### **Installing Providers**

In order to be used, a cryptographic provider must first be installed, then registered either statically or dynamically. There are a variety of Sun providers shipped with this release (SUN, SunJCE, SunJSSE, SunRsaSign, etc.) that are already installed and registered. The following sections describe how to install and register additional providers.

#### Installing the Provider Classes

There are two possible ways to install the provider classes:

1. **On the normal Java classpath**

Place a zip or JAR file containing the classes anywhere in your classpath. Some algorithms types (Ciphers) require the provider be a signed Jar file.

1. **As an Installed/Bundled Extension**

The provider will be considered an installed extension if it is placed in the standard extension directory. In the JDK, that would be located in:

* + Unix: <java-home>/lib/ext
  + Windows: <java-home>\lib\ext

Here <java-home> refers to the directory where the runtime software is installed, which is the top-level directory of the Java Runtime Environment (JRE) or the jre directory in the Java JDK software. For example, if you have JDK 6 installed on Solaris in a directory named /home/user1/JDK1.6.0, or on Microsoft Windows in a directory named C:\Java\JDK1.6.0, then you need to install the JAR file in the following directory:

* + Unix: /home/user1/JDK1.6.0/jre/lib/ext
  + Windows: C:\JDK1.6.0\jre\lib\ext

Similarly, if you have the JRE 6 installed on Solaris in a directory named /home/user1/jre1.6.0, or on Microsoft Windows in a directory named C:\jre1.6.0, you need to install the JAR file in the following directory:

* + Unix: /home/user1/jre1.6.0/lib/ext
  + Windows: C:\jre1.6.0\lib\ext

For more information on how to deploy an extension, see [How is an extension deployed?](https://docs.oracle.com/javase/7/docs/technotes/guides/extensions/spec.html#deployment)

#### Registering the Provider

The next step is to add the provider to your list of registered providers. Providers can be registered statically by editing a security properties configuration file before running a Java application, or dynamically by calling a method at runtime. To prevent the installation of rogue providers being added to the runtime environment, applications attempting to dynamically register a provider must possess the appropriate runtime privilege.

##### Static Registration

The configuration file is located in the following location:

* Unix: <java-home>/lib/security/java.security
* Windows: <java-home>\lib\security\java.security

For each registered provider, this file should have a statement of the following form:

security.provider.*n*=*masterClassName*

This declares a provider, and specifies its preference order *n*. The preference order is the order in which providers are searched for requested algorithms (when no specific provider is requested). The order is 1-based: 1 is the most preferred, followed by 2, and so on.

*masterClassName* must specify the fully qualified name of provider's master class. The provider's documentation will specify its master class. This class is always a subclass of the Provider class. The subclass constructor sets the values of various properties that are required for the Java Cryptography API to look up the algorithms or other facilities the provider implements.

The JDK comes standard with automatically installed and configured providers such as "SUN" and "SunJCE". The "SUN" provider's master class is the SUN class in the sun.security.provider package, and the corresponding java.security file entry is as follows:

security.provider.5=sun.security.provider.Sun

To utilize another JCA provider, add a line referencing the alternate provider, specify the preference order ( making corresponding adjustments to the other providers' orders, if needed).

Suppose that the master class of CompanyX's provider is com.companyx.provider.ProviderX, and that you would like to configure this provider as the eighth most-preferred. To do so, you would add the following line to the java.security file:

security.provider.8=com.companyx.provider.ProviderX

**Static Registration**

The configuration file is located in the following location:

* Unix: <java-home>/lib/security/java.security
* Windows: <java-home>\lib\security\java.security

For each registered provider, this file should have a statement of the following form:

security.provider.*n*=*masterClassName*

This declares a provider, and specifies its preference order *n*. The preference order is the order in which providers are searched for requested algorithms (when no specific provider is requested). The order is 1-based: 1 is the most preferred, followed by 2, and so on.

*masterClassName* must specify the fully qualified name of provider's master class. The provider's documentation will specify its master class. This class is always a subclass of the Provider class. The subclass constructor sets the values of various properties that are required for the Java Cryptography API to look up the algorithms or other facilities the provider implements.

The JDK comes standard with automatically installed and configured providers such as "SUN" and "SunJCE". The "SUN" provider's master class is the SUN class in the sun.security.provider package, and the corresponding java.security file entry is as follows:

security.provider.5=sun.security.provider.Sun

To utilize another JCA provider, add a line referencing the alternate provider, specify the preference order ( making corresponding adjustments to the other providers' orders, if needed).

Suppose that the master class of CompanyX's provider is com.companyx.provider.ProviderX, and that you would like to configure this provider as the eighth most-preferred. To do so, you would add the following line to the java.security file:

security.provider.8=com.companyx.provider.ProviderX

**Dynamic Registration**

To register providers dynamically, applications call either the addProvider or insertProviderAt method in the Security class. This type of registration is not persistent across VM instances, and can only be done by "trusted" programs with the appropriate privilege. See [Security](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Security).

**Setting Provider Permissions**

Whenever encryption providers are used (that is, those that supply implementations of Cipher, KeyAgreement, KeyGenerator, Mac, or SecretKeyFactory), and the provider is not an installed extension [Permissions](https://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html) may need to be granted for when applets or applications using JCA are run while a security manager is installed. There is typically a security manager installed whenever an applet is running, and a security manager may be installed for an application either via code in the application itself or via a command-line argument. Permissions do not need to be granted to installed extensions, since the default system [policy configuration file](https://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html) grants all permissions to installed extensions (that is, installed in the [extensions directory](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#ProviderInstalling)).

The documentation from the vendor of each provider you will be using should include information as to which permissions it requires, and how to grant such permissions. For example, the following permissions may be needed by a provider if it is not an installed extension and a security manager is installed:

* java.lang.RuntimePermission "getProtectionDomain" to get class protection domains. The provider may need to get its own protection domain in the process of doing self-integrity checking.
* java.security.SecurityPermission "putProviderProperty.{name}" to set provider properties, where {name} is replaced by the actual provider name.

For example, a sample statement granting permissions to a provider whose name is "MyJCE" and whose code is in myjce\_provider.jar appears below. Such a statement could appear in a policy file. In this example, the myjce\_provider.jar file is assumed to be in the /localWork directory.

grant codeBase "file:/localWork/myjce\_provider.jar" {

permission java.lang.RuntimePermission "getProtectionDomain";

permission java.security.SecurityPermission

"putProviderProperty.MyJCE";

};

### Provider Class Methods

Each Provider class instance has a (currently case-sensitive) name, a version number, and a string description of the provider and its services. You can query the Provider instance for this information by calling the following methods:

public String getName()

public double getVersion()

public String getInfo()

## The Security Class

The Security class manages installed providers and security-wide properties. It only contains static methods and is never instantiated. The methods for adding or removing providers, and for setting Security properties, can only be executed by a trusted program. Currently, a "trusted program" is either

* a local application not running under a security manager, or
* an applet or application with permission to execute the specified method (see below).

The determination that code is considered trusted to perform an attempted action (such as adding a provider) requires that the applet is granted the proper permission(s) for that particular action. The policy configuration file(s) for a JDK installation specify what permissions (which types of system resource accesses) are allowed by code from specified code sources. (See below and the ["Default Policy Implementation and Policy File Syntax"](https://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html) and ["Java Security Architecture Specification"](https://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc.html) files for more information.)

Code being executed is always considered to come from a particular "code source". The code source includes not only the location (URL) where the code originated from, but also a reference to any public key(s) corresponding to the private key(s) that may have been used to sign the code. Public keys in a code source are referenced by (symbolic) alias names from the user's [keystore](https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#KeyManagement).

In a policy configuration file, a code source is represented by two components: a code base (URL), and an alias name (preceded by signedBy), where the alias name identifies the keystore entry containing the public key that must be used to verify the code's signature.

Each "grant" statement in such a file grants a specified code source a set of permissions, specifying which actions are allowed.

Here is a sample policy configuration file:

grant codeBase "file:/home/sysadmin/", signedBy "sysadmin" {

permission java.security.SecurityPermission "insertProvider.\*";

permission java.security.SecurityPermission "removeProvider.\*";

permission java.security.SecurityPermission "putProviderProperty.\*";

};

This configuration file specifies that code loaded from a signed JAR file from beneath the /home/sysadmin/ directory on the local file system can add or remove providers or set provider properties. (Note that the signature of the JAR file can be verified using the public key referenced by the alias name sysadmin in the user's keystore.)

Either component of the code source (or both) may be missing. Here's an example of a configuration file where the codeBase is omitted:

grant signedBy "sysadmin" {

permission java.security.SecurityPermission "insertProvider.\*";

permission java.security.SecurityPermission "removeProvider.\*";

};

If this policy is in effect, code that comes in a JAR File signed by sysadmin can add/remove providers--regardless of where the JAR File originated.

Here's an example without a signer:

grant codeBase "file:/home/sysadmin/" {

permission java.security.SecurityPermission "insertProvider.\*";

permission java.security.SecurityPermission "removeProvider.\*";

};

In this case, code that comes from anywhere within the /home/sysadmin/ directory on the local file system can add/remove providers. The code does not need to be signed.

An example where neither codeBase nor signedBy is included is:

grant {

permission java.security.SecurityPermission "insertProvider.\*";

permission java.security.SecurityPermission "removeProvider.\*";

};

Here, with both code source components missing, any code (regardless of where it originates, or whether or not it is signed, or who signed it) can add/remove providers. Obviously, this is definitely **NOT** recommended, as this grant could open a security hole. Untrusted code could install a Provider, thus affecting later code that is depending on a properly functioning implementation. (For example, a rogue Cipher object might capture and store the sensitive information it receives.)

### Managing Providers

The following tables summarize the methods in the Security class you can use to query which Providers are installed, as well as to install or remove providers at runtime.

**Querying Providers**

|  |  |
| --- | --- |
| **Method** | **Description** |
| static Provider[] getProviders() | Returns an array containing all the installed providers (technically, the Provider subclass for each package provider). The order of the Providers in the array is their preference order. |
| static Provider getProvider (String providerName) | Returns the Provider named providerName. It returns null if the Provider is not found. |

**Adding Providers**

|  |  |
| --- | --- |
| **Method** | **Description** |
| static int addProvider(Provider provider) | Adds a Provider to the end of the list of installed Providers. It returns the preference position in which the Provider was added, or -1 if the Provider was not added because it was already installed. |
| static int insertProviderAt (Provider provider, int position) | Adds a new Provider at a specified position. If the given provider is installed at the requested position, the provider formerly at that position and all providers with a position greater than positionare shifted up one position (towards the end of the list). This method returns the preference position in which the Provider was added, or -1 if the Provider was not added because it was already installed. |

**Removing Providers**

|  |  |
| --- | --- |
| **Method** | **Description** |
| static void removeProvider(String name) | Removes the Provider with the specified name. It returns silently if the provider is not installed. When the specified provider is removed, all providers located at a position greater than where the specified provider was are shifted down one position (towards the head of the list of installed providers). |

**NOTE:** If you want to change the preference position of a provider, you must first remove it, and then insert it back in at the new preference position.

### Security Properties

The Security class maintains a list of system-wide security properties. These properties are similar to the System properties, but are security-related. These properties can be set statically or dynamically. We have already seen an example of static security properties (that is, registering a provider statically via the "security.provider.i" security property). If you want to set properties dynamically, trusted programs can use the following methods:

static String getProperty(String key)

static void setProperty(String key, String datum)

Note: the list of security providers is established during VM startup, therefore the methods described above must be used to alter the provider list.

As a reminder, the configuration file is located in the following location:

* Unix: <java-home>/lib/security/java.security
* Windows: <java-home>\lib\security\java.security