Certificates for localhost

letsencrypt.org/docs/certificates-for-localhost

Last updated: Dec 21, 2017 | See all Documentation

Sometimes people want to get a certificate for the hostname “localhost”, either for use in local development, or for distribution with a native application that needs to communicate with a web application. Let’s Encrypt can’t provide certificates for “localhost” because nobody uniquely owns it, and it’s not rooted in a top level domain like “.com” or “.net”. It’s possible to set up your own domain name that happens to resolve to `127.0.0.1`, and get a certificate for it using the DNS challenge. However, this is generally a bad idea and there are better options.

For local development

If you’re developing a web app, it’s useful to run a local web server like Apache or Nginx, and access it via `http://localhost:8000/` in your web browser. However, web browsers behave in subtly different ways on HTTP vs HTTPS pages. The main difference: On an HTTPS page, any requests to load JavaScript from an HTTP URL will be blocked. So if you’re developing locally using HTTP, you might add a script tag that works fine on your development machine, but breaks when you deploy to your HTTPS production site. To catch this kind of problem, it’s useful to set up HTTPS on your local web server. However, you don’t want to see certificate warnings all the time. How do you get the green lock locally?

The best option: Generate your own certificate, either self-signed or signed by a local root, and trust it in your operating system’s trust store. Then use that certificate in your local web server. See below for details.

For native apps talking to web apps

Sometimes developers want to offer a downloadable native app that can be used alongside a web site to offer extra features. For instance, the Dropbox and Spotify desktop apps scan for files from across your machine, which a web app would not be allowed to do. One common approach is for these native apps to offer a web service on localhost, and have the web app make requests to it via XMLHTTPRequest (XHR) or WebSockets. The web app almost always uses HTTPS, which means that browsers will forbid it from making XHR or WebSockets requests to non-secure URLs. This is called Mixed Content Blocking. To communicate with the web app, the native app needs to provide a secure web service.

Fortunately, modern browsers consider `http://127.0.0.1:8000/` to be a “potentially trustworthy” URL because it refers to a loopback address. Traffic sent to `127.0.0.1` is guaranteed not to leave your machine, and so is considered automatically secure against network interception. That means if your web app is HTTPS, and you offer a native app

web service on `127.0.0.1`, the two can happily communicate via XHR. Unfortunately, localhost doesn't yet get the same treatment. Also, WebSockets don't get this treatment for either name.

You might be tempted to work around these limitations by setting up a domain name in the global DNS that happens to resolve to `127.0.0.1` (for instance, `localhost.example.com`), getting a certificate for that domain name, shipping that certificate and corresponding private key with your native app, and telling your web app to communicate with `https://localhost.example.com:8000/` instead of `http://127.0.0.1:8000/`. *Don't do this.* It will put your users at risk, and your certificate may get revoked.

By introducing a domain name instead of an IP address, you make it possible for an attacker to Man in the Middle (MitM) the DNS lookup and inject a response that points to a different IP address. The attacker can then pretend to be the local app and send fake responses back to the web app, which may compromise your account on the web app side, depending on how it is designed.

The successful MitM in this situation is possible because in order to make it work, you had to ship the private key to your certificate with your native app. That means that anybody who downloads your native app gets a copy of the private key, including the attacker. This is considered a compromise of your private key, and your Certificate Authority (CA) is required to revoke your certificate if they become aware of it. Many native apps have had their certificates revoked for shipping their private key.

Unfortunately, this leaves native apps without a lot of good, secure options to communicate with their corresponding web site. And the situation may get trickier in the future if browsers further tighten access to localhost from the web.

Also note that exporting a web service that offers privileged native APIs is inherently risky, because web sites that you didn't intend to authorize may access them. If you go down this route, make sure to read up on Cross-Origin Resource Sharing, use Access-Control-Allow-Origin, and make sure to use a memory-safe HTTP parser, because even origins you don't allow access to can send preflight requests, which may be able to exploit bugs in your parser.

Making and trusting your own certificates

Anyone can make their own certificates without help from a CA. The only difference is that certificates you make yourself won't be trusted by anyone else. For local development, that's fine.

The simplest way to generate a private key and self-signed certificate for localhost is with this openssl command:

```
openssl req -x509 -out localhost.crt -keyout localhost.key \
  -newkey rsa:2048 -nodes -sha256 \
  -subj '/CN=localhost' -extensions EXT -config <( \
   printf "[dn]\nCN=localhost\n[req]\ndistinguished_name =
dn\n[EXT]\nsubjectAltName=DNS:localhost\nkeyUsage=digitalSignature\nextendedKeyUsag
```

You can then configure your local web server with localhost.crt and localhost.key, and install localhost.crt in your list of locally trusted roots.

If you want a little more realism in your development certificates, you can use minica to generate your own local root certificate, and issue end-entity (aka leaf) certificates signed by it. You would then import the root certificate rather than a self-signed end-entity certificate.

You can also choose to use a domain with dots in it, like `www.localhost`, by adding it to /etc/hosts as an alias to `127.0.0.1`. This subtly changes how browsers handle cookie storage.