Documentation

The Java™ Tutorials

Trail: Security Features in Java SE

Lesson: API and Tools Use for Secure Code and File Exchanges

This lesson explains why digital signatures, certificates, and keystores are needed. The lesson also compares use of the tools versus the JDK Security API with respect to generating signatures. Such tool usage is demonstrated in the next two lessons, Signing Code and Granting It Permissions and Exchanging Files. API usage is demonstrated in the Generating and Verifying Signatures lesson.

This lesson contains the following sections

Code and Document Security

If you electronically send someone an important document (or documents), or an applet or application to run, the recipient needs a way to verify that the document or code came from you and was not modified in transit (for example, by a malicious user intercepting it). Digital signatures, certificates, and keystores all help ensure the security of the files you send.

Digital Signatures

The basic idea in the use of digital signatures is as follows.

1. You "sign" the document or code using one of your **private keys**, which you can generate by using `keytool` or security API methods. That is, you generate a digital signature for the document or code, using the `jarsigner` tool or Security API methods.
2. You send your signed document to your recipient.
3. You also supply your recipient with your **public key.** This public key corresponds to the private key you originally used to generate the signature.
4. Your recipient uses your public key to verify that your document came from you and was not modified before it reached him/her.

A recipient needs to ensure that your public key *itself* is authentic before he/she can use it to verify that your signature is authentic. Therefore, you will usually supply a **certificate** that contains your public key together with the key of a **Certificate Authority** who can vouch for your key's authenticity. See the next section for details.

For more information about the terminology and concepts of signing and verification, and further explanation of the benefits, see the Signing JAR Files section of the "The Packaging Programs in JAR Files" lesson.

Certificates

A certificate contains:

- A public key.
- The "distinguished-name" information of the entity (person, company, or so on) whose certificate it is. This entity is referred to as the certificate *subject*, or *owner*. The distinguished-name information includes the following attributes (or a subset): the entity's name, organizational unit, organization, city or locality, state or province, and country code.
- A digital signature. A certificate is signed by one entity, the *issuer*, to vouch for the fact that the enclosed public key is the actual public key of another entity, the *owner*.
- The distinguished-name information for the signer (issuer).

One way for a recipient to check whether a certificate is valid is by verifying its digital signature, using its issuer's (signer's) public key. That key can itself be stored within another certificate whose signature can also be verified by using the public key of that next certificate's issuer, and that key may *also* be stored in yet *another* certificate, and so on. You can stop checking when you reach a public key that you already trust and use it to verify the signature on the corresponding certificate.

If the recipient cannot establish a trust chain, then he/she can calculate the the certificate **fingerprint(s)**, using the `keytool -import` or `-printcert` command. A fingerprint is a relatively short number that uniquely and reliably identifies the certificate. (Technically, the fingerprint is a hash value of the certificate information, using a message digest function.) The recipient can then phone the certificate owner and compare the fingerprint values of the received certificate with the certificate that was sent. If the fingerprints are the same, the certificates are the same.

Thus you can ensure that a certificate was not modified in transit. One other potential uncertainty when working with certificates is the identity of the sender. Sometimes a certificate is **self-signed**, that is, signed using the private key corresponding to the public key in the certificate; the issuer is the same as the subject.

Self-signing a certificate is useful for developing and testing an application. However, before deploying to users, obtain a certificate from a trusted third party, referred to as a certification authority (CA). To do so, you send a self-signed certificate signing request (CSR) to the CA. The CA verifies the signature on the CSR and your identity, perhaps by checking your driver's license or other information. The CA then vouches for your being the owner of the public key by issuing a certificate and signing it with its own (the CA's) private key. Anybody who trusts the issuing CA's public key can now verify the signature on the certificate. In many cases the issuing CA itself may have a certificate from a CA higher up in the CA hierarchy, leading to **certificate chains**.

Certificates of entities you trust are typically imported into your keystore as "**trusted certificates**." The public key in each such certificate may then be used to verify signatures generated using the corresponding private key. Such verifications can be accomplished by:

- the `jarsigner` tool (if the document/code and signature appear in a JAR file),
- API methods, or
- the runtime system, when a resource access is attempted and a policy file specifies that the resource access is allowed for the code attempting the access if its signature is authentic. The code's class file(s) and signature must be in a JAR file.

If you are sending signed code or documents to others, you need to supply them with the certificate containing the public key corresponding to the private key used to sign the code/document. The `keytool -export` command or API methods can export your certificate from your keystore to a file, which can then be sent to anyone needing it. A person who receives the certificate can import it into the keystore as a trusted certificate, using, for example, API methods or the `keytool -import` command.

If you use the `jarsigner` tool to generate a signature for a JAR file, the tool retrieves your certificate and its supporting certificate chain from your keystore. The tool then stores them, along with the signature, in the JAR file.

Keystores

Private keys and their associated public key certificates are stored in password-protected databases called **keystores**. A keystore can contain two types of entries: the trusted certificate entries discussed above, and key/certificate entries, each containing a private key and the corresponding public key certificate. Each entry in a keystore is identified by an *alias*.

A keystore owner can have multiple keys in the keystore, accessed via different aliases. An alias is typically named after a particular role in which the keystore owner uses the associated key. An alias may also identify the purpose of the key. For example, the alias `signPersonalEmail` might be used to identify a keystore entry whose private key is used for signing personal e-mail, and the alias `signJarFiles` might be used to identify an entry whose private key is used for signing JAR files.

The `keytool` tool can be used to

- Create private keys and their associated public key certificates
- Issue certificate requests, which you send to the appropriate certification authority
- Import certificate replies, obtained from the certification authority you contacted
- Import public key certificates belonging to other parties as trusted certificates
- Manage your keystore

API methods can also be used to access and to modify a keystore.

Tool and API Notes

Note the following regarding use of the tools and the API related to digital signatures.

- You can use the JDK Security API, tools, or a combination to generate keys and signatures and to import certificates. You can use these API or tool features to securely exchange documents with others.
- To use the *tools* for document exchange, the document(s) must be placed in a JAR (Java ARchive) file, which may be created by the `jar` tool. A JAR file is a good way of encapsulating multiple files in one spot. When a file is "signed", the resulting digital signature bytes need to be stored somewhere. When a JAR file is signed, the signature can go in the JAR file itself. This is what happens when you use the `jarsigner` tool to sign a JAR file.
- If you are creating applet code that you will sign, it needs to be placed in a JAR file. The same is true if you are creating application code that may be similarly restricted by running it with a security manager. The reason you need the JAR file is that when a policy file specifies that code signed by a particular entity is permitted one or more operations, such as specific file reads or writes, the code is expected to come from a signed JAR file. (The term "signed code" is an abbreviated way of saying "code in a class file that appears in a JAR file that was signed.")
- In order for the runtime system to check a code signature, the person/organization that will run the code first needs to import into their keystore a certificate authenticating the public key corresponding to the private key used to sign the code.
- In order for the `jarsigner` tool to verify the authenticity of a JAR file signature, the person/organization that received the JAR file first needs to import into their keystore a certificate authenticating the public key corresponding to the private key used to sign the code.
- At this time there are no APIs for certificate creation.

Use of the JDK Security API to Sign Documents

The Generating and Verifying Signatures shows you how to use the JDK Security API to sign documents. The lesson shows what one program, executed by the person who has the original document, would do to

- generate keys,
- generate a digital signature for the data using the private key, and
- export the public key and the signature to files.

Then it shows an example of another program, executed by the receiver of the data, signature, and public key. It shows how the program

- Imports the public key
- Verifies the authenticity of the signature.

This lesson also shows you alternative ways to import and supply keys, including certificates.

Use of the Tools to Sign Code or Documents

The Signing Code and Granting It Permissions lesson shows how to use Java Security tools to place your code into a JAR file, sign it, and export your public key. Then it shows how your recipient can use these same Java tools to import your public key certificate and then add an entry to a policy file that will grant your code the permission it needs to access system resources controlled by your recipient.

The Exchanging Files lesson you how to use Java security tools to sign a document and then export the public key certificate for the public key using `keytool`. corresponding to the private key used to sign that document using `keytool`. Then it shows how your recipient can verify your signature by installing your public key certificate and then using the `jarsigner` tool to verify your signature.

These two lessons have much in common. In both cases, the first two steps for the code or document sender are to:

- Create a JAR file containing the document or class file, using the `jar` tool.
- Generate keys (if they don't already exist), using the `keytool -genkey` command.

The next two steps are optional:

- Use the `keytool -certreq` command; then send the resulting certificate signing request to a certification authority (CA) such as VeriSign.
- Use the `keytool -import` command to import the CA's response.

The next two steps are required:

- Sign the JAR file, using the `jarsigner` tool and the private key generated earlier.
- Export the public key certificate, using the `keytool -export` command. Then supply the signed JAR file and the certificate to the receiver.

In both cases, the receiver of the signed JAR file and the certificate should import the certificate as a trusted certificate, using the `keytool -import` command. The `keytool` will attempt to construct a trust chain from the certificate to be imported to an already trusted certificate in the keystore. If that fails, the `keytool` will display the certificate fingerprint and prompt you to verify it.

If what was sent was code, the receiver also needs to modify a policy file to permit the required resource accesses to code signed by the private key corresponding to the public key in the imported certificate. The **Policy Tool** can be used to do this.

If what was sent was one or more documents, the receiver needs to verify the authenticity of the JAR file signature, using the `jarsigner` tool.

This lesson discusses the two optional steps. The other steps are covered in the next two lessons, Signing Code and Granting It Permissions and Exchanging Files .

Generating a Certificate Signing Request (CSR) for a Public Key Certificate

When `keytool` is used to generate public/private key pairs, it creates a keystore entry containing a private key and a self-signed certificate for the public key. (That is, the certificate is signed using the corresponding private key.) This is adequate when developing and testing an application.

However, a certificate is more likely to be trusted by others if it is signed by a certification authority (CA). To get a certificate signed by a CA, you first generate a certificate signing request (CSR), via a command such as the following:

```
keytool -certreq -alias alias -file csrFile
```

Here *alias* is used to access the keystore entry containing the private key and the public key certificate, and *csrFile* specifies the name to be used for the CSR created by this command.

You then submit this file to a CA, such as VeriSign, Inc. The CA authenticates you, the requestor ("subject"), and then signs and returns a certificate authenticating your public key. By signing the certificate, the CA vouches that you are the owner of the public key.

In some cases, the CA will return a chain of certificates, each one authenticating the public key of the signer of the previous certificate in the chain.

Importing the Response from the CA

After submitting a certificate signing request (CSR) to a certification authority (CA), you need to replace the original self-signed certificate in your keystore with a certificate chain by importing the certificate (or chain of certificates) returned to you by the CA.

But first you need a "trusted certificate" entry in your keystore (or in the `cacerts` keystore file, described below) that authenticates the *CA*'s public key. With such an entry the CA's signature can be verified. That is, the CA's signature on the certificate, or on the final certificate in the chain the CA sends to you in response to your CSR, can be verified.

Importing a Certificate from a CA as a "Trusted Certificate"

Before you import the certificate reply from a CA, you need one or more "trusted certificates" in your keystore or in the `cacerts` file.

- If the certificate reply is a certificate chain, you just need the top certificate of the chain -- the "root" CA certificate authenticating that CA's public key.
- If the certificate reply is a single certificate, you need a certificate for the issuing CA (the one that signed it). If that certificate is not self-signed, you need a certificate for its signer, and so on, up to a self-signed "root" CA certificate.

The `cacerts` file represents a system-wide keystore with CA certificates. This file resides in the JRE security properties directory, *java.home*`/lib/security`, where *java.home* is the JRE installation directory.

IMPORTANT: Verify Your `cacerts` File
Since you trust the CAs in the `cacerts` file as entities for signing and issuing certificates to other entities, you must manage the `cacerts` file carefully. The `cacerts` file should contain only certificates of the CAs you trust. It is your responsibility to verify the trusted root CA certificates bundled in the `cacerts` file and make your own trust decisions. To remove an untrusted CA certificate from the `cacerts` file, use the delete option of the `keytool` command. You can find the `cacerts` file in the JRE installation directory. Contact your system administrator if you do not have permission to edit this file.

The `cacerts` file contains a number of trusted CA certificates. If you sent your CSR to one of these trusted vendors (such as VeriSign), you won't need to import the vendor's root certificate as a trusted certificate in your keystore; you can go on to the next section to see how to import the certificate reply from the CA.

A certificate from a CA is usually either self-signed or signed by another CA, in which case you also need a certificate authenticating that CA's public key. Suppose that company ABC, Inc., is a CA and that you obtain a file named `ABCCA.cer`, purportedly a self-signed certificate from ABC, authenticating that CA's public key.

Be very careful to ensure that the certificate is valid prior to importing it as a "trusted" certificate! View it first (using the `keytool -printcert` command or the `keytool -import` command without the `-noprompt` option), and make sure that the displayed certificate fingerprint(s) match the expected ones. You can call the person who sent the certificate and compare the fingerprint(s) that you see with the ones that they show or that a secure public key repository shows. Only if the fingerprints are equal is it guaranteed that the certificate has not been replaced in transit with somebody else's (for example, an attacker's) certificate. If such an attack took place and you did not check the certificate before you imported it, you would end up trusting anything the attacker has signed.

If you trust that the certificate is valid, you can add it to your keystore via a command such as the following:

```
keytool -import -alias alias -file ABCCA.cer -keystore storefile
```

This command creates a "trusted certificate" entry in the keystore whose name is that specified in *storefile*. The entry contains the data from the file `ABCCA.cer`, and it is assigned the specified alias.

Importing the Certificate Reply from the CA

Once you've imported the required trusted certificate(s), as described in the previous section, or they are already in your keystore or in the `cacerts` file, you can import the certificate reply and thereby replace your self-signed certificate with a certificate chain. This chain will be either the one returned by the CA in response to your request (if the CA reply is a chain) or one constructed (if the CA reply is a single certificate) by using the certificate reply and trusted certificates that are already available in the keystore or in the `cacerts` keystore file.

As an example, suppose that you sent your certificate signing request to VeriSign. You can then import the reply via the following, which assumes that the returned certificate is in the file specified by *certReplyFile*:

```
keytool -import -trustcacerts
    -keystore storefile
    -alias alias
    -file certReplyFile
```

Type this command on one line.

The certificate reply is validated by using trusted certificates from the keystore and optionally by using the certificates configured in the `cacerts` keystore file (if the `-trustcacerts` option is specified). Each certificate in the chain is verified, using the certificate at the next level higher in the chain. You need to trust only the top-level "root" CA certificate in the chain. If you do not already trust the top-level certificate, `keytool` will display the fingerprint of that certificate and ask you whether you want to trust it.

The new certificate chain of the specified (by *alias*) entry replaces the old certificate (or chain) associated with this entry. The old chain can be replaced only if a valid *keypass*, the password used to protect the private key of the entry, is supplied. If no password is provided and if the private key password is different from the keystore password, the user is prompted for it.

For more detailed information about generating CSRs and importing certificate replies, see the `keytool` documentation:

- keytool documentation with Windows examples
- keytool documentation with UNIX examples

Previous page: Previous Lesson
Next page: Signing Code and Granting It Permissions