Code Signing

Code signing is a security technology that you use to certify that an app was created by you.

On macOS the system can detect any change to the app, whether the change is introduced accidentally or by malicious code.

On Windows, the system assigns a trust level to your code signing certificate which if you don't have, or if your trust level is low, will cause security dialogs to appear when users start using your application. Trust level builds over time so it's better to start code signing as early as possible.

While it is possible to distribute unsigned apps, it is not recommended. Both Windows and macOS will, by default, prevent either the download or the execution of unsigned applications. Starting with macOS Catalina (version 10.15), users have to go through multiple manual steps to open unsigned applications.

macOS Catalina Gatekeeper warning: The app cannot be opened because the developer cannot be verified

As you can see, users get two options: Move the app straight to the trash or cancel running it. You don't want your users to see that dialog.

If you are building an Electron app that you intend to package and distribute, it should be code-signed.

Signing & notarizing macOS builds

Properly preparing macOS applications for release requires two steps: First, the app needs to be code-signed. Then, the app needs to be uploaded to Apple for a process called "notarization", where automated systems will further verify that your app isn't doing anything to endanger its users.

To start the process, ensure that you fulfill the requirements for signing and notarizing your app:

1. Enroll in the Apple Developer Program (requires an annual fee)
2. Download and install Xcode - this requires a computer running macOS
3. Generate, download, and install signing certificates

Electron's ecosystem favors configuration and freedom, so there are multiple ways to get your application signed and notarized.

`electron-forge`

If you're using Electron's favorite build tool, getting your application signed and notarized requires a few additions to your configuration. Forge is a collection of the official Electron tools, using `electron-packager`, `electron-osx-sign`, and `electron-notarize` under the hood.

Let's take a look at an example configuration with all required fields. Not all of them are required: the tools will be clever enough to automatically find a suitable `identity`, for instance, but we recommend that you are explicit.

```
{
  "name": "my-app",
  "version": "0.0.1",
  "config": {
    "forge": {
      "packagerConfig": {
        "osxSign": {
          "identity": "Developer ID Application: Felix Rieseberg (LT94ZKYDCJ)",
          "hardened-runtime": true,
          "entitlements": "entitlements.plist",
          "entitlements-inherit": "entitlements.plist",
          "signature-flags": "library"
        },
        "osxNotarize": {
          "appleId": "felix@felix.fun",
          "appleIdPassword": "my-apple-id-password",
        }
      }
    }
  }
}
```

The `plist` file referenced here needs the following macOS-specific entitlements to assure the Apple security mechanisms that your app is doing these things without meaning any harm:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>com.apple.security.cs.allow-jit</key>
    <true/>
    <key>com.apple.security.cs.debugger</key>
    <true/>
  </dict>
</plist>
```

Note that up until Electron 12, the `com.apple.security.cs.allow-unsigned-executable-memory` entitlement was required as well. However, it should not be used anymore if it can be avoided.

To see all of this in action, check out Electron Fiddle's source code, especially its `electron-forge` configuration file.

If you plan to access the microphone or camera within your app using Electron's APIs, you'll also need to add the following entitlements:

```
<key>com.apple.security.device.audio-input</key>
<true/>
<key>com.apple.security.device.camera</key>
<true/>
```

If these are not present in your app's entitlements when you invoke, for example:

```
const { systemPreferences } = require('electron')
const microphone = systemPreferences.askForMediaAccess('microphone')
```

Your app may crash. See the Resource Access section in Hardened Runtime for more information and entitlements you may need.

`electron-builder`

Electron Builder comes with a custom solution for signing your application. You can find its documentation here.

`electron-packager`

If you're not using an integrated build pipeline like Forge or Builder, you are likely using `electron-packager` , which includes `electron-osx-sign` and `electron-notarize` .

If you're using Packager's API, you can pass in configuration that both signs and notarizes your application.

```
const packager = require('electron-packager')

packager({
  dir: '/path/to/my/app',
  osxSign: {
    identity: 'Developer ID Application: Felix Rieseberg (LT94ZKYDCJ)',
    'hardened-runtime': true,
    entitlements: 'entitlements.plist',
    'entitlements-inherit': 'entitlements.plist',
    'signature-flags': 'library'
  },
  osxNotarize: {
    appleId: 'felix@felix.fun',
    appleIdPassword: 'my-apple-id-password'
  }
})
```

The `plist` file referenced here needs the following macOS-specific entitlements to assure the Apple security mechanisms that your app is doing these things without meaning any harm:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>com.apple.security.cs.allow-jit</key>
    <true/>
    <key>com.apple.security.cs.debugger</key>
    <true/>
  </dict>
</plist>
```

Up until Electron 12, the `com.apple.security.cs.allow-unsigned-executable-memory` entitlement was required as well. However, it should not be used anymore if it can be avoided.

Mac App Store

See the Mac App Store Guide.

Signing Windows builds

Before signing Windows builds, you must do the following:

1. Get a Windows Authenticode code signing certificate (requires an annual fee)
2. Install Visual Studio to get the signing utility (the free Community Edition is enough)

You can get a code signing certificate from a lot of resellers. Prices vary, so it may be worth your time to shop around. Popular resellers include:

- digicert
- Sectigo
- Amongst others, please shop around to find one that suits your needs, Google is your friend 😄

There are a number of tools for signing your packaged app:

- `electron-winstaller` will generate an installer for windows and sign it for you
- `electron-forge` can sign installers it generates through the Squirrel.Windows or MSI targets.
- `electron-builder` can sign some of its windows targets

Windows Store

See the Windows Store Guide.