

DRAFT DOCUMENT – PRELIMINARY – NOT FINAL

IMPORTANT: This is a preliminary, unpublished document. The information presented in this document has not been reviewed for technical accuracy and its contents are subject to change without notice. All information in this document is confidential and it is not to be distributed or disclosed to anyone other than the intended recipient.

A primary category has yet to be selected for this document.

Technical Note TN2206

OS X Code Signing In Depth

The purpose of this technote is to provide a more in depth view of code signing. It is intended to expand upon the information given in the [Code Signing Guide](#) by supplying a more detailed analysis of the technology. The target audience for this document is OS X developers who have read and presumably understand the information given in the [Code Signing Guide](#) but want to learn a bit more.

This document is not meant to be applied to code signing on iOS, however. Xcode manages code signing on iOS; viewing the iOS documentation will give a clue as to the similarities.

-
- [Code Signing Recap](#)
 - [Trust and Code Signing](#)
 - [Code Requirements](#)
 - [Code Designated Requirement](#)
 - [Certificate Validity](#)
 - [Self-signed Identities and Self-created Certificate Authorities](#)
 - [Creating a Self-signed Code Signing Certificate using OpenSSL](#)
 - [Code signing changes in OS X Mavericks](#)
 - [Nested code](#)
 - [Using the --deep option to codesign correctly](#)
 - [Changes in OS X 10.9.5 and Yosemite Developer Preview 5](#)
 - [Troubleshooting](#)
 - [Signing Modifies the Executable](#)
 - [Signing Frameworks](#)
 - [Extended Key Usage](#)
 - [Shipping your Signed Code](#)
 - [Interpreting code signing failures](#)
 - [Document Revision History](#)
-

Code Signing Recap

Code signing is a facility by which developers can assign a digital identity to their programs. Apple provides the

tools necessary to sign your programs (see the [codesign](#) manual page).

Code signing on OS X is an integral part of the development process. Most code signing certificates are provided by Apple or internally provisioned by enterprise IT departments. While tools like Xcode handle much of the certificate management, you can also maintain your signing certificates yourself if your situation calls for it.

In short, code signing is a technology that allows you to dictate how validating mechanisms will interpret your code. Code signing does implement some policy checks. However, policy is mostly set by the specific subsystem carrying out validation; any policy decisions outside of those implemented by OS X subsystems are left up to you and your end users in how you interoperate between a specific set of subsystems.

Trust and Code Signing

Trust is determined by policy. A security trust policy determines whether a particular code identity, which is essentially the [designated requirement](#) (DR) for the code, should be accepted for allowing something to happen on the system, e.g., access to a resource or service, after testing for validity. Each OS X subsystem has its own policy, and makes this determination separately. Thus, it makes no sense to ask whether code signing trusts a particular signature. You have to ask based on the subsystem, and it is more meaningful to ask whether a specific subsystem trusts your signature.

In general, most subsystems do not care that your identity certificate chain leads to a trusted anchor, however, some do. Additionally, some subsystems track identities and some don't. Subsystem tracking alludes to how the subsystem verifies an identity **after** the initial policy decision has been acted upon. For a concrete example, below is a list of commonly-used subsystems that verify code signatures:

Table 1 : Examples of OS X subsystems that verify the validity of code.

Subsystem	Function	Initial Policy	Tracking Policy
App Sandbox	Gate access to system resources based on entitlements.	Allow if entitlement is present in the app's code signature.	Initial policy decision is verified against the application's DR.
Gatekeeper	Restrict launching of applications from unidentified developers	A configurable trusted anchor check (Developer ID or Mac App Store).	None (each request is evaluated by policy).
Application Firewall	Restrict inbound network access by applications.	Allow if a trusted anchor check succeeds; otherwise prompt the user.	Initial policy decision is verified against the application's DR.
Parental Controls (MCX)	Restrict what applications a managed user can run.	Explicit administrator decision (no code signing involved in the initial decision).	Initial policy decision is verified against the application's DR.
Keychain Access Controls	Controls what applications can do with specific keychain items.	The creating application is automatically trusted with its item, and determines the access policy using code signing requirements.	Free access to the keychain item by the creating application and tracked with its DR (No automatic tracking for custom ACLs).
Developer Tools Access (DTA)	Restrict what programs are allowed to call DTA APIs (task_for_pid, etc.)	A hard-coded trusted anchor check.	None (each request is evaluated by policy).

The above examples also further emphasize the fact that all policy decisions are determined by a specific subsystem and not by code signing itself. In addition, it highlights the diversity in how code signing can be used by a specific subsystem to carry out policy. For instance:

- DTA doesn't even have a tracking policy. It simply applies a set requirement to every requester without needing to retain any information.
- Parental controls show that you don't have to even use code signing at all in order to craft a usable policy.
- Application Firewall uses code signing for both its initial and tracking policy decisions.
- The keychain acts on the tracking policy by default but it can also allow arbitrary requirement-bearing ACLs to be added to express arbitrary policies determined by the owner of a specific keychain item.

Note: The keychain access controls can allow you to associate arbitrary code signing requirements with keychain items. This means that trusted anchor requirements can be attached to keychain items, either with explicit API calls, or by creating an item with an application whose designated requirement has been explicitly set to require a trusted anchor. However, this does not happen by default.

Many parts of OS X do not care about the identity of the signer. (Gatekeeper is a notable exception.) They care only whether the program is **validly signed and stable**. Stability is determined through the [designated requirement \(DR\)](#) mechanism, and does not depend on the nature of the certificate authority used. The keychain system and parental controls are examples of such usage. Self-signed identities and homemade certificate authorities (CA) work by default for this case.

Other parts of OS X constrain acceptable signatures to only those drawn from certificate authorities that are trusted on the system performing the validation. For those checks, the nature of the identity certificate used does matter. The [Application Firewall](#) is one example of this usage. Self-signed identities and self-created CA will not be verified as being valid for this check unless the verifying system has been told to trust them for Application Firewall purposes.

Note: In order for a new identity certificate to be designated as being a trusted anchor for a particular subsystem, the user must take action to accept this policy addition. For a system-wide trust entry higher privileges are needed, therefore, an administrator user is required to accept the policy addition.

Please keep in mind that using a signing identity that is system-wide trusted doesn't automatically mean that it's:

- a requirement for a signature to be valid.
- going to be ignored by the majority of subsystems.
- going to matter only to particular checks within particular subsystems.

[Back to Top](#) ▲

Code Requirements

A code requirement is a statement that expresses constraints on a validly signed application. Code signature validation can accept a requirement as input which will then be used to check whether the code is validly signed and satisfies the constraints of the requirement. When signing code, it is not normally necessary to concern yourself with code requirements. They will be managed implicitly by `codesign` and Xcode. However, in rare cases you may need to explicitly override the default settings to achieve particular effects.

To explicitly test whether a program satisfies a particular requirement, use the `-R` option to the `codesign` command; for example:

```
$ # Make a copy of the md5 tool.
$ cp /sbin/md5 .
$ # The copy still satisfies its DR.
```

```
$ codesign -vvvv ./md5
./md5: valid on disk
./md5: satisfies its Designated Requirement
$ # And we can check that it's signed by Apple.
$ codesign -vvvv -R="anchor apple" ./md5
./md5: valid on disk
./md5: satisfies its Designated Requirement
./md5: explicit requirement satisfied
$ # Modify the binary.
$ chmod u+w ./md5
$ dd if=/dev/zero bs=1 count=1 seek=8192 conv=notrunc of=./md5
1+0 records in
1+0 records out
1 bytes transferred in 0.000036 secs (27777 bytes/sec)
$ # The modified program no longer satisfies its DR.
$ codesign -vvvv ./md5
./md5: code or signature modified
$ # But we can resign the modified program with our signature.
$ codesign -s my-signing-identity -f ./md5
./md5: replacing existing signature
$ # And the modified program now satisfies its DR.
$ codesign -vvvv ./md5
./md5: valid on disk
./md5: satisfies its Designated Requirement
$ # But not our supplement requirement.
$ codesign -vvvv -R="anchor apple" ./md5
./md5: valid on disk
./md5: satisfies its Designated Requirement
test-requirement: failed to satisfy code requirement(s)
```

The requirement language is a set of rules that can be chained together using logical operators ("and", "or", "not", and parentheses to denote precedence) to form a requirement expression. Below is the current list of supported rules and their usage:

Table 2 : The requirement language syntax.

Rule Syntax Usage	Description
identifier <string>	The signing identifier is exactly the string given
certificate <slot> = <hash>	The certificate in the certificate chain has a SHA-1 hash as given
certificate <slot> trusted	The certificate is trusted as per Trust Settings API for code signing
certificate <slot> [<key>] = <value>	Some part of the certificate has the given value
info [<key>] = <value>	The Info.plist has a key with the given value
cdhash <hash>	The CodeDirectory's SHA-1 hash is the given value
anchor <string>	The root certificate given by its path
anchor trusted	The certificate chain must lead to a trusted root
anchor apple	The certificate chain must lead to an Apple root

Some important things to keep in mind:

- When you pass a path of a certificate to set as an anchor in the [designated requirement \(DR\)](#) the DR will automatically be transformed to store only the SHA-1 hash of that certificate. When the policy engine then evaluates the validity of the signed program it uses the stored hash value found in the DR to compare to the

actual anchor.

- The signing identifier is also embedded in the DR and will default to the `CFBundleIdentifier` found in the `Info.plist` for convenience if one is not supplied explicitly. The identifier has no meaning as far as code signing is concerned, other than as a means to make DRs unique.
- You may pass in negative integers as an index into the certificate chain array if you want the searching origin to start with the root certificate rather than the leaf as the origin; the value for the leaf certificate and root certificate are 0 and -1, respectively. The distance for the leaf origin (positive integer) is measured by the absolute value of the integer value passed in. Whereas, the distance for the root origin (negative integer) is measured by the absolute value of 1 + the integer value passed in.
- To manipulate or experiment with requirements, use the [csreq](#) command.

[Back to Top](#) ▲

Code Designated Requirement

All signed code has a designated requirement (DR). This requirement states, from the perspective of the developer of the program, what constraints a program needs to satisfy in order to be considered an instance of this program. Obviously, every program should satisfy its own DR, e.g., `codesign -vv` checks for this. More interestingly, a program's DR should also be satisfied by updates, i.e., new versions of that code, and by nothing else. This is how the OS X code signing policy engine recognizes updates and upgrades.

By default, the system synthesizes a suitable DR for your code when you sign your program. This will work fine in most cases. However, you may specify an explicit DR when signing your program, and there are situations where you should do so.

To see what DR a program has:

```
$ codesign -d -r- /sbin/md5
Executable=/sbin/md5

# designated => identifier "com.apple.md5" and anchor apple
```

Look for the line starting with "designated =>". If it is commented out (starts with a "#" mark), it is implicitly generated. If not, it is explicit.

Use the `-r` option to the `codesign` command to explicitly specify a DR when signing; for example:

```
$ codesign -vvvv -s my-signing-identity -r="designated => anchor trusted"
~/Desktop/CodesignTest
/Users/admin/Desktop/CodesignTest: signed Mach-O thin (i386) [CodesignTest]
```

If you do create a DR, you are responsible for crafting a suitable requirement to use. For example below is a DR for specifying to the policy engine that it should check that the signer of the program leads to a trusted anchor on the calling system and that the program's identifier matches the supplied parameter string.

```
$ codesign -vvvv -s my-signing-identity -r="designated => anchor trusted and \
identifier com.foo.bar" ~/Desktop/CodesignTest
/Users/admin/Desktop/CodesignTest: signed Mach-O thin (i386) [CodesignTest]
```

Note: If you're creating a CA for generating code signatures, then the organization (O) element of the subject distinguished name (DN) from the certificate should be kept consistent throughout your chain of certificates.

IMPORTANT: If you're using a certificate from a CA, they may require that you place certain criteria in your DR.

Consult with your CA on this matter.

[Back to Top](#) ▲

Certificate Validity

Except as stated in the next paragraph, the code signing and validation engines accept signatures made with expired certificates. This means that your signed code will not become invalid when your certificate expires. In simple applications, you can continue signing with an expired certificate and OS X will continue to accept this. Code signing will reject signatures made with identities that have been revoked according to standard X.509 processing rules (See [RFC 3280](#) for an example). Revocation check instructions have to be embedded in the certificates you want checked. Revocation checking is a per-user preference found in Keychain Access. When revocation checking is configured and enabled the system may still be unable to ascertain revocation status if it is disconnected from the network in which case the validation might succeed instead of fail, i.e., if certificate revocation Keychain Access preferences are set as "Best Attempt" instead of "Require if Cert Indicates".

Developer ID signatures carry cryptographic timestamps by default. Signatures with cryptographic timestamps are validated against the signing time, and signatures made with expired (at signing time) certificates are invalid. The previous discussion still applies to Developer ID signatures without secure timestamps.

[Back to Top](#) ▲

Self-signed Identities and Self-created Certificate Authorities

Depending on the policy used by the subsystem in question, a self-signed identity can usually be used (your program will reap all the benefits of being signed by it) as long as that identity is set following the respective policy. Obviously, one big downside in using a self-signed identity is that you will never be able to revoke it, however, it may be sufficient for your organization's certificate policy requirements or if you just want to test out the code signing machinery.

If you decide to create your own CA then you specify an explicit DR naming your own anchor certificate:

Listing 1: Creating a DR

```
$ codesign -vvvv -s my-signing-identity -r="designated => anchor rootCert and identifier  
com.foo.bar" \  
~/Desktop/CodesignTest  
/Users/admin/Desktop/CodesignTest: signed Mach-O thin (i386) [CodesignTest]
```

By using your own CA you gain the ability to issue new identities at will, since any signing identities issued from your CA will now satisfy the check. You can do this by selecting "Create a Certificate for Someone Else as a Certificate Authority" as option for the Certificate Assistant in Keychain Access. You can also do this with [openssl](#):

```
$ openssl ca -out cert.rsa -config ./openssl.cnf -infiles request.csr
```

Just as in a custom created CA, if you buy a signing certificate from a commercial CA you'll want to craft a DR that expresses the CA vendor's issuance policies. For instance, every time your CA reissues you a new certificate your identities will change which is something that your DR should take care to handle.

[Back to Top](#) ▲

Creating a Self-signed Code Signing Certificate using OpenSSL

If you already have an SSL identity, i.e., a public and private key pair generated through OpenSSL, and you want to use it for code signing then you will need to use something other than the Certificate Assistant. This is because converting an existing OpenSSL digital identity for use with code signing is currently unsupported by the Certificate Assistant. However, you can solve this problem using `openssl`. The steps that you need to take are:

- First, create a certificate signing request (CSR) using [openssl](#):

```
$ openssl req -new -key ./key.rsa -out ./key.csr -config ./openssl.cnf
```

- Second, create a certificate using [openssl](#):

```
$ openssl x509 -req -days 10 -in ./key.csr -signkey ./key.rsa -out ./key.crt -extfile  
./openssl.cnf  
-extensions codesign
```

- Third, create a keychain and import your private key using [certtool](#):

```
$ certtool i ./key.crt k="`pwd`/key.keychain" r=./key.rsa c p=moof
```

- Lastly, pass that keychain in for use with [codesign](#):

```
$ codesign -s my-signing-identity --keychain key.keychain ~/Desktop/CodesignTest
```

IMPORTANT: If you are generating a code signing identity from scratch, Apple strongly recommends you use the Certificate Assistant component of Keychain Access, which is equivalent to the OpenSSL approach but in addition it:

- has a GUI
- directly deposits the identity certificate into a keychain
- as the ability to form genuine CAs and issue invitations and certificates to other users

[Back to Top](#) ▲

Code signing changes in OS X Mavericks

OS X v10.9 Mavericks introduced a number of significant changes to the code signing machinery. Most of these changes apply to the **resource envelope** of a code signature, where the signature keeps a list of files in a bundle. The pre-Mavericks version, version 1, recorded only files in the Resources directory and ignored the rest. The Mavericks version, version 2, makes the following changes:

- It records substantially all files by default. There are no default "holes".
- It records nested code (frameworks, dylibs, helper tools and apps, plug-ins, etc.) by recording their code signature for verification.
- It records symbolic links. Version 1 resource envelopes ignore symlinks.

Note: Code signatures containing version 1 or version 2 resource envelopes are also known as **version 1 signatures** or **version 2 signatures**, respectively.

A signature may contain multiple versions of resource envelopes. Mavericks generates, by default, both version 2 and version 1 resource envelopes. Mavericks verifies version 2 resource envelopes. Pre-Mavericks systems ignore the version 2 resource envelope and use the version 1 resource envelope. If Mavericks sees a version 1 signature, it performs version 1 validation.

To determine which version of resource envelope a code signature has, use `codesign -dv` and note the version of the sealed resources, like this:

```
$ codesign -dv Chess.app/  
[...]  
Sealed Resources version=2 rules=15 files=53  
[...]
```

Note: `codesign` on OS X Mavericks and later does not show the version 1 resource envelope if a version 2 resource envelope is present, as only the version 2 resource envelope will be used on Mavericks and later.

Note: Signatures in single-file executables like command line tools do not have a resource envelope, so `codesign` won't show the `Sealed Resources` line for those files.

Systems before OS X Mavericks documented a signing feature (`--resource-rules`) to control which files in a bundle should be sealed by a code signature. This feature has been deprecated for Mavericks. Code signatures made in Mavericks and later always seal all files in a bundle; there is no need to specify this explicitly any more.

Nested code

From Mavericks onwards, signatures record nested code by its code signature and embed that information in the (outer) signature's resource envelope, recursively. This means that when a code signature is created, all nested code **must already be signed correctly** or the signing attempt will fail.

This is not a problem if you follow the standard Xcode build flow, because it will build and sign targets inside out: build the innermost code, copy it into the next-outer bundle, then sign that, etc. Always set signing identities for an entire project (not individual targets) to ensure that all targets are signed.

Nested code is expected in a number of standard locations within a bundle.

Table 3 : Standard locations for code inside a bundle

Location	Description
Contents	Top level of the bundle
Contents/MacOS	Helper apps and tools
Contents/Frameworks	Frameworks, dylibs
Contents/PlugIns	Plug-ins, both loadable and Extensions
Contents/XPCServices	XPC services
Contents/Helpers	Helper apps and tools
Contents/Library/Automator	Automator actions
Contents/Library/Spotlight	Spotlight importers
Contents/Library/LoginItems	Installable login items

These places are expected to contain **only** code. Putting arbitrary data files there will cause them to be rejected (since they're unsigned). Conversely, putting code into other places will cause it to be sealed as data (resource) files,

causing trouble during updates. Always put code and data into their proper places.

Note that a location where code is expected to reside cannot generally contain directories full of nested code, because those directories tend to be interpreted as bundles. So this occasional practice is not recommended and not officially supported.

[Back to Top](#) ▲

Using the `--deep` option to codesign correctly

When verifying signatures, add `--deep` to perform recursive validation of nested code. Without `--deep`, validation will be shallow: it will check the immediate nested content but not check that fully. Note that Gatekeeper always performs `--deep` style validation.

IMPORTANT: While the `--deep` option can be applied to a signing operation, this is not recommended. We recommend that you sign code inside out in individual stages (as Xcode does automatically). Signing with `--deep` is for emergency repairs and temporary adjustments only.

Note that signing with the combination `--deep --force` will forcibly re-sign all code in a bundle.

[Back to Top](#) ▲

Changes in OS X 10.9.5 and Yosemite Developer Preview 5

Beginning with OS X version 10.9.5, there will be changes in how OS X recognizes signed apps. Version 1 signatures created with OS X versions prior to Mavericks will no longer be recognized by Gatekeeper and are considered obsolete.

IMPORTANT: For your apps to run on updated versions of OS X they **must** be signed on OS X version 10.9 or later and thus have a version 2 signature.

If your team is using an older version of OS X to build your code, re-sign your app using OS X version 10.9 or later using the `codesign` tool to create version 2 signatures. Apps signed with version 2 signatures will work on older versions of OS X.

If your app is on the Mac App Store, submit your re-signed app as an update.

Structure your bundle according to the expectations for OS X version 10.9 or later:

- Only include signed code in directories that should contain signed code.
- Only include resources in directories that should contain resources.
- Do not use the `--resource-rules` flag or `ResourceRules.plist`. They have been obsoleted and will be rejected.

IMPORTANT: To ensure your current and upcoming releases work properly with Gatekeeper, test on OS X version 10.10 (Seed 5 or later) and OS X version 10.9.5.

Note: It is necessary to sign code while running OS X Mavericks to get a version 2 signature. The actual code signing machinery is part of the operating system, not the `codesign` tool. It will not work to copy the `codesign` tool from Mavericks to an older OS X version.

[Back to Top](#) ▲

Troubleshooting

Signing Modifies the Executable

Signing a program will modify its main executable file. There are some situations where this will cause you trouble:

- If your program has a self-verification mode that detects a change, your code may refuse to run.
- Appending data to a Mach-O executable is expressly prohibited. Signature verifications on such files will fail.

The obvious solution to these problems is to not meddle with your signed program after you've signed it with `codesign`. If you're modifying the executable or bundle in any way then the code signing validation engine will obviously pick up on that change and act appropriately with the set policy. If you've set your program to do self-integrity checking then it is possible that your preconceived notion of what constitutes "your program" is likely to have changed due to code signing. More specifically, whether you're checking the entire contents of the Mach-O file or just the aggregation of certain pieces of the file it's highly likely that code signing will break what you believe integrity checking is. For example:

```
$ # Let's see what the Mach-O file looks like pre-signing:
$ otool -l ~/Desktop/CodesignTest
CodesignTest:
Load command 0
    cmd LC_SEGMENT
    cmdsize 56
    segname __PAGEZERO
    vmaddr 0x00000000
    vmsize 0x00001000
    fileoff 0
    filesize 0
    maxprot 0x00000000
    initprot 0x00000000
    nsects 0
    flags 0x0
[...]
Load command 11
    cmd LC_LOAD_DYLIB
    cmdsize 52
    name /usr/lib/libSystem.B.dylib (offset 24)
    time stamp 2 Wed Dec 31 16:00:02 1969
    current version 111.0.0
compatibility version 1.0.0
$ # Now let's see what it looks like after signing:
$ codesign -vvvv -s my-signing-identity ~/Desktop/CodesignTest
CodesignTest: signed Mach-O thin (i386) [CodesignTest]
$ otool -l CodesignTest
CodesignTest:
Load command 0
    cmd LC_SEGMENT
    cmdsize 56
    segname __PAGEZERO
    vmaddr 0x00000000
    vmsize 0x00001000
    fileoff 0
    filesize 0
    maxprot 0x00000000
    initprot 0x00000000
    nsects 0
    flags 0x0
[...]
Load command 11
    cmd LC_LOAD_DYLIB
    cmdsize 52
    name /usr/lib/libSystem.B.dylib (offset 24)
    time stamp 2 Wed Dec 31 16:00:02 1969
```

```
current version 111.0.0
compatibility version 1.0.0
Load command 12
    cmd LC_CODE_SIGNATURE
    cmdsize 16
    dataoff 12816
    datasize 5232
$ # Notice the extra load command LC_CODE_SIGNATURE at number 12!
$ # Let's check it out even further with pagestuff:
$ pagestuff CodesignTest -a
File Page 0 contains Mach-O headers
[...]
File Page 3 contains local of code signature
File Page 4 contains local of code signature
```

[Back to Top](#) ▲

Signing Frameworks

Seeing as frameworks are bundles it would seem logical to conclude that you can sign a framework directly. Most frameworks contain a single version and can in fact be signed directly. Signing the framework as a whole signs its "Current" version by default.

Multi-versioned frameworks are discouraged in general. However, if you happen to have one, make sure that you sign each specific version as opposed to the whole framework:

```
$ # This is the wrong way to sign a multi-versioned framework:
$ codesign -s my-signing-identity ../FooBarBaz.framework
$ # This is the right way:
$ codesign -s my-signing-identity ../FooBarBaz.framework/Versions/A
$ codesign -s my-signing-identity ../FooBarBaz.framework/Versions/B
```

Multi-versioned frameworks are "versioned bundles", and each contained version should be signed and validated separately.

[Back to Top](#) ▲

Extended Key Usage

Standard X.509 certificates ([RFC 2459](#)) contain object identifiers (OID) which form key usage extensions to define what the public and private key can and cannot be used for. Extended key usages just further refine the key usage extensions. An extension is either critical or non-critical. If the extension is critical then the identity must only be used for indicated purpose(s).

The X.509 certificates and their code signing extended key usage is obviously required for an identity to be used for code signing on OS X. However, the code signing extended key usage should also be the **only** extended key usage for a certificate (this code signing extended usage is critical) in order to be valid to the OS X code signing subsystem. It is not possible to create one certificate that can be used for both code signing and other purposes.

You can check your certificate to find out whether the code signing extended key usage attribute is present by viewing the certificate in Keychain Access or by using any other X.509 compliant certificate parser. Dumping all the available information about a certificate can also show if the certificate has other usages which will cause it to be an invalid identity for use with code signing on Mac OS X. For example:

```
$ # Use security and certtool
$ security find-certificate -a -e clarus@apple.com -p > cert.pem
$ certtool d cert.pem
Serial Number : 01
Issuer Name :
    Common Name : Testing Code Signing
    Org : Apple Inc.
    OrgUnit : DTS
    State : CA
    Country : US
```

```
Locality : Cupertino
Email addr : clarus@apple.com
Subject Name :
Common Name : Testing Code Signing
Org : Apple Inc.
OrgUnit : DTS
State : CA
Country : US
Locality : Cupertino
Email addr : clarus@apple.com
Cert Sig Algorithm : OID : < 06 09 2A 86 48 86 F7 0D 01 01 05 >
alg params : 05 00
Not Before : 19:27:16 Nov 14, 2007
Not After : 19:27:16 Nov 13, 2008
Pub Key Algorithm : OID : < 06 09 2A 86 48 86 F7 0D 01 01 01 >
alg params : 05 00
Pub key Bytes : Length 270 bytes : 00 00 00 00 00 00 00 00 ...
CSSM Key :
Algorithm : RSA
Key Size : 2048 bits
Key Use : CSSM_KEYUSE_VERIFY
Signature : 256 bytes : FF FF FF FF FF FF FF FF ...
Other field: : OID : < 06 0C 60 86 48 01 86 F8 4D 02 01 01 01 17 >
Other field: : OID : < 06 0C 60 86 48 01 86 F8 4D 02 01 01 01 16 >
Extension struct : OID : < 06 03 55 1D 0F >
Critical : TRUE
usage : DigitalSignature
Extension struct : OID : < 06 03 55 1D 25 >
Critical : TRUE
purpose 0 : OID : < 06 08 2B 06 01 05 05 07 03 03 >
$admin>
```

[Back to Top](#) ▲

Shipping your Signed Code

Code signing uses [extended attributes](#) to store signatures in non-Mach-O executables such as script files. If the extended attributes are lost then the program's identity will be broken. Thus, when you ship your script, you must use a mechanism that preserves extended attributes.

One way to guarantee preservation of extended attributes is by packing up your signed code in a read-write disk image (DMG) file before signing and then, after signing, converting to read-only. You probably don't need to use a disk image until the final package stage so another less heavy-handed method would be to use ZIP or [XIP](#) files.

[Back to Top](#) ▲

Interpreting code signing failures

If signing or validation fails in the `codesign` command due to problems with nested code, the command will output an additional line

```
In subcomponent: path
```

indicating which nested code caused the problem. Always look for this line to correctly interpret a code signing failure.

If the Xcode Organizer produces a code signing error during distribution signing, the problem may be with improperly placed code or resources and not a problem with your signing identities.

[Back to Top](#) ▲

Document Revision History

Date	Notes
2014-07-29	

2014-07-28	Added discussion of significant code signing changes in OS X Mavericks. Other editorial changes.
2014-06-24	Omitting files from the signature's seal is deprecated on OS X Mavericks.
2008-08-06	New document that intermediate to expert level overview of OS X code signing that details specific options and gotchas

Posted: 2014-07-29