



## CHAPTER 5

# *Java Security*

Java programs can dynamically load Java classes from a variety of sources, including untrusted sources, such as web sites reached across an insecure network. The ability to create and work with such mobile code is one of the great strengths and features of Java. To make it work successfully, however, Java puts great emphasis on a security architecture that allows untrusted code to run safely, without fear of damage to the host system.

The need for a security system in Java is most acutely demonstrated by applets—miniature Java applications designed to be embedded in web pages.\* When a user visits a web page (with a Java-enabled web browser) that contains an applet, the web browser downloads the Java class files that define that applet and runs them. In the absence of a security system, an applet could wreak havoc on the user's system by deleting files, installing a virus, stealing confidential information, and so on. Somewhat more subtly, an applet could take advantage of the user's system to forge email, generate spam, or launch hacking attempts on other systems.

Java's main line of defense against such malicious code is *access control*: untrusted code is simply not given access to certain sensitive portions of the core Java API. For example, an untrusted applet is not typically allowed to read, write, or delete files on the host system or connect over the network to any computer other than the web server from which it was downloaded. This chapter describes the Java access control architecture and a few other facets of the Java security system.

## *Security Risks*

Java has been designed from the ground up with security in mind; this gives it a great advantage over many other existing systems and platforms. Nevertheless, no system can guarantee 100% security, and Java is no exception.

---

\* Applets are documented in *Java Foundation Classes in a Nutshell* (O'Reilly) and are not covered in this book. Still, they serve as good examples here.

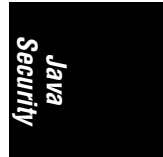
The Java security architecture was designed by security experts and has been studied and probed by many other security experts. The consensus is that the architecture itself is strong and robust, theoretically without any security holes (at least none that have been discovered yet). The implementation of the security architecture is another matter, however, and there is a long history of security flaws being found and patched in particular implementations of Java. For example, in April 1999, a flaw was found in Sun's implementation of the class verifier in Java 1.1. Patches for Java 1.1.6 and 1.1.7 were issued and the problem was fixed in Java 1.1.8. Even more recently, in August 1999, a severe flaw was found in Microsoft's Java Virtual Machine (which is used by the Internet Explorer 4.0 and 5.0 web browsers). The flaw was a particularly dangerous one because it allowed a malicious applet to gain unrestricted access to the underlying system. Microsoft has released a new version of their VM, and (as of this writing) there have not been any known attacks that took advantage of the flaw.

In all likelihood, security flaws will continue to be discovered (and patched) in Java VM implementations. Despite this, Java remains perhaps the most secure platform currently available. There have been few, if any, reported instances of malicious Java code exploiting security holes "in the wild." For practical purposes, the Java platform appears to be adequately secure, especially when contrasted with some of the insecure and virus-ridden alternatives.

## *Java VM Security and Class File Verification*

The lowest level of the Java security architecture involves the design of the Java Virtual Machine and the byte codes it executes. The Java VM does not allow any kind of direct access to individual memory addresses of the underlying system, which prevents Java code from interfering with the native hardware and operating system. These intentional restrictions on the VM are reflected in the Java language itself, which does not support pointers or pointer arithmetic. The language does not allow an integer to be cast to an object reference or vice versa, and there is no way whatsoever to obtain an object's address in memory. Without capabilities like these, malicious code simply cannot gain a foothold.

In addition to the secure design of the Virtual Machine instruction set, the VM goes through a process known as *byte-code verification* whenever it loads an untrusted class. This process ensures that the byte codes of a class (and their operands) are all valid; that the code never underflows or overflows the VM stack; that local variables are not used before they are initialized; that field, method, and class access control modifiers are respected; and so on. The verification step is designed to prevent the VM from executing byte codes that might crash it or put it into an undefined and untested state where it might be vulnerable to other attacks by malicious code. Byte-code verification is a defense against malicious hand-crafted Java byte codes and untrusted Java compilers that might output invalid byte codes.



## *Authentication and Cryptography*

In Java 1.1 and later, the `java.security` package (and its subpackages) provides classes and interfaces for *authentication*. As described in Chapter 4, this piece of the security architecture allows Java code to create and verify message digests and digital signatures. These technologies can ensure that any data (such as a Java class file) is authentic; that it originates from the person who claims to have originated it and has not been accidentally or maliciously modified in transit.

The Java Cryptography Extension, or JCE, consists of the `javax.crypto` package and its subpackages. These packages define classes for encryption and decryption of data. This is an important security-related feature for many applications, but is not directly relevant to the basic problem of preventing untrusted code from damaging the host system, so it is not discussed in this chapter.

## *Access Control*

As I noted at the beginning of this chapter, the heart of the Java security architecture is access control: untrusted code simply must not be granted access to the sensitive parts of the Java API that would allow it to do malicious things. As we'll discuss in the following sections, the Java access-control model evolved significantly between Java 1.0 and Java 1.2. The Java 1.2 access-control model is relatively stable; it has not changed significantly in Java 1.3.

### *Java 1.0: The Sandbox*

In this first release of Java, all Java code installed locally on the system is trusted implicitly. All code downloaded over the network, however, is untrusted and run in a restricted environment playfully called “the sandbox.” The access-control policies of the sandbox are defined by the currently installed `java.lang.SecurityManager` object. When system code is about to perform a restricted operation, such as reading a file from the local filesystem, it first calls an appropriate method (such as `checkRead()`) of the currently installed `SecurityManager` object. If untrusted code is running, the `SecurityManager` throws a `SecurityException` that prevents the restricted operation from taking place.

The most common user of the `SecurityManager` class is a Java-enabled web browser, which installs a `SecurityManager` object to allow applets to run without damaging the host system. The precise details of the security policy are an implementation detail of the web browser, of course, but applets are typically restricted in the following ways:

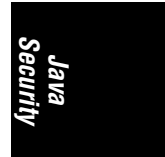
- An applet cannot read, write, rename, or delete files. It cannot query the length or modification date of a file or even check whether a given file exists. Similarly, an applet cannot create, list, or delete a directory.
- An applet cannot connect to or accept a connection from any computer other than the one it was downloaded from. It cannot use any privileged ports (i.e., ports below and including port 1024).

- An applet cannot perform system-level functions, such as loading a native library, spawning a new process, or exiting the Java interpreter. An applet cannot manipulate any threads or thread groups, except for those it creates itself. In Java 1.1 and later, applets cannot use the Java Reflection API to obtain information about the nonpublic members of classes, except for classes that were downloaded with the applet.
- An applet cannot access certain graphics- and GUI-related facilities. It cannot initiate a print job or access the system clipboard or event queue. In addition, all windows created by an applet typically display a prominent visual indicator that they are “insecure,” to prevent an applet from spoofing the appearance of some other application.
- An applet cannot read certain system properties, notably the `user.home` and `user.dir` properties, that specify the user's home directory and current working directory.
- An applet cannot circumvent these security restrictions by registering a new `SecurityManager` object.

#### *How the sandbox works*

Suppose that an applet (or some other untrusted code running in the sandbox) attempts to read the contents of the file `/etc/passwd` by passing this filename to the `FileInputStream()` constructor. The programmers who wrote the `FileInputStream` class were aware that the class provides access to a system resource (a file), so use of the class should therefore be subject to access control. For this reason, they coded the `FileInputStream()` constructor to use the `SecurityManager` class.

Every time `FileInputStream()` is called, it checks to see if a `SecurityManager` object has been installed. If so, the constructor calls the `checkRead()` method of that `SecurityManager` object, passing the filename (`/etc/passwd`, in this case) as the sole argument. The `checkRead()` method has no return value; it either returns normally or throws a `SecurityException`. If the method returns, the `FileInputStream()` constructor simply proceeds with whatever initialization is necessary and returns. Otherwise, it allows the `SecurityException` to propagate to the caller. When this happens, no `FileInputStream` object is created, and the applet does not gain access to the `/etc/passwd` file.



#### *Java 1.1: Digitally Signed Classes*

Java 1.1 retains the sandbox model of Java 1.0, but adds the `java.security` package and its digital signature capabilities. With these capabilities, Java classes can be digitally signed and verified. Thus, web browsers and other Java installations can be configured to trust downloaded code that bears a valid digital signature of a trusted entity. Such code is treated as if it were installed locally, so it is given full access to the Java APIs. In this release, the *jvakey* program manages keys and digitally signs JAR files of Java code. Although Java 1.1 adds the important ability to trust digitally signed code that would otherwise be untrusted, it sticks to the

basic sandbox model: trusted code gets full access and untrusted code gets totally restricted access.

### ***Java 1.2: Permissions and Policies***

Java 1.2 introduces major new access-control features into the Java security architecture. These features are implemented by new classes in the `java.security` package. The `Policy` class is one of the most important: it defines a Java security policy. A `Policy` object maps `CodeSource` objects to associated sets of `Permission` objects. A `CodeSource` object represents the source of a piece of Java code, which includes both the URL of the class file (and can be a local file) and a list of entities that have applied their digital signatures to the class file. The `Permission` objects associated with a `CodeSource` in the `Policy` define the permissions that are granted to code from a given source. Various Java APIs includes subclasses of `Permission` that represent different types of permissions. These include `java.lang.RuntimePermission`, `java.io.FilePermission`, and `java.net.SocketPermission`, for example.

Under this new access-control model, the `SecurityManager` class continues to be the central class; access-control requests are still made by invoking methods of a `SecurityManager`. However, the default `SecurityManager` implementation now delegates most of those requests to a new `AccessController` class that makes access decisions based on the `Permission` and `Policy` architecture.

The new Java 1.2 access-control architecture has several important features:

- Code from different sources can be given different sets of permissions. In other words, the new architecture supports fine-grained levels of trust. Even locally installed code can be treated as untrusted or partially untrusted. Under this new architecture, only system classes and standard extensions run as fully trusted.
- It is no longer necessary to define a custom subclass of `SecurityManager` to define a security policy. Policies can be configured by a system administrator by editing a text file or using the new *policytool* program.
- The new architecture is not limited to a fixed set of access control methods in the `SecurityManager` class. New `Permission` subclasses can be defined easily to govern access to new system resources (which might be exposed, for example, by new standard extensions that include native code).

### ***How policies and permissions work***

Let's return to the example of an applet that attempts to create a `FileInputStream` to read the file `/etc/passwd`. In Java 1.2, the `FileInputStream()` constructor behaves exactly the same as it does in Java 1.0 and Java 1.1: it looks to see if a `SecurityManager` is installed and, if so, calls its `checkRead()` method, passing the name of the file to be read.

What's new in Java 1.2 is the default behavior of the `checkRead()` method. Unless a program has replaced the default security manager with one of its own, the default implementation creates a `FilePermission` object to represent the access

being requested. This `FilePermission` object has a *target* of `"/etc/passwd"` and an *action* of `"read"`. The `checkRead()` method passes this `FilePermission` object to the static `checkPermission()` method of the `java.security.AccessController` class.

It is the `AccessController` and its `checkPermission()` method that do the real work of access control in Java 1.2. The method determines the `CodeSource` of each calling method and uses the current `Policy` object to determine the `Permission` objects associated with it. With this information, the `AccessController` can determine whether read access to the `/etc/passwd` file should be allowed.

The `Permission` class represents both the permissions granted by a `Policy` and the permissions requested by a method like the `FileInputStream()` constructor. When requesting a permission, Java typically uses a `FilePermission` (or other `Permission` subclass) with a very specific target, like `"/etc/passwd"`. When granting a permission, however, a `Policy` commonly uses a `FilePermission` object with a wildcard target, such as `"/etc/*"`, to represent many files. One of the key features of a `Permission` subclass such as `FilePermission` is that it defines an `implies()` method that can determine whether permission to read `"/etc/*"` implies permission to read `"/etc/passwd"`.

## *Security for Everyone*

Programmers, system administrators, and end users all have different security concerns and, thus, different roles to play in the Java 1.2 security architecture.

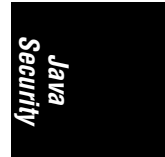
### *Security for System Programmers*

System programmers are the people who define new Java APIs that allow access to sensitive system resources. These programmers are typically working with native methods that have unprotected access to the system. They need to use the Java access-control architecture to prevent untrusted code from executing those native methods. To do this, system programmers must carefully insert `SecurityManager` calls at appropriate places in their code. A system programmer may choose to use an existing `Permission` subclass to govern access to the system resources exposed by her API, or she may decide to define a specialized subclass of `Permission`.

The system programmer carries a tremendous security burden: if she does not perform appropriate access control checks in her code, she compromises the security of the entire Java platform. The details are complex and are beyond the scope of this book. Fortunately, however, system programming that involves native methods is rare in Java; almost all of us are application programmers who can simply rely on the existing APIs.

### *Security for Application Programmers*

Programmers who use the core Java APIs and standard extensions, but do not define new extensions or write native methods, can simply rely on the security efforts of the system programmers who created those APIs. In other words, most



of us Java programmers can simply use the Java APIs and need not worry about introducing security holes into the Java platform.

In fact, application programmers rarely have to use the access-control architecture. If you are writing Java code that may be run as untrusted code, you should be aware of the restrictions placed on untrusted code by typical security policies. Keep in mind that some methods (such as methods that read or write files) can throw `SecurityException` objects, but don't feel you must write your code to catch these exceptions. Often, the appropriate response to a `SecurityException` is to allow it to propagate uncaught, so that it terminates the application.

Sometimes, as an application programmer, you want to write an application (such as an applet viewer) that can load untrusted classes and run them subject to access-control checks. To do this in Java 1.2, you must first install a security manager:

```
System.setSecurityManager(new SecurityManager());
```

Then use `java.net.URLClassLoader` to load the untrusted classes. `URLClassLoader` assigns a default set of safe permissions to the classes it loads, but in some cases you may want to modify the permissions granted to the loaded code through the `Policy` and `PermissionCollection` classes.

### ***Security for System Administrators***

In Java 1.2 and later, system administrators are responsible for defining the default security policy for the computers at their site. The default policy is stored in the file `lib/security/java.policy` in the Java installation. A system administrator can edit this text file by hand or use the *policytool* program from Sun to edit the file graphically. *policytool* is the preferred way to define policies, so the syntax of the underlying policy file is not documented in this book.

The default *java.policy* file defines a policy that is much like the policy of Java 1.0 and Java 1.1: system classes and installed extensions are fully trusted, while all other code is untrusted and only allowed a few simple permissions. While this default policy is adequate for many purposes, it may not be appropriate for all sites. For example, at some organizations, it may be appropriate to grant extra permissions to code downloaded from a secure intranet.

In order to define secure and effective security policies, a system administrator must understand the various `Permission` subclasses of the Java platform, the target and action names they support, and the security implications of granting any particular permission. These topics are explained well in a document titled "Permissions in the Java 2 SDK," which is part of the Java 1.2 release and also available (at the time of this writing) online at: <http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html>.

### ***Security for End Users***

Most end users do not have to think about security at all: their Java programs should simply run in a secure way with no intervention by them. Some sophisticated end users may want to define their own security policies, however. An end

user can do this by running *policytool* himself to define personal policy files that augment the system policy. The default personal policy is stored in a file named *.java.policy* in the user's home directory. By default, Java loads this policy file and uses it to augment the system policy file.

In Java 1.2 and later, a user can specify an additional policy file to use when starting up the Java interpreter, by defining the `java.security.policy` property with the `-D` option. For example:

```
C:\> java -Djava.security.policy=policyfile UntrustedApp
```

This line runs the class `UntrustedApp` after augmenting the default system and user policies with the policy specified in the file or URL *policyfile*. To replace the system and user policies instead of augmenting them, use a double equals sign in the property specification:

```
C:\> java -Djava.security.policy=policyfile UntrustedApp
```

Note, however, that specifying a policy file is only useful if there is a `SecurityManager` installed. If a user doesn't trust an application, he presumably doesn't trust that application to voluntarily install its own security manager. In this case, he can define the `java.security.manager` system property:

```
C:\> java -Djava.security.manager -Djava.security.policy=policyfile UntrustedApp
```

The value of this property does not matter; simply defining it is enough to tell the Java interpreter to automatically install a default `SecurityManager` object that subjects an application to the access control policies described in the system, user, and `java.security.policy` policy files.

## Permission Classes

Table 5-1 lists the various `Permission` subclasses defined by the core Java platform and summarizes the permissions they represent. See the reference section for more information on the individual classes. See <http://java.sun.com/j2se/1.4/docs/guide/security/permissions.html> for a detailed description of these permissions, along with their target and action names and a list of methods and the permissions they require (this document is also part of the standard documentation bundle that can be downloaded along with the JDK).

Table 5-1. Java permission classes

Permission class	Description
<code>java.security.AllPermission</code>	An instance of this special permission class implies all other permissions.
<code>javax.sound.sampled.AudioPermission</code>	Controls the ability to play and record sound.
<code>javax.security.auth.AuthPermission</code>	Controls access to authentication methods in <code>javax.security.auth</code> and its subpackages.



*Table 5-1. Java permission classes (continued)*

<i>Permission class</i>	<i>Description</i>
<code>java.awt.AWTPermission</code>	Controls access to sensitive methods in <code>java.awt</code> and its subpackages.
<code>java.io.FilePermission</code>	Governs access to the filesystem.
<code>java.net.NetPermission</code>	Governs access to networking-related resources such as stream handlers and HTTP authentication. See also <code>java.net.SocketPermission</code> .
<code>java.util.PropertyPermission</code>	Governs access to system properties.
<code>java.lang.reflect.ReflectPermission</code>	Governs access through the <code>java.lang.reflect</code> package to classes and class members that would normally be inaccessible.
<code>java.lang.RuntimePermission</code>	Governs access to a number of methods and resources. Many of the controlled methods are defined by <code>java.lang.System</code> and <code>java.lang.Runtime</code> .
<code>java.security.SecurityPermission</code>	Governs access to various security-related methods.
<code>java.io.SerializablePermission</code>	Governs access to serialization-related methods.
<code>java.net.SocketPermission</code>	Governs access to the network.
<code>java.sql.SQLPermission</code>	Governs the ability to specify logging streams in the <code>java.sql</code> JDBC API.