Technical Note TN2206

macOS 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

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 macOS 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 macOS 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 macOS 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 macOS 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 macOS 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 macOS 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.

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.

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 macOS 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.

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 macOS 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 10.9

OS X Mavericks 10.9 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" (omit rules).
- 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 only 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 macOS 10.9 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.

Resource Rules

Systems before OS X Mavericks 10.9 documented a signing feature `(--resource-rules)` to control which files in a bundle should be sealed by a code signature. This feature has been obsoleted 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. This also means that the Code Signing Resource Rules Path build setting in Xcode should no longer be used and should be left blank.

It is thus no longer possible to exclude parts of a bundle from the signature. Bundles should be treated as read-only once they have been signed.

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., signing the outermost bundle last.

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 content directory 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

Contents/Library/LaunchServices	Privileged helper tools installed by the ServiceManagement framework

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. This causes this code to be sealed twice; once in the nested code, and once in the outer signature. This wastes both signing and verification time and storage space. Also, this can break the outer signature of apps that use their own update mechanisms to replace nested code. If this nested code is being treated as resources, the outer signature doesn't know that this nested content is actually code.

Always put code and data into their proper places. This applies to all signed code and is enforced by the code signing machinery regardless of how the code is distributed.

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. If you do do this, do not use periods in the directory names. The code signing machinery interprets directories with periods in their names as code bundles and will reject them if they don't conform to the expected code bundle layout.

Store Python, Perl, shell, and other script files and other non-Mach-O executables in your app's `Contents/Resources` directory. While it's possible to sign such executables and store them in `Contents/MacOS`, this is not recommended.

To see why, it's necessary to understand that 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 signature will be broken. Many file transfer techniques do not preserve extended attributes, nor are they preserved when uploading to the Mac App Store.

Put another way, a properly-signed app that has all of its files in the correct places will not contain any signatures stored as extended attributes.

There may not be any content in the top level of a bundle. In other words, if a bundle has a `Contents` or `Versions` directory at its top level, there may be no other files or directories alongside them. The one exception is that alongside `Versions`, there may be symlinks to files in `Versions/Current`.

The code signing machinery performs some framework checks specifically on frameworks that are nested within other code. It's possible that signing a framework will succeed, but the result fails to validate when placed into another bundle's `Frameworks` directory. Make sure the framework is structured correctly per the requirements above.

Bundles must have their `Info.plist` in the proper location. For app bundles, this is in `Contents`. For frameworks, this is in `Versions/Current/Resources`.

Note: While strict compliance with these rules may not affect your app today, anything that doesn't meet these requirements note may be rejected by code signing verification (and the Mac App Store validator, in the case of Mac App Store apps) at any point in the future without notice.

Back to Top

Gatekeeper Changes in macOS 10.9.5 and Later

On macOS 10.9.5 and later, there are changes in how Gatekeeper recognizes signed apps. Version 1 signatures created with macOS 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 macOS they **must** be signed on macOS v10.9 or later and thus have a version 2 signature. They also must not contain any custom resource rules.

If your team is using an older version of macOS to build your code, re-sign your app using macOS v10.9 or later using the `codesign` tool to create version 2 signatures. Apps signed with version 2 signatures will work on older versions of macOS.

Structure your bundle according to the expectations for macOS 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 both OS X 10.9.5 and OS X Yosemite 10.10.

> **Note:** The changes in this section also apply to OS X 10.8.5 if Security Update 2014-004 for OS X Mountain Lion v10.8.5 has been installed.

Back to Top

Gatekeeper Changes in macOS 10.10.4 and Later

On macOS 10.10.4 and later, Gatekeeper verifies libraries loaded from outside an application bundle.

If an app uses `@rpath` or an absolute path to link to a dynamic library outside of the app, the app will be rejected by Gatekeeper. This restriction applies to the app's main executable and any other executable in the bundle, including libraries. This restriction applies even if the path does not exist (and normally causes the dynamic linker to fall back to a library inside the bundle).

If Gatekeeper rejects launching an app for this reason, the system log will note `Fails dylib check`, like this:

```
7/8/15 4:00:19.942 PM CoreServicesUIAgent[5940]: File
/Applications/MyApp.app/Contents/MacOS/MyApp failed on loadCmd /foo/libLibrary.dylib

7/8/15 4:00:19.942 PM CoreServicesUIAgent[5940]: Fails dylib check
```

This means that `MyApp.app` links against `/foo/libLibrary.dylib`. `/foo` is not a standard location for libraries on macOS, so the linkage isn't allowed.

Neither the `codesign` nor the `spctl` tool will show the error. The error will only appear in the system log.

This check is performed the first time the app is run. It does not apply to libraries the app loads itself using the `dlopen` function. It also does not apply to libraries loaded from paths where libraries are expected to reside, such as `/System`, `/Library`, and `/usr/`.

To see which libraries an app references, use the command `otool -L MyApp.app/Contents/MacOS/MyApp`.

> **Note:** The changes in this section also apply to OS X 10.9.5 if Security Update 2015-005 Mavericks has been installed.

The changes in this section also apply to OS X 10.8.5 if Security Update 2015-005 Mountain Lion has been installed.

Back to Top

Gatekeeper Changes in macOS 10.11 and Later

On macOS 10.11 and later, signatures that don't cover the entire code are rejected. This should not affect anyone using normal build tools.

Gatekeeper also rejects apps containing symbolic links that:

- point to nowhere
- point to places that are legitimately excluded from the app's signature
- point outside the app bundle, except to locations in `/System` and `/Library`.

However, a nested bundle may contain symlinks that point into the enclosing bundle.

While Gatekeeper always performs this check, the `codesign` tool does not perform this check unless the `--strict=symlinks` or `--strict` option is provided. `--strict` means to perform as strict a check as possible and apply the validation to the topmost bundle of a nested hierarchy to reproduce what Gatekeeper and `spctl` do.

Back to Top

macOS Code Signing Tips and Tricks

Checking Gatekeeper Conformance

To test Gatekeeper conformance, you must use macOS 10.9.5 or later. Follow these steps:

- Package your program the way you ship it, such as in a disk image.
- Download it from its website, or mail it to yourself, or send it to yourself using AirDrop or Message. This will quarantine the downloaded copy. This is necessary to trigger the Gatekeeper check as Gatekeeper only checks quarantined files the first time they're opened.

  Hint: keep the downloaded `.dmg` around; it will stay quarantined and you can use it again and again to test.
- Drag-install your app and launch it.
- Observe the results.

  Hint: Don't launch your app from inside the `.dmg`. Drag the app to /Applications and launch it from there.

What results did you get?

- If there is no dialog at all, a step was missed. Check the instructions and repeat the test.
- If you get the dialog `... is from the Internet` with an `Open` button, the test succeeded.

- If you're told that only apps from the Mac App Store or registered developers can be installed, your app isn't Developer ID–signed. Use the Xcode Organizer to export a Developer ID–signed copy of your app and repeat the test. It may also be that the system doesn't think your bundle is an app bundle because its `Info.plist` doesn't have its `CFBundlePackageType` property set to `APPL`.
- If you get any other complaint, your signature is weak or broken. Please review the Version 2 Code Signature FAQ.

This command:

```
$ codesign --verify --deep --strict --verbose=2 Foo.app
```

mimics what Gatekeeper does to check your app.

You can also use the `check-signature` tool to check both apps and installer packages.

Mount the disk image, then run the tool like this:

```
$ cd "/Volumes/Signature Check"
$ ./check-signature /Path/to/Foo.app /Path/to/Bar.pkg
```

For each target, the tool will present a simple `YES` answer if the signature meets Gatekeeper requirements, or `NO` if it does not.

Read the error messages carefully, with particular attention to the `in subcomponent:` part which, if present, tells you which nested code is giving you problems.

Understand that this validation will stop on many errors, and thus you must repeat it until you run out of problems.

You can also use the `spctl` tool to check if Gatekeeper will accept your app's signature. `spctl` is a command-line interface to the same security assessment policy subsystem that Gatekeeper uses.

Like Gatekeeper, `spctl` will only accept Developer ID–signed apps and apps downloaded from the Mac App Store by default. It will reject apps signed with Mac App Store development or distribution certificates.

Important: `spctl` must only be run on top-level app bundles. If it is run against other bundles such as embedded helper apps or frameworks, the tool will return an error.

Run `spctl` on your app like this:

```
$ spctl -a -t exec -vv Foo.app
```

This is the output if your app's signature will be accepted:

```
Foo.app: accepted
source=Developer ID
```

`source` will be `Mac App Store` for apps downloaded from the Mac App Store.

If `spctl` shows any result other than `accepted`, you must re-sign your app on Mavericks or later to ensure Gatekeeper compatibility.

Note: It is necessary to sign code while running macOS 10.9 or later 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 operating system version.

Users can override Gatekeeper's assessment of your app's signature using the steps described in this support article.

Moving Away From Custom Resource Rules

If you used custom resource rules because your program modifies itself as it is running, you should stop doing this as soon as possible. If you transport such a modified app to another computer by web download, email, AirDrop, etc., the modified copy at the receiving end will be quarantined. This will trigger the Gatekeeper check the first time the app is run on the receiving machine, and the app will not be allowed to run because of the broken signature.

If you used custom resource rules because your installation process relies on changing the bundle, your app will be rejected by Gatekeeper on first launch. These modifications are not permitted. Using an installation package instead of a drag-install will get you through Gatekeeper.

To support the creation of droplet-like apps or similar forms of personalization, it is acceptable to place the string representation of a UUID (e.g. `16F2ACA8-AA29-48ED-9ED9-F96F3A230505`) into an extended attribute named `com.apple.application-instance` of the top-level directory of your bundle (e.g. `MyApp.app`). This will not invalidate the code signature or affect the bundle's code identity. You may then link this UUID to storage outside the bundle that carries the necessary configuration data, such as NSUserDefaults, a file in the `Application Support` directory, or a web service.

Note: See Determining Where to Store Your App-Specific Files for guidance on using the `Application Support` directory.

Placing configuration data directly into extended attributes is not supported. Creating a configuration file outside of the app bundle is also not supported.

Changes That Don't Invalidate a Code Signature

There are a few changes you can make to a signed bundle that won't invalidate its signature.

If you have optional or replaceable content you wish to change without invalidating the code signature, nested code can be replaced with equivalent (conforms to the designated requirement) nested code without disturbing the outer signature. This is the design mechanism for indirection in code bundles. It is acceptable to code-sign a bundle containing no main executable, and then treat it as nested code (typically in `Contents`).

Removing files from `.lproj` directories inside `Contents/Resources` will not invalidate the code signature, but adding or changing files will.

Removing the Mach-O slice for a particular architecture from a universal binary will also not invalidate the code signature.

Signing with Xcode

Xcode will run all build phases for a target before it signs that target. This is usually what you want since you shouldn't tamper with the bundle after it's been signed. If for some reason you must do something to the target after signing, create a separate aggregate target, make it depend on the original target, and put your custom build phases there.

Make sure your target dependencies are set correctly so the targets are built and signed in the correct order. Confirm that the Identity section of every target's General pane has the same value for Team so every target is signed with the same identity.

Using the codesign Tool's --deep Option 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.

Installer Packages

Bundle-style installer packages are a legacy transition aid that is no longer supported. PackageMaker is also no longer supported. It is now necessary to convert to flat-file installer packages using tools like productbuild.

If you must make short-term modifications to existing `.pkg` or `.mpkg` bundles to get them through Gatekeeper, then specify explicit resource rules that declare all files in the package to be resources with no exceptions.

The default `codesign` behavior for bundle-style installer packages creates a legacy signature format that does not pass Gatekeeper any more.

Use the `check-signature` tool to check your installer packages as described earlier.

> **Note:** Installer packages are checked by Gatekeeper, but package signing is different from code signing and is not affected by these code signing changes. You do not need to re-sign your flat installer packages for them to remain compatible with Gatekeeper.

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.

Shipping your Signed Code

The preferred way to ship a signed app is via the Mac App Store. The Mac App Store provides a secure channel for app delivery and installation that requires minimal action on the part of the user.

For distribution outside of the Mac App Store, the preferred options are to use a signed disk image (DMG) or signed installer package. Signing these allows validation of the contents and their source. ZIP archives may also be used, but this is discouraged.

If using a disk image to ship an app, users should drag the app from the image to its desired installation location (usually `/Applications`) before launching it. This also applies to apps installed via ZIP or other archive formats or apps downloaded to the Downloads directory: ask the user to drag the app to `/Applications` and launch it from there.

This practice avoids an attack where a validly signed app launched from a disk image, ZIP archive, or ISO (CD/DVD) image can load malicious code or content from untrusted locations on the same image or archive. Starting with macOS Sierra, running a newly-downloaded app from a disk image, archive, or the Downloads directory will cause Gatekeeper to isolate that app at a unspecified read-only location in the filesystem. This will prevent the app from accessing code or content using relative paths.

Do not ship apps using ISO images. There is no provision for signing these.

Important: Starting with macOS Sierra, only XIP archives signed by Apple will be expanded. Developers who have been using XIP archives will need to move to using signed installer packages or disk images.

Signing Disk Images

Disk images can be signed using the `codesign` tool on macOS 10.11.5 and later. This allows the entire disk image to be validated by Gatekeeper the first time it is mounted.

Gatekeeper will validate the contents of the disk image as well.

Disk images should only be signed with your Developer ID Application identity.

On macOS Sierra and later, `spctl` can be used to assess a disk image's signature, like this:

```
$ spctl -a -t open --context context:primary-signature -v MyImage.dmg
/Users/me/Downloads/MyImage.dmg: accepted
source=Developer ID
```

Note: A disk image signed on OS X 10.11.5 or 10.11.6 may not be able to be re-signed. In this situation, the operation will appear to succeed, but the signature will be invalid. If you encounter this condition, sign a new (unsigned) copy of the image on macOS Sierra or later.

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 expectation 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
```

Kernel Extensions

Kernel extensions (kexts) must be signed in order to load on macOS 10.11 and later. Signed kexts to be loaded by the system must be located in `/Library/Extensions`.

To verify that a kext is signed correctly, use this command:

```
$ kextutil -nt MyKext.kext
```

Note: The `spctl` command cannot be used on kexts.

The primary reason a signed kext cannot be loaded is if it was signed using a Developer ID Application signing identity that was not enabled for kext signing.

Kext signing requires a special Developer ID Application signing identity with the custom extension with OID 1.2.840.113635.100.6.1.18. You can see this in Keychain Access if it's present. Otherwise, the identity is valid for signing apps only.

You can request a kext signing identity by visiting this link.

The next step is to request a new Developer ID Application signing identity from your developer account. You need to do this because any existing Developer ID Application identities that were generated before your kext signing request was approved are not enabled for kext signing.

You then need to use Keychain Access to remove any Developer ID Application identities you already have on your development system and install the new Developer ID Application identity that has been enabled for kext signing.

The last step is to set the Code Signing Identity build setting in your Xcode project to your Developer ID Application identity and do a clean build of your kext.

Note: Starting with macOS Sierra, signed kext bundles must follow the same signing rules as other bundles as described earlier for macOS 10.9.5 and later, such as version 2 signatures being required, no symlinks referencing outside the bundle, and so forth.

Important: Kexts containing both 32-bit and 64-bit architectures have never been supported and are now expressly prevented from loading on macOS Sierra and later. If you must support the 32-bit kernel, it is required that you ship separate 32-bit and 64-bit kexts.

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 than 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 macOS. 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 macOS 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 macOS. 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 addrs : clarus@apple.com
Subject Name :
   Common Name : Testing Code Signing
   Org : Apple Inc.
   OrgUnit : DTS
   State : CA
```

```
   Country : US
   Locality : Cupertino
   Email addrs : 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 >
```

Back to Top

Version 2 Code Signature FAQ

codesign says my signature is weak.

- Your bundle seal is incomplete: it excludes some files. All version 1 signatures are weak by definition. But if you re-sign as directed, the result should not be weak.
- You used resource rules to make your signature weak. This is no longer allowed.

I exclude files so I can fix my bundle after I build it.

- This is no longer allowed. If you must modify your bundle, do it before signing. If you modify a signed bundle, you must re-sign it afterwards.

I write a receipt/license file/magic marker into my bundle after installation.

- This is no longer allowed.

- Write data into files **outside** the bundle as described earlier.
- Use the extended attribute mechanism described earlier to point your bundle at this data if necessary.
- Use an online service to store it and use the extended attribute mechanism to point at data there.

I store data in or otherwise modify my bundle after I sign it.

- This is no longer allowed. You will need to locate this data elsewhere.
- It won't work to write the data before your app runs for the first time. Gatekeeper will reject your app because the signature will be invalid.
- It also won't work to write that data after your app first runs. That still breaks the signature. Gatekeeper will check your app again if your app is copied to another computer such that the new copy is quarantined. And on macOS Sierra and later, Gatekeeper will check your app every time it's launched if it is run from the location where it was downloaded.
- macOS APIs that rely on a valid identity will fail.
- In general, you should plan for future stricter runtime checks of code validity.

codesign says my code is unsigned when I try to sign it.

- Make sure all nested code is already signed and its signature is valid. Xcode will take care of this for you if you let it handle your code signing tasks.
- Make sure your target dependencies are set correctly so the targets are built and signed in the correct order. Confirm that the Identity section of every target's General pane has the same value for Team so every target is signed with the same identity.

The 'code' codesign complains about isn't code. It’s just some data files.

- If a file is in a code location, it must be code, and it must be signed.
- Do not put data files into code locations. Move these elsewhere, such as `Contents/Resources`.

I can't change where my data files are stored without significantly changing my code.

- You can work around this by moving the files to the correct locations and leave behind symlinks so your code can still find the files.
- Then fix your code, and remove the symlinks as soon as possible.

codesign says my code is broken when I try to sign it.

- The signature of your nested code is broken and should be fixed as described earlier.

codesign says my main executable failed strict validation.

- Your Mach-O executable does not conform to modern Mach-O layout rules.
- You may be using a third party development product that hasn't been brought up to date, or post-processed your file in unsupported ways.

I wrote some data to the Mach-O file before signing. Is that allowed?

- No. Do not tamper with Mach-O files, outside of using macOS build tools and Xcode workflows.

I embed my data in the Mach-O file.

- Use the `-sectcreate` linker option for this.

codesign says I have 'unsealed content' somewhere.

- Do not put files or directories into the top directory of an app or framework bundle
- All content must be inside `Contents` or `Versions` respectively as described earlier.

codesign says my bundle format is ambiguous.

- `codesign` thinks your bundle looks a bit like an app and a bit like a framework.
- Perhaps you have both `Contents` and `Versions` directories, or files in the top directory that match reserved names for directories in a bundle.
- Perhaps there's an `Info.plist` in an odd place.
- Perhaps a framework was copied incorrectly so the symlinks it contained were converted to normal files.
- In short, put everything in the correct places.

codesign says an embedded framework is wrong.

- Check the framework for top-level contents, and for proper placement and contents of its `Info.plist`.
- If your framework contains multiple versions, make sure they are **all** properly signed by the same signing authority.
- Make sure all symlinks in the top `.framework` directory are in fact symlinks and point to the matching directory inside `Versions/Current`.

codesign says there's an issue with a subcomponent of my app.

- 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
2016-08-09	Additional guidance for macOS Sierra.
2016-06-13	Discussed new Gatekeeper checks added since OS X v10.11. Added information about signed kexts. Made other editorial improvements.
2015-07-28	Cover changes to linking to dynamic libraries outside of an app bundle.
2014-11-13	Add reference to the new check-signature tool.
2014-10-23	Added v2 signature FAQ and additional guidance for developers transitioning to v2 signatures.
2014-08-26	Added clarifications of Mavericks and Yosemite code signing changes.
2014-07-31	Updated discussion of significant code signing changes in OS X Mavericks. Other editorial changes.
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 macOS code signing that details specific options and gotchas