Creating Distribution-Signed Code for Mac

This thread has been locked by a moderator.

This post is one of a pair of posts, the other one being Packaging Mac Software for Distribution, that replaces my earlier Signing a Mac Product For Distribution post.

Over the past year I've been trying to convert my most useful code signing posts here on DevForums to official documentation, namely:

- Placing Content in a Bundle
- Updating Mac Software
- Signing a Daemon with a Restricted Entitlement
- Embedding a Command-Line Tool in a Sandboxed App
- Embedding Nonstandard Code Structures in a Bundle

Unfortunately in the past month or so my Day Job™, answering developer questions for DTS, has become super busy, and so I've not had chance to complete this work by publish a replacement for Signing a Mac Product For Distribution. This post, and Packaging Mac Software for Distribution, represent the current state of that effort. I think these are sufficiently better than Packaging Mac Software for Distribution to warrant posting them here on DevForums while I wait for the quiet time needed to finish the official work.

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

Creating Distribution-Signed Code for Mac

Sign Mac code for distribution using either Xcode or command-line tools.

Overview

Before shipping a software product for the Mac, you must first create distribution-signed code, that is, code that you can package up and then submit to either the Mac App Store or the notary service. The way you do this depends on the nature of your product and how it was built:

- If your product is a standalone app, possibly with nested code such as an app extension, that you build using Xcode, use Xcode to export a distribution-signed app.
- If your product isn't a standalone app, but you build it using Xcode, create an Xcode archive, and then manually export distribution-signed code from that archive.
- If you build your product using an external build system, such as `make`, add a manual signing step to your build system.

Once you have distribution-signed code, package it for distribution. For more information, see Packaging Mac Software for Distribution.

Note If you use a third-party developer tool to build your app, consult its documentation for advice specific to that tool.

Export an App from Xcode

If your product is a standalone app that you build with Xcode, follow these steps to export a distribution-signed app:

1. Build an Xcode archive from your project.
2. Export a distribution-signed app from that Xcode archive.

You can complete each step from the Xcode app or automate the steps using `xcodebuild`.

To build an Xcode archive using the Xcode app, select your app's scheme and choose Product > Archive. This creates the Xcode archive and selects it in the organizer. To create a distribution-sign app from that archive, select the archive in the organizer, click Distribute App, and follow the workflow from there.

Note If the button says Distribute Content rather than Distribute App, your archive has multiple items in its Products directory. Make sure that every target whose output is embedded in your app has the Skip Install (`SKIP_INSTALL`) build setting set; this prevents the output from also being copied into the Xcode archive's `Products` directory. For more on this, see TN3110 Resolving generic Xcode archive issue.

For more information about the Xcode archives and the organizer, see Distributing Your App for Beta Testing and Releases.

To build an Xcode archive from the command line, run `xcodebuild` with the `archive` action. Once you have an Xcode archive, export a distribution-signed app by running `xcodebuild` with the `-exportArchive` option. For more information about `xcodebuild`, see its man page. For instructions on how to read a man page, see Reading UNIX Manual Pages. For information about the keys supported by the export options property list, run `xcodebuild` with the `-help` argument.

Export a Non-App Product Built with Xcode

If you build your product with Xcode but it's not a standalone app, you can build an Xcode archive using the techniques described in the previous section but you cannot export distribution-signed code from that archive. The Xcode organizer and the `-exportArchive` option only work for standalone apps.

To export a distribution-signed product from the Xcode archive:

1. Copy the relevant components from the archive.
2. Sign those components manually.

The exact commands for doing this vary depending on how your product is structured, so let's consider a specific example. Imagine your product is a daemon but it also has an associated configuration app. Moreover, the configuration app has a share extension, and an embedded framework to share code between the app and the extension. When you build an Xcode archive from this project it has this structure:

```
DaemonWithApp.xcarchive/
  Info.plist
  Products/
    usr/
      local/
        bin/
          Daemon
    Applications/
      ConfigApp.app/
        Contents/
          embedded.provisionprofile
          Frameworks/
            Core.framework/
              …
          PlugIns/
            Share.appex/
              Contents/
                embedded.provisionprofile
                …
          …
  …
```

The `Products` directory contains two items: the daemon itself (`Daemon`) and the configuration app (`ConfigApp.app`). To sign this product, first copy these items out of the archive:

```
% mkdir "to-be-signed"
% ditto "DaemonWithApp.xcarchive/Products/usr/local/bin/Daemon" "to-be-signed/Daemon"
% ditto "DaemonWithApp.xcarchive/Products/Applications/ConfigApp.app" "to-be-signed/ConfigApp.app"
```

IMPORTANT When you copy code, use `ditto` rather than `cp`. `ditto` preserves symlinks, which are critical to the structure of Mac frameworks. For more information on this structure, see Placing Content in a Bundle. Symlinks are also useful when dealing with nonstandard code structures. For more details, see Embedding Nonstandard Code Structures in a Bundle.

The code you copy from the Xcode archive is typically development-signed:

```
% codesign -d -vv to-be-signed/Daemon
…
Authority=Apple Development: …
…
```

To ship this code, you need to re-sign it for distribution.

Confirm Your Code Signing Identity

To sign code for distribution you need a code signing identity. Choose the right identity for your distribution channel:

- If you're distributing an app on the Mac App Store, use an Apple Distribution code signing identity. This is named `Apple Distribution: TTT`, where `TTT` identifies your team.
- Alternatively, you can use the old school Mac App Distribution code signing identity. This is named `3rd Party Mac Developer Application: TTT`, where `TTT` identifies your team.
- If you're distributing a product independently, use a Developer ID Application code signing identity. This is named `Developer ID Application: TTT`, where `TTT` identifies your team.

For information on how to set up these code signing identities, see Developer Account Help.

To confirm that your code-signing identity is present and correct, run the following command:

```
% security find-identity -p codesigning -v
  1) A06E7F3F8237330EE15CB91BE1A511C00B853358 "Apple Distribution: …"
  2) ADC03B244F4C1018384DCAFFC920F26136F6B59B "Developer ID Application: …"
     2 valid identities found
```

The `-p codesigning` argument filters for code-signing identities. The `-v` argument filters for valid identities only. If the code-signing identity that you need isn't listed, see Developer Account Help.

Each output line includes a SHA-1 hash that uniquely identifies the identity. If you have multiple identities with the same name, sign your code using this hash rather than the identity name.

Identify the Code to Sign

To sign your product, first identify each code item that you need to sign. For example, in the `DaemonWithApp` product, there are four code items: `ConfigApp.app`, `Core.framework`, `Share.appex`, and `Daemon`.

For each code item, determine the following:

- Is it bundled code?
- Is it a main executable?

IMPORTANT For a code item to be considered bundled code it must be the main code within a bundle. If, for example, you have an app with a nested helper tool, there are two code items: the app and the helper tool. The app is considered bundle code but the helper tool is not.

In some cases, it might not be obvious whether the code item is a main executable. To confirm, run the `file` command. A main executable says `Mach-O … executable`. For example:

```
% file "to-be-signed/ConfigApp.app/Contents/Frameworks/Core.framework/Versions/A/Core"
…
… Mach-O 64-bit dynamically linked shared library x86_64
…
% file "to-be-signed/ConfigApp.app/Contents/PlugIns/Share.appex/Contents/MacOS/Share"
…
… Mach-O 64-bit executable x86_64
…
```

The `Core.framework` is not a main executable but `Share.appex` is.

To continue the DaemonWithApp example, here's a summary of this info for each of its code items:

```
| Code Item      | Bundled Code? | Main Executable |
| ---------      | ------------- | --------------- |
| ConfigApp.app  | yes           | yes             |
| Core.framework | yes           | no              |
| Share.appex    | yes           | yes             |
| Daemon         | no            | yes             |
```

Determine the Signing Order

Sign code from the inside out. That is, if A depends on B, sign B before you sign A. For the DaemonWithApp example, the signing order for the app is:

1. `Core.framework`
2. `Share.appex`
3. `ConfigApp.app`

The app and daemon are independent, so you can sign them in either order.

Configure Your Entitlements

A code signature may include entitlements. These key-value pairs grant an executable permission to use a service or technology. For more information about this, see Entitlements.

Entitlements only make sense on a main executable. When a process runs an executable, the system grants the process the entitlements claimed by its code signature. Do not apply entitlements to library code. It doesn't do anything useful and can prevent your code from running.

When signing a main executable, decide whether it needs entitlements. If so, create an entitlements file to use when signing that executable. This entitlements file is a property list containing the key-value pairs for the entitlements that the executable claims.

If you build your product with Xcode, you might be able to use the `.entitlements` file that Xcode manages in your source code. If not, create the `.entitlements` file yourself.

IMPORTANT The entitlements file must be a property list in the standard XML format with LF line endings, no comments, and no BOM. If you're not sure of the file's provenance, use `plutil` to convert it to the standard format. For specific instructions, see Ensure Properly Formatted Entitlements.

If you have a development-signed version of your program you can get a head start on this by dumping its entitlements. For example:

```
% codesign -d --entitlements - --xml "to-be-signed/ConfigApp.app" | plutil -convert xml1 -o - -
…
<dict>
  <key>com.apple.application-identifier</key>
  <string>SKMME9E2Y8.com.example.apple-samplecode.DaemonWithApp.App</string>
  <key>com.apple.developer.team-identifier</key>
  <string>SKMME9E2Y8</string>
  <key>com.apple.security.app-sandbox</key>
  <true/>
  <key>keychain-access-groups</key>
  <array>
    <string>SKMME9E2Y8.com.example.apple-samplecode.DaemonWithApp.SharedKeychain</string>
  </array>
</dict>
</plist>
```

Keep in mind that some entitlements vary between development and distribution builds. For example:

- The value of the APS Environment (macOS) Entitlement changes from `development` to `production`.
- The `com.apple.security.get-task-allow` entitlement allows the debugger to attach to your program, so you rarely apply it to a distribution-signed program.

To check whether an entitlement varies in distribution builds, see the documentation for that specific entitlement in Entitlements.

For information about when it makes sense to distribute a program signed with the get-task-allow entitlement, see *Avoid the Get-Task-Allow Entitlement* section in Resolving Common Notarization Issues).

Embed Distribution Provisioning Profiles

In general, all entitlement claims must be authorized by a provisioning profile. This is an important security feature. For example, the fact that the `keychain-access-groups` entitlement must be authorized by a profile prevents other developers from shipping an app that impersonates your app in order to steal its keychain items.

However, macOS allows programs to claim some entitlements without such authorization. These unrestricted entitlements include:

- `com.apple.security.get-task-allow`
- `com.apple.security.application-groups`
- Those used to enable and configure the App Sandbox
- Those used to configure the Hardened Runtime

If your program claims a restricted entitlement, include a distribution provisioning profile to authorize that claim:

1. Create the profile on the developer web site.
2. Copy that profile into your program's bundle.

Note If your product includes a non-bundled executable that uses a restricted entitlement, package that executable in an app-like structure. For details on this technique, see Signing a Daemon with a Restricted Entitlement.

To create a distribution provisioning profile, follow the instructions in Developer Account Help. Make sure to choose a profile type that matches your distribution channel (Mac App Store or Developer ID).

Once you have a distribution provisioning profile, copy it into your program's bundle. For information about where to copy it, see Placing Content in a Bundle.

To continue the DaemonWithApp example, the configuration app and its share extension use a keychain access group to share secrets. The system grants the programs access to that group based on their `keychain-access-groups` entitlement claim, and such claims must be authorized by a provisioning profile. The app and the share extension each have their own profile. To distribute the app, update the app and share extension bundles with the corresponding distribution provisioning profile:

```
% cp "ConfigApp-Dist.provisionprofile" "to-be-signed/ConfigApp.app/Contents/embedded.provisionprofile"
% cp "Share-Dist.provisionprofile" "to-be-signed/ConfigApp.app/Contents/PlugIns/Share.appex/Contents/embedded.provisionprofile"
```

Modifying the app in this way will break the seal on its code signature. This is fine because you are going to re-sign the app before distributing it.

IMPORTANT If you're building your product with Xcode then you might find that Xcode has embedded a provisioning profile within your bundle. This is a development provisioning profile. You must replace it with a distribution provisioning profile.

Sign Each Code Item

For all code types, the basic `codesign` command looks like this:

```
% codesign -s III PPP
```

Here `III` is the name of the code signing identity to use and `PPP` is the path to the code to sign.

The specific identity you use for `III` varies depending on your distribution channel, as discussed in *Confirm Your Code Signing*, above.

Note If you have multiple identities with the same name, supply the identity's SHA-1 hash to specify it unambiguously. For information on how to get this hash, see *Confirm Your Code Signing*, above.

When signing bundled code, as defined in *Identify the Code to Sign*, above, use the path to the bundle for `PPP`, not the path to the bundle's main code.

If you're re-signing code — that is, the code you're signing is already signed — add the `-f` option.

If you're signing a main executable that needs entitlements, add the `--entitlements EEE` option, where `EEE` is the path to the entitlements file for that executable. For information on how to create this file, see *Configure Your Entitlements*, above.

If you're signing for Developer ID distribution, add the `--timestamp` option to include a secure timestamp.

If you're signing a main executable for Developer ID distribution, add the `-o runtime` option to enable the Hardened Runtime. For more information about the Hardened Runtime, see Hardened Runtime.

If you're signing non-bundled code, add the `-i BBB` option to set the code signing identifier. Here `BBB` is the bundle ID the code would have if it had a bundle ID. For example, if you have an app whose bundle ID is `com.example.flying-animals` that has a nested command-line tool called `pig-jato`, the bundle ID for that tool would logically be `com.example.flying-animals.pig-jato`, and that's a perfectly fine value to use for `BBB`.

Note For bundled code, you don't need to supply a code signing identifier because `codesign` defaults to using the bundle ID.

Repeat this signing step for every code item in your product, in the order you established in *Determine the Signing Order*, above. If you have a complex product with many code items to sign, create a script to automate this process.

Here's the complete sequence of commands to sign the `DaemonWithApp` example for Developer ID distribution:

```
% codesign -s "Developer ID Application" -f --timestamp "to-be-signed/ConfigApp.app/Contents/Frameworks/Core.framework"
to-be-signed/ConfigApp.app/Contents/Frameworks/Core.framework: replacing existing signature
% codesign -s "Developer ID Application" -f --timestamp -o runtime --entitlements "Share.entitlements" "to-be-signed/ConfigApp.app/Contents/PlugIns/Share.appex"
to-be-signed/ConfigApp.app/Contents/PlugIns/Share.appex: replacing existing signature
% codesign -s "Developer ID Application" -f --timestamp -o runtime --entitlements "ConfigApp.entitlements" "to-be-signed/ConfigApp.app"
to-be-signed/ConfigApp.app: replacing existing signature
% codesign -s "Developer ID Application" -f --timestamp -o runtime -i "com.example.apple-samplecode.DaemonWithApp.Daemon" "to-be-signed/Daemon"
to-be-signed/Daemon: replacing existing signature
```

Consider Deep Harmful

When signing code, do not pass the `--deep` option to `codesign`. This option is helpful in some specific circumstances but it will cause problems when signing a complex product. Specifically:

- It applies the same code signing options to every code item that it signs, something that's not appropriate. For example, you might have an app with an embedded command-line tool, where the app and the tool need different entitlements. The `--deep` option will apply the same entitlements to both, which is a serious mistake.
- It only signs code that it can find, and it only finds code in nested code sites. If you put code in a place where the system is expecting to find data, `--deep` won't sign it.

The first issue is fundamental to how `--deep` works, and is the main reason you should avoid it. The second issue is only a problem if you don't follow the rules for nesting code and data within a bundle, as documented in Placing Content in a Bundle.

Revision History

- **2022-08-17** Updated the *Confirm Your Code Signing Identity* section to cover Apple Distribution code signing identities. Added a link to TN3110.
- **2022-03-01** First posted.

Code Signing | Gatekeeper | Developer ID

Reply

Posted 7 months ago by eskimo

Add a Comment