A Peek Behind the Code Signing Curtain

This thread has been locked by a moderator.

533

IMPORTANT This post is now retired in favour of TN3126 Inside Code Signing: Hashes. I'm leaving the original post here just for the record, but you should consider the official documentation authoritative.

I was explaining code signing to someone today and, well, at the end of that conversation I found that I'd 'accidentally' written up a high-level description of how code signing works (-: Rather than let that go to waste, I thought I'd post a copy here.

IMPORTANT This is *all* implementation details. You don't need this info in order to sign your product for Apple platforms. If you use Xcode, it takes care of code signing for you. If you build a Mac product outside of Xcode, follow the advice in:

- Creating Distribution-Signed Code for Mac
- Packaging Mac Software for Distribution

However, sometimes it's useful to know a little bit about the man behind the curtain. It allows you to make better design choices and debug leaky abstraction layers. And hey, sometimes it's just fun to know this stuff!

If you have questions about code signing, put them in a new thread here on DevForums. Tag it with *Code Signing* so that I see it.

Finally, if you *really* want to know how code signing works, most, maybe even all, of the implementation is available in Darwin. Specifically, look in the `xnu` project for the kernel side of it and the `Security` for the user space side.

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

A Peek Behind the Code Signing Curtain

Code signing is a fundamental technology on Apple platforms. The vast majority of developers don't need to know how code signing works. They use Xcode, which takes care of the fiddly details. And, if they run into problems, those are usually related to high-level concepts — signing identities, entitlements, provisioning profiles — not the core code signing implementation.

However, that's not always the case. Every now and again an issue crops up where you actually need to understand how code signing works. For example:

- My Signing code for older versions of macOS on Apple Silicon post explains how to work around a code signing issue that cropped up during the early days of Apple silicon. It's easier to understand issues like this if you understand some core code signing concepts.
- Using the Latest Code Signature Format has a diagnostic process that involves code signing hash slots. While the diagnostic itself is useful, it makes more sense if you know what those slots are for.
- The issue covered by Updating Mac Software makes more sense once you understand code signing's lazy per-page signature checking.

The goal of this post is to explain some of these core code signing concepts so that you can better understand, and potentially debug, weird code signing issues.

I want to stress that this description is not comprehensive. I've elided various details for various reasons: Some stuff I haven't yet researched in depth, some stuff is only interesting from a historical perspective, and some stuff is not relevant to my overall goal.

My focus here is on macOS but these core code signing concepts apply to all Apple platfroms.

Finally, a word of warning…

WARNING Code signing has changed in the past and you can reasonably expect it to change again in the future. If you build a product that relies on these details, you must accept the compatibility risks that this entails.

Reading this post is like reading the Darwin source: You'll learn a lot, but you need to be careful about how you apply that knowledge.

Code Signature Storage

The code signature for an item is stored in one of four ways:

- If the item is a Mach-O image, or is a bundle wrapped around a Mach-O image, the code signature is stored within the image.

```
% otool -l "/Applications/PCalc.app/Contents/MacOS/PCalc" | grep LC_CODE_SIGNATURE -B 1 -A 3
Load command 38
      cmd LC_CODE_SIGNATURE
  cmdsize 16
  dataoff 3170576
 datasize 60656
```

- If the item is a bundle without a Mach-O image, the code signature is stored in the bundle's `_CodeSignature` directory:

```
% codesign -d -vvv Test.bundle
…
Format=bundle
…
Authority=Apple Development: Quinn Quinn (7XFU7D52S4)
…
% find Test.bundle
Test.bundle
Test.bundle/Contents
Test.bundle/Contents/_CodeSignature
Test.bundle/Contents/_CodeSignature/CodeResources
Test.bundle/Contents/_CodeSignature/CodeDirectory
Test.bundle/Contents/_CodeSignature/CodeRequirements-1
Test.bundle/Contents/_CodeSignature/CodeSignature
Test.bundle/Contents/_CodeSignature/CodeRequirements
Test.bundle/Contents/Info.plist
```

- If the item exists within a bundle, it's covered by the bundle's code signature, as discussed below.
- Otherwise, the code signature is stored in extended attributes (EAs) on the item:

```
% cat hello.txt
Hello Cruel World!
% codesign -d -vvv hello.txt
…
Format=generic
…
Authority=Apple Development: Quinn Quinn (7XFU7D52S4)
…
% ls -l@ hello.txt
-rw-r--r--@ 1 quinn  staff  19 14 Mar 22:09 hello.txt
    com.apple.cs.CodeDirectory      129
    com.apple.cs.CodeRequirements   168
    com.apple.cs.CodeRequirements-1 165
    com.apple.cs.CodeSignature      4860
```

IMPORTANT Storing a code signature in EAs is brittle because many file transfer mechanisms drop these. If you follow the rules in Placing Content in a Bundle, none of your files will have code signature EAs.

Code Directory

The central concept in a code signature is the *code directory*. This is a data structure that holds all of the info about the code being signed. It's this data structure that's signed as part of the signing process [1], and it's the hashes in this data structure that seal the code, resources, and metadata of your product.

If you have a universal binary then each architecture is signed independently, so each has its own code directory.

Hashing the code directory results in a *code directory hash*, or *cdhash*. This value uniquely identifies the code being signed. It crops up in a variety of places, most notably notarisation. A notarised ticket is actually a set of cdhash values that have been signed by the notary service.

A code directory can be hashed by different algorithms, resulting in different cdhash values. Most code has a SHA-1 and a SHA-256 cdhash:

```
% codesign -d -vvv "/Applications/PCalc.app"
…
CandidateCDHash sha1=eca4d50c1736d878a5c4f5f7994960735311f314
CandidateCDHashFull sha1=eca4d50c1736d878a5c4f5f7994960735311f314
CandidateCDHash sha256=a9b23d5ca054140c96fb770ee8391d96d8515923
CandidateCDHashFull sha256=a9b23d5ca054140c96fb770ee8391d96d8515923804608438e2629770d5792c6
Hash choices=sha1,sha256
…
CDHash=a9b23d5ca054140c96fb770ee8391d96d8515923
…
```

The `CDHash` property is the One True CDHash™. The `CandidateCDHash` and `CandidateCDHashFull` properties are alternative cdhash values, each specifying a hash algorithm. The `Full` variant includes the full hash, while the other variant is truncated to 20 bytes to match SHA-1.

This structure was set up to enable stronger hashes (SHA-256) while still allowing the code to run on systems that only understand the old hashes (SHA-1). There's a bunch of fine detail for this that I've not researched in depth.

Also note that you'll get different values for each architecture. The example above was run on Intel, and so you get the Intel architecture's cdhash. To get the Apple silicon one, specify the `--arch` argument:

```
% codesign -d -vvv --arch arm64 "/Applications/PCalc.app"
…
CDHash=28cf639f72ac35f8d57d45a118c101a2285f0a31
…
```

[1] This uses CMS (Cryptographic Message Syntax) but I'm not going to go into those details here.

Per-Page Hashes

Within a code directory there are a set of hash slots. For a summary, look at the `CodeDirectory` property:

```
% codesign -d -vvv "/Applications/PCalc.app"
…
CodeDirectory v=20500 size=25155 flags=0x10000(runtime) hashes=775+7 location=embedded
…
```

This means there are 775 per-page hash slots and 7 special hash slots. Add more `-v` options to see a dump of these hashes:

```
% codesign -d -vvvvvv "/Applications/PCalc.app"
…
Page size=4096
    -7=86f48b256e85ed00d054bbe3bad62b424baf1028c91cb7b193d1d3f8ebbe3f4e
    -6=0000000000000000000000000000000000000000000000000000000000000000
    -5=bffee242018e6a39c46389fd913486fa29b3304712d32dd433f22d33b5a95da3
    -4=0000000000000000000000000000000000000000000000000000000000000000
    -3=8bee5419a9d27b02eee8f9897b6b94007b972cba348e4c863aa10df039b4b6c3
    -2=a9ff3a03b94f75fcdea2208d8a8fadabc4cf5e4b2d4e6835a14508d1686a99b6
    -1=f88548eecf39aa2e159365a9d4f0274a429e0793480abc8026d49fb2a8bf9ee9
     0=07b91feb157b8238efb3d8149bacf546dc1ded500449cfd7d5b6a8bf1ebb4b34
     1=f2f49595ca9daa599b4c41b69d99b6fede89eea39fa2000b452c7e26e06b8ddc
     2=d62187f6573798dce6752a05aea083ae14c050e429fbc76d2092379c3533ebe5
     3=a5a23c9cd825492a0ed5b5f541f2bbf12941bb8056f987f9d4e0f04d0885471f
     …
   774=f6d1072647ec9f0e79a55c772f5e31699bbb1c3c424a2d48f64d0b896ae5c255
…
```

The negative slots are special. More on that below.

The non-negative slots are for per-page hashes, that is, 0 is the hash for the first page of code, 1 for the second, and so on.

This per-page architecture means that the kernel can check each page as its loaded into memory. That is, if you take a page fault on a memory mapped file that's code signed then, as part of satisfying that fault, the kernel may choose to verify the hash of the page's contents. This allows the system to run a code-signed executable and check its code signature lazily [2].

macOS does not *always* check code as it is paged in. One key feature of the hardened runtime is that it opts the process in to this checking by default. The `com.apple.security.cs.disable-executable-page-protection` entitlement opts you out of this and other security features. Don't do that!

[2] I'm using the computer science definition of *lazy* here.

Special Slots

Within the code directory the negative hash slots are special. They don't correspond to code but rather to other data structures. Each slot number corresponds to a specific type of data. I'm not going to give a comprehensive list, but here's some highlights:

- Slot -1 holds a hash of the `Info.plist`.
- Slot -3 holds a hash of the resources.
- Slot -5 holds a hash of the entitlements.

Consider this:

```
% codesign -d -vvvvvv "/Applications/PCalc.app"
…
Page size=4096
    …
    -1=f88548eecf39aa2e159365a9d4f0274a429e0793480abc8026d49fb2a8bf9ee9
    …
% shasum -a 256 "/Applications/PCalc.app/Contents/Info.plist"
f88548eecf39aa2e159365a9d4f0274a429e0793480abc8026d49fb2a8bf9ee9 …
```

The `Info.plist` hash matches the value in its hash slot. Neat-o!

Now consider this advice from Using the Latest Code Signature Format:

> If -5 contains a value and -7 contains a zero value, or is not present, you need to re-sign your app to include the new DER entitlements.

You should now have a better handle on this diagnostic. Slot -5 holds the hash for the old school property list entitlements while slot -7 holds the hash for the new-style DER entitlements. If you have an entry in -5, that is, you have entitlements, but have no entry in -7, then you're missing your new-style DER entitlements and you must re-sign to run on iOS 15.

Resources

If your code exist in a bundle then the code signature protects not just your code but the resources in your bundle. Central to this is the `CodeResources` file. Slot -3 in the code directory holds the hash of that file:

```
% codesign -d -vvvvvv "/Applications/PCalc.app"
…
Page size=4096
    …
    -3=8bee5419a9d27b02eee8f9897b6b94007b972cba348e4c863aa10df039b4b6c3
    …
% shasum -a 256 "/Applications/PCalc.app/Contents/_CodeSignature/CodeResources"
8bee5419a9d27b02eee8f9897b6b94007b972cba348e4c863aa10df039b4b6c3 …
```

So, if that file changes, the code directory hash changes and you break the seal on the code signature.

Now let's look at that file:

```
% plcat "/Applications/PCalc.app/Contents/_CodeSignature/CodeResources"
…
<dict>
    <key>files</key>
    <dict>
        …
    </dict>
    <key>files2</key>
    <dict>
        …
    </dict>
    <key>rules</key>
    <dict>
        …
    </dict>
    <key>rules2</key>
    <dict>
        …
    </dict>
</dict>
</plist>
```

It's a property list with four top-level dictionaries: `files`, `files2`, `rules`, and `rules2`. Amusingly, three out of four of these items are vestigial. The one that matters is `files2`.

Note The `files` dictionary contains SHA-1 hashes and is present for compatibility purposes. The `rules` and `rules2` dictionaries are associated with resource rules, a concept that's now obsolete. For more on the move away from resource rules, see Technote 2206 macOS Code Signing In Depth.

The `files2` dictionary contains two kinds of items. Firstly, there are references to resources. For example:

```
% plutil -convert xml1 -o - "/Applications/PCalc.app/Contents/_CodeSignature/CodeResources"
…
<dict>
    …
    <key>files2</key>
    <dict>
        …
        <key>Resources/PCalc.help/Contents/Resources/PCalc.png</key>
        <dict>
            <key>hash</key>
            <data>
            wLi4SwDuno06WheWgINYKVT6LMg=
            </data>
            <key>hash2</key>
            <data>
            srK8T824L2l18d9eBixJC9nr3NUGCBh7L6qwsglQqFo=
            </data>
        </dict>
        …
    </dict>
    …
</dict>
</plist>
% shasum -a 256 "/Applications/PCalc.app/Contents/Resources/PCalc.help/Contents/Resources/PCalc.png"
b2b2bc4fcdb82f6975f1df5e062c490bd9ebdcd50608187b2faab0b20950a85a  …
% base64 -D | xxd -p
srK8T824L2l18d9eBixJC9nr3NUGCBh7L6qwsglQqFo=
b2b2bc4fcdb82f6975f1df5e062c490bd9ebdcd50608187b2faab0b20950
a85a
```

As you can see, the `hash2` property contains the SHA-256 checksum of the resource.

The other kind of item is nested code. For example:

```
% plutil -convert xml1 -o - "/Applications/PCalc.app/Contents/_CodeSignature/CodeResources"
…
<dict>
    …
    <key>files2</key>
    <dict>
        <key>Library/LoginItems/PCalc Widget.app</key>
        <dict>
            <key>cdhash</key>
            <data>
            3pJUbWYNPPnlN8SNi1CAgdzIRWg=
            </data>
            <key>requirement</key>
            <string>(anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.9] /* exists */ or anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] /* exists */ and certificate leaf[field.1.2.840.113635.100.6.1.13] /* exists */ and certificate leaf[subject.OU] = "9A6GM5K6XE") and identifier "com.pcalc.helper"</string>
        </dict>
        …
    </dict>
    …
</dict>
</plist>
```

The `cdhash` property contains the code directory hash of the nested code:

```
% codesign -d -vvv "/Applications/PCalc.app/Contents/Library/LoginItems/PCalc Widget.app"
…
CDHash=de92546d660d3cf9e537c48d8b508081dcc84568
…
% base64 -D | xxd -p
3pJUbWYNPPnlN8SNi1CAgdzIRWg=
de92546d660d3cf9e537c48d8b508081dcc84568
```

See, I told you cdhash values crop up all over the place!

The `requirement` property contains the designated requirement (DR) of that nested code:

```
% codesign -d -r - "/Applications/PCalc.app/Contents/Library/LoginItems/PCalc Widget.app"
…
designated => (anchor apple generic and certificate leaf[field.1.2.840.113635.100.6.1.9] /* exists */ or anchor apple generic and certificate 1[field.1.2.840.113635.100.6.2.6] /* exists */ and certificate leaf[field.1.2.840.113635.100.6.1.13] /* exists */ and certificate leaf[subject.OU] = "9A6GM5K6XE") and identifier "com.pcalc.helper"
```

In theory this lets you update the nested code with a new version, as long as it has the same DR. No one ever does this in practice (-:

For more information about code signing requirements, see Code Signing Guide, and specifically the Code Requirements section and the Code Signing Requirement Language appendix.

Code Signing

Reply

Posted 7 months ago by eskimo

Add a Comment