NAME

codesign – Create and manipulate code signatures

SYNOPSIS

codesign -s *identity* [-i identifier] [-r requirements] [**-fv**] [*path* *...*]
codesign -v [-R requirement] [**-v**] [*path|pid* *...*]
codesign -d [**-v**] [*path|pid* *...*]
codesign -h [**-v**] [*pid* *...*]

DESCRIPTION

The **codesign** command is used to create, check, and display code signatures, as well as inquire into the dynamic status of signed code in the system.

codesign requires exactly one *operation* option to determine what action is to be performed, as well as any number of other options to modify its behavior. It can act on any number of objects per invocation, but performs the same operation on all of them.

codesign accepts single-character (classic) options, as well as GNU-style long options of the form --name and --name=value. Common options have both forms; less frequent and specialized options have only long form. Note that the form --name value (without equal sign) will not work as expected on options with optional values.

OPTIONS

The options are as follows:

--all-architectures

When verifying a code signature on code that has a universal ("fat") Mach-O binary, separately verify each architecture contained. This is the default unless overridden with the -a (--architecture) option.

-a, --architecture *architecture*

When verifying or displaying signatures, explicitly select the Mach-O architecture given. The *architecture* can be specified either by name (e.g. i386) or by number; if by number, a sub-architecture may be appended separated by a comma. This option applies only to Mach-O binary code and is ignored for other types. If the *path* uses the Mach-O format and contains no code of the given architecture, the command will fail. The default for verification is --all-architectures, to verify all architectures present. The default for display is to report on the native architecture of the host system. When signing, **codesign** will always sign all architectures contained in a universal Mach-O file.

--bundle-version *version-string*

When handling versioned bundles such as frameworks, explicitly specify the version to operate on. This must be one of the names in the "Versions" directory of the bundle. If not specified, **codesign** uses the bundle's default version. Note that most frameworks delivered with the system have only one version, and thus this option is irrelevant for them. There is currently no facility for operating on all versions of a bundle at once.

--check-notarization

When verifying the code at the path(s) given, force an online notarization check to see if a notarization ticket is available.

-d, --display

Display information about the code at the path(s) given. Increasing levels of verbosity produce more output. The format is designed to be moderately easy to parse by simple scripts while still making sense to human eyes. In addition, the -r, --file-list, --extract-certificates, and --entitlements options can be used to retrieve additional information.

-D, --detached filename

When signing, designates that a detached signature should be written to the specified file. The code being signed is not modified and need not be writable. When verifying, designates a file containing a detached signature to be used for verification. Any embedded signature in the code is ignored.

--deep When signing a bundle, specifies that nested code content such as helpers, frameworks, and plug-ins, should be recursively signed in turn. Beware that all signing options you specify will apply, in turn, to such nested content.

When verifying a bundle, specifies that any nested code content will be recursively verified as to its full content. By default, verification of nested content is limited to a shallow investigation that may not detect changes to the nested code.

When displaying a signature, specifies that a list of directly nested code should be written to the display output. This lists only code directly nested within the subject; anything nested indirectly will require recursive application of the **codesign** command.

--detached-database

When signing, specifies that a detached signature should be generated as with the --detached option, but that the resulting signature should be written into a system database, from where it is made automatically available whenever apparently unsigned code is validated on the system.

Writing to this system database requires elevated process privileges that are not available to ordinary users.

-f, --force

When signing, causes **codesign** to replace any existing signature on the path(s) given. Without this option, existing signatures will not be replaced, and the signing operation fails.

--generate-entitlement-der

When signing, convert the supplied entitlements XML data to DER and embed the entitlements as both XML and DER in the signature. Embedding DER entitlements is default behavior as of macOS 12.0 when signing for all platforms. This argument was introduced in macOS 10.14 (Mojave).

-h, --hosting

Constructs and prints the hosting chain of a running program. The *pid* arguments must denote running code (pids etc.) With verbose options, this also displays the individual dynamic validity status of each element of the hosting chain.

-i, --identifier *identifier*

During signing, explicitly specify the unique identifier string that is embedded in code signatures. If this option is omitted, the identifier is derived from either the Info.plist (if present), or the filename of the executable being signed, possibly modified by the --prefix option. It is a **very bad idea** to sign different programs with the same identifier.

-o, --options *flag,...*

During signing, specifies a set of option flags to be embedded in the code signature. The value takes the form of a comma-separated list of names (with no spaces). Alternatively, a numeric value can be used to directly specify the option mask (CodeDirectory flag word). See OPTION FLAGS below.

-P, --pagesize *pagesize*

Indicates the granularity of code signing. Pagesize must be a power of two. Chunks of pagesize bytes are separately signed and can thus be independently verified as needed. As a special case, a pagesize of zero indicates that the entire code should be signed and verified as a single, possibly gigantic page. This option only applies to the main executable and has no effect on the sealing of associated data, including resources.

--remove-signature

Removes the current code signature from the path(s) given.

-r, --requirements *requirements*

During signing, indicates that internal requirements should be embedded in the code path(s) as specified. See "specifying requirements" below. Defaults will be applied to requirement types that are not explicitly specified; if you want to defeat such a default, specify "never" for that type.

During display, indicates where to write the code's internal requirements. Use -r- to write them to standard output.

-R, --test-requirement *requirement*

During verification, indicates that the path(s) given should be verified against the code requirement specified. If this option is omitted, the code is verified only for internal integrity and against its own designated requirement.

-s, --sign _identity_

Sign the code at the path(s) given using this identity. See SIGNING IDENTITIES below.

-v, --verbose

Sets (with a numeric value) or increments the verbosity level of output. Without the verbose option, no output is produced upon success, in the classic UNIX style. If no other options request a different action, the first -v encountered will be interpreted as --verify instead (and does not increase verbosity).

-v, --verify

Requests verification of code signatures. If other actions (sign, display, etc.) are also requested, -v is interpreted to mean --verbose.

--continue

Instructs **codesign** to continue processing path arguments even if processing one fails. If this option is given, exit due to operational errors is deferred until all path arguments have been considered. The exit code will then indicate the most severe failure (or, with equal severity, the first such failure encountered).

--dryrun

During signing, performs almost all signing operations, but does not actually write the result anywhere. Cryptographic signatures are still generated, actually using the given signing identity and triggering any access control checks normally, though the resulting signature is then discarded.

--entitlements _path_

When signing, take the file at the given _path_ and embed its contents in the signature as entitlement data. If the data at _path_ does not already begin with a suitable binary ("blob") header, one is attached automatically.
When displaying a signature, extract any entitlement data from the signature and write it to the _path_ given in an abstract representation. If needed **--xml** or **--der** may be passed in to output the entitlements in a desired format, if you pass in both then DER will be printed. Use "-" as the path to write to standard output. If the signature has no entitlement data, nothing is written (this is not an error).

--extract-certificates _prefix_

When displaying a signature, extract the certificates in the embedded certificate chain and write them to individual files. The _prefix_ argument is appended with numbers 0, 1, ... to form the filenames, which can be relative or absolute. Certificate 0 is the leaf (signing) certificate, and as many files are written as there are certificates in the signature. The files are in ASN.1 (DER) form. If *prefix* is omitted, the default prefix is "codesign" in the current directory.

--file-list *path*

When signing or displaying a signature, **codesign** writes to the given path a list of files that may have been modified as part of the signing process. This is useful for installer or patcher programs that need to know what was changed or what files are needed to make up the "signature" of a program. The file given is appended-to, with one line per absolute path written. An argument of "-" (single dash) denotes standard output. Note that the list may be somewhat pessimistic - all files not listed are guaranteed to be unchanged by the signing process, but some of the listed files may not actually have changed. Also note that changes may have been made to extended attributes of these files.

--ignore-resources

During static validation, do not validate the contents of the code's resources. In effect, this will pass validation on code whose resources have been corrupted (or inappropriately signed). On large programs, it will also substantially speed up static validation, since all the resources will not be read into memory. Obviously, the outcome of such a validation should be considered on its merits.

--keychain *filename*

During signing, only search for the signing identity in the keychain file specified. This can be used to break any matching ties if you have multiple similarly-named identities in several keychains on the user's search list. Note that the standard keychain search path is still consulted while constructing the certificate chain being embedded in the signature.
Note that *filename* will not be searched to resolve the signing identity's certificate chain unless it is also on the user's keychain search list.

--prefix *string*

If no explicit unique identifier is specified (using the -i option), and if the implicitly generated identifier does not contain any dot (.) characters, then the given string is prefixed to the identifier before use. If the implicit identifier contains a dot, it is used as-is. Typically, this is used to deal with command tools without Info.plists, whose default identifier is simply the command's filename; the conventional prefix used is com.domain. (note that the final dot needs to be explicit).

--preserve-metadata=list

When re-signing code that is already signed, reuse some information from the old signature. If new data is specified explicitly, it is preferred. You still need to specify the -f (--force) option to

enable overwriting signatures at all. If this option is absent, any old signature has no effect on the signing process.
Note: if the *linker-signed* flag is present on the previous binary, then this option is ignored.
This option takes a comma-separated list of names, which you may reasonably abbreviate:

identifier Preserve the signing identifier (--identifier) instead of generating a default identifier.

entitlements Preserve the entitlement data (--entitlements).

requirements Preserve the internal requirements (--requirements option), including any explicit Designated Requirement. Note that all internal requirements are preserved or regenerated as a whole; you cannot pick and choose individual elements with this option.

flags Preserve the option flags (-o), see the OPTION FLAGS section below.

runtime Preserve the hardened runtime version (-o runtime flag, --runtime-version option) instead of overriding or deriving the version.

For historical reasons, this option can be given without a value, which preserves all of these values as presently known. This use is deprecated and will eventually be removed; always specify an explicit list of preserved items.

--strict *options*

When validating code, apply additional restrictions beyond the defaults.

symlinks Check that symbolic links inside the code bundle point to sealed files inside its bundle. This means that broken symbolic links are rejected, as are links to places outside the bundle and to places that are not, for whatever reason, sealed by the signature.

sideband Check that no resource forks, Finder attributes, or similar sideband data is present in the signed code. This is now automatically enforced by signing operations.

Options can be specified as a comma-separated list. Use plain --strict or --strict=all to be as strict as possible. Note that --strict=all may include more checking types over time.
Not all strictness check make sense in all circumstances, which is why these behaviors are not the defualt.

--timestamp [=*URL*]

During signing, requests that a *timestamp* *authority* *server* be contacted to authenticate the time of signing. The server contacted is given by the *URL* value. If this option is given without a value, a default server provided by Apple is used. Note that this server may not support signatures made with identities not furnished by Apple. If the timestamp authority service cannot be contacted over the Internet, or it malfunctions or refuses service, the signing operation will **fail**.

If this option is not given at all, a system-specific default behavior is invoked. This may result in some but not all code signatures being timestamped.

The special value *none* explicitly disables the use of timestamp services.

--runtime-version *version*

During signing, when the *runtime* OPTION FLAG is set, explicitly specify the hardened runtime version stored in the code signature. If this option is omitted, but the *runtime* OPTION FLAG is set then the hardened runtime version is omitted for non-Mach-O files and derived from the SDK version of Mach-O files.

OPERATION

In the first synopsis form, **codesign** attempts to sign the code objects at the *path(s)* given, using the *identity* provided. Internal *requirements* and *entitlements* are embedded if requested. Internal requirements not specified may be assigned suitable default values. Defaulting applies separately to each type of internal requirement. If an *identifier* is explicitly given, it is sealed into all *path(s)*. Otherwise, each path derives its *identifier* independently from its Info.plist or pathname. Code nested within bundle directories must already be signed or the signing operation will *fail*, unless the **--deep** option is given, in which case any unsigned nested code will be recursively signed before proceeding, using the same signing options and parameters. If the **--force** option is given, any existing top-level signature is replaced, subject to any **--preserve-metadata** options also present. Combining the **--force** and **--deep** options results in forcible replacement of all signatures within the target bundle.

In the second synopsis form, **codesign** verifies the code signatures on all the *path(s)* given. The verification confirms that the code at those *path(s)* is signed, that the signature is valid, and that all sealed components are unaltered. If a *requirement* is given, each *path* is also checked against this requirement (but see DIAGNOSTICS below). If verbose verification is requested, the program is also checked against its own designated requirement, which should never fail for a properly signed program.

If a *path* begins with a decimal digit, it is interpreted as the process id of a running process in the system, and dynamic validation is performed on that process instead. This checks the code's dynamic status and just enough static data to close the nominal security envelope. Add at least one level of verbosity to also perform a full static check.

In the third synopsis form, **codesign** displays the contents of the signatures on the *path(s)* given. More information is displayed as the verbosity level increases. This form may not completely verify the signatures on the *path(s)*; though it may perform some verification steps in the process of obtaining information about the *path(s)*. If the *-r path* option is given, internal requirements will be extracted from the *path(s)* and written to *path*; specify a dash "-" to write to standard output. If the code does not contain an explicit designated requirement, the implied one will be retrieved and written out as a source comment. If the *--entitlements path* option is given, embedded entitlement data will be extracted likewise and written to the file specified.

In the fourth synopsis form, **codesign** constructs the hosting path for each *pid* given and writes it, one host per line, to standard output. 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). If the *--verbose* option is given, the dynamic validity status of each host is also displayed, separated from the path by a tab character. Note that hosting chains can at times be constructed for invalid or even unsigned code, and the output of this form of the **codesign** command should not be taken as a statement of formal code validity. Only **codesign** *--verify* can do that; and in fact, formal verification constructs the hosting chain as part of its operation (but does not display it).

SIGNING IDENTITIES

To be used for code signing, a digital identity must be stored in a keychain that is on the calling user's keychain search list. All keychain sources are supported if properly configured. In particular, it is possible to sign code with an identity stored on a supported smart card. If your signing identity is stored in a different form, you need to make it available in keychain form to sign code with it.

If the *--keychain* argument is used, *identity* is only looked-for in the specific keychain given. This is meant to help disambiguate references to identities. Even in that case, the full keychain search list is still consulted for additional certificates needed to complete the signature.

The *identity* is first considered as the full name of a **keychain identity preference**. If such a preference exists, it directly names the identity used. Otherwise, the identity is located by searching all keychains for a certificate whose subject **common name** (only) contains the *identity* string given. If there are multiple matches, the operation fails and no signing is performed; however, an exact match is preferred over a partial match. These comparisons are case sensitive. Multiple instances of the exactly same certificate in multiple keychains are tolerated as harmless.

If *identity* consists of exactly forty hexadecimal digits, it is instead interpreted as the SHA-1 hash of the certificate part of the desired identity. In this case, the identity's subject name is not considered.

Both **identity preferences** and certificate hashes can be used to identify a particular signing identity regardless of name. Identity preferences are global settings *for each user* and provide a layer of indirection. Certificate hashes are very explicit and local. These choices, combined with what is placed into Xcode project and target build variables and/or script settings, allows for very flexible designation of signing identities.

If *identity* is the single letter "-" (dash), **ad-hoc signing** is performed. Ad-hoc signing does not use an identity at all, and identifies exactly one instance of code. Significant restrictions apply to the use of ad-hoc signed code; consult documentation before using this.

codesign will attempt to embed the entire certificate chain documenting the signing identity in the code signature it generates, including any intermediate certificates and the anchor certificate. It looks for those in the keychain search list of the user performing the signing operation. If it cannot generate the entire certificate chain, signing may still succeed, but verification may fail if the verifying code does not have an independent source for the missing certificates (from *its* keychains).

SPECIFYING REQUIREMENTS

The *requirement(s)* arguments (-r and -R) can be given in various forms. A plain text argument is taken to be a path to a file containing the requirement(s). **codesign** will accept both binary files containing properly compiled requirements code, and source files that are automatically compiled before use. An argument of "-" requests that the requirement(s) are read from standard input. Finally, an argument that begins with an equal sign "=" is taken as a literal requirements source text, and is compiled accordingly for use.

OPTION FLAGS

When signing, a set of option flags can be specified to change the behavior of the system when using the signed code. The following flags are recognized by **codesign**; other flags may exist at the API level. Note that you can specify any valid flags by giving a (single) numeric value instead of a list of option names.

kill   Forces the signed code's kill flag to be set when the code begins execution. Code with the kill flag set will die when it becomes dynamically invalid. It is therefore safe to assume that code marked this way, once validated, will have continue to have a valid identity while alive.

hard   Forces the signed code's hard flag to be set when the code begins execution. The hard flag is a hint to the system that the code prefers to be denied access to resources if gaining such access would invalidate its identity.

host   Marks the code as capable of hosting guest code. You must set this option if you want the code to act as a code signing host, controlling subsidiary ("guest") code. This flag is set automatically if you specify an internal guest requirement.

expires Forces any validation of the code to consider expiration of the certificates involved. Code signatures generated with this flag will fail to verify once any of the certificates in the chain has expired, regardless of the intentions of the verifier. Note that this flag does not affect any other checks that may cause signature validation to fail, including checks for certificate revocation.

library Forces the signed code's library validation flag to be set when the code begins execution. The code will only be able to link against system libraries and frameworks, or libraries, frameworks, and plug-in bundles with the same team identifier embedded in the code directory. Team identifiers are automatically recorded in signatures when signing with suitable Apple-issued signing certificates. Note that the flag is not supported for i386 binaries, and only applies to the main executable. The flag has no effect when set on frameworks and libraries.

runtime On macOS versions >= 10.14.0, opts signed processes into a hardened runtime environment which includes runtime code signing enforcement, library validation, hard, kill, and debugging restrictions. These restrictions can be selectively relaxed via entitlements. Note: macOS versions older than 10.14.0 ignore the presence of this flag in the code signature.

linker-signed
Identifies a signature as signed by the linker. Linker signatures are very similar to adhoc signatures, except:

- linker signatures can be replaced without using the *--force* option.

- linker signatures are never preserved regardless of the use of the *--preserve-metadata* option.

- linker signatures will usually not contain any embedded code requirements including a designated requirement.

Note that code can set the hard and kill flags on itself at any time. The signing options only affect their initial state. Once set by any means, these flags cannot be cleared for the lifetime of the code. Therefore, specifying such flags as signing options guarantees that they will be set whenever the signed code runs.

If the code being signed has an Info.plist that contains a key named CSFlags, the value of that key is taken as the default value for the options. The value of CSFlags can be a string in the same form as the --options option, or an integer number specifying the absolute numeric value. Note however that while you can abbreviate flag names on the command lines, you must spell them out in the Info.plist.

EXAMPLES

To sign application Terminal.app with a signing identity named "authority":

```
      codesign --sign authority Terminal.app
```

To sign the command-line tool "helper" with the same identity, overwriting any existing signature, using the signing identifier "com.mycorp.helper", and embedding a custom designated requirement

```
codesign -f --sign authority --prefix=com.mycorp. -r="designated =>
anchor /tmp/foo" helper
```

To enable the hardened runtime on Terminal.app and sign with the signing identity named "authority":

```
codesign --sign authority --options runtime Terminal.app
```

To verify the signature on Terminal.app and produce some verbose output:

```
codesign --verify --verbose Terminal.app
```

To verify the dynamic validity of process 666:

```
codesign --verify +666
```

To display all information about Terminal.app's code signature:

```
codesign --display --verbose=4 Terminal.app
```

To extract the internal requirements from Terminal.app to standard output:

```
codesign --display -r- Terminal.app
```

To display the entitlements of a binary or bundle:

```
codesign --display --entitlements - /sbin/launchd
codesign --display --entitlements - --der Terminal.app
```

To display the entitlements of process 666:

```
codesign --display --entitlements - +666
```

To display the XML entitlements of process 1337:

```
codesign --display --entitlements - --xml +1337
```

TROUBLESHOOTING

A common source of confusion when using **codesign** arises from the ordering of command line options. If **codesign** is not behaving as expected, consult this manual and check the ordering of your arguments. As a general rule **codesign** follows a **verb noun** rule. For example **--sign** should be placed before **--options** in the invocation. This is because you are performing a "sign" action with a given set of options.

If these are inverted and **--options** is provided before **--sign** in the invocation, the value of **--options** is ignored silently.

DIAGNOSTICS

codesign exits 0 if all operations succeed. This indicates that all codes were signed, or all codes verified properly as requested. If a signing or verification operation fails, the exit code is 1. Exit code 2 indicates invalid arguments or parameters. Exit code 3 indicates that during verification, all path(s) were properly signed but at least one of them failed to satisfy the requirement specified with the -R option.

For verification, all path arguments are always investigated before the program exits. For all other operations, the program exits upon the first error encountered, and any further path arguments are ignored, unless the

--continue option was specified, in which case **codesign** will defer the failure exit until after it has attempted to process all path arguments in turn.

SIGNING ATOMICITY

When a signing operation fails for a particular code, the code may already have been modified in certain ways by adding requisite signature data. Such information will not change the operation of the code, and the code will not be considered signed even with these pieces in place. You may repeat the signing operation without difficulty. Note however that a previous valid signature may have been effectively destroyed if you specified the -f option.
If you require atomicity of signing stricter than provided by **codesign**, you need to make an explicit copy of your code and sign that.

ENVIRONMENT

If the CODESIGN_ALLOCATE environment variable is set, it identifies a substitute codesign_allocate tool used to allocate space for code signatures in Mach-O binaries. This is used by Xcode SDK distributions to provide architectural support for non-native platforms such as iPhones. The system will not accept such substitutes unless they are specially signed (by Apple).

FILES

/var/db/DetachedSignatures	System-wide database of detached code signatures for unsigned code.

SEE ALSO

csreq(1), xcodebuild(1), codesign_allocate(1)

HISTORY

The **codesign** command first appeared in Mac OS 10.5.0 (Leopard).

BUGS

Some options only apply to particular operations, and **codesign** ignores them (without complaining) if you specify them for an operation for which they have no meaning.

The --preserve-metadata option used to take no value, and varied across releases in what exactly it preserved. The ensuing confusion is still with you if you need to make backward-compatible scripts.

The dual meaning of the -v option, indicating either verbosity or verification, confuses some people. If you find it confusing, use the unambiguous long forms --verbose and --verify instead.

The **--verify** option can take either a file or a pid. If your file path starts with a number you should prefix it with "./" to force **codesign** to interpret the argument as a path. For example:

```
codesign --verify 666
```

would become:

```
codesign --verify ./666
```

NOTES

The Xcode build system invokes **codesign** automatically if the CODE_SIGN_IDENTITY build variable is set. You can express any combination of **codesign** options with additional build variables there.

codesign is fundamentally a shell around the code signing APIs, and performs nothing of the underlying work. Replacing it with older or newer versions is unlikely to have a useful effect.

codesign has several operations and options that are purposely left undocumented in this manual page because they are either experimental (and subject to change at any time), or unadvised to the unwary. The interminably curious are referred to the published source code.

[Process completed]