Code Signing Guide

Developer

Contents

About Code Signing 4
At a Glance 4
Prerequisites 5
See Also 5

Code Signing Overview 6
The Benefits Of Signing Code 6
Digital Signatures and Signed Code 8
Code Requirements 8
The Role of Trust in Code Signing 9

Code Signing Tasks 11
Obtaining a Signing Identity 11
Adding an Info.plist to Single-File Tools 15
Signing Your Code 17
    What to Sign 17
    When to Sign 17
    Using the codesign Command 18
Shipping and Updating Your Product 20

Code Signing Requirement Language 22
Language Syntax 22
Evaluation of Requirements 23
Constants 23
    String Constants 23
    Integer Constants 24
    Hash Constants 24
Variables 24
Logical Operators 24
Comparison Operations 25
    Equality 25
    Inequality 26
    Existence 26
Constraints 26

Identifier 26
Info 27
Certificate 27
Trusted 29
Entitlement 30
Code Directory Hash 30
Requirement Sets 31

Document Revision History 33

About Code Signing

Code signing is a Mac OS X security technology that allows you to certify that an app was created by you. Once an app is signed, the system can detect any change to the app—whether the change is introduced accidentally or by malicious code.

Users appreciate code signing. After installing a new version of a code-signed app, a user is not bothered with alerts asking for permission to access the keychain or similar resources. As long as the new version uses the same signature, Mac OS X can treat it exactly as it treated the previous version.

Other Mac OS X security features, such as App Sandbox and parental controls, also depend on code signing.

In most cases, you can rely on Xcode's automatic code signing, which requires only that you specify a code signing identity in the build settings for your project. This document is for readers who must go beyond automatic code signing—perhaps to troubleshoot an unusual problem, or to incorporate the `codesign` tool into a build system.

At a Glance

The elements of code signing include code signatures, code signing identities, code signing certificates, and security trust policies. Be sure to understand these concepts if you need to perform code signing outside of Xcode.

Relevant chapter "Code Signing Overview" (page 6)

Before you can sign code, you must obtain or create a code signing identity. You then sign your code and prepare it for distribution.

Relevant chapter "Code Signing Tasks" (page 11)

To specify recommended criteria for verifiers to use when evaluating your app's code signature, you use a requirements language specific to the `codesign` and `csreq` commands. You then save your criteria to a binary file as part of your Xcode project.

Relevant chapter "Code Signing Requirement Language" (page 22)

Prerequisites

Read *Security Overview* to understand the place of code signing in the Mac OS X security picture.

See Also

For descriptions of the command-line tools for performing code signing, see the `codesign` and `csreq` man pages.

Code Signing Overview

Code signing is a technique that can be used to ensure the integrity of code, allow the system to unambiguously determine the source (developer) of the code, and allow any application to determine the purposes for which the developer intended the code to be used. The code signing solution in Mac OS X is intended to be completely managed by the developer. This means that it is up to you to create or purchase and maintain signing certificates, sign your code, specify the meaning of the signature, and distribute the signed code in a way that that is convenient for users. Although the code signing system will carry out policy checks based on a code signature, it is up to the caller to make policy decisions based on the results of those checks. When it is the operating system that makes the policy checks, whether your code will be allowed to run in a given situation depends on whether you signed the code and on the requirements you included in the signature.

This chapter describes the benefits of signing code and introduces some of the basic concepts you need to understand in order to carry out the code signing process.

The Benefits Of Signing Code

When code is signed, it is possible to determine reliably whether that code has been modified by someone other than the signer, no matter whether the modification was intentional (by a hacker, for example) or accidental (as when a file gets corrupted). In addition, by adding a code signature, a developer can ensure that updates to a program are valid and can be treated by the system as the same program as the previous version.

For example, suppose the user gives the SurfWriter application permission to access a keychain item. Each time SurfWriter attempts to access the keychain, the keychain must determine whether this is the same application as the one to which the user gave permission. If the application is signed, the keychain (in Mac OS X v10.5 and later) can determine this with certainty. If the developer of SurfWriter updates the program and signs the new version of SurfWriter with the same unique identifier as the old version, keychain will recognize the update as the same application and will give it access without requesting verification from the user. On the other hand, if SurfWriter is corrupted or hacked, keychain will detect the change and will refuse access.

Similarly, if you use Parental Controls (in Mac OS X v10.5 or later) to prevent your child from running a specific game, and that game has been signed by its manufacturer, your child cannot circumvent the control by renaming or moving files on the disk. Parental Controls can use the signature to unambiguously identify the game regardless of its name, location, or version number.

All sorts of code can be signed, including tools, applications, scripts, libraries, plug-ins, and other "code-like" data.

Code signing can be seen as having three distinct purposes. It can be used to:

- ensure the integrity of the code; that is, that it has not been altered
- identify the code as coming from a specific source (the developer or signer)
- determine whether the code is trustworthy for a specific purpose (for example, to access a keychain item).

To enable signed code to fulfill all of these purposes, a code signature consists of three parts:

- A unique identifier, which can be used to identify the code or to determine to which groups or categories the code belongs. This identifier can be derived from the contents of the `Info.plist` for the program, or can be provided explicitly by the signer.
- A seal, which is a collection of checksums or hashes of the various parts of the program, such as the identifier, the `Info.plist`, the main executable, the resource files, and so on. The seal can be used to detect alterations to the code and to the program identifier.
- A digital signature, which signs the seal to guarantee its integrity. The signature includes information that can be used to determine who signed the code and whether the signature is valid.

For more discussion of digital signatures, see the following section, "Digital Signatures and Signed Code."

To learn more about how the code signature is used to determine the code's trustworthiness for a specific purpose, see "Code Requirements" (page 8).

Note that code signing deals primarily with running code. Although it can be used to ensure the integrity of stored code (on disk, for example), that's a secondary use.

In order to fully appreciate the uses of code signing, it is important to be aware of some things that signing code *cannot* do:

- It can't guarantee that the code is free of security vulnerabilities.
- It can't guarantee that a program will not load unsafe or altered code—such as untrusted plug-ins—during execution.
- It is not a digital rights management (DRM) or copy protection technology. Although the system could determine that a copy of your program had not been properly signed by you, or that its copy protection had been hacked, thus making the signature invalid, there is nothing to prevent the user from running the program anyway.

Digital Signatures and Signed Code

A digital signature uses public key cryptography to ensure the integrity of data.Like traditional signatures written with ink on paper, they can be used to identify and authenticate the signer of the data. However, digital signatures go beyond traditional signatures in that they can also ensure that the data itself has not been altered. This is like designing a check in such a way that if someone alters the amount of the sum written on the check, an “Invalid” watermark becomes visible on the face of the check.

To create a digital signature, the signer generates a **message digest** of the data and then uses a private key to sign the digest. The signer must have a valid digital certificate containing the public key that corresponds to the private key. The combination of a certificate and related private key is called an **identity**. The signature includes the signed digest and information about the signer’s digital certificate. The certificate includes the public key and the algorithm needed to verify the signature.

To verify that the signed document has not been altered, the recipient uses the algorithm to create their own message digest and applies the public key to the signed digest. If the two digests prove identical, then the message cannot have been altered and must have been sent by the owner of the public key.

To ensure that the person who provided the signature is not only the same person who provided the data but is also who they say they are, the certificate is also signed—in this case by the certification authority who issued the certificate. Digital certificates are described in “Security Concepts”.

Signed code uses several digital signatures:

- If the code is universal, the object code for each architecture is signed separately
- Various components of the application bundle (such as the `Info.plist` file, if there is one) are also signed

Code Requirements

It is up to the system or program that is launching or loading signed code to decide whether to verify the signature and, if it does, to determine how to evaluate the results of that verification. The criteria used to evaluate a code signature are called **code requirements**. The signer can specify requirements when signing the code; such requirements are referred to as **internal requirements**. A verifier can read any internal requirements before deciding how to treat signed code. However, it is up to the verifier to decide what requirements to use. For example, Safari could require a plug-in to be signed by Apple in order to be loaded, regardless of whether that plug-in’s signature included internal requirements.

One major purpose of code signatures is to allow the verifier to identify the code (such as a program, plug-in, or script) to determine whether it is the same code the verifier has seen before. The criteria used to make this determination are referred to as the code’s **designated requirement**. For example, the designated requirement for Apple Mail might be "was signed by Apple and the identifier is `com.apple.Mail`".

To see how this works in practice, assume the user has granted permission to the Apple Mail application to access a keychain item. The keychain uses Mail's designated requirement to identify it: the keychain records the identifier (`com.apple.Mail`) and the signer of the application (Apple) to identify the program allowed to access the keychain item. Whenever Mail attempts to access this keychain item, the keychain looks at Mail's signature to make sure that the program has not been corrupted, that the identifier is `com.apple.Mail`, and that the program was signed by Apple. If everything checks out, the keychain gives Mail access to the keychain item. When Apple issues a new version of Mail, the new version includes a signature, signed by Apple, that identifies the application as `com.apple.Mail`. Therefore, when the user installs the new version of Mail and it attempts to access the keychain item, the keychain recognizes the updated version as the same program and does not prompt the user for verification.

The program identifier or the entire designated requirement can be specified by the signer, or can be inferred by the `codesign` utility at the time of signing. In the absence of an explicitly specified designated requirement, the `codesign` utility typically builds a designated requirement from the name of the program found in its `Info.plist` file and the chain of signatures securing the code signature.

Architecturally, a code requirement is a script, written in a dedicated language, that describes conditions (restrictions) the code must satisfy to be acceptable for some purpose. It is up to you whether to specify internal requirements when you sign code.

Note that validation of signed code against a set of requirements is performed only when the system or some other program needs to determine whether it's all right to trust that code. The Finder, for example, might run a program that has an invalid code identifier as long as there is no reason to check the identifier. Even if that code requests access to a keychain item and the keychain checks the identifier, the only consequence of the identifier being invalid is that the keychain will refuse access to the keychain item; the process will be permitted to continue running.

The Role of Trust in Code Signing

Trust is determined by policy. A security trust policy determines whether a particular code identity (assuming it is valid) should be accepted for allowing something, such as access to a resource or service. Each Mac OS X component has its own policy, and makes this determination separately. Thus it makes no sense to ask whether the code signing system trusts a particular signature. Instead, it is more meaningful to ask whether a specific subsystem of Mac OS X trusts the signature. Therefore, in general, in order for an application that is signed to be trusted for a particular purpose it must have been signed either by an identity whose root certificate is already trusted by default on Mac OS X or by one that has previously been designated by the caller as being trusted.

Note that many parts of Mac OS X do not care about the identity of the signer—they care only whether the signer has changed since the last time the signature was checked. They use the code signature's designated requirement for this purpose. The keychain system and parental controls are examples of this use of signatures. Self-signed identities and home-made certificate authorities (CAs) work for this purpose as well as commercial signing certificates.

Other parts of Mac OS X constrain acceptable signatures to only those drawn from certificate authorities (root certificates) that are trusted anchors on the system performing the validation. For those checks, the nature of the identity used matters. The Application Firewall is one example of this type of policy. Self-signed identities and self-created CAs do not work for this unless the validating system has been told to trust them for code signing purposes.

Code Signing Tasks

This chapter gives procedures and examples for the code signing process. It covers what you need to do before you begin to sign code, how to sign code, and how to ship the code you signed.

Obtaining a Signing Identity

To sign code, you need a code signing digital identity, which is a private cryptographic key plus a digital certificate. The digital certificate must have a usage extension that enables it to be used for signing and it must contain the public key that corresponds to the private key. You can use more than one signing identity, each for its own purpose, such as one to be used for beta seeds and one for final, released products. However, most organizations use only one identity.

You can obtain a signing identity from a certificate authority such as VeriSign, RSA, or Thawte. If your company already has a signing identity that you use to sign code on other systems, you can use it with the Mac OS X `codesign` command as well. Apple uses the industry-standard form and format of code signing digital certificates. Some companies are certificate issuing authorities; in this case, you need to contact your IT department to find out how to get a signing certificate issued by your company.

To import a signing identity with Keychain Access . . .

1. In Keychain Access (available in /Applications/Utilities), choose File > Import Items.
2. Choose a destination keychain for the identity.
3. Choose the certificate file.
4. Click Open.

However, if the only reason you need a certificate is for a signing identity to use with Mac OS X, you can create your own identity by using the Certificate Assistant, which is provided as part of the Keychain Access application.

Before you obtain a code signing identity and sign your code, consider the following points:

- A self-signed certificate created with the Certificate Assistant is not recognized by users' operating systems as a valid certificate for any purpose other than validating the designated requirement of your signed code. Because a self-signed certificate has not been signed by a recognized root certificate authority, the

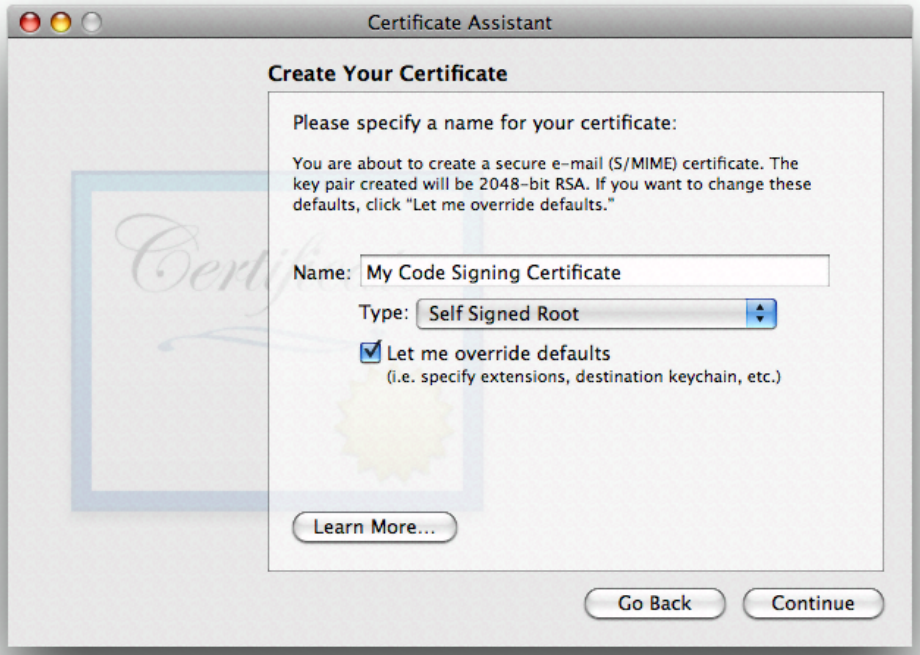
user can only verify that two versions of your application came from the same source; they cannot verify that your company is the true source of the code. For more information about root authorities, see "Security Concepts".

- Depending on your company's internal policies, you might have to involve your company's Build and Integration, Legal, and Marketing departments in decisions about what sort of signing identity to use and how to obtain it. You should start this process well in advance of the time you need to actually sign the code for distribution to customers.
- Any signed version of your code that gets into the hands of users will appear to have been endorsed by your company for use. Therefore, you might not want to use your "final" signing identity to sign code that is still in development.
- A signing identity, no matter how obtained, is completely compromised if it is ever out of the physical control of whoever is authorized to sign the code. That means that the signing identity's private key must never, under any circumstances, be given to end users, and should be restricted to one or a small number of trusted persons within your company. Before obtaining a signing identity and proceeding to sign code, you must determine who within your company will possess the identity, who can use it, and how it will be kept safe. For example, if the identity must be used by more than one person, you can keep it in the keychain of a secure computer and give the password of the keychain only to authorized users, or you can put the identity on a smart card to which only authorized users have the PIN.
- A self-signed certificate created by the Certificate Assistant is adequate for internal testing and development, regardless of what procedures you put in place to sign released products.

To use the Certificate Assistant to create a signing identity . . .

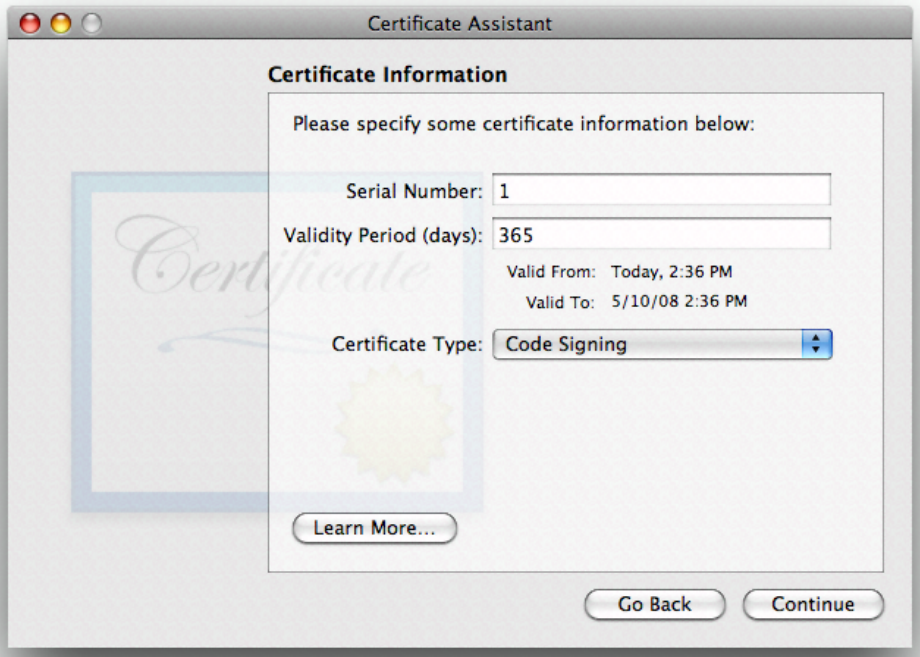
1. Open Applications > Utilities > Keychain Access.

2. From the Keychain Access menu, choose Certificate Assistant > Create a Certificate.



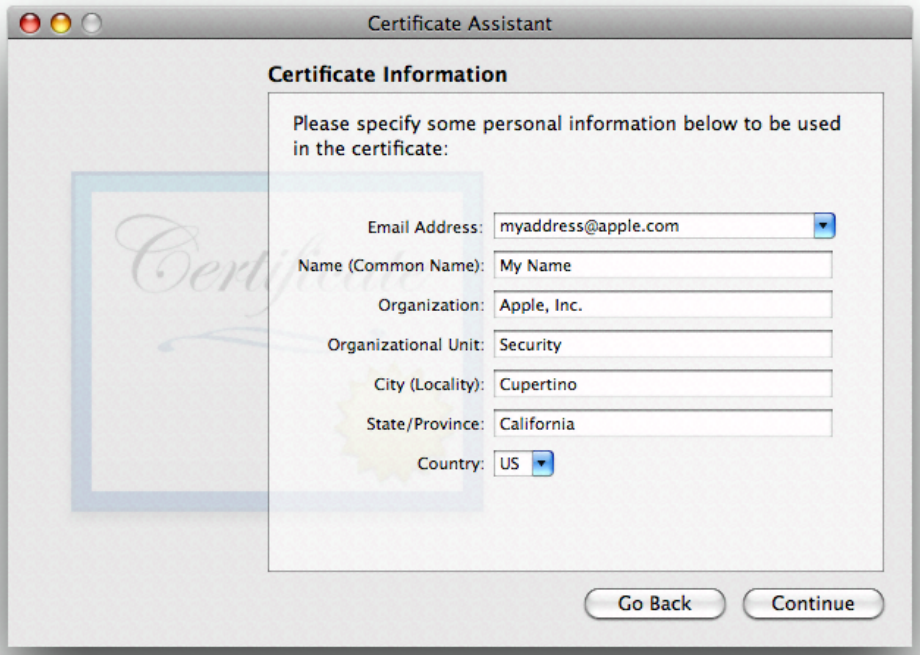
3. Fill in a name for the certificate. This name appears in the Keychain Access utility as the name of the certificate.
4. Choose `Self Signed Root` from the `Type` popup menu.

5. Check the `Let me override defaults` checkbox. Click `Continue`.



6. Specify a serial number for the certificate. Any number will do as long as you have no other certificate with the same name and serial number.

7. Choose `Code Signing` from the `Certificate Type` popup menu. Click `Continue`.



8. Fill in the information for the certificate. Click `Continue`.
9. Accept the defaults for the rest of the dialogs.

Adding an Info.plist to Single-File Tools

As discussed in "Code Requirements" (page 8), the system often uses the `Info.plist` file of an application bundle to determine the code's designated requirement. Although single-file tools don't normally have an `Info.plist`, you can add one. To do so, use the following procedure:

1. Add an `Info.plist` file to your project (including adding it to your source control).
2. Make sure the `Info.plist` file has the following keys:
   - `CFBundleIdentifier`
   - `CFBundleName`

3. The value for `CFBundleIdentifier` is used as the default unique name of your program for Code Signing purposes. Because the `CFBundleIdentifier` value is also used when your application accesses resources in the application bundle, it may sometimes be necessary to use a non-unique `CFBundleIdentifier` value for a helper. If you do this, you must provide a different, unique identifier for code signing purposes by passing the `-i` or `--identifier` flag to the `codesign` command.

   The identifier used for signing must be globally unique. To ensure uniqueness, you should include your company's name in the value. The usual form for this identifier is a hierarchical name in reverse DNS notation, starting with the top level domain, followed by the company name, followed by the organization within the company, and ending with the product name. For example, the `CFBundleIdentifier` value for the `codesign` command is `com.apple.security.codesign`.

4. The value for `CFBundleName` shows up in system dialogs as the name of your program, so it should match your marketing name for the product.

5. Add the following arguments to your linker flags:

   `-sectcreate __TEXT __info_plist` *Info.plist_path*

   where *Info.plist_path* is the complete path of the `Info.plist` file in your project.

   In Xcode, for example, you would add these linker flags to the `OTHER_LDFLAGS` build variable (Other Linker Flags in the target's build rules).

For example, here are the contents of the `Info.plist` file for the `codesign` command:

```
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.security.codesign</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>codesign</string>
    <key>CFBundleVersion</key>
    <string>0.3</string>
</dict>
</plist>
```

Signing Your Code

You use the `codesign` command to sign your code. This section discusses what to sign and gives some examples of the use of `codesign`. See the `codesign(1)` manual page for a complete description of its use.

What to Sign

You should sign every executable in your product, including applications, tools, hidden helper tools, utilities and so forth. Signing an application bundle covers its resources, but not its subcomponents such as tools and sub-bundles. Each of these must be signed independently.

If your application consists of a big UI part with one or more little helper tools that try to present a single face to the user, you can make them indistinguishable to code signing by giving them all the exact same code signing identifier. (You can do that by making sure that they all have the same `CFBundleIdentifier` value in their `Info.plist`, or by using the `-i` option in the `codesign` command, to assign the same identifier.) In that case, all your program components have access to the same keychain items and validate as the same program. Do this only if the programs involved are truly meant to form a single entity, with no distinctions made.

A universal binary (bundle or tool) automatically has individual signatures applied to each architecture component. These are independent, and usually only the native architecture on the end user's system is verified.

In the case of installer packages (.pkg and .mpkg bundles), everything is implicitly signed: The CPIO archive containing the payload, the CPIO archive containing install scripts, and the bill of materials (BOM) each have a hash recorded in the XAR header, and that header in turn is signed. Therefore, if you modify an install script (for example) after the package has been signed, the signature will be invalid.

You may also want to sign your plug-ins and libraries. Although this is not currently required, it will be in the future, and there is no disadvantage to having signatures on these components.

Depending on the situation, `codesign` may add to your Mach-O executable file, add extended attributes to it, or create new files in your bundle's Contents directory. None of your other files is modified.

When to Sign

You can run `codesign` at any time on any system running Mac OS X v10.5 or later, provided you have access to the signing identity. You can run it from a shell script phase in Xcode if you like, or as a step in your Makefile scripts, or anywhere else you find suitable. Signing is typically done as part of the product mastering process, after quality assurance work has been done. Avoid signing pre-final copies of your product so that no one can mistake a leaked or accidentally released incomplete version of your product for the real thing.

Your final signing must be done after you are done building your product, including any post-processing and assembly of bundle resources. Code signing detects any change to your program after signing, so if you make any changes at all after signing, your code will be rejected when an attempt is made to verify it. Sign your code before you package the product for delivery.

Because each architecture component is signed independently, it is all right to perform universal-binary operations (such as running the `lipo` command) on signed programs. The result will still be validly signed as long as you make no other changes.

Using the codesign Command

The `codesign` command is fully described in the `codesign(1)` manual page. This section provides some examples of common uses of the command. Note that your signing identity must be in a keychain for these commands to work.

Signing Code

To sign the code located at `<code-path>`, using the signing identity `<identity>`, use the following command:

```
codesign -s <identity> <code-path> …
```

The `<code-path>` value may be a bundle folder or a specific code binary. See "What to Sign" (page 17) for more details.

The identity can be named with any (case sensitive) substring of the certificate's common name attribute, as long as the substring is unique throughout your keychains. (Signing identities are discussed in "Obtaining a Signing Identity" (page 11).)

As is typical of Unix-style commands, this command gives no confirmation of success. To get some feedback, include the `-v` option:

```
codesign -s <identity> -v <code-path> …
```

Use the `-r` option to specify an internal requirement. With this option you can specify a text file containing the requirements, a precompiled requirements binary, or the actual requirement text prefixed with an equal sign (=). For example, to add an internal requirement that all libraries be signed by Apple, you could use the following option:

```
-r="library => anchor apple"
```

The code requirement language is described in "Code Signing Requirement Language" (page 22).

If you have built your own certificate hierarchy (perhaps using Certificate Assistant—see "Obtaining a Signing Identity" (page 11)), and want to use your certificate's anchor to form a designated requirement for your program, you could use the following command:

```
codesign -s signing-identity -r="designated => anchor /my/anchor/cert and identifier
 com.mycorp.myprog"
```

Note that the requirement source language accepts either an SHA1 hash of a certificate (for example `H"abcd...."`) or a path to the DER encoded certificate in a file. It does not currently accept a reference to the certificate in a keychain, so you have to export the certificate before executing this command.

You can also use the `csreq` command to write the requirements out to a file, and then use the path to that file as the input value for the `-r` option in the `codesign` command. See the manual page for `csreq(1)` for more information on that command.

Here are some other samples of requirements:

- `anchor apple` –the code is signed by Apple
- `anchor trusted` –the anchor is trusted (for code signing) by the system
- `certificate leaf = /path/to/certificate` –the leaf (signing) certificate is the one specified
- `certificate leaf = /path/to/certificate and identifier "com.mycorp.myprog"` –the leaf certificate and program identifier are as specified
- `info[mykey] = myvalue` – the `Info.plist` key `mykey` exists and has the value `myvalue`

Except for the explicit `anchor trusted` requirement, the system does not consult its trust settings database when verifying a code requirement. Therefore, as long as you don't add this designated requirement to your code signature, the anchor certificate you use for signing your code does not have to be introduced to the user's system for validation to succeed.

Verifying Code

To verify the signature on a signed binary, use the `-v` option with no other options:

```
codesign -v <code-path> …
```

This checks that the code binaries at <code–path> are actually signed, that the signature is valid, that all the sealed components are unaltered, and that the whole thing passes some basic consistency checks. It does not by default check that the code satisfies any requirements except its own designated requirement. To check a particular requirement, use the –R option. For example, to check that the Apple Mail application is identified as Mail, signed by Apple, and secured with Apple's root signing certificate, you could use the following command:

```
codesign –v –R="identifier com.apple.mail and anchor apple" /Applications/Mail.app
```

Note that, unlike the –r option, the –R option takes only a single requirement rather than a requirements collection (no => tags). Add one or more additional –v options to get details on the validation process.

If you pass a number rather than a path to the verify option, codesign takes the number to be the process ID (pid) of a running process, and performs dynamic validation instead.

Getting Information About Code Signatures

To get information about a code signature, use the –d option. For example, to output the code signature's internal requirements to standard out, use the following command:

```
codesign –d –r code–path
```

Note that this option does not verify the signature.

Shipping and Updating Your Product

The only thing that matters to the code signing system is that the signed code installed on the user's system identical to the code that you signed. It does not matter how you package, deliver, or install your product as long as you don't introduce any changes into the product. Compression, encoding, encrypting, and even binary patching the code are all right as long as you end up with exactly what you started with. You can use any installer you like, as long as it doesn't write anything into the product as it installs it. Drag-installs are fine as well.

When you have qualified a new version of your product, sign it just as you signed the previous version, with the same identifier and the same designated requirement. The user's system will consider the new version of your product to be the same program as the previous version. In particular, the keychain will not distinguish older and newer versions of your program as long as both were signed and the unique Identifier hasn't changed.

You can take a partial-update approach to revising your code on the user's system. To do so, sign the new version as usual, then calculate the differences between the new and the old signed versions, and transmit the differences. Because the differences include the new signature data, the result of installing the changes on the end-user's system will be the newly signed version. You cannot patch a signed application in the field. If you do so, the system will notice that the application has changed and will invalidate the signature, and there is no way to re-validate or resign the application in the field.

Code Signing Requirement Language

When you use the `codesign` command to sign a block of code, you can specify internal requirements; that is, the criteria that you recommend should be used to evaluate the code signature. It is up to the verifier to decide whether to apply the internal requirements or some other set of requirements when deciding how to treat the signed code. You use the code requirement language described in this chapter when specifying requirements to the `codesign` or `csreq` command (see the manual pages for `codesign(1)` and `csreq(1)`).

This chapter describes the requirement language source code. You can compile a set of requirements and save them in binary form using the `csreq` command. You can provide requirements to the `codesign` command either as source code or as a binary file. Both the `codesign` and `csreq` commands can convert a binary requirement set to text. Although there is some flexibility in the source code syntax (for example, quotes can always be used around string constants but are not always required), conversion from binary to text always uses the same form:

- Parentheses are placed (usually only) where required to clarify operator precedence.
- String constants are quoted (usually only) where needed.
- Whether originally specified as constants or through file paths, certificate hashes are always returned as hash constants.
- Comments in the original source are not preserved in the reconstructed text.

Language Syntax

Some basic features of the language syntax are:

- Expressions use conventional infix notation (that is, the operator is placed between the two entities being acted on; for example *quantity* < *constant*).
- Keywords are reserved, but can be quoted to be included as part of ordinary strings.
- Comments are allowed in C, Objective C, and C++.
- Unquoted whitespace is allowed between tokens, but strings containing whitespace must be quoted.
- Line endings have no special meaning and are treated as whitespace.

Evaluation of Requirements

A requirement constitutes an expression without side effects. Each requirement can have any number of subexpressions, each of which is evaluated with a Boolean (succeed-fail) result. There is no defined order of evaluation. The subexpressions are combined using logical operators in the expression to yield an overall Boolean result for the expression. Depending on the operators used, an expression can succeed even if some subexpressions fail. For example, the expression

```
anchor apple or anchor = "/var/db/yourcorporateanchor.cert"
```

succeeds if either subexpression succeeds—that is, if the code was signed either by Apple or by your company—even though one of the subexpressions is sure to fail.

If an error occurs during evaluation, on the other hand, evaluation stops immediately and the `codesign` or `csreq` command returns with a result code indicating the reason for failure.

Constants

This section describes the use of string, integer, hash-value, and binary constants in the code signing requirement language.

String Constants

String constants must be enclosed by double quotes (`" "`) unless the string contains only letters, digits, and periods (.), in which case the quotes are optional. Absolute file paths, which start with a slash, do not require quotes unless they contain spaces. For example:

```
com.apple.mail                      //no quotes are required
"com.apple.mail"                    //quotes are optional
"My Company's signing identity"     //requires quotes for spaces and apostrophe
/Volumes/myCA/root.crt              //no quotes are required
"/Volumes/my CA/root.crt"           //space requires quotes
"/Volumes/my_CA/root.crt"           //underscore requires quotes
```

It's never incorrect to enclose the string in quotes—if in doubt, use quotes.

Use a backslash to "escape" any character. For example:

```
"one \" embedded quote"             //one " embedded quote
"one \\ embedded backslash"         //one \ embedded backslash
```

There is nothing special about the single quote character (').

Integer Constants

Integer constants are written as decimal constants are in C. The language does not allow radix prefixes (such as `0x`) or leading plus or minus (`+` or –) signs.

Hash Constants

Hash values are written either as a hexadecimal number in quotes preceded by an `H`, or as a path to a file containing a binary certificate. If you use the first form, the number must include the exact number of digits in the hash value. A SHA-1 hash (the only kind currently supported) requires exactly 40 digits; for example:

```
H"0123456789ABCDEFFEDCBA98765432100A2BC5DA"
```

You can use either uppercase or lowercase letters (`A..F` or `a..f`) in the hexadecimal numbers.

If you specify a file path, the compiler reads the binary certificate and calculates the hash for you. The compiled version of the requirement code includes only the hash; the certificate file and the path are not retained. If you convert the requirement back to text, you get the hexadecimal hash constant. The file path must point to a file containing an X.509 DER encoded certificate. No container forms (PKCS7, PKCS12) are allowed, nor is the OpenSSL "PEM" form supported.

Variables

There are currently no variables in the requirement language.

Logical Operators

The requirement language includes the following logical operators, in order of decreasing precedence:

- `!` (negation)
- `and` (logical AND)

- `or` (logical OR)

These operators can be used to combine subexpressions into more complex expressions. The negation operator (`!`) is a unary prefix operator. The others are infix operators. Parentheses can be used to override the precedence of the operators.

Because the language is free of side effects, evaluation order of subexpressions is unspecified.

Comparison Operations

The requirement language includes the following comparison operators:

- = (equals)
- < (less than)
- > (greater than)
- <= (less than or equal to)
- >= (greater than or equal to)
- `exists` (value is present)

The value-present (`exists`) operator is a unary suffix operator. The others are infix operators.

There are no operators for non-matches (not equal to, not greater than, and so on). Use the negation operator (`!`) together with the comparison operators to make non-match comparisons.

Equality

All equality operations compare some value to a constant. The value and constant must be of the same type: a string matches a string constant, a data value matches a hexadecimal constant. The equality operation evaluates to `true` if the value exists and is equal to the constant. String matching uses the same matching rules as `CFString` (see *CFString Reference*).

In match expressions (see "Info" (page 27), "Part of a Certificate" (page 29), and "Entitlement" (page 30)), substrings of string constants can be matched by using the `*` wildcard character:

- `value = *constant*` is `true` if the value exists and any substring of the value matches the constant; for example:
  - `thunderbolt = *under*`
  - `thunderbolt = *thunder*`
  - `thunderbolt = *bolt*`

- `value = constant*` is `true` if the value exists and begins with the constant; for example:
  - `thunderbolt = thunder*`
  - `thunderbolt = thun*`
- `value = *constant` is `true` if the value exists and ends with the constant; for example:
  - `thunderbolt = *bolt`
  - `thunderbolt = *underbolt`

If the constant is written with quotation marks, the asterisks must be outside the quotes. An asterisk inside the quotation marks is taken literally. For example:

- `"ten thunderbolts" = "ten thunder"*` is `true`
- `"ten thunder*bolts" = "ten thunder*"*` is `true`
- `"ten thunderbolts" = "ten thunder*"` is `false`

Inequality

Inequality operations compare some value to a constant. The value and constant must be of the same type: a string matches a string constant, a data value matches a hexadecimal constant. String comparisons use the same matching rules as `CFString` with the `kCFCompareNumerically` option flag; for example, `"17.4"` is greater than `"7.4"`.

Existence

The existence operator tests whether the value exists. It evaluates to `false` only if the value does not exist at all or is exactly the Boolean value `false`. An empty string and the number `0` are considered to exist.

Constraints

Several keywords in the requirement language are used to require that specific certificates be present or other conditions be met.

Identifier

The expression

`identifier =` *constant*

succeeds if the unique identifier string embedded in the code signature is exactly equal to *constant*. The equal sign is optional in identifier expressions. Signing identifiers can be tested only for exact equality; the wildcard character (`*`) can not be used with the identifier constraint, nor can identifiers be tested for inequality.

Info

The expression

`info [`*key*`]` *match expression*

succeeds if the value associated with the top-level key in the code's `info.plist` file matches *match expression*, where *match expression* can include any of the operators listed in "Logical Operators" (page 24) and "Comparison Operations" (page 25). For example:

```
info [CFBundleShortVersionString] < "17.4"
```

or

```
info [MySpecialMarker] exists
```

You must specify *key* as a string constant.

If the value of the specified key is a string, the match is applied to it directly. If the value is an array, it must be an array of strings and the match is made to each in turn, succeeding if any of them matches. Substrings of string constants can be matched by using any match expression (see "Comparison Operations" (page 25)).

If the code has no `info.plist` file, or the `info.plist` does not contain the specified key, this expression evaluates to `false` without returning an error.

Certificate

Certificate constraints refer to certificates in the certificate chain used to validate the signature. Most uses of the `certificate` keyword accept an integer that indicates the position of the certificate in the chain: positive integers count from the anchor (0) toward the leaf. Negative integers count backward from the signing certificate (-1). For example, `certificate 1` indicates the certificate that was directly signed by the anchor, and `certificate -2` is the intermediate certificate that was used to sign the leaf (that is, the signing certificate). Note that this convention is the same as that used for array indexing in the Perl and Ruby programming languages:

Anchor	First intermediate	Second intermediate	Leaf
`certificate 0`	`certificate 1`	`certificate 2`	`certificate 3`
`certificate -4`	`certificate -3`	`certificate -2`	`certificate -1`

Other keywords include:

- `certificate root`—the anchor certificate; same as certificate 0
- `anchor`—same as `certificate root`
- `certificate leaf`—the signing certificate; same as `certificate -1`

> **Note** The short form `cert` is allowed for the keyword `certificate`.

If there is no certificate at the specified position, the constraint evaluates to `false` without returning an error.

If the code was signed using an ad-hoc signature, there are no certificates at all and all certificate constraints evaluate to `false`. (An ad-hoc signature is created by signing with the pseudo-identity – (a dash). An ad-hoc signature does not use or record a cryptographic identity, and thus identifies exactly and only the one program being signed.)

If the code was signed by a self-signed certificate, then the leaf and root refer to the same single certificate.

Whole Certificate

To require a particular certificate to be present in the certificate chain, use the form

`certificate` *position* `=` *hash*

or one of the equivalent forms discussed above, such as `anchor =` *hash*. Hash constants are described in "Hash Constants" (page 24).

For Apple's own code, signed by Apple, you can use the short form

```
anchor apple
```

For code signed by Apple, including code signed using a signing certificate issued by Apple to other developers, use the form

```
anchor apple generic
```

Part of a Certificate

To match a well-defined element of a certificate, use the form

`certificate` *position* `[`*element*`]` *match expression*

where *match expression* can include the * wildcard character and any of the operators listed in "Logical Operators" (page 24) and "Comparison Operations" (page 25). The currently supported elements are as follows:

> **Note** Case is significant in element names.

Element name	Meaning	Comments
`subject.CN`	Subject common name	Shown in Keychain Access utility
`subject.C`	Subject country name	
`subject.D`	Subject description	
`subject.L`	Subject locality	
`subject.O`	Subject organization	Usually company or organization
`subject.OU`	Subject organizational unit	
`subject.STREET`	Subject street address	

Certificate field by OID

To check for the existence of any certificate field identified by its X.509 object identifier (OID), use the form

`certificate` *position* `[field.`*OID*`] exists`

The object identifier must be written in numeric form (*x*`.`*y*`.`*z*...) and can be the OID of a certificate extension or of a conventional element of a certificate as defined by the CSSM standard (see Chapter 31 in *Common Security: CDSA and CSSM*, version 2 (with corrigenda) by the Open Group (http://www.opengroup.org/security/cdsa.htm)).

Trusted

The expression

`certificate` *position* `trusted`

succeeds if the certificate specified by *position* is marked trusted for the code signing certificate policy in the system's Trust Settings database. The *position* argument is an integer or keyword that indicates the position of the certificate in the chain; see the discussion under "Certificate" (page 27).

The expression

```
anchor trusted
```

succeeds if any certificate in the signature's certificate chain is marked trusted for the code signing certificate policy in the system's Trust Settings database, provided that no certificate closer to the leaf certificate is explicitly untrusted.

Thus, using the `trusted` keyword with a certificate position checks only the specified certificate, while using it with the `anchor` keyword checks all the certificates, giving precedence to the trust setting found closest to the leaf.

> **Important** The syntax `anchor trusted` is *not* a synonym for `certificate anchor trusted`. Whereas the former checks all certificates in the signature, the latter checks only the anchor certificate.

Certificates can have per-user trust settings and system-wide trust settings, and trust settings apply to specific policies. The `trusted` keyword in the code signing requirement language causes trust to be checked for the specified certificate or certificates for the user performing the validation. If there are no settings for that user, then the system settings are used. In all cases, only the trust settings for the code-signing policy are checked. Policies and trust are discussed in *Certificate, Key, and Trust Services Programming Guide*.

> **Important** If you do not include an expression using the `trusted` keyword in your code signing requirement, then the verifier does not check the trust status of the certificates in the code signature at all.

Entitlement

The expression

`entitlement [`*key*`]` *match expression*

succeeds if the value associated with the specified key in the signature's embedded entitlement dictionary matches *match expression*, where *match expression* can include the * wildcard character and any of the operators listed in "Logical Operators" (page 24) and "Comparison Operations" (page 25). You must specify *key* as a string constant. The entitlement dictionary is included in signatures for certain platforms.

Code Directory Hash

The expression

`cdhash` *hash-constant*

computes a SHA-1 hash of the program's CodeDirectory resource and succeeds if the value of this hash exactly equals the specified hash constant.

The CodeDirectory resource is the master directory of the contents of the program. It consists of a versioned header followed by an array of hashes. This array consists of a set of optional special hashes for other resources, plus a vector of hashes for pages of the main executable. The CodeSignature and CodeDirectory resources together make up the signature of the code.

You can use the codesign utility with (at least) three levels of verbosity to obtain the hash constant of a program's CodeDirectory resource:

```
$ codesign –dvvv /bin/ls
...
CodeDirectory v=20001 size=257 flags=0x0(none) hashes=8+2 location=embedded
CDHash=4bccbc576205de37914a3023cae7e737a0b6a802
...
```

Because the code directory changes whenever the program changes in a nontrivial way, this test can be used to unambiguously identify one specific version of a program. When the operating system signs an otherwise unsigned program (so that the keychain or Parental Controls can recognize the program, for example), it uses this requirement.

Requirement Sets

A requirement set is a collection of distinct requirements, each indexed (tagged) with a type code. The expression

tag `=>` *requirement*

applies *requirement* to the type of code indicated by *tag*, where possible tags are

- `host`—this requirement is applied to the direct host of this code module; each code module in the hosting path can have its own host requirement, where the hosting path is the chain of code signing hosts starting with the most specific code known to be running, and ending with the root of trust (the kernel)
- `guest`—this requirement is applied to each code module that is hosted by this code module
- `library`—this requirement is applied to all libraries mounted by the signed code
- `designated`—this is an explicitly specified designated requirement for the signed code; if there is no explicitly specified designated requirement for the code, then there is no `designated` internal requirement

The primary use of requirement sets is to represent the internal requirements of the signed code. For example:

```
    codesign -r='host => anchor apple and identifier com.apple.perl designated =>
 anchor /my/root and identifier com.bar.foo'
```

sets the internal requirements of some code, having a host requirement of `anchor apple and identifier com.apple.perl` ("I'm a Perl script and I want to be run by Apple's Perl interpreter") and an explicit designated requirement of `anchor /my/root and identifier com.bar.foo`. Note that this command sets no guest or library requirements.

You can also put the requirement set in a file and point to the file:

```
    codesign -r myrequirements.rqset
```

where the file `myrequirements.rqset` might contain:

```
    //internal requirements
      host => anchor apple and identifier com.apple.perl //require Apple's Perl
interpreter
      designated => anchor /my/root and identifier com.bar.foo
```

Document Revision History

This table describes the changes to *Code Signing Guide*.

Date	Notes
2011-09-28	Revised document to focus exclusively on code signing. Some of the content in this document was previously in *Code Signing and Application Sandboxing Guide*.
2011-07-11	Added information about application sandboxing.
2009-10-19	Fixed typographical errors.
2009-10-13	Clarified explanation of CFBundleIdentifier and uniqueness.
2008-11-19	Added a chapter describing the code signing requirement language.
2007-05-15	New document that explains why you should sign your code and provides code signing procedures.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010