaf first, then each intermediate, concatenated in order. PEM files stack cleanly, so this is one command.

Combine a lab leaf and intermediate for server use:

```
cat app.crt lab-intermediate.crt > app-fullchain.crt
```

Order matters. The leaf comes first because TLS requires the server's own certificate at the start; each following certificate should sign the one before it. This is the same layout tools like certbot produce when they hand you a `fullchain.pem`.

Point the server at the full chain by loading it as the `cert` value:

```
cert: fs.readFileSync('app-fullchain.crt'),
```

Node sends every certificate in the file, in file order. The `key` option does not change — the private key still corresponds only to the leaf. Intermediates are public certificates; no key material for them belongs anywhere near your server.

Verify what the server now presents with `s_client -showcerts`:

```
openssl s_client -connect app.lab.test:8443 -showcerts </dev/null 2>/dev/null | grep -c 'BEGI
# 2
```

Two PEM blocks means both the leaf and the intermediate went out. Run without the `grep` to read the `s:` and `i:` lines and confirm the order: certificate `0` should be `app.lab.test`, certificate `1` the intermediate.

The classic mistake here is subtle because it half-works. A server sending only the leaf passes tests in browsers that cached the intermediate from an earlier visit somewhere else, then fails in curl, CI, or mobile apps with `unable to get local issuer certificate`. If a TLS error appears only on some clients, suspect the chain before anything else.

Two things never belong in a full-chain file. Do not include a private key — the file gets sent to every client. And do not include the root: clients must already hold it, they ignore a root you send, and you waste bytes on every handshake.

Check your understanding: serve local https

Check the commands, decisions, safety boundaries, and failure cases from serve local https.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Diagnose TLS failures

Reproduce hostname, time, chain, protocol, and key failures and identify the exact failed check.

Diagnose a hostname mismatch

Compare the requested name with certificate SAN entries and repair the identity instead of disabling checks.

A trusted certificate can still be wrong for the requested hostname. Chain verification and name verification are two separate checks: the first asks "did a trusted CA sign this?", the second asks "is this certificate for the name I typed?". A certificate for `app.lab.test` is genuinely valid — for `app.lab.test`. Ask for any other name and the identity check must fail.

Let's force a mismatch against the local server. Our lab certificate includes `DNS:localhost` in its SAN, so pick a name it does not cover:

```
curl --cacert lab-ca.crt https://127.0.0.1:8443/
# curl: (60) SSL: no alternative certificate subject name matches target host name '127.0.0.1
```

Read the exact error. curl trusted the chain — `--cacert` handled that — and failed only on the name comparison. Depending on your curl version the wording mentions the subject name or the alternative names, but the shape is the same: the name you requested, and the statement that no certificate name matched it.

Now inspect what names the leaf actually covers:

```
openssl s_client -connect 127.0.0.1:8443 </dev/null 2>/dev/null | openssl x509 -noout -ext su
# X509v3 Subject Alternative Name:
#     DNS:app.lab.test, DNS:localhost
```

This pipe grabs the leaf from the live server and decodes its SAN. Diagnosis is now a comparison: requested `127.0.0.1`, offered `app.lab.test` and `localhost`. No overlap, so the failure is correct.

From there you have exactly two honest fixes. Fix the requested hostname — connect using a name the certificate covers, such as `https://localhost:8443/` in our lab. Or fix the certificate issuance policy — reissue with a SAN that includes the name clients really use. In production the second is usually right: clients use the name they use, and the certificate must match reality.

In real systems this error usually means a URL pointing at an internal IP, a certificate that lists `www.example.org` but not `example.org`, or a service migrated to a new name with the old certificate still deployed.

Do not solve a name mismatch with `--insecure`; that removes the identity check you need. It does not skip just this error — it accepts any certificate from anyone, which is precisely the attack TLS names exist to stop.

Diagnose certificate validity time

Use certificate dates and the client clock to distinguish expired, not-yet-valid, and clock-skew failures.

Certificates have `notBefore` and `notAfter` timestamps, and every verification compares them against one more input people forget about: the client's own clock. Verification uses the client's current clock, so a wrong clock can make a perfectly valid deployment appear invalid — and a real expiry look like a mystery.

That gives you three distinct time failures that produce similar-looking errors:

- **Expired**: the clock is right, `notAfter` is in the past. Someone missed a renewal.
- **Not yet valid**: the clock is right, `notBefore` is in the future. Rare, usually a certificate issued by a machine with a fast clock.
- **Clock skew**: the certificate is fine, the client's clock is wrong. Common on devices that sat powered off, containers with a broken time source, and machines without NTP.

Diagnosis means putting the two timestamps and the clock side by side. Inspect dates and current UTC time:

```
openssl x509 -in app.crt -noout -dates
# notBefore=Aug  3 10:45:00 2026 GMT
# notAfter=Aug 17 10:45:00 2026 GMT
date -u
# Mon Aug  3 15:02:11 UTC 2026
```

Use `date -u` and not plain `date` — certificate timestamps are in GMT, and comparing across timezones by eye is how off-by-a-few-hours mistakes happen. Classify the result as currently valid, expired, or not yet valid.

For a live server, take the dates from the connection itself:

```
openssl s_client -connect example.org:443 -servername example.org </dev/null 2>/dev/null | op
# notAfter=Jan 15 08:21:00 2027 GMT
```

An expired certificate also shows up in `s_client` as `Verify return code: 10 (certificate has expired)` — that code 10 is your fastest confirmation.

If the dates look sane but verification still complains about time, suspect the clock. Check whether the machine is synchronized (`timedatectl` on systemd Linux shows `System clock synchronized: yes`). A clock that is off by minutes already breaks certificates issued moments ago.

The prevention side: keep renewal monitoring ahead of the deadline. Alert when a certificate has, say, 14 days left, not when it dies. Expiry is the most predictable outage in computing — the date has been printed inside the certificate since the day it was issued.

One thing you must never do: changing the clock to hide an error breaks other security and operational systems — logs, tokens, schedulers, and every other certificate. Fix time synchronization or renew the certificate. Those are the only two real repairs.

Diagnose a missing intermediate

Identify an incomplete server chain and verify the same leaf with the required intermediate supplied.

This is the sneakiest chain failure because it hides. A server can present a correct leaf but omit the intermediate needed to reach a trusted root. Some clients may hide the mistake because they cached that intermediate from an earlier connection to a different site — browsers do this all the time. So the site works in Chrome, works on your laptop, and fails in curl, in CI, and on a fresh phone.

The tell is exactly that pattern: a TLS failure that only some clients see.

Inspect and verify the presented chain:

```
openssl s_client -connect app.lab.test:8443 -servername app.lab.test -showcerts -CAfile lab-c
```

Two parts of the output matter. First, count the certificates in the `Certificate chain` section. A leaf that was signed by an intermediate, presented alone, looks like this:

```
Certificate chain
 0 s:CN = app.lab.test
   i:CN = Practical TLS Lab Intermediate CA
...
verify error:num=20:unable to get local issuer certificate
Verify return code: 20 (unable to get local issuer certificate)
```

One certificate, and its issuer is an intermediate that never arrived. Error 20 is OpenSSL saying "I have the leaf, but nothing in what the server sent or in my trust store signs it". Error 21, `unable to verify the first certificate`, points at the same family of problem.

Second, prove the leaf itself is fine. Save certificate `0` from the `-showcerts` output as `presented.crt`, then verify it with the intermediate supplied by hand:

```
openssl verify -CAfile lab-ca.crt -untrusted lab-intermediate.crt presented.crt
# presented.crt: OK
```

`-untrusted` provides certificates that may be used to build the path without being trust anchors themselves. If this returns `OK`, the diagnosis is settled: the leaf is good, the server's configuration is incomplete.

The fix belongs on the server, once. Compare the certificates sent by the server with the intended chain, add the intermediate to the server full-chain configuration — the `cat leaf intermediate` file from earlier in this course — and retest from a clean client that has no cached state.

Do not add the leaf itself to every client trust store. That "fix" multiplies across every client, breaks again at the next renewal, and trains people to click through trust prompts. Repair the server chain and every client heals at the same moment.

Diagnose protocol and key failures

Separate no-shared-protocol failures, unsupported algorithms, and certificate/private-key mismatches.

Not every TLS failure is about certificates. A handshake can fail before certificate verification even starts, when the two peers cannot agree on a protocol version or algorithm. And a server can fail to start at all when its key does not match the certificate. These three problems produce very different errors, and telling them apart saves you from debugging the wrong layer.

No shared protocol

An old client that only speaks TLS 1.0 cannot talk to a server that requires TLS 1.2 or newer. The handshake dies immediately, before any certificate is exchanged. You can reproduce the shape of this by forcing a version during a controlled test:

```
openssl s_client -tls1_2 -connect example.org:443 -servername example.org </dev/null
```

Compare with `-tls1_3` where supported. Against a server that refuses the version you forced, the output contains no `Certificate chain` section at all — just an alert like `tlsv1 alert protocol version` or a `no protocols available` error. That absence is the clue: if you never received a certificate, stop staring at certificate fields.

Key does not match certificate

The other failure lives on the server. After a renewal or a copy-paste deployment, the certificate on disk and the private key on disk can come from different generations. Node refuses to start with a key-mismatch error; nginx says `SSL_CTX_use_PrivateKey_file ... key values mismatch`.

You confirm it without exposing secrets. A key pair matches when both files contain the same public key, so for a key pair, compare public-key fingerprints rather than exposing private material:

```
openssl x509 -in app.crt -noout -pubkey | openssl sha256
# SHA2-256(stdin)= 7f3c9e...
openssl pkey -in app.key -pubout | openssl sha256
# SHA2-256(stdin)= 7f3c9e...
```

The first command extracts the public key from the certificate, the second derives it from the private key. Identical hashes mean the pair belongs together. Different hashes mean you deployed mismatched files — find the key that was generated with this certificate's CSR.

The tempting wrong fix

When an ancient device cannot connect, someone will suggest re-enabling TLS 1.0 on the server. Do not enable obsolete protocols just to satisfy an abandoned client — that downgrades every connection's floor, not just one client's. Upgrade or isolate the dependency instead, for example behind a dedicated proxy that only that client uses, while the main endpoint keeps a modern minimum.

Check your understanding: diagnose tls failures

Check the commands, decisions, safety boundaries, and failure cases from diagnose tls failures.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Mutual TLS and operations

Require client certificates, convert formats, rotate identities, and complete an evidence-based TLS review.

Issue a client certificate

Create a separate client identity with client-auth usage instead of reusing the server certificate.

Until now, only the server proved its identity. **Mutual TLS** flips the second half on: the client also presents a certificate, and the server gets a certificate-backed client identity instead of a password or an API key. It's common between backend services, for admin endpoints, and anywhere a stolen bearer token would hurt.

A client identity is its own credential. Issue it under a policy and key separate from the HTTPS server — never reuse the server certificate for a client, because the two roles have different lifetimes, different holders, and different revocation stories.

Create a client key and CSR:

```
openssl req -new -newkey rsa:2048 -nodes -keyout client.key -out client.csr -subj '/CN=lab-cl
```

Note what's missing compared to the server CSR: no DNS names in a SAN. A client certificate identifies a party, not a hostname, so a common name like `lab-client` is the identity the server will see.

Now sign it with a client-auth extension. The **Extended Key Usage** extension declares what a certificate may be used for, and for a client it must say client authentication:

```
printf 'extendedKeyUsage = clientAuth\n' > client-ext.cnf
openssl x509 -req -in client.csr -CA lab-ca.crt -CAkey lab-ca.key -CAcreateserial -out client
```

The `-extfile` option injects the extension during signing. Without it you would get a generic certificate, and strict servers reject client certificates that lack the `clientAuth` usage.

Inspect Extended Key Usage before using it:

```
openssl x509 -in client.crt -noout -subject -ext extendedKeyUsage
# subject=CN = lab-client
# X509v3 Extended Key Usage:
#     TLS Web Client Authentication
```

`TLS Web Client Authentication` is the line that makes this a client certificate. If you see `TLS Web Server Authentication` instead, you signed with the wrong extension file — reissue before touching the server.

A verification failure you can predict: sign without the extension, then wonder why a picky server rejects an otherwise valid certificate. Decode the EKU first, every time; it costs one command.

Treat the result like what it is. A client private key authenticates its holder. Protect, scope, rotate, and revoke it like any credential — one certificate per client, short lifetimes, and a plan for what happens when a laptop carrying `client.key` goes missing.

Require client authentication

Configure the local server to request and verify certificates from the lab CA.

In mutual TLS, the client verifies the server and the server also verifies the client certificate chain. The server needs three pieces of configuration for that: ask for a certificate, refuse connections that fail verification, and know which CA to verify against.

In our Node lab server, that is three new options. Add client verification to the Node HTTPS options:

```
https.createServer({
  key: fs.readFileSync('app.key'),
  cert: fs.readFileSync('app.crt'),
  requestCert: true,
  rejectUnauthorized: true,
  ca: fs.readFileSync('lab-ca.crt'),
}, (request, response) => {
  response.end(`hello ${request.socket.getPeerCertificate().subject.CN}\n`)
}).listen(8443, '127.0.0.1')
```

`requestCert: true` makes the server ask every client for a certificate during the handshake. `rejectUnauthorized: true` drops clients whose certificate does not verify. `ca` sets the trust anchor for that verification — only certificates chaining to our lab CA pass. The handler shows the payoff: `getPeerCertificate()` exposes the verified client identity, so the application can read `CN = lab-client` from the socket.

Now test both sides of the door. Connect once without a client certificate:

```
curl --cacert lab-ca.crt --resolve app.lab.test:8443:127.0.0.1 https://app.lab.test:8443/
# curl: (56) ... alert certificate required
```

The handshake fails — not with an HTTP 401, but at the TLS layer, with an alert telling you a certificate was required. Unauthenticated clients never reach the application code at all.

Then connect with the client credentials from the previous lesson, using `curl --cert client.crt --key client.key`:

```
curl --cacert lab-ca.crt --cert client.crt --key client.key --resolve app.lab.test:8443:127.0
# hello lab-client
```

Record the different handshake results. Same server, same URL: one connection rejected during the handshake, one greeted by name.

A failure worth recognizing: everything configured, but the client certificate was signed by a different CA than the one in the server's `ca` option. The symptom is the same handshake alert as having no certificate. When mTLS ”doesn't work”, verify the client certificate against the server's expected CA with `openssl verify -CAfile lab-ca.crt client.crt` before touching code.

One boundary to keep sharp: authentication is not authorization. Do not treat a valid client certificate as unlimited authorization. The handshake proved who is connecting; the application still decides what that identity may do. Map the verified identity to narrow application permissions, exactly as you would with any logged-in user.

Convert certificate formats

Convert between PEM and PKCS#12 when a platform requires a bundle, without losing track of private-key handling.

Sooner or later a platform refuses your neat pile of `.crt` and `.key` files and demands "a PKCS#12 file" or "a .pfx". Nothing about the cryptography changes — it is the same key and the same certificates in a different container. Knowing the two main containers saves you from panicked searching at deploy time.

PEM is what we have used all course: text files with `-----BEGIN CERTIFICATE-----` markers, one concern per file. PEM commonly stores text-encoded certificates and keys in separate files, which is why servers like nginx and Node take separate `cert` and `key` paths.

PKCS#12 (extensions `.p12` or `.pfx`) is a binary bundle. It can package a key, leaf, and chain under password protection in a single file. Browsers importing a client certificate, Java keystores, and Windows services usually want this one.

Create a lab client bundle from the mTLS files:

```
openssl pkcs12 -export -out client.p12 -inkey client.key -in client.crt -certfile lab-ca.crt
```

`-export` switches `pkcs12` into bundle-creation mode. `-inkey` and `-in` take the private key and its certificate; `-certfile` adds the CA certificate so the receiving system gets the chain context too. You are prompted for an export password — choose a real one, because this file contains the private key and will probably travel to another machine, which is exactly when files get copied around carelessly.

Check what landed inside without decrypting the secrets to your terminal:

```
openssl pkcs12 -in client.p12 -info -noout
# MAC: sha256, Iteration 2048
# PKCS7 Encrypted data: ...
# Certificate bag
# Certificate bag
# Shrouded Keybag: ...
```

List the bundle structure with `-info -noout`: two certificate bags (the client certificate and the CA) and one shrouded key bag (the encrypted private key). Avoid printing private key material — if you ever extract back to PEM, be aware the output includes the decrypted key, so send it to a protected file, never to a shared screen or log.

The failure mode here is not technical but operational: the bundle works, gets mailed around "because it's just one file", and now the private key exists in inboxes with a password sitting in the same thread. Format conversion does not improve a weak password or insecure file permissions. Treat a `.p12` with the same care as the raw key, because that is what it contains.

Plan rotation and recovery

Track owners, expiry, renewal, deployment, rollback, revocation, and private-key compromise.

Certificates expire and private keys can be exposed. Both are certainties, not risks — one has a printed date, the other has an unknown one. Operations need renewal before expiry and an emergency replacement path, and neither can depend on the one person who remembers where the files live.

Start with knowing what you have. Create a small inventory from certificate files:

```
for file in *.crt; do
  echo "$file"
  openssl x509 -in "$file" -noout -subject -issuer -enddate -fingerprint -sha256
done
```

For each certificate you get the identity, the signer, the deadline, and a SHA-256 fingerprint that pins exactly which certificate is deployed where:

```
app.crt
subject=CN = app.lab.test
issuer=CN = Practical TLS Lab CA
notAfter=Aug 17 10:45:00 2026 GMT
SHA2-256 Fingerprint=7A:1B:0C:...
```

The certificate cannot tell you the rest. Add owner, service, deployment location, renewal method, and recovery contact outside the certificate itself — a tracked file or a wiki page beats memory. When `notAfter` is 30 days out, this row is what turns an alert into an action.

Renewal should be boring. Public-facing certificates belong on ACME automation — a Let's Encrypt client renews and deploys with no human in the loop, and your monitoring only confirms it keeps happening. For an internal CA like our lab, the renewal method column names the commands and who runs them. Alert on remaining lifetime, not on expiry day; an alert that fires with two weeks left is a task, one that fires at midnight is an outage.

Then there is the emergency path: the private key leaks. Renewal on a schedule does not cover this — you need replacement now. The drill is always the same: issue a new key and certificate, deploy, then revoke the old certificate so the CA stops vouching for it. The order matters; revoking first takes the service down.

Rehearse replacing the lab certificate without disabling HTTPS: issue a second leaf from your lab CA, swap the files, restart, and confirm with `openssl s_client` that the new fingerprint is being served. If that drill takes an afternoon of guessing in the lab, it will fail at 2am in production.

One policy note for the inventory: never copy old keys into a new certificate unless the rotation policy deliberately permits key reuse. A rotation that keeps the compromised key has rotated nothing.

Check your understanding: mutual tls and operations

Check the commands, decisions, safety boundaries, and failure cases from mutual tls and operations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/tls-certificates/.