# Notarisation Fundamentals

318

I regularly help developers with 'fun' notarisation edge cases, where the best path forward is only clear when you understand a little bit about how the notary service works. This post is my attempt to pass on my accumulated knowledge on this subject, in the hope that it'll allow folks to make better notarisation choices.

If you have questions or feedback about any of the points raised here, start a new thread with the *Notarization* tag.

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

## Notarisation Fundamentals

Many Mac developers find notarisation to be a trivial exercise: If you've built an app with Xcode, notarizing it involves a few clicks in the Xcode organiser. However, not all Mac products are that simple. If you're building a more complex product, and especially if you're using third-party tools, it pays to know a little bit about how notarisation works so that you can make educated choices about how to notarise your product.

Notarisation involves four steps:

1. You submit a product to the notary service.
2. The notary service checks that product.
3. If those checks pass, it issues a notarised ticket.
4. When the user runs your product, macOS checks that it is authorised by a notarised ticket.

### Viewing the JSON Ticket

To see what's in the ticket, fetch the notary log and look in the `ticketContents` property. For example:

```
"ticketContents": [
  {
    "path": "Test710726.zip/Test710726.app",
    "digestAlgorithm": "SHA-256",
    "cdhash": "e69065d292b7f222975262de06d2c4e0f72036da4",
    "arch": "x86_64"
  },
  {
    "path": "Test710726.zip/Test710726.app",
    "digestAlgorithm": "SHA-256",
    "cdhash": "f042bf085079b5bc3201751f51db626c4976f47a",
    "arch": "arm64"
  },
  {
    "path": "Test710726.zip/Test710726.app/Contents/MacOS/Test710726",
    "digestAlgorithm": "SHA-256",
    "cdhash": "e69065d292b7f222975262de06d2c4e0f72036da4",
    "arch": "x86_64"
  },
  {
    "path": "Test710726.zip/Test710726.app/Contents/MacOS/Test710726",
    "digestAlgorithm": "SHA-256",
    "cdhash": "f042bf085079b5bc3201751f51db626c4976f47a",
    "arch": "arm64"
  },
  {
    "path": "Test710726.zip/Test710726.app/Contents/Frameworks/CoreWaffleVarnishing.framework/Versions/A/CoreWaffleVarnishing",
    "digestAlgorithm": "SHA-256",
    "cdhash": "b394af6af863e226f2bd0c36cd20dd5e7282f26e",
    "arch": "x86_64"
  },
  {
    "path": "Test710726.zip/Test710726.app/Contents/Frameworks/CoreWaffleVarnishing.framework/Versions/A/CoreWaffleVarnishing",
    "digestAlgorithm": "SHA-256",
    "cdhash": "f557d2753d286f2dc0c6ad4cf5d36e362c50f474",
    "arch": "arm64"
  }
  {
    "path": "Test710726.zip/Test710726.app/Contents/Frameworks/CoreWaffleVarnishing.framework/Versions/Current",
    "digestAlgorithm": "SHA-256",
    "cdhash": "b394af6af863e226f2bd0c36cd20dd5e7282f26e",
    "arch": "x86_64"
  },
  {
    "path": "Test710726.zip/Test710726.app/Contents/Frameworks/CoreWaffleVarnishing.framework/Versions/Current",
    "digestAlgorithm": "SHA-256",
    "cdhash": "f557d2753d286f2dc0c6ad4cf5d36e362c50f474",
    "arch": "arm64"
  },
],
```

**Note** For instructions on how to fetch the notary log, see Fetching the Notary Log.

There should be at least one entry for every item of executable code in your product, but there may be more than one. If the executable code is universal, you'll find separate items for each architecture. And if the executable code is bundled, you'll often find separate items for the code itself and the bundle as a whole.

**IMPORTANT** In this context, **executable code** means a Mach-O image. It doesn't include things like shell scripts, Python scripts, and AppleScripts.

What really matters in this output ticket is the `cdhash` properties. Each of these is a code directory hash (cdhash) value that uniquely identifies executable code that's been notarised.

**Note** For more information about cdhash values, see TN3126 Inside Code Signing: Hashes.

### Viewing the Binary Ticket

The JSON structure shown above tells you what's in the ticket but it's *not* the actual ticket. That is a compact binary data structure containing just the cdhash values and a digital signature. To see the actual ticket, run `stapler` with the `validate` subcommand:

```
% xcrun stapler validate -v Test710726.app
…
Downloaded ticket has been stored at file:///var/folders/ts/89wlmlw971x8k8ds6y_48kn80000gp/T/ffbc6e29-ad1a-48ac-bf8d-
6a3b298aaf06.ticket.
The validate action worked!
```

**IMPORTANT** The trailing dot (`.`) is not part of the ticket's path.

Now dump the ticket as hex:

```
% xxd -p /var/folders/ts/89wlmlw971x8k8ds6y_48kn80000gp/T/ffbc6e29-ad1a-48ac-bf8d-6a3b298aaf06.ticket
% xxd -p /var/folders/ts/89wlmlw971x8k8ds6y_48kn80000gp/T/ffbc6e29-ad1a-48ac-bf8d-6a3b298aaf06.ticket
…
…e69065d292b7f222975262de06d2c4e0f72036da4…f557d2753d286f2d
cbc6ad4cf5d36e362c50f474…b394af6af863e226f2bd0c36cd20dd5e72
82f26e…f042bf085079b5bc3201751f51db626c4976f47a…
…
```

Note how every cdhash value from the `ticketContents` JSON property is present in the ticket.

## Using the Ticket

When you load Developer ID signed code, the trusted execution system checks that the code is authorised by a notarised ticket [1]. You can see evidence of this in the system log:

```
type: info
time: 2022-07-22 03:53:36.134013 -0700
process: syspolicyd
subsystem: com.apple.syspolicy
category: default
message: looking up ticket: {length = 20, bytes = 0xe69065d292b7f222975262de06d2c4e0f72036da4}, 2, 1

type: info
time: 2022-07-22 03:53:36.134062 -0700
process: syspolicyd
subsystem: com.apple.syspolicy
category: default
message: cloudkit record fetch: https://api.apple-cloudkit.com/database/1/com.apple.gk.ticket-
delivery/production/public/records/lookup, 2/2/e69065d292b7f222975262de06d2c4e0f72036da4
```

**Note** Some of these values may not be present if you enable private data. For the details, see Your Friend the System Log.

The first entry shows `syspolicyd` looking for a ticket that authorises `e69065d292b7f222975262de06d2c4e0f72036da4`, that is, the cdhash of the Intel code for the Test710726 app itself. The second entry shows `syspolicyd` starting to fetch that ticket from CloudKit.

If you try to run Developer ID signed code and it's not authorised by a notarised ticket, the trusted execution system will prevent the code from running. For an example of what that looks like in practice, see the *Identifying a Notarisation Problem* section of Resolving Gatekeeper Problems.

[1] The exact circumstances under which it does this check can change over time. In macOS 12 and earlier, this is done by Gatekeeper only when the code is quarantined (except for high-security items, like KEXTs). This is changing in macOS 13 beta, as explained in WWDC 2022 Session 10096 What's new in privacy.

## Packaging for Notarisation

It's critical that all Developer ID signed executable code be authorised by a notarised ticket. If not, the trusted execution system will block that code from running.

On the other hand, if you ship some code that you never intend to run, it's fine for it to not be included in the ticket.

These two points affect how you package your product for notarisation. Let's walk through some examples.

### Executable Code Inside Archives

Imagine you have a third-party runtime that typically ships code inside an archive file. In this case it's best to use an archive format that's *transparent* to the notary service. That is, the notary service must be able to unpack the archive so that it can find the executable code, check it, and then include it in the ticket.

**Note** If you can't use a notary-transparent archive format, see the advice in the *Installers* section below.

The JAR files used by Java are a classic example of this. A JAR file is essentially a zip archive, and thus notary-transparent. This is a good thing, because it means that the executable code within the JAR file is authorised by the ticket. However, it also means that the executable in the JAR file must pass the notary service checks.

Another common example is the zip archives used by Node.js.

In many cases you can fix these problems by unpacking the archive, signing the unpacked executable code, and then repacking the archive. Alternatively, you might be able to reconfigure your build system to include the executable code outside of the archive. Consult the support resources for the third-party runtime you're using for specific advice about this issue.

The one thing you mustn't do is disguise the archive so that the notary service can't see inside it. This gets you over the notarisation hurdle but you'll then stumble at the next hurdle, namely Gatekeeper. If your executable code isn't authorise by a notarised ticket, the trusted execution system will block it from loading.

### Disguising Executable Code

There are, however, circumstances where disguising the contents of the archive is the right thing to do. Imagine you're creating an app that contains executable code that's never intended to execute directly. For example you might be building an IDE that includes project templates with executable code. Your app never runs these templates. Rather, they are included solely to be copied in to other apps that the user creates. Given that, there's no point notarising this code and so it's fine to disguise it. Moreover, in some cases it might be necessary to do that [1].

The standard way to disguise code like this is to put it in an encrypted zip archive. The notary service can't possibly see into this because it doesn't have the password.

**IMPORTANT** Do not disguise your archive by slightly munging it; the notary service is sometimes smart enough to see past such things. Also don't use a standard archive format that the notary service doesn't currently support. Such support might be added in the future.

Remember that disguising code that's intended to run on the Mac is pointless. It'll let you pass notarisation but then the code will be blocked by the trusted execution system.

[1] I've seen cases where the notary service gets confused by cross-platform code. To continue the IDE example, imagine that your IDE builds iOS products and supports the iOS Simulator. That means you have to ship `arm64` templates for iOS and both `arm64` and `x86_64` templates for the iOS Simulator. The notary service should notice that this code is not for the macOS platform and ignore it but, at least in the past, that check has not always worked correctly.

### Installers

In some cases you *must* ship code within an archive that's not notary-transparent. The classic example of this is a third-party installer that uses a proprietary archive format. In that case you must notarise twice:

1. First notarise the contents of the installer.
2. Then build the installer and notarise that.

Each step generates a ticket. The first covers the contents of the installer, while the second covers the installer itself.

**Note** This requirement is called out in the Important box within the *Prepare your software for notarization* section of Notarizing macOS software before distribution.

If you only do the second step, the contents of the installer isn't seen by the notary service and hence isn't included in any notarise ticket. Annoyingly, this might seem to work because:

1. The files laid down by the installer are not quarantined.
2. On macOS 12 and earlier, Gatekeeper only checks quarantined files.

So you might not notice the problem until you trigger Gatekeeper in some other way, for example, by sharing the installed code over AirDrop.

Notarization

Reply

Posted 2 months ago by eskimo

Add a Comment