Windows - Code signing guide locally & with GitHub Actions

Intro

Code signing your application lets users know that they downloaded the official executable of your app and not some 3rd party malware that poses as your app. While it is not required, it improves users' confidence in your app.

DANGER

This guide only applies to OV code signing certificates acquired before June 1st 2023! For code signing with EV certificates and OV certificates received after that date please consult the documentation of your certificate issuer instead.

Prerequisites

- Windows - you can likely use other platforms, but this tutorial uses Powershell native features.
- A working Tauri application
- Code signing certificate - you can acquire one of these on services listed in Microsoft's docs. There are likely additional authorities for non-EV certificates than included in that list, please compare them yourself and choose one at your own risk.
  - Please make sure to get a **code signing** certificate, SSL certificates do not work!

This guide assumes that you have a standard code signing certificate> If you have an EV certificate, which generally involves a hardware token, please follow your issuer's documentation instead.

NOTE

If you sign the app with an EV Certificate, it'll receive an immediate reputation with Microsoft SmartScreen and won't show any warnings to users.

> If you opt for an OV Certificate, which is generally cheaper and available to individuals, Microsoft SmartScreen will still show a warning to users when they download the app. It might take some time until your certificate builds enough reputation. You may opt for submitting your app to Microsoft for manual review. Although not guaranteed, if the app does not contain any malicious code, Microsoft may grant additional reputation and potentially remove the warning for that specific uploaded file.

Getting Started

There are a few things we have to do to get Windows prepared for code signing. This includes converting our certificate to a specific format, installing this certificate, and decoding the required information from the certificate.

A. Convert your `.cer` to `.pfx`

1. You will need the following:
   - certificate file (mine is `cert.cer`)
   - private key file (mine is `private-key.key`)
2. Open up a command prompt and change to your current directory using `cd Documents/Certs`
3. Convert your `.cer` to a `.pfx` using `openssl pkcs12 -export -in cert.cer -inkey private-key.key -out certificate.pfx`
4. You should be prompted to enter an export password **DON'T FORGET IT!**

B. Import your `.pfx` file into the keystore.

We now need to import our `.pfx` file.

1. Assign your export password to a variable using `$WINDOWS_PFX_PASSWORD = 'MYPASSWORD'`
2. Now Import the certificate using `Import-PfxCertificate -FilePath certificate.pfx -CertStoreLocation Cert:\CurrentUser\My -Password (ConvertTo-SecureString -String $WINDOWS_PFX_PASSWORD -Force -AsPlainText)`

C. Prepare Variables

1. Start ➡️ `certmgr.msc` to open Personal Certificate Management, then open Personal/Certificates.

2. Find the certificate we just imported and double-click on it, then click on the Details tab.

3. The Signature hash algorithm will be our `digestAlgorithm`. (Hint: this is likely `sha256`)

4. Scroll down to Thumbprint. There should be a value like `A1B1A2B2A3B3A4B4A5B5A6B6A7B7A8B8A9B9A0B0`. This is our `certificateThumbprint`.

5. We also need a timestamp URL; this is a time server used to verify the time of the certificate signing. I'm using `http://timestamp.comodoca.com`, but whoever you got your certificate from likely has one as well.

Prepare `tauri.conf.json` file

1. Now that we have our `certificateThumbprint`, `digestAlgorithm`, & `timestampUrl` we will open up the `tauri.conf.json`.

2. In the `tauri.conf.json` you will look for the `tauri` -> `bundle` -> `windows` section. You see, there are three variables for the information we have captured. Fill it out like below.

```
"windows": {
        "certificateThumbprint": "A1B1A2B2A3B3A4B4A5B5A6B6A7B7A8B8A9B9A0B0",
        "digestAlgorithm": "sha256",
        "timestampUrl": "http://timestamp.comodoca.com"
}
```

3. Save and run `yarn | yarn build`

4. In the console output, you should see the following output.

```
info: signing app
info: running signtool "C:\\Program Files (x86)\\Windows
Kits\\10\\bin\\10.0.19041.0\\x64\\signtool.exe"
info: "Done Adding Additional Store\r\nSuccessfully signed: APPLICATION FILE PATH HERE
```

Which shows you have successfully signed the `.exe`.

And that's it! You have successfully signed your .exe file.

BONUS: Sign your application with GitHub Actions.

We can also create a workflow to sign the application with GitHub actions.

GitHub Secrets

We need to add a few GitHub secrets for the proper configuration of the GitHub Action. These can be named however you would like.

- You can view the encrypted secrets guide on how to add GitHub secrets.

The secrets we used are as follows

GitHub Secrets	Value for Variable
WINDOWS_CERTIFICATE	Base64 encoded version of your .pfx certificate, can be done using this command `certutil -encode certificate.pfx base64cert.txt`
WINDOWS_CERTIFICATE_PASSWORD	Certificate export password used on creation of certificate .pfx

Workflow Modifications

1. We need to add a step in the workflow to import the certificate into the Windows environment. This workflow accomplishes the following

    i. Assign GitHub secrets to environment variables
    ii. Create a new `certificate` directory
    iii. Import `WINDOWS_CERTIFICATE` into tempCert.txt
    iv. Use `certutil` to decode the tempCert.txt from base64 into a `.pfx` file.
    v. Remove tempCert.txt
    vi. Import the `.pfx` file into the Cert store of Windows & convert the `WINDOWS_CERTIFICATE_PASSWORD` to a secure string to be used in the import command.

2. We will be using the `tauri-action` publish template.

```yaml
name: 'publish'
on:
  push:
    branches:
      - release

jobs:
  publish-tauri:
    strategy:
      fail-fast: false
      matrix:
        platform: [macos-latest, ubuntu-latest, windows-latest]

    runs-on: ${{ matrix.platform }}
    steps:
      - uses: actions/checkout@v2
      - name: setup node
        uses: actions/setup-node@v1
        with:
          node-version: 12
      - name: install Rust stable
        uses: actions-rs/toolchain@v1
        with:
          toolchain: stable
      - name: install webkit2gtk (ubuntu only)
        if: matrix.platform == 'ubuntu-latest'
        run: |
          sudo apt-get update
          sudo apt-get install -y webkit2gtk-4.0
      - name: install app dependencies and build it
        run: yarn && yarn build
      - uses: tauri-apps/tauri-action@v0
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        with:
          tagName: app-v__VERSION__ # the action automatically replaces \_\_VERSION\_\_ with
the app version
          releaseName: 'App v__VERSION__'
          releaseBody: 'See the assets to download this version and install.'
          releaseDraft: true
          prerelease: false
```

3. Right above `-name: install app dependencies and build it` you will want to add the following step

```yaml
- name: import windows certificate
  if: matrix.platform == 'windows-latest'
  env:
    WINDOWS_CERTIFICATE: ${{ secrets.WINDOWS_CERTIFICATE }}
    WINDOWS_CERTIFICATE_PASSWORD: ${{ secrets.WINDOWS_CERTIFICATE_PASSWORD }}
  run: |
    New-Item -ItemType directory -Path certificate
    Set-Content -Path certificate/tempCert.txt -Value $env:WINDOWS_CERTIFICATE
    certutil -decode certificate/tempCert.txt certificate/certificate.pfx
    Remove-Item -path certificate -include tempCert.txt
    Import-PfxCertificate -FilePath certificate/certificate.pfx -CertStoreLocation
Cert:\CurrentUser\My -Password (ConvertTo-SecureString -String
$env:WINDOWS_CERTIFICATE_PASSWORD -Force -AsPlainText)
```

4. Save and push to your repo.

5. Your workflow can now import your windows certificate and import it into the GitHub runner, allowing for automated code signing!