

Mutual TLS with a Private CA

Running an internal certificate authority to support mTLS and zero trust.

- [Concepts](#)
- [Architecture](#)
- [Standing up the CA](#)
- [Issuing and Installing Certificates](#)
- [Enforcing mTLS](#)
- [Keeping Renewal Boring](#)

Concepts

Why nothing inside the network is trusted without proof, and how to run the CA that makes it possible.

This book is a working reference for putting mutual TLS (mTLS) between every internal component of a self-hosted environment, backed by a private certificate authority you run yourself. It opens with the case for doing it, then walks through standing up the CA, issuing and installing certificates, enforcing mTLS at the proxy layer, and the part that makes this sustainable, keeping renewal boring. The specific use-case I provide as an example is Web Application Firewall (WAF)-to-host, but the same pattern can be applied to other services.

Why internal traffic shouldn't be trusted

For a long time, internal networks ran on an assumption of trust by location. A request coming from inside the perimeter was treated as trustworthy simply because of where it came from. But location is not identity. Anything that gains ingress to the internal network including a compromised workstation, a rogue device, or a service host an attacker has gained a foothold on, inherits the trust the perimeter was supposed to gate. On a flat network where internal callers are trusted by default, one compromised device can reach every service that assumes internal traffic is safe, because nothing along the way asks it to prove who it is. The blast radius of a single foothold is **everything**.

Mutual TLS (mTLS) breaks that assumption. It does two things. It encrypts every internal connection, so a device that manages to intercept traffic cannot read it, and it requires both ends of every connection to present a certificate proving who they are, issued by an authority both sides trust. A backend configured for mTLS does not just encrypt, it refuses connections that cannot present a valid certificate. Being on the network is no longer enough; you have to hold a key.

This does not make a compromised host harmless, and it's worth being precise about what it does and doesn't do. An attacker who compromises a host can use whatever certificate that host legitimately holds, to reach whatever that certificate was authorized to reach. What mTLS changes is the scope of a compromise. On a network based on location trust, one foothold reaches everything. With mTLS, one foothold reaches only what that specific identity was permitted to reach, and a device holding no valid certificate cannot open an authenticated connection to a protected service at all. The boundary moves from "inside the network" to "holds a valid identity,"

and the blast radius shrinks from everything to one identity's authorized reach.

What "identity" means here: When a person logs in, they prove identity with something they know or hold, a password, a second factor, a passkey. A machine can't do that, so it proves identity a different way; with a certificate. A certificate is a small signed document that says "this host is service.int.example," countersigned by an authority both sides trust. When a service presents its certificate, the other end can verify that signature and know it is talking to the real holder of that name, not an impostor. That is all a machine identity is, a name, bound to a cryptographic key, vouched for by a trusted authority. Where a user identity answers "which person is this," a machine identity answers "which host or service is this," and mTLS is simply both ends asking that question of each other before exchanging any traffic.

Why a private certificate authority

Certificates need an issuer both ends trust. For public websites that's a public CA like Let's Encrypt. For internal service identity, a public CA is the wrong tool:

- **Public CAs won't issue for internal names.** Your internal services have names like `service.int.example` that don't exist in public DNS and that no public CA will ever certify. Internal identity needs an issuer that will.
- **You want short lifetimes and full control.** A private CA lets you issue certificates that live weeks, not years, and rotate constantly, which shrinks the value of a stolen key. You set the policy.
- **The CA becomes your internal root of trust.** Every host is configured to trust one CA, and from then on "do you hold a certificate this CA signed?" is the question that gates internal communication. That root of trust is yours to run, not rented.

A private CA is, in effect, the identity system for machines, the same role an IdP plays for users. Users authenticate to an identity provider; services authenticate to each other with certificates from the CA. Both answer the same underlying question: prove you are who you claim to be before I trust you.

Why step-ca

The CA in this environment is [step-ca](#) from Smallstep. The reasons it fit:

- **Self-hostable and lightweight.** It runs as a single service on an internal host, with no dependency on anyone else's infrastructure. For a self-hosted, privacy-focused environment, the machine identity root shouldn't live in someone else's cloud any more

than the user identity root should.

- **Built around short-lived certificates.** Step-ca's whole design philosophy is issue-often, expire-fast, automate-renewal, which is exactly the posture that makes stolen certs low-value. It makes the secure path the easy path.
- **Standards-based and ACME-capable.** It issues ordinary X.509 certificates and can speak ACME, so the knowledge and the certificates work with standard tooling (nginx, curl, anything that speaks TLS).
- **Right-sized.** It gives you a real CA, root, intermediate, provisioners, issuance policy, without standing up an enterprise PKI suite that would be absurd at a small scale.

Architecture

Before the configuration, the model. Three ideas make the rest of this book make sense: where the CA lives, what the trust chain looks like, and the fact that every internal connection uses *two* certificates, not one.

Where the CA lives

The certificate authority runs on an internal control-plane host, reachable only from inside the private network. It is never exposed publicly. Its only job is to answer certificate requests from hosts that have already proven they belong in the environment, and to hold the signing key that every host trusts. Because that key is the root of all internal trust, the CA host is treated as one of the most sensitive systems in the environment, on par with the identity provider and the secrets manager.

The trust chain

Step-ca is initialized with a **root** certificate and an **intermediate** certificate. The root signs the intermediate; the intermediate signs the certificates issued to hosts. Every host in the environment is configured to trust the root. From then on, a certificate is trusted if it chains back to that root; root signed the intermediate, intermediate signed the host cert. This is why nginx is configured with `ssl_verify_depth 2` later on: the chain is two links deep, and verification has to be allowed to trace both.

Trust is established once, by pinning the root's fingerprint when a host bootstraps (covered on the next page). After that, no host ever has to be told about individual certificates, it trusts anything the CA signs, and distrusts everything else.

The two-certificate model

A single mutually-authenticated connection involves **two certificates**, one presented by each end.

- The **WAF**, or other service, presents a **client certificate** identifying itself. This proves to the backend that the caller is the authorized service and not some other host that happens to be on the network.

- The **backend** presents a **server certificate** identifying it as `service.int.example`. This proves to the service, such as a (WAF), that it reached the right backend and not an impostor.

Each end verifies the other's certificate against the shared CA root. If either certificate is missing, expired, or not signed by the CA, the connection is refused. That mutual check is the whole point. The backend won't serve just anyone who can reach it, and the service won't forward to just any host claiming to be the backend.

Standing up the CA

This is the identity root for every machine in the environment. Everything else in the book depends on it existing and being trusted. Treat its keys with the same care as any other root secret.

Initialize step-ca

On the control-plane host, initialize the CA:

```
sudo curl -fsSL https://packages.smallstep.com/keys/apt/repo-signing-key.gpg -o
/etc/apt/trusted.gpg.d/smallstep.asc && echo 'deb [signed-
by=/etc/apt/trusted.gpg.d/smallstep.asc] https://packages.smallstep.com/stable/debian debs
main' | sudo tee /etc/apt/sources.list.d/smallstep.list

sudo apt update && sudo install step-cli step-ca

sudo step ca init
```

Make sure port 9000 is open on your firewall, this is the default port step-ca listens on.

The prompts and the choices made here:

Prompt	Value	Why
Deployment type	Standalone	A single self-contained CA; no need for the more complex modes at this scale.
PKI name	(your CA name)	Labels the root and intermediate.
DNS / IP for the CA	(internal CA host IP)	The address hosts will reach the CA at, inside the private network only.
CA bind address	(internal IP):9000	step-ca listens here for issuance requests.
First provisioner name	admin-provisioner	The identity allowed to request certificates. Referenced later when issuing.
Password for the CA keys	(strong secret)	Encrypts the CA's private keys at rest. Store it in the password manager.

`step ca init` produces the root certificate, the intermediate certificate, and the signing keys. The root is the thing every host will be told to trust.

The CA key password goes in a file the service reads at startup:

```
sudo nano /root/.step/secrets/password.txt
```

This file, and the key material it unlocks, are the crown jewels for your CA. Anyone who can read both can mint certificates the entire environment trusts. Lock down the host accordingly.

Set the issuance policy

Short-lived certificates are one of the main arguments for a private CA, and the issuance policy is where you enforce them. Edit the CA config:

```
sudo nano /root/.step/config/ca.json
```

Add a claims block setting the minimum, maximum, and default certificate lifetimes:

```
"claims": {  
  "minTLSCertDuration": "1h",  
  "maxTLSCertDuration": "2160h",  
  "defaultTLSCertDuration": "720h"  
}
```

What this says: a certificate can live no less than an hour, no more than 2160 hours (90 days), and defaults to 720 hours (30 days) if a duration isn't specified. The 90-day ceiling is the important one, it means a stolen certificate is worthless within a quarter no matter what, and in practice renewal happens far more often than that. Setting a *maximum* is what stops a convenient long-lived cert from quietly becoming a long-lived liability.

Capture the root fingerprint

Every host that joins the trust domain will pin the root by its fingerprint. Get it:

```
sudo step certificate fingerprint /root/.step/certs/root_ca.crt
```

Record this. It's used on every host bootstrap on the next page, and pinning by fingerprint means a host checks the CA against a value you already know, so a substituted or malicious root is rejected

instead of trusted.

Run it as a service

Run step-ca under systemd so it starts on boot and restarts on failure:

```
[Unit]
Description=Smallstep Certificate Authority
After=network-online.target

[Service]
User=root
Environment=STEPPATH=/root/.step
ExecStart=/usr/bin/step-ca /root/.step/config/ca.json --password-file
/root/.step/secrets/password.txt
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

The `--password-file` is what lets it start unattended, it reads the key password from the file rather than prompting. That convenience is also why that file's permissions matter so much: it is the thing standing between "a service that restarts cleanly" and "anyone who reads this file owns the CA."

Issuing and Installing Certificates

Every host follows the same bootstrap: make the internal names resolve, trust the CA root, then request the certificates it needs.

First: make the internal names resolve

Internal services are addressed by names like `service.int.example`. These names deliberately don't exist in public DNS, they're internal-only, so each host needs to resolve them locally. Rather than run an internal DNS server, my environment is small enough to just use `/etc/hosts` entries, which is a right-sized choice for a fixed, small set of hosts.

On each host, `/etc/hosts` maps every internal service name to its private IP:

```
10.x.x.2   jump.int.example
10.x.x.3   tools.int.example
10.x.x.4   pass.int.example
10.x.x.6   files.int.example
10.x.x.7   applications.int.example
10.x.x.8   devbox.int.example
10.x.x.9   auth.int.example
10.x.x.10  waf.int.example
# ...one line per host
```

This matters to mTLS directly, and it's a subtle failure mode worth understanding: **mTLS verifies the certificate against the name being connected to**. When the WAF connects to `service.int.example`, it checks the backend's certificate against that name. If `/etc/hosts` resolves the name to the wrong IP, or doesn't resolve it at all, you get failures that *look* like certificate problems, such as verification errors and connection refusals, but are actually name-resolution problems. When an mTLS connection misbehaves, confirm the name resolves to the right host before you start suspecting the certificate.

A gotcha while you're in here

Make sure you map the host's own short hostname to the loopback range:

```
127.0.1.1    thishostname
```

Without it, `sudo` incurs a noticeable delay on every invocation, because it tries and fails to resolve the machine's own hostname before proceeding. The symptom (slow sudo) looks nothing like the cause (a missing hosts entry). It isn't strictly an mTLS concern, but every host should get this line as part of base configuration and it's the kind of thing you only debug once if you've seen it before.

Second: trust the CA root

Bootstrap the host's trust in the CA, pinning the root by the fingerprint captured when the CA was initialized:

```
sudo step ca bootstrap \  
  --ca-url https://10.x.x.2:9000 \  
  --fingerprint <CA_ROOT_FINGERPRINT>
```

Pinning by fingerprint is what makes this first trust decision safe. The host isn't blindly accepting whatever the CA presents; it's accepting only the specific root whose fingerprint you supplied out of band. Then install the root cert where the system and nginx will look for it:

```
sudo install -m 644 -D /root/.step/certs/root_ca.crt /etc/ssl/certs/example-ca.crt
```

From here on, this host trusts anything the CA signed and nothing it didn't.

Third: request the certificates

Recall the two-certificate model. Each internal connection uses a server cert (held by the backend) and a client cert (held by the caller). For a backend host that serves an application, both get issued.

The host's server certificate, identifies the backend as itself:

```
sudo step ca certificate \  
  --san service.int.example \  
  --not-after 2160h --kty EC --curve P-256 \  
  --
```

```
service.int.example \  
/etc/ssl/certs/service.crt \  
/etc/ssl/private/service.key
```

The WAF's client certificate for reaching this host, issued on the WAF, identifies the WAF as the authorized caller:

```
sudo step ca certificate \  
  --not-after 2160h --kty EC --curve P-256 \  
waf-to-service \  
/etc/ssl/certs/waf-to-service.crt \  
/etc/ssl/private/waf-to-service.key
```

A few things about these commands worth noting:

- `--not-after 2160h` requests the 90-day maximum. In practice the renewal automation reissues long before then; the explicit duration just pins it to the policy ceiling.
- `--kty EC --curve P-256` issues elliptic-curve certificates rather than RSA, smaller, faster, and entirely sufficient. A reasonable modern default.
- **The names are the identity.** `service.int.example` and `waf-to-service` aren't cosmetic; they're what the other end verifies against. This is why name resolution had to be sorted first.

Enforcing mTLS

This is where the policy becomes real. The two configurations below, one on the calling side, one on the serving side, are what turn "both ends should authenticate" into "no valid certificate, no connection." Everything before this page was setup, this is the enforcement.

Recall the two ends of an internal connection: We'll use a WAF as an example, but this can apply to any communication. The WAF must present its client certificate and verify the backend's server certificate, and the backend must present its server certificate and require the WAF's client certificate. Both halves have to be configured.

The calling side (WAF)

On the WAF, inside the `location` block that proxies to the backend:

```
location / {
    proxy_pass https://service.int.example;

    # Verify the backend's server certificate against the CA
    proxy_ssl_server_name on;
    proxy_ssl_name service.int.example;
    proxy_ssl_trusted_certificate /etc/ssl/certs/example-ca.crt;
    proxy_ssl_verify on;
    proxy_ssl_verify_depth 2;

    # Present the WAF's own client certificate (this is the "mutual" half)
    proxy_ssl_certificate      /etc/ssl/certs/waf-to-service.crt;
    proxy_ssl_certificate_key /etc/ssl/private/waf-to-service.key;
}
```

Line by line, what each directive enforces:

- `proxy_pass https://...`: the upstream is HTTPS, not HTTP. The connection to the backend is itself TLS, separate from the public TLS the user terminated at the WAF.
- `proxy_ssl_server_name on` / `proxy_ssl_name`: send the backend's name during the handshake and verify the certificate matches it. This is what makes the name the identity: the WAF checks it reached `service.int.example` and not something else answering on that

IP.

- `proxy_ssl_trusted_certificate`: the CA root. The backend's certificate is trusted only if it chains to this.
- `proxy_ssl_verify on`: actually enforce the check. Without this, nginx would happily connect to a backend presenting any certificate. This directive is the difference between "encrypted" and "verified."
- `proxy_ssl_verify_depth 2`: allow the chain to be two links deep (host cert → intermediate → root), matching the CA's structure.
- `proxy_ssl_certificate` / `_key`: the WAF's own client certificate. This is the half that makes it *mutual*: the WAF proves its identity to the backend, not just the other way around.

The serving side (backend)

On the backend host, the internal gateway server block:

```
server {
    listen 443 ssl;
    server_name service.int.example;

    # This host's server certificate
    ssl_certificate      /etc/ssl/certs/service.crt;
    ssl_certificate_key  /etc/ssl/private/service.key;

    # Require and verify the caller's client certificate
    ssl_client_certificate /etc/ssl/certs/example-ca.crt;
    ssl_verify_client on;
    ssl_verify_depth 2;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        # hand off to the local application
        proxy_pass http://127.0.0.1:80;
        proxy_set_header Host $host;
        # ...forwarding headers
    }
}
```

The three directives that do the enforcing:

- `ssl_client_certificate`: the CA root, here used to validate *incoming* client certificates. The backend trusts a caller whose certificate chains to this.
- `ssl_verify_client on`: the critical one. This makes the client certificate *mandatory*. A connection without a valid CA-signed client certificate is rejected at the TLS layer, before the request ever reaches the application. This single directive is what enforces "being on the network isn't enough."
- `ssl_verify_depth 2`: same two-link chain allowance as the other end.

The backend terminates the mutually-authenticated TLS, then hands the request to the local application over plain localhost HTTP (`127.0.0.1:80`). The application itself doesn't need to know anything about certificates. The gateway enforces mTLS in front of it. That keeps the application simple and puts all the certificate logic in one place per host.

Gotchas

- **The name and the certificate must agree.** The WAF's `proxy_ssl_name`, the backend's `server_name`, and the name in the backend's certificate all have to refer to the same thing. If the WAF connects under one name and the backend's server block answers to another, nginx rejects the handshake and you get a misdirected-request error that looks like a proxy bug, but is really a name mismatch. When adding a service to a host that already serves others, the name the caller uses is what routes it. Get that wrong and the symptom is confusing.

Adapt this for...

Any internal service-to-service connection, not just WAF-to-backend. Anywhere two internal components talk and you want that traffic authenticated rather than merely encrypted, this is the shape, and `ssl_verify_client on` (or its equivalent on the serving side) is the line that enforces it.

Keeping Renewal Boring

Short-lived certificates are only a good idea if renewal is reliably automated. The price of short lifetimes is that expiry must be a non-event, and the only way to get there is automation.

Certificates in this setup presented live 90 days at most, usually less. Across a dozen hosts, each with a server certificate and one or more client certificates, manual renewal is tedious, and risks that something will expire unnoticed, and take a service down. The goal is a system where certificates are renewed before expiry, and the only time a human is involved is when renewal fails.

Per-host renewal

Each host runs the step-ca renewal service that watches its own certificates and renews them as they approach expiry. As a systemd unit:

```
[Unit]
Description=Auto-renew internal host certificate
After=network.target

[Service]
ExecStart=/usr/bin/step ca renew /etc/ssl/certs/service.crt /etc/ssl/private/service.key --
daemon --exec "systemctl reload nginx"
Restart=always
RestartSec=10
User=root
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=step-renew

[Install]
WantedBy=multi-user.target
```

The important pieces:

- **--daemon**: step runs continuously, waking on its own schedule to check whether the certificate is close enough to expiry to warrant renewal. It renews in place, so the certificate and key files are refreshed without intervention.

- **Restart=always**: if the renewal daemon dies, systemd brings it back. A renewal daemon that has quietly stopped is the silent failure this whole system exists to avoid.
- **SyslogIdentifier=step-renew**: renewal events are tagged in syslog. That tag is what makes renewal auditable so the logs can be forwarded, reviewed, and have alerts triggered on renewal failure.
- **--exec "systemctl reload nginx"**: reloads the nginx service after certificate renewal to refresh the certificates cached in memory.

The WAF's client certificates

The WAF holds a client certificate for every backend it talks to (**waf-to-service**, one per host). These are renewed on a weekly timer which executes a script to check and renew the certificates approaching expiry:

The **timer** runs it weekly. This is enabled in systemd, not the service file. The timer is the scheduler; it holds no logic beyond when.

```
[Unit]
Description=Run WAF mTLS renewal weekly

[Timer]
# Run every Monday at 02:00
OnCalendar=Mon *-*- * 02:00:00
Persistent=true
AccuracySec=1h
Unit=waf-mtls-renew.service

[Install]
WantedBy=timers.target
```

- **OnCalendar**: fires it every Monday at 02:00.
- **Persistent=true**: runs a missed occurrence at next boot if the machine was down at the scheduled time.
- **AccuracySec=1h**: lets systemd batch the wake-up for efficiency rather than firing on the exact second.

The **service** file performs the renewal pass, reissuing any **waf-to-<host>** certificate that expires within the next 21 days. The service is a oneshot: it runs the renewal script once and exits, which is the correct shape for timer-driven work rather than a long-running daemon.

```
[Unit]
Description=Renew WAF mTLS client certificates (waf-to-*.crt)
After=network-online.target
Wants=network-online.target

[Service]
Type=oneshot
ExecStart=/usr/local/bin/renew-waf-mtls.sh
User=root
PrivateTmp=yes
ProtectSystem=full
ProtectHome=read-only
NoNewPrivileges=yes

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=waf-mtls-renew
```

- **The hardening directives** `ProtectSystem`, `ProtectHome`, `NoNewPrivileges`, and `PrivateTmp` constrain what the script can touch since it runs as root.
- `SyslogIdentifier=waf-mtls-renew`: tags its output so renewals are greppable in the logs and can feed alerting.

The script walks every waf-to-*.crt in the certificate directory, inspects each one's expiry with step certificate inspect, and renews only those inside the threshold window. Certificates with time to spare are logged as healthy and skipped, so a run that renews nothing still records the full certificate inventory and its expiry dates. When any certificate is renewed, nginx is reloaded once at the end so the WAF picks up the new certificates. When nothing is renewed, the reload is skipped to avoid needless churn.

```
#!/bin/bash
set -euo pipefail

STEP_BIN="${STEP_BIN:-/usr/bin/step}"
CERT_DIR="${CERT_DIR:-/etc/ssl/certs}"
KEY_DIR="${KEY_DIR:-/etc/ssl/private}"
CA_URL="${CA_URL:-https://10.x.x.2:9000}"
RELOAD_CMD="${RELOAD_CMD:-systemctl reload nginx}"
RENEW_THRESHOLD_HOURS=504 # 21 days

log() {
```

```

logger -p local5.info -t waf-mtls-renew "$1"
echo "$(date -Is) [waf-mtls-renew] $1"
}

RENEWED_ANY=0
SKIPPED=()
NOW_EPOCH=$(date +%s)
THRESHOLD_SECONDS=$((RENEW_THRESHOLD_HOURS * 3600))

for CRT_PATH in "$CERT_DIR"/waf-to-*.crt; do
    [[ -e "$CRT_PATH" ]] || continue

    BASENAME=$(basename "$CRT_PATH" .crt)
    KEY_PATH="$KEY_DIR/${BASENAME}.key"

    EXPIRES_TS=$("$STEP_BIN" certificate inspect --format json "$CRT_PATH" \
        | jq -r '.validity.end')
    EXPIRES_EPOCH=$(date -d "$EXPIRES_TS" +%s)
    EXPIRES_HUMAN=$(date -d "$EXPIRES_TS" '+%Y-%m-%d %H:%M %Z')
    DAYS_LEFT=$(( (EXPIRES_EPOCH - NOW_EPOCH) / 86400 ))

    if (( EXPIRES_EPOCH - NOW_EPOCH < THRESHOLD_SECONDS )); then
        log "Renewing $BASENAME (expires $EXPIRES_HUMAN, ${DAYS_LEFT}d remaining)"

        "$STEP_BIN" ca renew \
            --ca-url "$CA_URL" \
            --force \
            "$CRT_PATH" "$KEY_PATH"

        log "Renewed $BASENAME"
        RENEWED_ANY=1
    else
        SKIPPED+=("$BASENAME|$EXPIRES_HUMAN|${DAYS_LEFT}d")
    fi
done

if [[ $RENEWED_ANY -eq 1 ]]; then
    log "Reloading nginx"
    $RELOAD_CMD
else

```

```
log "No renewals needed. Current certificate status:"
for ENTRY in "${SKIPPED[@]}"; do
    IFS='|' read -r NAME EXPIRY DAYS <<< "$ENTRY"
    log "  OK  $NAME – expires $EXPIRY ($DAYS remaining)"
done
fi
```

Three pieces divide the work cleanly. The timer decides when (weekly, with catch-up for missed runs), the service defines how it runs (once, sandboxed, logged), and the script does the actual work (inspect every client certificate, renew what's near expiry, reload nginx only if something changed). The design goal behind all three is that certificate renewal is observable and self-correcting. It logs what it did and what it skipped, it recovers from a missed run, and it renews with enough margin that no single failure reaches an outage. Expiry becomes a routine log entry rather than an incident.