NAME

jarsigner − sign and verify Java Archive (JAR) files

SYNOPSIS

`jarsigner` [*options*] *jar−file alias*

`jarsigner −verify` [*options*] *jar−file* [*alias ...*]

options The command−line options. See **Options for jarsigner**.

`−verify`
The **`−verify`** option can take zero or more keystore alias names after the JAR file name. When the **`−verify`** option is specified, the **`jarsigner`** command checks that the certificate used to verify each signed entry in the JAR file matches one of the keystore aliases. The aliases are defined in the keystore specified by **`−keystore`** or the default keystore.

If you also specify the **`−strict`** option, and the **`jarsigner`** command detects severe warnings, the message, "jar verified, with signer errors" is displayed.

jar−file The JAR file to be signed.

If you also specified the **`−strict`** option, and the **`jarsigner`** command detected severe warnings, the message, "jar signed, with signer errors" is displayed.

alias The aliases are defined in the keystore specified by **`−keystore`** or the default keystore.

DESCRIPTION

The **`jarsigner`** tool has two purposes:

- To sign Java Archive (JAR) files.
- 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 de velopers to produce JAR files. (Technically, any ZIP file can also be considered a JAR file, although when created by the **`jar`** command or processed by the **`jarsigner`** command, 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, and so on). Similar to a handwritten signature, a digital signature has many useful characteristics:

- Its authenticity can be verified by a computation that uses the public key corresponding to the private key used to generate the signature.
- It can't 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 can't be changed. If the data is changed, then the signature can't be verified as authentic.

To generate an entity's signature for a file, the entity must first have a public/private key pair associated with it and one or more certificates that authenticate its public key. A certificate is a digitally signed statement from one entity that says that the public key of another entity has a particular value.

The **`jarsigner`** command 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 that authenticate the corresponding public keys. The**`keytool`** command is used to create and administer keystores.

The **`jarsigner`** command 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. The **`jarsigner`** command can verify the digital signature of the signed JAR file using the certificate inside it (in its signature block file).

The **`jarsigner`** command can generate signatures that include a time stamp that enables a systems or deployer to check whether the JAR file was signed while the signing certificate was still valid.

In addition, APIs allow applications to obtain the timestamp information.

At this time, the **`jarsigner`** command can only sign JAR files created by the **`jar`** command or zip files. JAR files are the same as zip files, except they also have a **`META−INF/MANIFEST.MF`** file. A **`META−INF/MANIFEST.MF`** file is created when the **`jarsigner`** command signs a zip file.

The default **`jarsigner`** command behavior is to sign a JAR or zip file. Use the **`−verify`** option to verify a signed JAR file.

The **`jarsigner`** command also attempts to validate the signer's certificate after signing or verifying. During validation, it checks the revocation status of each certificate in the signer's certificate chain when the **`−revCheck`** option is specified. If there is a validation error or any other problem, the command generates warning messages. If you specify the **`−strict`** option, then the command treats severe warnings as errors. See**Err ors and Warnings**.

KEYSTORE ALIASES

All keystore entities are accessed with unique aliases.

When you use the **`jarsigner`** command to sign a JAR file, you must specify the alias for the keystore entry that contains the private key needed to generate the signature. If no output file is specified, it overwrites the original JAR file with the signed JAR file.

Keystores are protected with a password, so the store password must be specified. You are prompted for it when 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 must be specified, and you are prompted for the password when you don't specify it on the command line and it isn't the same as the store password.

KEYSTORE LOCATION

The **`jarsigner`** command 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.

Linux and OS X: `user.home` defaults to the user's home directory.

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 when the **`KeyStore`** class isn't file based, for example, when it resides on a hardware token device.

KEYSTORE IMPLEMENTATION

The **`KeyStore`** class provided in the **`java.security`** package supplies a number of well−defined interfaces to access and modify the information in a keystore. You can have multiple different concrete implementations, where each implementation is for a particular type of keystore.

Currently, there are two command−line tools that use keystore implementations (**`keytool`** and **`jarsigner`**).

The default keystore implementation is **`PKCS12`**. This is a cross platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This standard is primarily meant for storing or transporting a user's private keys, certificates, and miscellaneous secrets. There is another built−in implementation, provided by Oracle. It implements the keystore as a file with 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, which means the application interfaces supplied by the **`KeyStore`** class are implemented in terms of a Service Provider Interface (SPI). There is a corresponding abstract **`KeystoreSpi`** class, also in the **`java.security package`**, that 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. To provide a keystore implementation, clients must implement a provider and supply a **`KeystoreSpi`** subclass implementation, as described in **How to Implement a Provider in the Java Cryptography Architecture** [https://www.oracle.com/pls/topic/lookup?ctx=en/java/javase/11/tools&id=JSSEC−GUID−2BCFDD85−D533−4E6C−8CE9−29990DEB0190].

Applications can choose different types of keystore implementations from different providers, with the **getInstance** factory method 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 aren't compatible.

The **jarsigner** commands can read file−based keystores from any location that can be specified using a URL. In addition, these commands can read non−file−based keystores such as those provided by MSCAPI on Windows and PKCS11 on all platforms.

For the **jarsigner** and **keytool** commands, you can specify a keystore type at the command line with the **−storetype** option.

If you don't explicitly specify a keystore type, then the tools choose a keystore implementation based 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 JDK security properties directory, **java.home/conf/security**.

Each tool gets the **keystore.type** value and then examines all the 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 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.getDefault-
Type());
```

The default keystore type is **pkcs12**, which is a cross platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This is specified by the following line in the security properties file:

```
keystore.type=pkcs12
```

Case doesn't matter in keystore type designations. For example, **JKS** is the same as **jks**.

To have the tools utilize a keystore implementation other than the default, you can change that line to specify a different keystore type. For example, if you want to use the Oracle's **jks** keystore implementation, then change the line to the following:

```
keystore.type=jks
```

SUPPORTED ALGORITHMS

By default, the **jarsigner** command signs a JAR file using one of the following algorithms and block file extensions depending on the type and size of the private key:

keyalg	keysize	default sigalg	block file extension
DSA	any size	SHA256withDSA	.DSA
RSA	<= 3072	SHA256withRSA	.RSA
	<= 7680	SHA384withRSA	
	> 7680	SHA512withRSA	
EC	< 384	SHA256withECDSA	.EC
	< 512	SHA384withECDSA	
	= 512	SHA512withECDSA	
RSASSA−PSS	<= 3072	RSASSA−PSS (with SHA−256)	.RSA
	<= 7680	RSASSA−PSS (with SHA−384)	
	> 7680	RSASSA−PSS (with SHA−512)	
EdDSA	255	Ed25519	.EC

448	Ed448

- If an RSASSA−PSS key is encoded with parameters, then jarsigner will use the same parameters in the signature. Otherwise, jarsigner will use parameters that are determined by the size of the key as specified in the table above. For example, an 3072−bit RSASSA−PSS key will use RSASSA−PSS as the signature algorithm and SHA−256 as the hash and MGF1 algorithms.

These default signature algorithms can be overridden by using the **`−sigalg`** option.

The **`jarsigner`** command uses the **`jdk.jar.disabledAlgorithms`** and **`jdk.security.legacyAlgorithms`** security properties to determine which algorithms are considered a security risk. If the JAR file was signed with any algorithms that are disabled, it will be treated as an unsigned JAR file. If the JAR file was signed with any legacy algorithms, it will be treated as signed with an informational warning to inform users that the legacy algorithm will be disabled in a future update. For detailed verification output, include **`−J−Djava.security.debug=jar`**. The**`jdk.jar.disabledAlgorithms`** and **`jdk.security.legacyAlgorithms`** security properties are defined in the **`java.security`** file (located in the JDK's **`$JAVA_HOME/conf/security`** directory).

Note:

In order to improve out of the box security, default key size and signature algorithm names are periodically updated to stronger values with each release of the JDK. If interoperability with older releases of the JDK is important, please make sure the defaults are supported by those releases, or alternatively use the **`−sigalg`** option to override the default values at your own risk.

THE SIGNED JAR FILE

When the **`jarsigner`** command 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 an **`.SF`** extension
- A signature block file with a **`.DSA`**, **`.RSA`**, or **`.EC`** extension

The base file names for these two files come from the value of the **`−sigfile`** option. For example, when the option is **`−sigfile MKSIGN`**, the files are named **`MKSIGN.SF`** and **`MKSIGN.RSA`**. In this document, we assume the signer always uses an RSA key.

If no **`−sigfile`** option appears on the command line, then the base file name for the **`.SF`** and the signature block files is the first 8 characters of the alias name specified on the command line, all converted to uppercase. If the alias name has fewer than 8 characters, then the full alias name is used. If the alias name contains any characters that aren't allowed in a signature file name, then each such character is converted to an underscore (_) character in forming the file name. Valid characters include letters, digits, underscores, and hyphens.

SIGNATURE FILE

A signature file (**`.SF`** file) looks similar to the manifest file that is always included in a JAR file when the **`jarsigner`** command is used to sign the file. For each source file included in the JAR file, the **`.SF`** file has two lines, such as in the manifest file, that list the following:

- File name
- Name of the digest algorithm (SHA)
- SHA digest value

Note:

The name of the digest algorithm (SHA) and the SHA digest value are on the same line.

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, the digest value for a specified source file is the hash of the two lines in the manifest file for the source file.

The signature file, by default, includes a header with a hash of the whole manifest file. The header also contains a hash of the manifest header. The presence of the header enables verification optimization. See

JAR File Verification.

SIGNATURE BLOCK FILE

The **.SF** file is signed and the signature is placed in the signature block file. This file also contains, encoded inside it, the certificate or certificate chain from the keystore that authenticates the public key corresponding to the private key used for signing. The file has the extension **.DSA**, **.RSA**, or **.EC**, depending on the key algorithm used. See the table in **Supported Algorithms**.

SIGNATURE TIME STAMP

The **jarsigner** command used with the following options generates and stores a signature time stamp when signing a JAR file:

- **−tsa** *url*
- **−tsacert** *alias*
- **−tsapolicyid** *policyid*
- **−tsadigestalg** *algorithm*

See **Options for jarsigner**.

JAR FILE VERIFICATION

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

1. Verify the signature of the **.SF** file.

   The verification ensures that the signature stored in each signature block file was generated using the private key corresponding to the public key whose certificate (or certificate chain) also appears in the signature block file. It also ensures that the signature is a valid signature of the corresponding signature (**.SF**) file, and thus the **.SF** file wasn't 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 that contains a hash of the entire manifest file. When the header is present, the verification can check to see whether or not the hash in the header matches the hash of the manifest file. If there is a match, then verification proceeds to the next step.

   If there is no match, then 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 Signature File.

   One reason the hash of the manifest file that is stored in the **.SF** file header might not equal the hash of the current manifest file is that it might contain sections for newly added files after the file was signed. For example, suppose one or more files were added to the signed JAR file (using the **jar** tool) that already contains a signature and a **.SF** file. If the JAR file is signed again by a different signer, then the manifest file is changed (sections are added to it for the new files by the **jarsigner** tool) and a new **.SF** file is created, but the original **.SF** file is unchanged. A verification is still considered successful if none of the files that were in the JAR file when the original signature was generated have been changed since then. This is because 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 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, then the process is stopped and a security exception is thrown. The**jarsigner** command catches and displays the e xception.

4. Check for disabled algorithm usage. See **Supported Algorithms**.

Note:

You should read any addition warnings (or errors if you specified the **−strict** option), as well as the content of the certificate (by specifying the **−verbose** and **−certs** options) to determine if the signature can be trusted.

MULTIPLE SIGNATURES FOR A JAR FILE

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

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

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

```
SUSAN.SF
SUSAN.RSA
KEVIN.SF
KEVIN.RSA
```

OPTIONS FOR JARSIGNER

The following sections describe the options for the **jarsigner**. Be aware of the following standards:

- All option names are preceded by a hyphen sign (−).
- The options can be provided in any order.
- Items that are in italics or underlined (option values) represent the actual values that must be supplied.
- The **−storepass**, **−keypass**, **−sigfile**, **−sigalg**, **−digestalg**, **−signedjar**, and TSA−related options are only relevant when signing a JAR file; they aren't relevant when verifying a signed JAR file. The**−keystore** option is rele vant for signing and verifying a JAR file. In addition, aliases are specified when signing and verifying 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. You must explicitly specify a keystore when the default keystore doesn't exist or if you want to use one other than the default.

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

The **−keystore** argument can be a file name and path specification rather than a URL, in which case it is treated the same as a file: URL, for example, the following are equivalent:

- **−keystore** *filePathAndName*
- **−keystore file:***filePathAndName*

If the Sun PKCS #11 provider was configured in the **java.security** security properties file (located in the JDK's **$JAVA_HOME/conf/security** directory), then the **keytool** and **jarsigner** tools can operate on the PKCS #11 token by specifying these options:

```
−keystore NONE −storetype PKCS11
```

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

```
keytool −keystore NONE −storetype PKCS11 −list
```

−storepass [**:env** | **:file**] *argument*

Specifies the password that is required to access the keystore. This is only needed when signing (not verifying) a JAR file. In that case, if a **−storepass** option isn't provided at the command line, then the user is prompted for the password.

If the modifier **env** or **file** isn't specified, then the password has the value **argument**. Otherwise, the password is retrieved as follows:

- **`env`**: Retrieve the password from the environment variable named *argument*.
- **`file`**: Retrieve the password from the file named *argument*.

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.

`-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 PKCS #11 token can also be specified with the **`-storepass`** option. If none is specified, then the **`keytool`** and **`jarsigner`** commands 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.

`-keypass` [**`:env`** | **`:file`**] *argument* **`-certchain`** *file*
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, then the user is prompted for it.

If the modifier **`env`** or **`file`** isn't specified, then the password has the value **`argument`**. Otherwise, the password is retrieved as follows:

- **`env`**: Retrieve the password from the environment variable named *argument*.
- **`file`**: Retrieve the password from the file named *argument*.

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.

`-certchain` *file*
Specifies the certificate chain to be used when the certificate chain associated with the private key of the keystore entry that is addressed by the alias specified on the command line isn't complete. This can happen when the keystore is located on a hardware token where there isn't enough capacity to hold a complete certificate chain. The file can be a sequence of concatenated X.509 certificates, or a single PKCS#7 formatted data block, either in binary encoding format or in printable encoding format (also known as Base64 encoding) as defined by **Internet RFC 1421 Certificate Encoding Standard** [http://tools.ietf.org/html/rfc1421].

`-sigfile` *file*
Specifies the base file name to be used for the generated **`.SF`** and signature block files. For example, if file is **`DUKESIGN`**, then the generated **`.SF`** and signature block files are named **`DUKESIGN.SF`** and **`DUKESIGN.RSA`**, and placed in the **`META-INF`** directory of the signed JAR file.

The characters in the file must come from the set **`a-zA-Z0-9_-`**. Only letters, numbers, underscore, and hyphen characters are allowed. All lowercase characters are converted to uppercase for the **`.SF`** and signature block file names.

If no **`-sigfile`** option appears on the command line, then the base file name for the **`.SF`** and signature block files is 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, then the full alias name is used. If the alias name contains any characters that aren't valid in a signature file name, then each such character is converted to an underscore (_) character to form the file name.

`-signedjar` *file*
: Specifies the name of signed JAR file.

`-digestalg` *algorithm*
: Specifies the name of the message digest algorithm to use when digesting the entries of a JAR file.

For a list of standard message digest algorithm names, see Java Security Standard Algorithm Names.

If this option isn't specified, then **`SHA256`** is used. There must either be a statically installed provider supplying an implementation of the specified algorithm or the user must specify one with the **`-addprovider`** or **`-providerClass`** options; otherwise, the command will not succeed.

`-sigalg` *algorithm*
: Specifies the name of the signature algorithm to use to sign the JAR file.

This algorithm must be compatible with the private key used to sign the JAR file. If this option isn't specified, then use a default algorithm matching the private key as described in the **Supported Algorithms** section. There must either be a statically installed provider supplying an implementation of the specified algorithm or you must specify one with the **`-addprovider`** or **`-providerClass`** option; otherwise, the command doesn't succeed.

For a list of standard message digest algorithm names, see Java Security Standard Algorithm Names.

`-verify`
: Verifies a signed JAR file.

`-verbose`[**`:`***suboptions*]
: When the **`-verbose`** option appears on the command line, it indicates that the **`jarsigner`** use the verbose mode when signing or verifying with the suboptions determining how much information is shown. This causes the , which causes**`jarsigner`** to output e xtra information about the progress of the JAR signing or verification. The*suboptions* can be **`all`**, **`grouped`**, or **`summary`**.

If the **`-certs`** option is also specified, then the default mode (or suboption **`all`**) displays each entry as it is being processed, and after that, the certificate information for each signer of the JAR file.

If the **`-certs`** and the **`-verbose:grouped`** suboptions are specified, then entries with the same signer info are grouped and displayed together with their certificate information.

If **`-certs`** and the **`-verbose:summary`** suboptions are specified, then entries with the same signer information are grouped and displayed together with their certificate information.

Details about each entry are summarized and displayed as *one entry (and more)*. See**Example of V erifying a Signed JAR File** and **Example of Verification with Certificate Information**.

`-certs`
: If the **`-certs`** option appears on the command line with the **`-verify`** and **`-verbose`** options, then 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 signature block file) that certifies the signer's public key, and if the certificate is an X.509 certificate (an instance of the **`java.security.cert.X509Certificate`**), then the distinguished name of the signer.

The keystore is also examined. If no keystore value is specified on the command line, then the default keystore file (if any) is checked. If the public key certificate for a signer matches an entry in the keystore, then the alias name for the keystore entry for that signer is displayed in parentheses.

`-revCheck`
: This option enables revocation checking of certificates when signing or verifying a JAR file. The **`jarsigner`** command attempts to make network connections to fetch OCSP responses and CRLs if the **`-revCheck`** option is specified on the command line. Note that revocation checks are not enabled unless this option is specified.

`-tsa` *url*
: If **`-tsa http://example.tsa.url`** appears on the command line when signing a JAR file then a time stamp is generated for the signature. The URL, **`http://example.tsa.url`**, identifies the lo-

cation of the Time Stamping Authority (TSA) and overrides any URL found with the **`-tsacert`** option. The **`-tsa`** option doesn't require the TSA public key certificate to be present in the keystore.

To generate the time stamp, **`jarsigner`** communicates with the TSA with the Time−Stamp Protocol (TSP) defined in RFC 3161. When successful, the time stamp token returned by the TSA is stored with the signature in the signature block file.

`-tsacert` *alias*
When **`-tsacert`** *alias* appears on the command line when signing a JAR file, a time stamp is generated for the signature. The alias identifies the TSA public key certificate in the keystore that is 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 public key certificate must be present in the keystore when using the **`-tsacert`** option.

`-tsapolicyid` *policyid*
Specifies the object identifier (OID) that identifies the policy ID to be sent to the TSA server. If this option isn't specified, no policy ID is sent and the TSA server will choose a default policy ID.

Object identifiers are defined by X.696, which is an ITU Telecommunication Standardization Sector (ITU−T) standard. These identifiers are typically period−separated sets of non−negative digits like **`1.2.3.4`**, for example.

`-tsadigestalg` *algorithm*
Specifies the message digest algorithm that is used to generate the message imprint to be sent to the TSA server. If this option isn't specified, SHA−256 will be used.

See **Supported Algorithms**.

For a list of standard message digest algorithm names, see Java Security Standard Algorithm Names.

`-internalsf`
In the past, the signature block file generated when a JAR file was signed included 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 signature block file by default doesn't contain a copy of the **`.SF`** file anymore. If**`-internalsf`** appears on the command line, then the old beha vior is utilized. This option is useful for testing. In practice, don't use the **`-internalsf`** option because it incurs higher overhead.

`-sectionsonly`
If the **`-sectionsonly`** option appears on the command line, then the **`.SF`** file (signature file) generated when a JAR file is signed doesn't include a header that contains a hash of the whole manifest file. It contains only the information and hashes related to each individual source file included in the JAR file. See Signature File.

By default, this header is added, as an optimization. When the header is present, whenever the JAR file is verified, the verification can first check to see whether the hash in the header matches the hash of the whole manifest file. When there is a match, verification proceeds to the next step. When there is no match, 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. See **JAR File Verification**.

The **`-sectionsonly`** option is primarily used for testing. It shouldn't be used other than for testing because using it incurs higher overhead.

`-protected`
Values can be either **`true`** or **`false`**. Specify**`true`** when a passw ord must be specified through a protected authentication path such as a dedicated PIN reader.

`-providerName` *providerName*
If more than one provider was configured in the **`java.security`** security properties file, then 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 Oracle 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 −provider-
Name SunPKCS11−SmartCard −list
```

−addprovider *name* [**−providerArg** *arg*]

Adds a security provider by name (such as SunPKCS11) and an optional configure argument. The value of the security provider is the name of a security provider that is defined in a module.

Used with the **−providerArg ConfigFilePath** option, the **keytool** and **jarsigner** tools install the provider dynamically and use **ConfigFilePath** for the path to the token configuration file. The following example shows a command to list a **PKCS #11** keystore when the Oracle PKCS #11 provider wasn't configured in the security properties file.

```
jarsigner −keystore NONE −storetype PKCS11 −ad-
dprovider SunPKCS11 −providerArg /mydir1/mydir2/token.config
```

−providerClass *provider−class−name* [**−providerArg** *arg*]

Used to specify the name of cryptographic service provider's master class file when the service provider isn't listed in the **java.security** security properties file. Adds a security provider by fully−qualified class name and an optional configure argument.

Note:

The preferred way to load PKCS11 is by using modules. See **−addprovider**.

−J*javaoption*

Passes through the specified *javaoption* string directly to the Java interpreter. The**jarsigner** command is a wrapper around the interpreter. This option shouldn't 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.

−strict

During the signing or verifying process, the command may issue warning messages. If you specify this option, the exit code of the tool reflects the severe warning messages that this command found. See **Errors and Warnings**.

−conf *url*

Specifies a pre−configured options file. Read the **keytool documentation** for details. The property keys supported are "jarsigner.all" for all actions, "jarsigner.sign" for signing, and "jarsigner.verify" for verification. **jarsigner** arguments including the JAR file name and alias name(s) cannot be set in this file.

DEPRECATED OPTIONS

The following **jarsigner** options are deprecated as of JDK 9 and might be removed in a future JDK release.

−altsigner *class*

This option specifies an alternative signing mechanism. 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, then the **jarsigner** command uses the signing mechanism provided by the specified class. Otherwise, the **jarsigner** command 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 and any JAR file it depends on. The class file name is specified with the **−altsigner** option. If the class file is in a JAR file, then this option specifies the path to that JAR file.

An absolute path or a path relative to the current directory can be specified. If *classpathlist* contains multiple paths or JAR files, then they should be separated with a:

- Colon (`:`) on Linux and macOS
- Semicolon (`;`) on Windows

This option isn't necessary when the class is already in the search path.

The following example shows how to specify the path to a JAR file that contains the class file. The JAR file name is included.

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

The following example shows how to specify the path to the JAR file that contains the class file. The JAR file name is omitted.

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

ERRORS AND WARNINGS

During the signing or verifying process, the **`jarsigner`** command may issue various errors or warnings.

If there is a failure, the **`jarsigner`** command exits with code 1. If there is no failure, but there are one or more severe warnings, the **`jarsigner`** command exits with code 0 when the **`−strict`** option is **not** specified, or exits with the OR−value of the warning codes when the **`−strict`** is specified. If there is only informational warnings or no warning at all, the command always exits with code 0.

For example, if a certificate used to sign an entry is expired and has a KeyUsage extension that doesn't allow it to sign a file, the **`jarsigner`** command exits with code 12 (=4+8) when the **`−strict`** option is specified.

Note: Exit codes are reused because only the values from 0 to 255 are legal on Linux and OS X.

The following sections describes the names, codes, and descriptions of the errors and warnings that the **`jarsigner`** command can issue.

FAILURE

Reasons why the **`jarsigner`** command fails include (but aren't limited to) a command line parsing error, the inability to find a keypair to sign the JAR file, or the verification of a signed JAR fails.

failure Code 1. The signing or verifying fails.

SEVERE WARNINGS

Note:

Severe warnings are reported as errors if you specify the **`−strict`** option.

Reasons why the **`jarsigner`** command issues a severe warning include the certificate used to sign the JAR file has an error or the signed JAR file has other problems.

hasExpiredCert
Code 4. This JAR contains entries whose signer certificate has expired.

hasExpiredTsaCert
Code 4. The timestamp has expired.

notYetValidCert
Code 4. This JAR contains entries whose signer certificate isn't yet valid.

chainNotValidated
Code 4. This JAR contains entries whose certificate chain isn't validated.

tsaChainNotValidated
Code 64. The timestamp is invalid.

signerSelfSigned
: Code 4. This JAR contains entries whose signer certificate is self signed.

disabledAlg
: Code 4. An algorithm used is considered a security risk and is disabled.

badKeyUsage
: Code 8. This JAR contains entries whose signer certificate's KeyUsage extension doesn't allow code signing.

badExtendedKeyUsage
: Code 8. This JAR contains entries whose signer certificate's ExtendedKeyUsage extension doesn't allow code signing.

badNetscapeCertType
: Code 8. This JAR contains entries whose signer certificate's NetscapeCertType extension doesn't allow code signing.

hasUnsignedEntry
: Code 16. This JAR contains unsigned entries which haven't been integrity−checked.

notSignedByAlias
: Code 32. This JAR contains signed entries which aren't signed by the specified alias(es).

aliasNotInStore
: Code 32. This JAR contains signed entries that aren't signed by alias in this keystore.

tsaChainNotValidated
: Code 64. This JAR contains entries whose TSA certificate chain is invalid.

INFORMATIONAL WARNINGS

Informational warnings include those that aren't errors but regarded as bad practice. They don't have a code.

extraAttributesDetected
: The POSIX file permissions and/or symlink attributes are detected during signing or verifying a JAR file. The **`jarsigner`** tool preserves these attributes in the newly signed file but warns that these attributes are unsigned and not protected by the signature.

hasExpiringCert
: This JAR contains entries whose signer certificate expires within six months.

hasExpiringTsaCert
: The timestamp will expire within one year on **`YYYY−MM−DD`**.

legacyAlg
: An algorithm used is considered a security risk but not disabled.

noTimestamp
: This JAR contains signatures that doesn't include a timestamp. Without a timestamp, users may not be able to validate this JAR file after the signer certificate's expiration date (**`YYYY−MM−DD`**) or after any future revocation date.

EXAMPLE OF SIGNING A JAR FILE

Use the following command to sign **`bundle.jar`** with the private key of a user whose keystore alias is **`jane`** in a keystore named **`mystore`** in the **`working`** directory and name the signed JAR file **`sbundle.jar`**:

```
jarsigner −keystore /working/mystore −storepass keystore_password −keypass private_key_password −signedjar sbundle.jar bundle.jar jane
```

There is no **`−sigfile`** specified in the previous command so the generated **`.SF`** and signature block files to be placed in the signed JAR file have default names based on the alias name. They are named **`JANE.SF`** and **`JANE.RSA`**.

If you want to be prompted for the store password and the private key password, then you could shorten the previous command to the following:

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

If the **keystore** is the default **keystore** (**.keystore** in your home directory), then you don't need to specify a **keystore**, as follows:

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

If you want the signed JAR file to overwrite the input JAR file (**bundle.jar**), then you don't need to specify a **-signedjar** option, as follows:

```
jarsigner bundle.jar jane
```

EXAMPLE OF VERIFYING A SIGNED JAR FILE

To verify a signed JAR file to ensure that the signature is valid and the JAR file wasn't been tampered with, use a command such as the following:

```
jarsigner -verify ButtonDemo.jar
```

When the verification is successful, **jar verified** is displayed. Otherwise, an error message is displayed. You can get more information when you use the **-verbose** option. A sample use of **jarsigner** with the **-verbose** option follows:

```
jarsigner -verify -verbose ButtonDemo.jar

s       866 Tue Sep 12 20:08:48 EDT 2017 META-INF/MANIFEST.MF
        825 Tue Sep 12 20:08:48 EDT 2017 META-INF/ORACLE_C.SF
       7475 Tue Sep 12 20:08:48 EDT 2017 META-INF/ORACLE_C.RSA
          0 Tue Sep 12 20:07:54 EDT 2017 META-INF/
          0 Tue Sep 12 20:07:16 EDT 2017 components/
          0 Tue Sep 12 20:07:16 EDT 2017 components/images/
sm      523 Tue Sep 12 20:07:16 EDT 2017 components/ButtonDemo$1.class
sm     3440 Tue Sep 12 20:07:16 EDT 2017 components/ButtonDemo.class
sm     2346 Tue Sep 12 20:07:16 EDT 2017 components/ButtonDemo.jnlp
sm      172 Tue Sep 12 20:07:16 EDT 2017 components/images/left.gif
sm      235 Tue Sep 12 20:07:16 EDT 2017 components/images/middle.gif
sm      172 Tue Sep 12 20:07:16 EDT 2017 components/images/right.gif

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

- Signed by "CN="Oracle America, Inc.", OU=Software Engineering, O="Oracle
    Digest algorithm: SHA-256
    Signature algorithm: SHA256withRSA, 2048-bit key
  Timestamped by "CN=Symantec Time Stamping Services Signer - G4, O=Symante
    Timestamp digest algorithm: SHA-1
    Timestamp signature algorithm: SHA1withRSA, 2048-bit key

jar verified.

The signer certificate expired on 2018-02-01. However, the JAR will be vali
```

EXAMPLE OF VERIFICATION WITH CERTIFICATE INFORMATION

If you specify the **-certs** option with the **-verify** and **-verbose** options, then the output includes certificate information for each signer of the JAR file. The information includes the certificate type, the signer distinguished name information (when it is an X.509 certificate), and in parentheses, the keystore alias for the signer when the public key certificate in the JAR file matches the one in a keystore entry, for example:

```
jarsigner −keystore $JAVA_HOME/lib/security/cacerts −verify −verbose −certs

s k     866 Tue Sep 12 20:08:48 EDT 2017 META−INF/MANIFEST.MF

      >>> Signer
      X.509, CN="Oracle America, Inc.", OU=Software Engineering, O="Oracle 
      [certificate is valid from 2017−01−30, 7:00 PM to 2018−02−01, 6:59 PM
      X.509, CN=Symantec Class 3 SHA256 Code Signing CA, OU=Symantec Trust 
      [certificate is valid from 2013−12−09, 7:00 PM to 2023−12−09, 6:59 PM
      X.509, CN=VeriSign Class 3 Public Primary Certification Authority − G
      [trusted certificate]
      >>> TSA
      X.509, CN=Symantec Time Stamping Services Signer − G4, O=Symantec Cor
      [certificate is valid from 2012−10−17, 8:00 PM to 2020−12−29, 6:59 PM
      X.509, CN=Symantec Time Stamping Services CA − G2, O=Symantec Corpora
      [certificate is valid from 2012−12−20, 7:00 PM to 2020−12−30, 6:59 PM

        825 Tue Sep 12 20:08:48 EDT 2017 META−INF/ORACLE_C.SF
       7475 Tue Sep 12 20:08:48 EDT 2017 META−INF/ORACLE_C.RSA
          0 Tue Sep 12 20:07:54 EDT 2017 META−INF/
          0 Tue Sep 12 20:07:16 EDT 2017 components/
          0 Tue Sep 12 20:07:16 EDT 2017 components/images/
smk     523 Tue Sep 12 20:07:16 EDT 2017 components/ButtonDemo$1.class

      [entry was signed on 2017−09−12, 4:08 PM]
      >>> Signer
      X.509, CN="Oracle America, Inc.", OU=Software Engineering, O="Oracle 
      [certificate is valid from 2017−01−30, 7:00 PM to 2018−02−01, 6:59 PM
      X.509, CN=Symantec Class 3 SHA256 Code Signing CA, OU=Symantec Trust 
      [certificate is valid from 2013−12−09, 7:00 PM to 2023−12−09, 6:59 PM
      X.509, CN=VeriSign Class 3 Public Primary Certification Authority − G
      [trusted certificate]
      >>> TSA
      X.509, CN=Symantec Time Stamping Services Signer − G4, O=Symantec Cor
      [certificate is valid from 2012−10−17, 8:00 PM to 2020−12−29, 6:59 PM
      X.509, CN=Symantec Time Stamping Services CA − G2, O=Symantec Corpora
      [certificate is valid from 2012−12−20, 7:00 PM to 2020−12−30, 6:59 PM

smk    3440 Tue Sep 12 20:07:16 EDT 2017 components/ButtonDemo.class
...
smk    2346 Tue Sep 12 20:07:16 EDT 2017 components/ButtonDemo.jnlp
...
smk     172 Tue Sep 12 20:07:16 EDT 2017 components/images/left.gif
...
smk     235 Tue Sep 12 20:07:16 EDT 2017 components/images/middle.gif
...
smk     172 Tue Sep 12 20:07:16 EDT 2017 components/images/right.gif
...

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

```
- Signed by "CN="Oracle America, Inc.", OU=Software Engineering, O="Oracle A
    Digest algorithm: SHA-256
    Signature algorithm: SHA256withRSA, 2048-bit key
  Timestamped by "CN=Symantec Time Stamping Services Signer - G4, O=Symantec
    Timestamp digest algorithm: SHA-1
    Timestamp signature algorithm: SHA1withRSA, 2048-bit key

jar verified.

The signer certificate expired on 2018-02-01. However, the JAR will be vali
```

If the certificate for a signer isn't an X.509 certificate, then 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**, then you would get: **PGP, (bob)**.