Embedding a Command-Line Tool in a Sandboxed App

This thread has been locked. Questions are automatically locked after two months of inactivity, or sooner if deemed necessary by a moderator.

I regularly help developers — both here on DevForums and as part of my Day Job™ in DTS — who have a sandboxed app, built with Xcode, and want to embed a helper tool within that app. For example:

- They have some of their own code that they want to run in a separate process. In many cases an XPC Service is a better choice for this, but sometimes it's just easier to embed a command-line tool.
- They have a command-line tool that was built by an external build system (makefiles and so on).

Doing this is a bit tricky, so I thought I'd write down the process for the benefit of all.

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

Embedding a Command-Line Tool in a Sandboxed App

Using Xcode to embed a command-line tool in a sandboxed app is a little tricky. The exact process you use depends whether you want to build the tool using Xcode or you have an existing tool that was built by an external build system. I'll cover each scenario in turn.

Note This post focuses on building an app for the App Store because that's where most sandboxed apps are destined. However, the same basic process works for Developer ID; you just need to choose a different distribution path in the Organizer.

The post assumes Xcode 12.5 running on macOS 11.3.

Embed a Tool Built With Xcode

This section describes how to use Xcode to create a sandboxed app and then embed a helper tool, also built with Xcode, into that app.

Create an App Project

To get started, first create a new project from the macOS > App template. In this example I named it `Test768613894`, resulting in a bundle ID of `com.example.apple-samplecode.Test768613894`.

In the project editor, set the deployment target to 10.15. This isn't strictly necessary but I'll use it to show how to configure the app and its embedded helper tool to use the same deployment target.

In the General tab of the app target editor, set the App Category to Utilities. This avoids a warning when you try to submit to the store.

In the Signing & Capabilities tab of the app target editor, make sure that "Automatically manage signing" is checked and then select the appropriate team. The Signing Certificate popup will switch to Development, which is exactly what you want for day-to-day development.

Add the Hardened Runtime capability. This isn't necessary for App Store submission but it's a good idea to use it on all new projects.

Choose Product > Archive. This builds the app into an Xcode archive and reveals that archive in the Organizer. The goal here is to simply check that everything is working so far.

In the Organizer, delete the new archive, just to reset things back to the original state.

Create the Helper Tool

With the app target in the project building correctly, it's time to create a helper tool target so that you can embed its product into the app. To start, create a new target from the macOS > Command Line Tool template. I named this `ToolX`, where the *X* stands for *built with Xcode*.

In the General tab of the tool target editor, clear the Deployment Target field. The tool will now inherit its deployment target (macOS 10.15) from the project.

In the Signing & Capabilities tab of the tool target editor, make sure that "Automatically manage signing" is checked and then select the appropriate team. Again, the Signing Certificate popup will switch to Development.

Fill in the Bundle Identifier field. As my app's bundle ID is `com.example.apple-samplecode.Test768613894` I set this to `com.example.apple-samplecode.Test768613894.ToolX`. Bundle IDs generally don't play a big part in command-line tools but it's necessary in this case because of the way that I set up the code signing identifier (see below).

Add the App Sandbox and Hardened Runtime capabilities. Again, the Hardened Runtime isn't required but it's good to start as you mean to go on.

In the Build Settings tab, set the Skip Install (`SKIP_INSTALL`) build setting to true. Without this Xcode copies the tool into the 'root' of your Xcode archive, which causes grief later on.

Also set Code Signing Inject Base Entitlements (`CODE_SIGN_INJECT_BASE_ENTITLEMENTS`) to false. If you leave this set then Xcode will include the get-task-allow entitlement in development builds of your tool, but this entitlement is incompatible with the `com.apple.security.inherit` entitlement.

IMPORTANT This means that you won't be able to debug your tool. If you need to do this, create a new target for the tool, one that's not sandboxed at all. Be warned, however, that this target may behave differently from the embedded tool target because it's not running under the sandbox.

Finally, set Other Code Signing Flags (`OTHER_CODE_SIGN_FLAGS`) to `$(inherited) -i $(PRODUCT_BUNDLE_IDENTIFIER)`. This ensures that the tool's code signing identifier matches its bundle ID, a matter of best practice.

Select `ToolX.entitlements` in the Project navigator and added `com.apple.security.inherit` to it, with a Boolean value of true.

Select the `ToolX` scheme and chose Product > Build, just to make sure that it builds correctly.

Now switch back to the app (`Test768613894`) scheme.

Embed the Helper Tool

In the Build Phases tab of the app target editor, add the `ToolX` target to the Dependencies build phase. This ensures that Xcode builds the tool before building the app.

Add a new Copy Files build phase. Named this `Embed Helper Tools` (the exact name doesn't matter but it's best to pick a descriptive one) and set the Destination popup to Executables.

Note This will place the helper tool in your app's `Contents/MacOS` directory, one of the locations recommended by the *Nested Code* section of Technote 2206 macOS Code Signing In Depth.

Add the `ToolX` executable to that build phase, making sure that Code Sign On Copy is checked.

Build and Validate

With the project now set up it's time to test that everything builds correctly. To start, do another Product > Archive. This will build the tool target and then the app target, embedding the former within the latter.

In the Organiser, select the newly-created archive and click Distribute App.

Note If the button says Distribute Content rather than Distribute App, go back and check that you set the Skip Install build setting on the *tool* target.

Select App Store Connect, clicked Next, then Export and clicked Next.

Go through the rest of the export workflow. The end result is a directory with a name like `Test768613894 2021-05-17 14-07-21`.

In that directory is an installer package. Unpack that.

Note I used Pacifist for this but, if you don't have that app, and you should, see Unpacking Apple Archives.

Run the following commands to validate that Xcode constructed everything correctly.

```
% codesign -d -vvv --entitlements :- Test768613894.app
Executable=/…/Test768613894.app/Contents/MacOS/Test768613894
Identifier=com.example.apple-samplecode.Test768613894
Format=app bundle with Mach-O universal (x86_64 arm64)
CodeDirectory v=20500 size=822 flags=0x10000(runtime) hashes=14+7 location=embedded
…
Authority=Apple Distribution: Quinn Quinn (5KMM8E9E2Y8)
…
TeamIdentifier=5KMM8E9E2Y8
…
<dict>
    <key>com.apple.security.app-sandbox</key>
    <true/>
    <key>com.apple.security.files.user-selected.read-only</key>
    <true/>
</dict>
</plist>
% codesign -d -vvv --entitlements :- Test768613894.app/Contents/MacOS/ToolX
…
Identifier=com.example.apple-samplecode.Test768613894.ToolX
Format=Mach-O universal (x86_64 arm64)
CodeDirectory v=20500 size=796 flags=0x10000(runtime) hashes=13+7 location=embedded
…
Authority=Apple Distribution: Quinn Quinn (5KMM8E9E2Y8)
…
TeamIdentifier=5KMM8E9E2Y8
…
<dict>
    <key>com.apple.security.app-sandbox</key>
    <true/>
    <key>com.apple.security.inherit</key>
    <true/>
</dict>
</plist>
```

I want to highlight some things in this output:

- The `Identifier` field is the code signing identifier.
- The `Format` field shows that both executables are universal.
- The `runtime` flag, in the `CodeDirectory` field, shows that the hardened runtime is enabled.
- The `Authority` field shows that the code was signed by my Apple Distribution signing identity, which is what you'd expect for an App Store submission.
- The `TeamIdentifier` is, well, the Team ID.
- The app's entitlements include `com.apple.security.app-sandbox` and whatever other entitlements are appropriate for this app.
- The tool's entitlements include just `com.apple.security.app-sandbox` and `com.apple.security.inherit`.

IMPORTANT Any other entitlements here can cause problems. If you find that, when your app runs the tool, it immediately crashes with a code signing error, check that the tool is signed with just these two entitlements.

Embedding an Externally-Built Tool

With the app and Xcode-built helper tool working correctly, it's time to repeat the process for a tool built using an external build system. In this example we'll create a dummy helper tool from the command line and then embed that in the Test768613894 app.

Build the Tool

Create a new directory and change into it:

```
% mkdir ToolC
% cd ToolC
```

Here *C* stands for *built with Clang*.

Create a source file in that directory that looks like this:

```
% cat main.c
#include <stdio.h>

int main(int argc, char ** argv) {
    printf("Hello Cruel World!\n");
    return 0;
}
```

Build that with `clang` twice, once for each architecture, and then `lipo` them together:

```
% clang -o ToolC-x86_64 -mmacosx-version-min=10.15 -arch x86_64 main.c
% clang -o ToolC-arm64 -mmacosx-version-min=10.15 -arch arm64 main.c
% lipo ToolC-x86_64 ToolC-arm64 -create -output ToolC
```

The `-mmacosx-version-min` option sets the deployment target to match the Test768613894 app.

Create an entitlements file for the tool:

```
% cat ToolC.entitlements
<?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.app-sandbox</key>
    <true/>
    <key>com.apple.security.inherit</key>
    <true/>
</dict>
</plist>
```

Sign the tool as shown below:

```
% codesign -s - -i com.example.apple-samplecode.Test768613894.ToolC -o runtime --entitlements ToolC.entitlements -f ToolC
```

This breaks down as follows:

- The `-s -` argument applies an ad-hoc signature (in Xcode parlance this is Sign to Run Locally). More on this below.
- The `-i com.example.apple-samplecode.Test768613894.ToolC` option sets the code signing identifier.
- The `-o runtime` option enables the hardened runtime. Again, this isn't necessary for App Store distribution but it's a good idea in general.
- The `--entitlements ToolC.entitlements` option supplies the signature's entitlements.
- The `-f` option overrides any existing signature. This isn't strictly necessary but it avoids a confusion that the existing ad-hoc signature applied by the linker to the `arm64` architecture.

IMPORTANT Setting up the code signature here is critical. It sets up a 'pattern' that Xcode uses when it re-signs the tool while embedding it into the final app. The only thing that doesn't matter here is the signing identity. Xcode will override that with the project's signing identity during this embedding process. That's why we can get away with an ad-hoc signature.

Add the `ToolC` executable to your Test768613894 project. When you do this:

- Enable "Copy items if needed".
- Select "Create groups" rather than "Create folder reference."
- Uncheck all the boxes in the "Add to targets" list.

In the Build Phases tab of the app target editor, add `ToolC` to the build phase, making sure that Code Sign On Copy is checked.

Validate

To validate your work, follow the same process as described in the *Build and Validate* section, substituting `ToolC` for `ToolX` everywhere.

Xcode Code Signing

Asked 17 minutes ago by eskimo

Reply to this question