jarsigner - JAR Signing and Verification Tool

Generates signatures for Java ARchive (JAR) files, and verifies the signatures of signed JAR files.

SYNOPSIS

```
jarsigner [ options ] jar-file alias
jarsigner -verify [ options ] jar-file
```

DESCRIPTION

The **jarsigner** tool is used for two purposes:

1. to sign Java ARchive (JAR) files, and
2. to verify the signatures and integrity of signed JAR files.

The JAR feature enables the packaging of class files, images, sounds, and other digital data in a single file for faster and easier distribution. A tool named **jar** enables developers to produce JAR files. (Technically, any zip file can also be considered a JAR file, although when created by **jar** or processed by **jarsigner**, JAR files also contain a META-INF/MANIFEST.MF file.)

A *digital signature* is a string of bits that is computed from some data (the data being "signed") and the private key of an entity (a person, company, etc.). Like a handwritten signature, a digital signature has many useful characteristics:

- Its authenticity can be verified, via a computation that uses the public key corresponding to the private key used to generate the signature.
- It cannot be forged, assuming the private key is kept secret.
- It is a function of the data signed and thus can't be claimed to be the signature for other data as well.
- The signed data cannot be changed; if it is, the signature will no longer verify as being authentic.

In order for an entity's signature to be generated for a file, the entity must first have a public/private key pair associated with it, and also one or more certificates authenticating its public key. A *certificate* is a digitally signed statement from one entity, saying that the public key of some other entity has a particular value.

jarsigner uses key and certificate information from a *keystore* to generate digital signatures for JAR files. A keystore is a database of private keys and their associated X.509 certificate chains authenticating the corresponding public keys. The **keytool** utility is used to create and administer keystores.

jarsigner uses an entity's private key to generate a signature. The signed JAR file contains, among other things, a copy of the certificate from the keystore for the public key corresponding to the private key used to sign the file. **jarsigner** can verify the digital signature of the signed JAR file using the certificate inside it (in its signature block file).

Starting in J2SE 5.0, **jarsigner** can generate signatures that include a timestamp, thus enabling systems/deployer (including Java Plug-in) to check whether the JAR file was signed while the signing certificate was still valid. In addition, APIs were added in J2SE 5.0 to allow applications to obtain the timestamp information.

At this time, **jarsigner** can only sign JAR files created by the SDK **jar** tool or zip files. (JAR files are the same as zip files, except they also have a META-INF/MANIFEST.MF file. Such a file will automatically be created when **jarsigner** signs a zip file.)

The default **jarsigner** behavior is to *sign* a JAR (or zip) file. Use the `-verify` option to instead have it *verify* a signed JAR file.

Compatibility with JDK 1.1

The **keytool** and **jarsigner** tools completely replace the **javakey** tool provided in JDK 1.1. These new tools provide more features than **javakey**, including the ability to protect the keystore and private keys with passwords, and the ability to verify signatures in addition to generating them.

The new keystore architecture replaces the identity database that **javakey** created and managed. There is no backwards compatibility between the keystore format and the database format used by **javakey** in 1.1. However,

- It is possible to import the information from an identity database into a keystore, via the **keytool** `-identitydb` command.
- **jarsigner** can sign JAR files also previously signed using **javakey**.

- **jarsigner** can verify JAR files signed using **javakey**. Thus, it recognizes and can work with signer aliases that are from a JDK 1.1 identity database rather than a Java 2 SDK keystore.

The following table explains how JAR files that were signed in JDK 1.1.x are treated in the Java 2 platform.

JAR File Type	Identity in 1.1 database	Trusted Identity imported into Java 2 Platform keystore from 1.1 database (4)	Policy File grants privileges to Identity/Alias	Privileges Granted
Signed JAR	NO	NO	NO	Default privileges granted to all code.
Unsigned JAR	NO	NO	NO	Default privileges granted to all code.
Signed JAR	NO	YES	NO	Default privileges granted to all code.
Signed JAR	YES/Untrusted	NO	NO	Default privileges granted to all code. (3)
Signed JAR	YES/Untrusted	NO	YES	Default privileges granted to all code. (1,3)
Signed JAR	NO	YES	YES	Default privileges granted to all code plus privileges granted in policy file.
Signed JAR	YES/Trusted	YES	YES	Default privileges granted to all code plus privileges granted in policy file. (2)
Signed JAR	YES/Trusted	NO	NO	All privileges
Signed JAR	YES/Trusted	YES	NO	All privileges (1)
Signed JAR	YES/Trusted	NO	YES	All privileges (1)

Notes:

1. If an identity/alias is mentioned in the policy file, it must be imported into the keystore for the policy file to have any effect on privileges granted.
2. The policy file/keystore combination has precedence over a trusted identity in the identity database.
3. Untrusted identities are ignored in the Java 2 platform.
4. Only trusted identities can be imported into Java 2 SDK keystores.

Keystore Aliases

All keystore entities are accessed via unique *aliases*.

When using **jarsigner** to sign a JAR file, you must specify the alias for the keystore entry containing the private key needed to generate the signature. For example, the following will sign the JAR file named "MyJARFile.jar", using the private key associated with the alias "duke" in the keystore named "mystore" in the "working" directory. Since no output file is specified, it overwrites MyJARFile.jar with the signed JAR file.

```
    jarsigner -keystore /working/mystore -storepass myspass
      -keypass dukekeypasswd MyJARFile.jar duke
```

Keystores are protected with a password, so the store password (in this case "myspass") must be specified. You will be prompted for it if you don't specify it on the command line. Similarly, private keys are protected in a keystore with a password, so the private key's password (in this case "dukekeypasswd") must be specified, and you will be prompted for it if you don't specify it on the command line and it isn't the same as the store password.

Keystore Location

jarsigner has a `-keystore` option for specifying the URL of the keystore to be used. The keystore is by default stored in a file named `.keystore` in the user's home directory, as determined by the `user.home` system property. On Solaris systems `user.home` defaults to the user's home directory.

Note that the input stream from the `-keystore` option is passed to the `KeyStore.load` method. If `NONE` is specified as the URL, then a null stream is passed to the `KeyStore.load` method. `NONE` should be specified if the `KeyStore` is not file-based, for example, if it resides on a hardware token device.

Keystore Implementation

The `KeyStore` class provided in the `java.security` package supplies well-defined interfaces to access and modify the information in a keystore. It is possible for there to be multiple different concrete implementations, where each implementation is that for a particular *type* of keystore.

Currently, there are two command-line tools that make use of keystore implementations (**keytool** and **jarsigner**), and also a GUI-based tool named **Policy Tool**. Since `KeyStore` is publicly available, Java 2 SDK users can write additional security applications that use it.

There is a built-in default implementation, provided by Sun Microsystems. It implements the keystore as a file, utilizing a proprietary keystore type (format) named "JKS". It protects each private key with its individual password, and also protects the integrity of the entire keystore with a (possibly different) password.

Keystore implementations are provider-based. More specifically, the application interfaces supplied by `KeyStore` are implemented in terms of a "Service Provider Interface" (SPI). That is, there is a corresponding abstract `KeystoreSpi` class, also in the `java.security` package, which defines the Service Provider Interface methods that "providers" must implement. (The term "provider" refers to a package or a set of packages that supply a concrete implementation of a subset of services that can be accessed by the Java Security API.) Thus, to provide a keystore implementation, clients must implement a provider and supply a KeystoreSpi subclass implementation, as described in How to Implement a Provider for the Java Cryptography Architecture.

Applications can choose different *types* of keystore implementations from different providers, using the "getInstance" factory method supplied in the `KeyStore` class. A keystore type defines the storage and data format of the keystore information, and the algorithms used to protect private keys in the keystore and the integrity of the keystore itself. Keystore implementations of different types are not compatible.

keytool works on any file-based keystore implementation. (It treats the keytore location that is passed to it at the command line as a filename and converts it to a FileInputStream, from which it loads the keystore information.) The **jarsigner** and **policytool** tools, on the other hand, can read a keystore from any location that can be specified using a URL.

For **jarsigner** and **keytool**, you can specify a keystore type at the command line, via the *-storetype* option. For **Policy Tool**, you can specify a keystore type via the "Change Keystore" command in the Edit menu.

If you don't explicitly specify a keystore type, the tools choose a keystore implementation based simply on the value of the `keystore.type` property specified in the security properties file. The security properties file is called `java.security`, and it resides in the SDK security properties directory, *`java.home`*`/lib/security`, where *java.home* is the runtime environment's directory (the `jre` directory in the SDK or the top-level directory of the Java 2 Runtime Environment).

Each tool gets the `keystore.type` value and then examines all the currently-installed providers until it finds one that implements keystores of that type. It then uses the keystore implementation from that provider.

The `KeyStore` class defines a static method named `getDefaultType` that lets applications and applets retrieve the value of the `keystore.type` property. The following line of code creates an instance of the default keystore type (as specified in the `keystore.type` property):

```
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
```

The default keystore type is "jks" (the proprietary type of the keystore implementation provided by Sun). This is specified by the following line in the security properties file:

```
keystore.type=jks
```

Note: Case doesn't matter in keystore type designations. For example, "JKS" would be considered the same as "jks".

To have the tools utilize a keystore implementation other than the default, change that line to specify a different keystore type. For example, if you have a provider package that supplies a keystore implementation for a keystore type called "pkcs12", change the line to

```
keystore.type=pkcs12
```

Note that if you us the PKCS#11 provider package, you should refer to the KeyTool and JarSigner section of the Java PKCS#11 Reference Guide for details.

Supported Algorithms

By default, **jarsigner** signs a JAR file using either

- DSA (Digital Signature Algorithm) with the SHA-1 digest algorithm, or
- the RSA algorithm with the MD5 digest algorithm.

That is, if the signer's public and private keys are DSA keys, **jarsigner** will sign the JAR file using the "SHA1withDSA" algorithm. If the signer's keys are RSA keys, **jarsigner** will attempt to sign the JAR file using the "MD5withRSA" algorithm.

These default signature algorithms can be overridden using the *-sigalg* option.

The Signed JAR File

When **jarsigner** is used to sign a JAR file, the output signed JAR file is exactly the same as the input JAR file, except that it has two additional files placed in the META-INF directory:

- a signature file, with a .SF extension, and
- a signature block file, with a .DSA extension.

The base file names for these two files come from the value of the `-sigFile` option. For example, if the option appears as

```
-sigFile MKSIGN
```

the files are named "MKSIGN.SF" and "MKSIGN.DSA".

If no `-sigfile` option appears on the command line, the base file name for the .SF and .DSA files will be the first 8 characters of the alias name specified on the command line, all converted to upper case. If the alias name has fewer than 8 characters, the full alias name is used. If the alias name contains any characters that are not allowed in a signature file name, each such character is converted to an underscore ("_") character in forming the file name. Legal characters include letters, digits, underscores, and hyphens.

The Signature (.SF) File

A signature file (the .SF file) looks similar to the manifest file that is always included in a JAR file when **jarsigner** is used to sign the file. That is, for each source file included in the JAR file, the .SF file has three lines, just as in the manifest file, listing the following:

- the file name,
- the name of the digest algorithm used (SHA), and
- a SHA digest value.

In the manifest file, the SHA digest value for each source file is the digest (hash) of the binary data in the source file. In the .SF file, on the other hand, the digest value for a given source file is the hash of the three lines in the manifest file for the source file.

The signature file also, by default, includes a header containing a hash of the whole manifest file. The presence of the header enables verification optimization, as described in JAR File Verification.

The Signature Block (.DSA) File

The .SF file is signed and the signature is placed in the .DSA file. The .DSA file also contains, encoded inside it, the certificate or certificate chain from the keystore which authenticates the public key corresponding to the private key used for signing.

Signature Timestamp

As of the J2SE 5.0 release, the `jarsigner` tool can now generate and store a signature timestamp when signing a JAR file. In addition, `jarsigner` supports alternative signing mechanisms. This behavior is optional and is controlled by the user at the time of signing through these options:

- `-tsa url`
- `-tsacert alias`
- `-altsigner class`
- `-altsignerpath classpathlist`

Each of these options is detailed in the Options section below.

JAR File Verification

A successful JAR file verification occurs if the signature(s) are valid, and none of the files that were in the JAR file when the signatures were generated have been changed since then. JAR file verification involves the following steps:

1. Verify the signature of the .SF file itself.
   That is, the verification ensures that the signature stored in each signature block (.DSA) file was in fact generated using the private key corresponding to the public key whose certificate (or certificate chain) also appears in the .DSA file. It also ensures that the signature is a valid signature of the corresponding signature (.SF) file, and thus the .SF file has not been tampered with.

2. Verify the digest listed in each entry in the .SF file with each corresponding section in the manifest.
   The .SF file by default includes a header containing a hash of the entire manifest file. When the header is present, then

the verification can check to see whether or not the hash in the header indeed matches the hash of the manifest file. If that is the case, verification proceeds to the next step.

If that is not the case, a less optimized verification is required to ensure that the hash in each source file information section in the .SF file equals the hash of its corresponding section in the manifest file (see The Signature (.SF) File).

One reason the hash of the manifest file that is stored in the .SF file header may not equal the hash of the current manifest file would be because one or more files were added to the JAR file (using the `jar` tool) after the signature (and thus the .SF file) was generated. When the `jar` tool is used to add files, the manifest file is changed (sections are added to it for the new files), but the .SF file is not. A verification is still considered successful if none of the files that were in the JAR file when the signature was generated have been changed since then, which is the case if the hashes in the non-header sections of the .SF file equal the hashes of the corresponding sections in the manifest file.

3. Read each file in the JAR file that has an entry in the .SF file. While reading, compute the file's digest, and then compare the result with the digest for this file in the manifest section. The digests should be the same, or verification fails.

If any serious verification failures occur during the verification process, the process is stopped and a security exception is thrown. It is caught and displayed by **jarsigner**.

Multiple Signatures for a JAR File

A JAR file can be signed by multiple people simply by running the **jarsigner** tool on the file multiple times, specifying the alias for a different person each time, as in:

```
jarsigner myBundle.jar susan
jarsigner myBundle.jar kevin
```

When a JAR file is signed multiple times, there are multiple .SF and .DSA files in the resulting JAR file, one pair for each signature. Thus, in the example above, the output JAR file includes files with the following names:

```
SUSAN.SF
SUSAN.DSA
KEVIN.SF
KEVIN.DSA
```

Note: It is also possible for a JAR file to have mixed signatures, some generated by the JDK 1.1 **javakey** tool and others by **jarsigner**. That is, **jarsigner** can be used to sign JAR files already previously signed using **javakey**.

OPTIONS

The various **jarsigner** options are listed and described below. Note:

- All option names are preceded by a minus sign (-).
- The options may be provided in any order.
- Items in italics (option values) represent the actual values that must be supplied.
- The `-keystore`, `-storepass`, `-keypass`, `-sigfile`, `-sigalg`, `-digestalg`, and `-signedjar` options are only relevant when signing a JAR file, not when verifying a signed JAR file. Similarly, an alias is only specified on the command line when signing a JAR file.

`-keystore` ***url***

Specifies the URL that tells the keystore location. This defaults to the file *.keystore* in the user's home directory, as determined by the "user.home" system property.

A keystore is required when signing, so you must explicitly specify one if the default keystore does not exist (or you want to use one other than the default).

A keystore is *not* required when verifying, but if one is specified, or the default exists, and the `-verbose` option was also specified, additional information is output regarding whether or not any of the certificates used to verify the JAR file are contained in that keystore.

Note: the `-keystore` argument can actually be a file name (and path) specification rather than a URL, in which case it will be treated the same as a "file:" URL. That is,

```
-keystore filePathAndName
```

is treated as equivalent to

```
-keystore file:filePathAndName
```

If the Sun PKCS#11 provider has been configured in the `java.security` security properties file (located in the JRE's `$JAVA_HOME/lib/security` directory), then keytool and jarsigner can operate on the PKCS#11 token by specifying these options:

- `-keystore NONE`
- `-storetype PKCS11`

For example, this command lists the contents of the configured PKCS#11 token:

```
jarsigner -keystore NONE -storetype PKCS11 -list
```

`-storetype` ***storetype***

Specifies the type of keystore to be instantiated. The default keystore type is the one that is specified as the value of the "keystore.type" property in the security properties file, which is returned by the static `getDefaultType` method in `java.security.KeyStore`.

The PIN for a PCKS#11 token can also be specified using the `-storepass` option. If none has been specified, keytool and jarsigner will prompt for the token PIN. If the token has a protected authentication path (such as a dedicated PIN-pad or a biometric reader), then the `-protected` option must be specified and no password options can be specified.

`-storepass` ***password***

Specifies the password which is required to access the keystore. This is only needed when signing (not verifying) a JAR file. In that case, if a `-storepass` option is not provided at the command line, the user is prompted for the password.

Note: The password shouldn't be specified on the command line or in a script unless it is for testing purposes, or you are on a secure system.

`-keypass` ***password***

Specifies the password used to protect the private key of the keystore entry addressed by the alias specified on the command line. The password is required when using **jarsigner** to sign a JAR file. If no password is provided on the command line, and the required password is different from the store password, the user is prompted for it.

Note: The password shouldn't be specified on the command line or in a script unless it is for testing purposes, or you are on a secure system.

`-sigfile` ***file***

Specifies the base file name to be used for the generated .SF and .DSA files. For example, if *file* is "DUKESIGN", the generated .SF and .DSA files will be named "DUKESIGN.SF" and "DUKESIGN.DSA", and will be placed in the "META-INF" directory of the signed JAR file.

The characters in *file* must come from the set "a-zA-Z0-9_-". That is, only letters, numbers, underscore, and hyphen characters are allowed. Note: All lowercase characters will be converted to uppercase for the .SF and .DSA file names.

If no `-sigfile` option appears on the command line, the base file name for the .SF and .DSA files will be the first 8 characters of the alias name specified on the command line, all converted to upper case. If the alias name has fewer than 8 characters, the full alias name is used. If the alias name contains any characters that are not legal in a signature file name, each such character is converted to an underscore ("_") character in forming the file name.

`-sigalg` ***algorithm***

Specifies the name of the signature algorithm to use to sign the JAR file.

See Appendix A of the Java Cryptography Architecture for a list of standard signature algorithm names. This algorithm must be compatible with the private key used to sign the JAR file. If this option is not specified, SHA1withDSA or MD5withRSA will be used depending on the type of private key. There must either be a statically installed provider supplying an implementation of the specified algorithm or the user must specify one with the *-providerClass* option, otherwise the command will not succeed.

`-digestalg` ***algorithm***

Specifies the name of the message digest algorithm to use when digesting the entries of a jar file.

See Appendix A of the Java Cryptography Architecture for a list of standard message digest algorithm names. If this option is not specified, SHA-1 will be used. There must either be a statically installed provider supplying an implementation of the specified algorithm or the user must specify one with the *-providerClass* option, otherwise the command will not succeed.

`-signedjar` ***file***

Specifies the name to be used for the signed JAR file.

If no name is specified on the command line, the name used is the same as the input JAR file name (the name of the JAR file to be signed); in other words, that file is overwritten with the signed JAR file.

`-verify`

If this appears on the command line, the specified JAR file will be verified, not signed. If the verification is successful, "jar verified" will be displayed. If you try to verify an unsigned JAR file, or a JAR file signed with an unsupported algorithm (e.g., RSA when you don't have an RSA provider installed), the following is displayed: "jar is unsigned. (signatures missing or not parsable)"

It is possible to verify JAR files signed using either **jarsigner** or the JDK 1.1 **javakey** tool, or both.

For further information on verification, see JAR File Verification.

`-certs`

If this appears on the command line, along with the `-verify` and `-verbose` options, the output includes certificate information for each signer of the JAR file. This information includes

- the name of the type of certificate (stored in the .DSA file) that certifies the signer's public key
- if the certificate is an X.509 certificate (more specifically, an instance of `java.security.cert.X509Certificate`): the distinguished name of the signer

The keystore is also examined. If no keystore value is specified on the command line, the default keystore file (if any) will be checked.

If the public key certificate for a signer matches an entry in the keystore, then the following information will also be displayed:

- in parentheses, the alias name for the keystore entry for that signer. If the signer actually comes from a JDK 1.1 identity database instead of from a keystore, the alias name will appear in brackets instead of parentheses.

`-verbose`

If this appears on the command line, it indicates "verbose" mode, which causes **jarsigner** to output extra information as to the progress of the JAR signing or verification.

`-internalsf`

In the past, the .DSA (signature block) file generated when a JAR file was signed used to include a complete encoded copy of the .SF file (signature file) also generated. This behavior has been changed. To reduce the overall size of the output JAR file, the .DSA file by default doesn't contain a copy of the .SF file anymore. But if `-internalsf` appears on the command line, the old behavior is utilized. **This option is mainly useful for testing; in practice, it should not be used, since doing so eliminates a useful optimization.**

`-sectionsonly`

If this appears on the command line, the .SF file (signature file) generated when a JAR file is signed does *not* include a header containing a hash of the whole manifest file. It just contains information and hashes related to each individual source file included in the JAR file, as described in The Signature (.SF) File .

By default, this header is added, as an optimization. When the header is present, then whenever the JAR file is verified, the verification can first check to see whether or not the hash in the header indeed matches the hash of the whole manifest file. If so, verification proceeds to the next step. If not, it is necessary to do a less optimized verification that the hash in each source file information section in the .SF file equals the hash of its corresponding section in the manifest file.

For further information, see JAR File Verification.

This option is mainly useful for testing; in practice, it should not be used, since doing so eliminates a useful optimization.

`-provider` ***provider-class-name***

Used to specify the name of cryptographic service provider's master class file when the service provider is not listed in the security properties file, `java.security`.

Used in conjunction with the `-providerArg` *ConfigFilePath* option, keytool and jarsigner will install the provider dynamically (where *ConfigFilePath* is the path to the token configuration file). Here's an example of a command to list a PKCS#11 keystore when the Sun PKCS#11 provider has not been configured in the security properties file.

```
jarsigner -keystore NONE -storetype PKCS11 \
          -providerClass sun.security.pkcs11.SunPKCS11 \
          -providerArg /foo/bar/token.config \
          -list
```

`-providerName` ***providerName***

If more than one provider has been configured in the `java.security` security properties file, you can use the `-providerName` option to target a specific provider instance. The argument to this option is the name of the provider.

For the Sun PKCS#11 provider, *providerName* is of the form `SunPKCS11-`*`TokenName`*, where *`TokenName`* is the name suffix that the provider instance has been configured with, as detailed in the configuration attributes table. For example, the following command lists the contents of the PKCS#11 keystore provider instance with name suffix `SmartCard`:

```
jarsigner -keystore NONE -storetype PKCS11 \
        -providerName SunPKCS11-SmartCard \
        -list
```

-J*javaoption*

Passes through the specified *javaoption* string directly to the Java interpreter. (**jarsigner** is actually a "wrapper" around the interpreter.) This option should not contain any spaces. It is useful for adjusting the execution environment or memory usage. For a list of possible interpreter options, type `java -h` or `java -X` at the command line.

`-tsa` ***url***

If `"-tsa http://example.tsa.url"` appears on the command line when signing a JAR file then a timestamp is generated for the signature. The URL, `http://example.tsa.url`, identifies the location of the Time Stamping Authority (TSA). It overrides any URL found via the `-tsacert` option. The `-tsa` option does not require the TSA's public key certificate to be present in the keystore.

To generate the timestamp, `jarsigner` communicates with the TSA using the Time-Stamp Protocol (TSP) defined in RFC 3161. If successful, the timestamp token returned by the TSA is stored along with the signature in the signature block file.

`-tsacert` ***alias***

If `"-tsacert alias"` appears on the command line when signing a JAR file then a timestamp is generated for the signature. The `alias` identifies the TSA's public key certificate in the keystore that is currently in effect. The entry's certificate is examined for a Subject Information Access extension that contains a URL identifying the location of the TSA.

The TSA's public key certificate must be present in the keystore when using `-tsacert`.

`-altsigner` ***class***

Specifies that an alternative signing mechanism be used. The fully-qualified class name identifies a class file that extends the `com.sun.jarsigner.ContentSigner abstract class`. The path to this class file is defined by the `-altsignerpath` option.

If the `-altsigner` option is used, `jarsigner` uses the signing mechanism provided by the specified class. Otherwise, `jarsigner` uses its default signing mechanism.

For example, to use the signing mechanism provided by a class named `com.sun.sun.jarsigner.AuthSigner`, use the `jarsigner` option "`-altsigner com.sun.jarsigner.AuthSigner`"

`-altsignerpath` ***classpathlist***

Specifies the path to the class file (the class file name is specified with the `-altsigner` option described above) and any JAR files it depends on. If the class file is in a JAR file, then this specifies the path to that JAR file, as shown in the example below.

An absolute path or a path relative to the current directory may be specified. If `classpathlist` contains multiple paths or JAR files, they should be separated with a colon (`:`) on Solaris and a semi-colon (`;`) on Windows. This option is not necessary if the class is already in the search path.

Example of specifying the path to a jar file that contains the class file:

```
  -altsignerpath /home/user/lib/authsigner.jar
```

Note that the JAR file name is included.

Example of specifying the path to the jar file that contains the class file:

```
  -altsignerpath /home/user/classes/com/sun/tools/jarsigner/
```

Note that the JAR file name is omitted.

EXAMPLES

Signing a JAR File

Suppose you have a JAR file named "bundle.jar" and you'd like to sign it using the private key of the user whose keystore alias is "jane" in the keystore named "mystore" in the "working" directory. Suppose the keystore password is "myspass" and the password for *jane*'s private key is "j638klm". You can use the following to sign the JAR file and name the signed JAR file "sbundle.jar":

```
jarsigner -keystore /working/mystore -storepass myspass
  -keypass j638klm -signedjar sbundle.jar bundle.jar jane
```

Note that there is no `-sigfile` specified in the command above, so the generated .SF and .DSA files to be placed in the signed JAR file will have default names based on the alias name. That is, they will be named `JANE.SF` and `JANE.DSA`.

If you want to be prompted for the store password and the private key password, you could shorten the above command to

```
jarsigner -keystore /working/mystore
  -signedjar sbundle.jar bundle.jar jane
```

If the keystore to be used is the default keystore (the one named ".keystore" in your home directory), you don't need to specify a keystore, as in:

```
jarsigner -signedjar sbundle.jar bundle.jar jane
```

Finally, if you want the signed JAR file to simply overwrite the input JAR file (`bundle.jar`), you don't need to specify a `-signedjar` option:

```
jarsigner bundle.jar jane
```

Verifying a Signed JAR File

To verify a signed JAR file, that is, to verify that the signature is valid and the JAR file has not been tampered with, use a command such as the following:

```
jarsigner -verify sbundle.jar
```

If the verification is successful,

```
jar verified.
```

is displayed. Otherwise, an error message appears.

You can get more information if you use the `-verbose` option. A sample use of **jarsigner** with the `-verbose` option is shown below, along with sample output:

```
jarsigner -verify -verbose sbundle.jar

       198 Fri Sep 26 16:14:06 PDT 1997 META-INF/MANIFEST.MF
       199 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.SF
      1013 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.DSA
smk   2752 Fri Sep 26 16:12:30 PDT 1997 AclEx.class
smk    849 Fri Sep 26 16:12:46 PDT 1997 test.class
```

```
      s = signature was verified
      m = entry is listed in manifest
      k = at least one certificate was found in keystore

    jar verified.
```

Verification with Certificate Information

If you specify the `-certs` option when verifying, along with the `-verify` and `-verbose` options, the output includes certificate information for each signer of the JAR file, including the certificate type, the signer distinguished name information (iff it's an X.509 certificate), and, in parentheses, the keystore alias for the signer if the public key certificate in the JAR file matches that in a keystore entry. For example,

```
    jarsigner -keystore /working/mystore -verify -verbose -certs myTest.jar

           198 Fri Sep 26 16:14:06 PDT 1997 META-INF/MANIFEST.MF
           199 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.SF
          1013 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.DSA
           208 Fri Sep 26 16:23:30 PDT 1997 META-INF/JAVATEST.SF
          1087 Fri Sep 26 16:23:30 PDT 1997 META-INF/JAVATEST.DSA
    smk   2752 Fri Sep 26 16:12:30 PDT 1997 Tst.class

      X.509, CN=Test Group, OU=Java Software, O=Sun Microsystems, L=CUP, S=CA, C=US (javatest)
      X.509, CN=Jane Smith, OU=Java Software, O=Sun, L=cup, S=ca, C=us (jane)

      s = signature was verified
      m = entry is listed in manifest
      k = at least one certificate was found in keystore

    jar verified.
```

If the certificate for a signer is not an X.509 certificate, there is no distinguished name information. In that case, just the certificate type and the alias are shown. For example, if the certificate is a PGP certificate, and the alias is "bob", you'd get

```
      PGP, (bob)
```

Verification of a JAR File that Includes Identity Database Signers

If a JAR file has been signed using the JDK 1.1 **javakey** tool, and thus the signer is an alias in an identity database, the verification output includes an "i" symbol. If the JAR file has been signed by both an alias in an identity database and an alias in a keystore, both "k" and "i" appear.

When the `-certs` option is used, any identity database aliases are shown in square brackets rather than the parentheses used for keystore aliases. For example:

```
    jarsigner -keystore /working/mystore -verify -verbose -certs writeFile.jar

           198 Fri Sep 26 16:14:06 PDT 1997 META-INF/MANIFEST.MF
           199 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.SF
          1013 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.DSA
           199 Fri Sep 27 12:22:30 PDT 1997 META-INF/DUKE.SF
          1013 Fri Sep 27 12:22:30 PDT 1997 META-INF/DUKE.DSA
   smki   2752 Fri Sep 26 16:12:30 PDT 1997 writeFile.html

      X.509, CN=Jane Smith, OU=Java Software, O=Sun, L=cup, S=ca, C=us (jane)
      X.509, CN=Duke, OU=Java Software, O=Sun, L=cup, S=ca, C=us [duke]

      s = signature was verified
      m = entry is listed in manifest
      k = at least one certificate was found in keystore
      i = at least one certificate was found in identity scope

    jar verified.
```

Note that the alias "duke" is in brackets to denote that it is an identity database alias, not a keystore alias.

SEE ALSO

- jar tool documentation
- keytool tool documentation
- the **Security** trail of the **Java Tutorial** for examples of the use of the **jarsigner** tool

ORACLE
Java Technology

Contact Us