





Andrew Lock | .NET Escapades

[Home](#)
[About](#)
[Subscribe](#)
[Dark](#)

Sponsored by **MailBee.NET Objects**—send, receive, process email and Outlook file formats in .NET apps.
Now with **TLS 1.3 support**. [↗](#)

October 03, 2017 in [ASP.NET CORE](#) [ASP.NET CORE 2.0](#) [SECURITY](#) ~ 6 min read.

Creating and trusting a self-signed certificate on Linux for use in Kestrel and ASP.NET Core

Share on: [f](#) [t](#) [g](#) [in](#)

These days, running your apps over HTTPS is pretty much required, so you need an SSL certificate to encrypt the connection between your app and a user's browser.

I was recently trying to create a self-signed certificate for use in a Linux development environment, to serve requests with ASP.NET Core over SSL when developing locally. Playing with certs is always harder than I think it's going to be, so this post describes the process I took to create and trust a self-signed cert.

Disclaimer I'm very much a Windows user at heart, so I can't give any guarantees as to whether this process is correct. It's just what I found worked for me!

Using Open SSL to create a self-signed certificate

On Windows, creating a self-signed development certificate for development is often not necessary - Visual Studio automatically creates a development certificate for use with IIS Express, so if you run your apps this way, then you shouldn't have to deal with certificates directly.

On the other hand, if you want to host Kestrel directly over HTTPS, then you'll need to work with certificates directly one way or another. On Linux, you'll either need to create a cert for Kestrel to use, or for a reverse-proxy like Nginx or HAProxy. After much googling, I took the approach described in this post.


Creating a basic certificate using openssl

Creating a self-signed cert with the `openssl` library on Linux is theoretically pretty simple. My first attempt was to use a script something like the following:

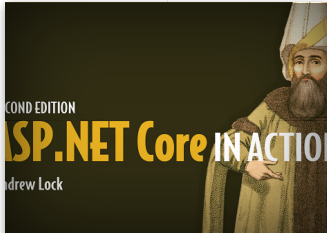
```
openssl req -new -x509 -newkey rsa:2048 -keyout localhost.key -out localhost.cer
openssl pkcs12 -export -out localhost.pfx -inkey localhost.key -in localhost.cer
```

This creates 3 files:

- `localhost.cer` - The public key for the SSL certificate





What if Your Project Management Tool Was Fast and Intuitive? Try **Shortcut** (formerly Clubho) ADS VIA CARBON



My new book *ASP.NET Core in Action, Second Edition* is available now! It supports .NET 5.0, and is available as an eBook or paperback. You even get a free copy of the first edition of *ASP.NET Core in Action*!

ENJOY THIS BLOG?

 Buy me a coffee





The script creates a certificate with a "Common Name" for the `localhost` domain (the `-subj /CN=localhost` part of the script). That means we can use it to secure connections to the `localhost` domain when developing locally.

The problem with this certificate is that it only includes a common name so [the latest Chrome versions will not trust it](#). Instead, we need to create a certificate with a [Subject Alternative Name \(SAN\)](#) for the DNS record (i.e. `localhost`).

The easiest way I found to do this was to use a `.conf` file containing all our settings, and to pass it to `openssl`.

Creating a certificate with DNS SAN

The following file shows the `.conf` config file that specifies the particulars of the certificate that we're going to create. I've included all of the details that you must specify when creating a certificate, such as the company, email address, location etc.

If you're creating your own self signed certificate, be sure to change these details, and to add any extra DNS records you need.

```
[ req ]
prompt            = no
default_bits      = 2048
default_keyfile   = localhost.pem
distinguished_name = subject
req_extensions    = req_ext
x509_extensions  = x509_ext
string_mask       = utf8only

# The Subject DN can be formed using X501 or RFC 4514 (see RFC 4519 for a desc
# Its sort of a mashup. For example, RFC 4514 does not provide emailAddress.
[ subject ]
countryName       = GB
stateOrProvinceName = London
localityName      = London
organizationName  = .NET Escapades

# Use a friendly name here because its presented to the user. The server's DNS
# names are placed in Subject Alternate Names. Plus, DNS names here is depre
# by both IETF and CA/Browser Forums. If you place a DNS name here, then you
# must include the DNS name in the SAN too (otherwise, Chrome and others tha
# strictly follow the CA/Browser Baseline Requirements will fail).
commonName        = localhost dev cert
emailAddress       = test@test.com

# Section x509_ext is used when generating a self-signed certificate. I.e., op
[ x509_ext ]

subjectKeyIdentifier      = hash
authorityKeyIdentifier    = keyid,issuer

# You only need digitalSignature below. *If* you don't allow
# RSA Key transport (i.e., you use ephemeral cipher suites), then
# omit keyEncipherment because that's key transport.
basicConstraints          = CA:FALSE
keyUsage                  = digitalSignature, keyEncipherment
subjectAltName             = @alternate_names
nsComment                 = "OpenSSL Generated Certificate"

# RFC 5280, Section 4.2.1.12 makes EKU optional
# CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
# In either case, you probably only need serverAuth.
# extendedKeyUsage        = serverAuth, clientAuth

# Section req_ext is used when generating a certificate signing request. I.e.,
[ req_ext ]

subjectKeyIdentifier      = hash

basicConstraints          = CA:FALSE
keyUsage                  = digitalSignature, keyEncipherment
subjectAltName             = @alternate_names
nsComment                 = "OpenSSL Generated Certificate"
```







```
# RFC 5280, Section 4.2.1.12 makes EKU optional
# CA/Browser Baseline Requirements, Appendix (B)(3)(G) makes me confused
# In either case, you probably only need serverAuth.
# extendedKeyUsage = serverAuth, clientAuth

[ alternate_names ]

DNS.1      = localhost

# Add these if you need them. But usually you don't want them or
# need them in production. You may need them for development.
# DNS.5     = localhost
# DNS.6     = localhost.localdomain
# DNS.7     = 127.0.0.1

# IPv6 localhost
# DNS.8     = ::1
```

We save this config to a file called `localhost.conf`, and use it to create the certificate using a similar script as before. Just run this script in the same folder as the `localhost.conf` file.

```
openssl req -config localhost.conf -new -x509 -sha256 -newkey rsa:2048 -nodes
-keyout localhost.key -days 3650 -out localhost.crt
openssl pkcs12 -export -out localhost.pfx -inkey localhost.key -in localhost.c
```

This will ask you for an export password for your pfx file. Be sure that you provide a password and keep it safe - ASP.NET Core requires that you don't leave the password blank. You should now have an X509 certificate called `localhost.pfx` that you can use to add HTTPS to your app.



Before we use the certificate in our apps, we need to trust it on our local machine. Exactly how you go about this [varies depending on which flavour of Linux](#) you're using. On top of that, some apps seem to use their own certificate stores, so trusting the cert globally won't necessarily mean it's trusted in all of your apps.

The following example worked for me on Ubuntu 16.04, and kept Chrome happy, but I had to explicitly add an exception to Firefox when I first used the cert.

```
#Install the cert utils
sudo apt install libnss3-tools
# Trust the certificate for SSL
pk12util -d sql:$HOME/.pki/nssdb -i localhost.pfx
# Trust a self-signed server certificate
certutil -d sql:$HOME/.pki/nssdb -A -t "P,," -n 'dev cert' -i localhost.crt
```

As I said before, I'm not a Linux guy, so I'm not entirely sure if you need to run both of the trust commands, but I did just in case! If anyone knows a better approach I'm all ears :)

We've now created a self-signed certificate with a DNS SAN name for `localhost`, and we trust it on the development machine. The last thing remaining is to use it in our app.

Configuring Kestrel to use your self-signed certificate

For simplicity, I'm just going to show how to load the `localhost.pfx` certificate in your app from the `.pfx` file, and how to configure Kestrel to use it to serve requests over HTTPS. I've hard-coded the `.pfx` password in this example for simplicity, but you should load it from configuration instead.

Warning You should never include the password directly like this in a production app.

The following example is for ASP.NET Core 2.0 - [Shawn Wildermuth has an example](#) of how to add SSL in ASP.NET Core 1.X (as well as how to create a self-signed cert on Windows).

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHost(args).Run();
    }

    public static IWebHost BuildWebHost(string[] args) =>
        return WebHost.CreateDefaultBuilder()
            .UseKestrel(options =>
            {
                // Configure the Url and ports to bind to
                // This overrides calls to UseUrls and the ASPNETCORE_URLS env
                // overridden if you call UseIisIntegration() and host behind
                options.Listen(IPAddress.Loopback, 5001);
                options.Listen(IPAddress.Loopback, 5002, listenOptions =>
                {
                    listenOptions.UseHttps("localhost.pfx", "testpassword");
                });
            })
            .UseStartup<Startup>()
            .Build();
}
```

Although `CreateDefaultBuilder()` adds Kestrel to the app anyway, you can call `UseKestrel()` again and specify additional options. Here we are defining two URLs and ports to listen on (The `IPAddress.Loopback` address corresponds to `localhost` or `127.0.0.1`):

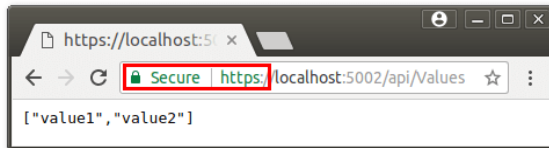
- <http://localhost:5001> - An unsecured end point
- <https://localhost:5002> - Secured using our SSL cert

We add HTTPS to the second `Listen()` call with the `UseHttps()` extension method. There are





If everything is configured correctly, you should be able to view the app in Chrome, and see a nice, green, Secure padlock:



As I said at the start of this post, I'm not 100% on all of this, so if anyone has any suggestions or improvements, please let me know in the comments.

Resources

- [The Most Common OpenSSL Commands](#)
- [Chrome deprecates subject CN matching](#)
- [How to create a self-signed certificate with openssl?](#)

FOLLOW ME

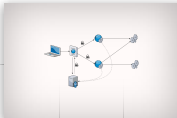


ENJOY THIS BLOG?



PREVIOUS

Using anonymous types and tuples to attach correlation IDs to scope state with Serilog and Seq in ASP.NET Core



NEXT

Debugging JWT validation problems between an OWIN app and IdentityServer4

Loading comments powered by Disqus, please wait...