Automating TLS certificate management in Docker

smallstep.com/blog/automate-docker-ssl-tls-certificates

25 October 2021

Carl Tashian

2021-10-25

follow smallstep on Twitter

As we set out to create our Practical Zero Trust guide to server TLS, we wanted to help DevOps folks automate certificate management for services that run in three different contexts: Linux, Docker, and Kubernetes.

Docker has proven to be the most difficult environment for certificate automation. While pure Linux services can leverage cron or systemd timers and clients like certbot for certificate renewal, and Kubernetes has packages like cert-manager for certificate management, Docker containers have minimal tooling around them. **There's no single answer for TLS certificate management in Docker**.

In this post, I'll share some patterns for automating TLS certificate management in Docker when using an internal PKI, and show several examples. The examples use our open-source `step` CLI tool, which connects to either your own `step-ca` server or a Smallstep Certificate Manager hosted CA, but many of the approaches here could be adapted for other CAs, or for Web PKI certificates.

"Why get certificates to containers at all? Just use a reverse proxy."

It's true: The simplest way to add TLS to Docker services is to not add TLS to Docker services. Just terminate TLS at a reverse proxy, and pass the traffic to containers on an internal network via HTTP.

This is a very common approach, because it works with almost any service. And the proxy server can also act as a load balancer, and it can aggregate services from several containers as well.

For a lot of people, this may be sufficient. But **a TLS-terminating proxy is not Zero Trust**.

Given that Zero Trust is a vague marketing term, let me be clear…

Zero Trust or BeyondProd approaches require authenticated and encrypted communications *everywhere*, including between services on internal networks.

With a reverse proxy in front of your Docker services, you're still running HTTP. You still have unauthenticated, unencrypted traffic flowing across your internal network.

That said, **there are some cases where a reverse proxy configuration can still be quite secure**. For example, you might have a containerized service that maps itself to a port on `localhost`, and you run a reverse proxy on the host side that adds TLS.

Finally, it's worth noting that TLS-terminating proxies preclude the addition of TLS client authentication or mutual TLS between your clients and servers, any you may want to add that down the road (for full Beast Mode Zero Trust).

And if you need load balancing, you can always use an L4 load balancer for TLS traffic.

"Why have an internal PKI? Just use Let's Encrypt certificates."

You could get Web PKI certificates for all of your Docker containers and internal services. We don't recommend this approach. **Your internal TLS certificate metadata will be published** in public Certificate Transparency logs. This opens you up to an infrastructure enumeration risk.

Requirements for Certificate Automation in Containers

When working with certificates in containers, you will need to decide how and where and when to handle each of these tasks for any service:

- **Bootstrapping**: Getting the container to trust your internal PKI.
- **Enrollment**: Authenticating the container with the CA and requesting a TLS certificate for the container
- **Renewal**: Renewing the container's certificate when it nears expiry
- **Deployment**: Moving the renewed certificate into place and signaling a certificate reload (or restarting the service)

Why is this so hard?

Docker runs in a lot of different contexts, even outside of the Kubernetes world. There's pure Docker containers running on VMs, lightweight orchestration environments like Docker Compose or Portainer, managed container environments like Amazon ECS, Heroku Runtime, Digital Ocean App Platform, or Azure Container Instances. There's also alternative container engines and runtimes, like Podman or gVisor.

Bottom line, there's no single place for certificate management to live in a Docker deployment:

- It could live in the host environment (eg. with a volume mount and a systemd renewal timer); but most managed PaaS environments disallow host access.

- It could be managed via a dedicated certificate management container and a shared volume mount; but most managed PaaS environments disallow shared volume mounts.
- It could happen in a custom image that adds certificate management to a service; but it's painful to build and maintain your own repository of custom images.

Meanwhile, how a service handles TLS certificates will favor one approach over another:

- Will the service reload its certificates at all? If not, the container will need to be restarted when the certificate is rotated; a custom image may not be sufficient.
- Does the service require a signal (eg. `SIGHUP`) to rotate certificates? If so, that signal will need to come from inside the container's namespace; a dedicated certificate management container may not be sufficient.
- Does the service store its certificates in a database instead of the filesystem? If so, a host-based volume mount solution may not be sufficient.

So, there's a lot of possibilities here—and no one-size-fits-all solution for certificate management in containers—but we've found that **building custom images is the most flexible approach**.

Doesn't that violate the principle of a container having just one responsibility?

I believe that if a service offers TLS support, **certificate management is part of the service**. Now that the ACME protocol has gained widespread adoption, it's easier than ever to build an ACME client into a service, and do full certificate management internally. Several services have done this. If everyone did it, this blog post would be much shorter.

> Speaking of built-in ACME, our own Mariano Cano put together an example Hello World gRPC service written in Go that has a built-in ACME client. Automatic TLS certificate management with ACME only added 40 lines of code compared to a non-ACME version of the service!

Bootstrapping: Trusting your CA from a container

Getting a container to trust your internal CA for the purpose of managing its own certificates is easy if you're running `step-ca` : Just run `step ca bootstrap` with the CA's URL and the root CA certificate's fingerprint, either when building the container, or when starting the container.

Example: Adding trust anchors when the container is built

Here's an example that bootstraps CA trust for a MongoDB server container. In this example, we use the `step` tool to download our root CA certificates from a `step-ca` server and install them.

```
FROM smallstep/step-cli as step
FROM mongo
COPY --from=step /usr/local/bin/step /usr/local/bin/
ARG CA_URL
ARG CA_FINGERPRINT
ENV CA_URL=${CA_URL} CA_FINGERPRINT=${CA_FINGERPRINT}
RUN step ca bootstrap --ca-url $CA_URL --fingerprint $CA_FINGERPRINT --install
```

Notice the `--install` flag. This adds your root CA to the container's certificate trust store. Use this flag if your container needs to run any sort of client application that uses TLS. If you're only running a server in the container, you probably won't need it.

`CA_URL` and `CA_FINGERPRINT` must be supplied as build arguments, eg.

```
$ docker build . --build-arg "CA_FINGERPRINT=c8de28e...620ecaa" \
        --build-arg "CA_URL=https://ca:4443/
```

Adding trust anchors on container startup

The example above might be all you need. But here are some common alternative approaches to adding trust anchors to a Docker container.

> With these alternatives, to use the `step` client within a container, you'll still want to add `--ca-url` and `--root` to your certificate requests, passing your CA's root certificate PEM file via the `--root` parameter.

1. Volume mount your CA certificate bundle

If you have the ability to use volume mounts, the simplest approach is to use a volume mount to inject your CA root certificate(s).

Confusingly, there are many trust store libraries and trust store locations in the filesystem, depending on which Docker base image you're using. While many popular Linux distributions have standardized around the `ca-certificates` package to manage their internal trust store, they don't all use the same paths for the trust store itself.

Unfortunately, there is no single place on a Linux system where you can reliably mount this volume, because **applications use several methods to access system CA certificates:**

- OpenSSL-based applications use the `/etc/ssl/certs` directory as the system trust store. If you don't want to run any extra commands, you can simply mount a PEM file containing your CA bundle to `/etc/ssl/certs/ca-certificates.crt` in the container, and the container will trust those CA certificates. Note that in this case, if you want to trust the Web PKI, your `ca-certificates.crt` file will have to include all Web PKI CAs as well as your own.
- NSS uses a hard-coded list of trusted CA certificates inside the `libnssckbi.so` library, and it stores user-supplied certificates in SQL databases inside `$HOME/.pki/nssdb` or `/etc/pki/nssdb`.
- Java applications use a special Java Keystore file.

- Some applications don't trust *any* system trust store, or they have their own independent trust store or need to be configured with a CA bundle filename. For example, servers that need to validate client certificates will require a CA bundle be passed in as a config parameter.

For base images that use standard `ca-certificates` paths

- Debian
- BusyBox
- Ubuntu
- Alpine

The recommended approach is to mount your pem file into `/usr/local/share/ca-certificates/`, and run the `update-ca-certificates` command inside the container. This will generate the system-managed `/etc/ssl/certs/ca-certificates.crt` list of trusted certificates.

Other base images

Arch Linux

Mount your PEM file into `/etc/ca-certificates/trust-source/anchors/`, then run `update-ca-trust extract` in the container to update the trust store.

RHEL 8

Put CA certificates into `/etc/pki/ca-trust/source/anchors/` or `/usr/share/pki/ca-trust-source/anchors/`, and run `update-ca-trust`.

Further reading on Linux trust anchors

- Storing Trust Policy by Stef Walter, Red Hat
- Gist: Trusted TLS Certificate Stores on Linux OS and Applications

2. Store your CA bundle on the web, and download it on container startup

This is a more complex approach, but it allows you to fetch the CA bundle dynamically when the container starts up.

> Perhaps surprisingly, most OS base images (with the exception of Alpine) do not ship with any CA certificates baked in. You usually have to install the `ca-certificates` package to get the Web PKI to become trusted by the container. I think the reasoning is that CA certificates expire, and some downstream images are very rarely updated, so it's better to leave their installation up to the user.

In the `Dockerfile`, install the `ca-certificates` package:

```
FROM ubuntu:focal

ARG CA_BUNDLE_URL=https://pki.example.com/ca-bundle.crt
ENV CA_BUNDLE_URL=${CA_BUNDLE_URL}
RUN apt update; \
    apt install -y --no-install-recommends \
        ca-certificates \
        curl; \
    mkdir -p /usr/local/share/ca-certificates; \
    rm -rf /var/lib/apt/lists/*

COPY ./init.sh /init.sh
ENTRYPOINT ["/init.sh"]
```

The assumption here is that your `CA_BUNDLE_URL` has a Web PKI certificate, not a private certificate, so we need to build the `ca-certificates` package into the container.

Then, on container startup, in the entrypoint script, download a fresh CA bundle and trust it:

```
curl -s -o /usr/local/share/ca-certificates/ca-bundle.crt "${CA_BUNDLE_URL}"
/usr/sbin/update-ca-certificates
```

`CA_BUNDLE_URL` can be baked into the `Dockerfile`, or supplied as arguments to `docker build`.

3. Embed a root CA in the image, and download a CA bundle on container startup

If you don't want to depend on the Web PKI, a third option is to **embed a single root CA into the image, and then download a full CA bundle** from an internal website when the container starts.

This approach updates the system-managed `/etc/ssl/certs/ca-certificates.crt` file. The expectation here is that the container will never install or run `update-ca-certificates`, which will rewrite that file.

For this approach, you'll need to bake the single URL and fingerprint of the root CA into the image that you will verify your internal website's TLS certificate. Here's a `Dockerfile`:

```
FROM ubuntu:focal

# Change these, or supply them via build args:
# CA_URL: The root CA certificate for pki.internal
# CA_FINGERPRINT: The SHA256 fingerprint of pki.internal's root CA cert
# CA_BUNDLE_URL: The full CA bundle to be downloaded and trusted
ARG CA_URL=https://pki.internal/root_ca.crt
ARG
CA_FINGERPRINT=c8de28e620ec4367f734a0c405d92d7350dbec351cec3e4f6a6d1fc9512387aa
ARG CA_BUNDLE_URL=https://pki.internal/ca-certificates.crt

ENV CA_URL=${CA_URL} CA_FINGERPRINT=${CA_FINGERPRINT}
CA_BUNDLE_URL=${CA_BUNDLE_URL}
RUN apt update; \
    apt install -y --no-install-recommends \
        curl \
        jq \
        openssl \
    ; \
    rm -rf /var/lib/apt/lists/*

COPY ./init.sh /init.sh
ENTRYPOINT ["/init.sh"]
```

Then, in the `init.sh` entrypoint script, download and install the full CA bundle:

```
if [ ! -f /etc/ssl/certs/ca-certificates.crt ]; then
  # Insecurely download the root certificate for the CA_URL server
  curl -ks -o /tmp/root-ca.crt "${CA_URL}"

  # Confirm the certificate matches the hardcoded fingerprint
  fingerprint=$(openssl x509 -in /tmp/root-ca.crt -noout -sha256 -fingerprint \
               | tr -d ":" \
               | cut -d "=" -f 2 \
               | tr "[:upper:]" "[:lower:]")

  if [[ "$fingerprint" != "$CA_FINGERPRINT" ]]; then
    echo >&2
    echo >&2 "error: CA certificate fingerprint $fingerprint does not match
expected value ${CA_FINGERPRINT}"
    echo >&2
    exit 1
  fi

  # Now download the full CA bundle, without -k
  curl -s --cacert /tmp/root-ca.crt -o /etc/ssl/certs/ca-certificates.crt
"${CA_BUNDLE_URL}"
fi
```

The variables `CA_URL`, `CA_FINGERPRINT`, and `CA_BUNDLE_URL` can be baked into the Dockerfile, or supplied as arguments to `docker build`.

Enrollment with a CA

Once the container trusts our root CA, the next question is: How will the CA trust the container to issue it a TLS certificate? There's a few options for authenticating a container to a CA:

1. Single-use CA token enrollment

`step-ca` allows for the creation of single-use JWTs that the `step` client can use to get a TLS certificate. Valid for 5 minutes, a CA token is designed to be securely generated in one context (eg. on the host machine), then passed into the context where it will be used (eg. a Docker container). The token is then used to authenticate with the CA and sign a certificate.

2. ACME enrollment

ACME is the protocol used by Let's Encrypt, and it's supported by `step-ca` too. The client configuration can be as simple as a CA URL and a root CA certificate.

An ACME server authenticates its clients via challenges. The client must prove to the ACME CA that it controls the hostnames being requested in the certificate (by setting up a server on port 80 or 443 that can answer an HTTP or TLS-based challenges from the CA), or that it controls the DNS records for those hostnames (by creating a special DNS record to answer a challenge from the CA).

While ACME Is widely supported, it can be tricky to deploy internally. For an HTTP challenge, the CA needs to be able to reach port 80 on the hostnames in the certificate request. Because containers live inside their own network namespace and may not share the same IP address as the host, the host may not be able to easily get certificates on behalf of its containers using ACME.

For DNS challenges, the benefit is that the CA doesn't need to able to reach the container directly. But the downside is that the ACME client needs to be able to edit DNS records. The setup for it depends on your DNS provider's API and is beyond the scope of this post.

For an HTTP challenge, you could map port 80 in the container to a different host port, and run a reverse proxy on the host that passes ACME challenges from the `/.well-known/acme-challenge/<TOKEN>` endpoint into an ACME client in the container that's requesting a certificate.

For certificates that will be used for internal traffic within a bridge or overlay network, you could run an CA or an RA server within your container network that offers ACME to containers. A CA in this context will be able to resolve internal container DNS names and reach the container without needing a port mapping.

3. Bake a birth certificate into the image

A long-lived certificate and private key could be added to the image and used to authenticate to the CA, using `step-ca`'s x5c provisioner. This special certificate (sometimes called an "identity certificate" or a "birth certificate") would have a separate root CA that's specific to the x5c provisioner configuration, and could be used when the container starts up, to get a server certificate for the container from the CA.

Renewal

Once you have bootstrapping and enrollment worked out, the next step is to get the certificate renewed when it reaches two-thirds of its lifetime, and deploy the renewed certificate to the service.

The simplest answer is to use `step-ca`'s certificate-based renewal API. The `step ca renew` command will renew a certificate with the CA using Mutual TLS (mTLS), where the certificate-to-be-renewed and its key are used for authentication.

Alternatively, you can use ACME for renewals. `certbot` and other ACME clients can renew certificates using ACME.

Bringing it all together

Option 1: Fully self-contained certificate management

It would be great if a single container could run the service *and* handle certificate management.

We believe certificate management should be a concern of any service that supports TLS! And, in fact, some modern services now ship with a built-in ACME client.

To add certificate management to non-ACME services in a self-contained way, we need to create a custom Docker image on top of the official image for the service, overriding the `ENTRYPOINT` script and adding a Bash "shim" script that can handle all of the cert management tasks listed above.

When to use it

The self-contained approach supports the broadest range of environments where containers are run. While it requires building a custom Docker image, it solves the problems of root distribution, enrollment, renewal, and deployment without requiring volume mounts or changes to the host environment.

- You're comfortable building a custom Docker image for the service
- You don't have access to the host environment
- You can't use shared volume mounts
- You really want the service to run in a single container
- The service does not need to be restarted to reload its certificates.

Example: MongoDB server

See the complete example files in GitHub.

Here's a Docker image that adds certificate management (using a `step-ca` Certificate Authority) to the official MongoDB image. This example features:

- A custom image with a baked-in root CA certificate
- ACME enrollment on container startup
- Renewal via the `step ca renew` daemon

- Deployment via MongoDB’s `rotateCertificates` command

The Dockerfile bootstraps with the root CA:

```
FROM smallstep/step-cli as step
FROM mongo
# https://github.com/docker-library/mongo/blob/master/Dockerfile-linux.template
# mongo image comes with jq installed
COPY --from=step /usr/local/bin/step /usr/local/bin/
ARG CA_URL
ARG CA_FINGERPRINT
ENV CA_URL=${CA_URL} CA_FINGERPRINT=${CA_FINGERPRINT} STEPPATH=/.step TLS_PEM_LO
CATION=/run/tls/server.pem TLS_CERT_LOCATION=/run/tls/server.crt TLS_KEY_LOCATIO
N=/run/tls/server.key TLS_CA_CERT_LOCATION=/run/tls/ca.crt
RUN step ca bootstrap --ca-url $CA_URL --fingerprint $CA_FINGERPRINT --install;
\
    mkdir -p /run/tls; \
    cp /.step/certs/root_ca.crt /run/tls/ca.crt; \
    chown -R mongodb:mongodb /run/tls /.step

COPY entrypoint-shim.sh cert-*.sh /usr/local/bin/
ENTRYPOINT ["entrypoint-shim.sh"]
```

The entrypoint “shim” script enrolls the host with `step-ca` and starts up a certificate renewal daemon that watches the certificate and triggers renewal when it has reached two-thirds of its lifetime. Finally, the entrypoint “shim” script passes control to `mongo` ’s official entrypoint script.

When the certificate renewal daemon renews the certificate, it will run a script to trigger certificate rotation in MongoDB:

```
#!/bin/bash

cat $TLS_CERT_LOCATION $TLS_KEY_LOCATION > $TLS_PEM_LOCATION
chown mongodb:mongodb $TLS_PEM_LOCATION
mongosh --tlsAllowInvalidCertificates \
     "mongodb://localhost:27017?
tls=true&tlsCAFile=${TLS_CA_CERT_LOCATION}&tlsCertificateKeyFile=${TLS_PEM_LOCATION
 \
     -f <(echo "db.adminCommand( { rotateCertificates: 1 } )")
```

This is the piece that needs to be customized for a given service. In the case of MongoDB, the renewed certificate and private key need to be concatenated together, and then MongoDB needs to be sent a `rotateCertificates` command to reload the file.

Other Examples

We have a couple other self-contained examples in our docker-tls GitHub repo:

- Nginx Unit uses an API to manage its certificates, so the enrollment and renewal scripts have to account for that.
- Redis is similar to MongoDB in that it has a command for hot rotation of certificates.

If you're already building a custom image, root distribution is easy: the root CA certificate can be baked into the image when it's built.

The big downside here is that you have to modify the official container image for your service, rather than just running it as-is. For many, that's just not a feasible option.

Option 2: Host-based certificate management

With host-based certificate management, the unmodified, official Docker image for the service can be used. The host is responsible for renewing certificates and triggering a reload of certificate files (or restarting the containers) when they are renewed. Certificates can be renewed from the host via volume mounts, or copied into the container.

- The easiest option for host-based certificate management is to use a volume mount to store your server certificates, and the `step ca renew` command with a systemd-based renewal timer or a cron-based renewal, which we have documented in Production Considerations.

- You could also map the container service to a port on localhost, and run a reverse proxy on the host that does its own certificate management (Caddy, for example).

- If you need ACME support via the host, one approach we've seen is to sneak into a container's network namespace and run an ACME client binary (`certbot` , `acme.sh` , or `step`) in order to respond to ACME `HTTP-01` or `TLS-ALPN-01` challenges. The `nsenter` command in Linux can facilitate this. On the host, you'd run something like:

  ```
  id="$(docker ps --filter 'name=my_container' --format '{{.ID}}')"
  netns="$(docker inspect "$id" --format '{{.NetworkSettings.SandboxKey}}')"
  nsenter --net --target "$netns" certbot certonly --deploy-hook "cp
  \"$RENEW_LINEAGE/{fullchain,privkey}.pem\" \"$PWD/certificates\"; ..."
  ```

  Source: this GitHub Issue

When to use it

This approach is especially useful when:

- You want to use unmodified, official Docker images (the image may still need to be augmented to trust your root CA certificate)
- The service doesn't support hot reloading of certificates. In this case, the container needs to be restarted from the host side.
- You don't want to pass any CA authentication credentials into the container
- You already use a systemd service to spin up the container

Option 3: Multi-container certificate management

There's a couple ways this could be done, both of which are beyond the scope of this post.

- If your service allows triggering the reloading of certificates via an API, or if it reloads certificate files with each new connection, you could have a sidecar container that only manages the certificate files. It would run `step ca renew --daemon`. This sidecar could share a volume mount with the service container, and enroll and renew the certificate files as needed.
- If the service needs to be restarted in order to reload its certificates, you could potentially use Docker Socket Proxy to give a sidecar container the ability to restart the service. We haven't tested this but it seems promising.
- Similar to a services mesh like Istio, you could set up an internal reverse proxy that handles TLS for each service, and configure that with certificate management. The reverse proxy and the service live together in the same network namespace, and only the reverse proxy exposes its traffic to the outside.

When to use it

This approach is useful when:

- You want to use unmodified, official Docker images (the image may still need to be augmented to trust your root CA certificate)
- The service supports reloading triggered via an API
- A shared volume mount can be used to store the certificate files

Wrapping up

Certificate management in Docker is not easy, but I hope this guide has helped you in some way. Have you found a novel way to handle certificates in Docker containers? Something that we haven't covered here? We'd love to hear about it. Hit us up on our Twitter, open a GitHub Discussion, or join our Discord to chat with the Smallstep community!

Technical