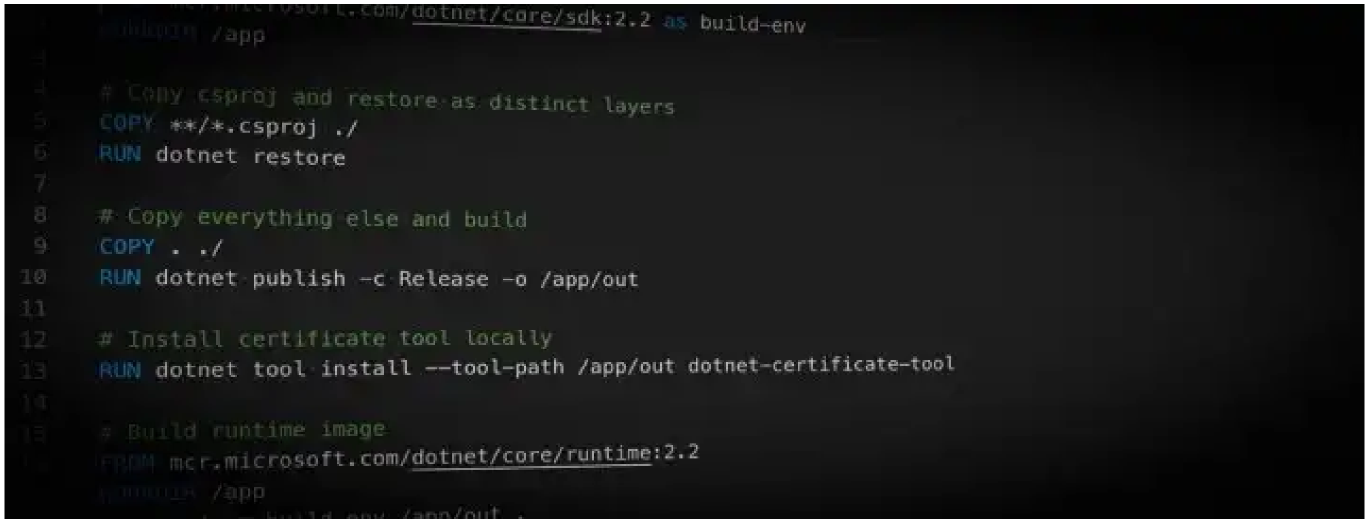
Published in GSoft Tech

Yohan Belval Follow
Oct 9, 2019 · 3 min read · Listen

Certificates in .NET Core on Linux and Docker


```
com/dotnet/core/sdk:2.2 as build-env
WORKDIR /app

# Copy csproj and restore as distinct layers
COPY **/*.csproj ./
RUN dotnet restore

# Copy everything else and build
COPY . ./
RUN dotnet publish -c Release -o /app/out

# Install certificate tool locally
RUN dotnet tool install --tool-path /app/out dotnet-certificate-tool

# Build runtime image
FROM mcr.microsoft.com/dotnet/core/runtime:2.2
WORKDIR /app
```

I recently wanted to run a .NET Core application on Docker and ran into a little challenge. You see, the application needs to authenticate with an Azure Active Directory Application with Certificate Credentials.

Disclaimer: I am in no stretch of the imagination a savant in the field of cryptography nor do I know Linux and Docker inside out. I'm simply a .NET C# developer who has embraced the opportunity to explore different platforms 🙅.

Cross-Platform Challenges

Historically, the .NET framework was written only with the Windows platform in mind. Now with .NET Core, porting the framework's functionality to other platforms presents significant challenges and some paradigm shifts such as *how to handle certificates.*

For example, in Windows, the certificate store has the concept of *Local Machine* and *Current User* locations which doesn't quite translate to the way Linux stores certificates. There is no *Current User* store in Linux, which begs the question; **What happens when I try to access it with .NET Core on Linux like this?**

```
new X509Store(StoreName.My, StoreLocation.CurrentUser);
```

Well, it turns out the fallback is to search in a custom location that is handled by the .NET Core framework: *~/.dotnet/corefx/cryptography/x509stores/my*. For more information, please consult this github issue.

That being said, adding a certificate to the *Current User* store has to be done programmatically with the .NET Core framework (adding the certificate file manually to that custom location is not a recommended approach).

.NET Core Global Tools — WIN!

Since programmatically adding certificates is a bit of a pain in the butt, I decided to develop a little console application to help with the process. My colleague suggested I should publish it as a .NET Core global tool to further simplify the installation/usage process — good idea!

Now all you have to do to use it is:

```
# Install the tool
dotnet tool install --global dotnet-certificate-tool

#Use it like so
certificate-tool add --file ./cert.pfx --password xxx
```

And you're off 🏃‍♀️! Fo more information on how to add/remove certificates with the tool, have a look at the repository:

gsoft-inc/dotnet-certificate-tool

Command line tool to install and remove pfx certificates from the current user's store. Mainly used to workaround the…

github.com

Using it with Docker

There are a few “gotchas” when using it with Docker.

First off, the .NET global tool CLI is only available with the SDK, which means that if your image depends on a runtime images (which it should), you won’t be able to install and use the tool.

To work around this limitation, you can use Docker multi-stage builds to use the SDK to install the tool locally (to a specific folder) and then copy it to your runtime image like so:

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.2 as build-env

WORKDIR /app
...

# Install certificate tool locally
RUN dotnet tool install --tool-path ./ dotnet-certificate-tool

# Build runtime image
FROM mcr.microsoft.com/dotnet/core/runtime:2.2

WORKDIR /app

# Make sure to copy the tool (and other stuff) to runtime
COPY --from=build-env /app .

ENTRYPOINT ["./docker-entrypoint.sh"]
```

In order to install the certificate when initializing the container, the image entry point needs to do the work. For example, a bash script could do the following:

```
#!/usr/bin/env bash

# Install certificate
./certificate-tool add --file $1 --password $2

# Then you can run the app
dotnet ./ConsoleApp.dll
```

And the call the Docker run CLI with the necessary arguments:

```
docker run -v /cert-directory:/app my-image /app/mycert.pfx 'the
password'
```

Tada 🎉 you now have a .NET Core application running in Docker which can load a certificate from the *X509Store* in your code. Hopefully this little tool will have saved you some wasted time. Cheers!

Note: For a full example of the previous notions, see the ‘docker-example’ directory in the certificate tool repository.

82 | 3

Docker Net Core Security Best Practices Linux

Thanks to Alexandra Gifuni