Programmer to Programmer™

BEGINNING Java Networking

Networking in JDK 1.3 and 1.4, Java I/O, Threads, Java security model, JCA, JAAS, JCE and JSSE, TCP, UDP, Multicasting, Java URL handler architecture, Serialization, RMI, CORBA, Servlets, JavaMail and JMS

Chad Darby, John Griffin, Pascal de Haan, Peter den Haan,
Alexander V. Konstantinou, Sing Li, Sean MacLean,
Glenn E. Mitchell II, Joel Peach, Peter Wansch, William Wright

Summary of Contents

7

Java Security Model

If you are studying Java Networking with an eye on writing networked client software, you should be acutely aware of the security implications of downloading executable content over the network. A basic knowledge of the Java security framework can help you to decide when you can trust code, and how to limit the trust you extend to downloaded code. Should you be planning to write network services, such as web servers, security is absolutely vital. You should assume that the server *will* be attacked sooner or later. Most of the Java preliminaries on network programming have been covered now, but the final ingredient needed before we can safely embark on it is security.

In this chapter, you will see why Java is exceptionally well suited to writing network-enabled code, and be shown how the Java security model can help you make it even more secure. You will:

- See why security is important in a heavily networked environment
- Understand the fundamentals of access control as introduced in Java 1.0 and extended in Java 1.1, notably the "sandbox model"
- Sign your code to authenticate it to others and make it tamper-proof
- Understand the Java 2 platform security model which introduces permissions and policies
- Give Java classes specific permissions to access parts of the system using a security policy
- Create your own permissions, and securely grant access to services you provide
- Use the Java 2 security tools

Although some cryptography terms cannot be avoided, this chapter covers only those that are absolutely necessary. An in-depth treatment of the topic follows later in Chapter 14.

Why Security?

Don't just trust any executable code – not even when it's coming from an apparently trustworthy source. This lesson was first learned the hard way with the famous Internet worm of November 3, 1988, which effectively shut down the net. A 23-year-old student, Robert Morris, had created a program that would propagate over the network, installing a copy of itself on every computer it met along the way. But the bit of code that was supposed to ensure that each machine only got a single copy of the worm didn't work, and thousands of Internet hosts ground to a halt as they became infested with hundreds of worms each. Administrators of clean machines axed their net connections to avoid infection, and for a few days the Internet ceased to exist.

Amazingly enough, this lesson is still being learned at great cost every day. Some of the most popular e-mail applications and word processors are perfectly happy to execute macros, scripts, and active controls without any user intervention. Such executable content can gain access to the most sensitive system services, such as the hard drive or the operating system. This is possible because what little security is in place appears to be an afterthought. More than a decade after the worm, its descendants thrive. The worm was, in fact, a fairly benign creature, which had run amok due to a programming error. Most of the current crop is spread by design, and some of them have been created to cause as much damage as possible.

The Internet worm actually spread through a vulnerability in the commonly-used **sendmail** mail transport program. Sendmail featured a handy facility allowing programmers to debug it over the net. For the worm, this turned out to be a handy facility allowing it to run just about anything on any mail host. Clearly, it pays to be very careful about the services provided by an Internet server. Sensitive services need to be carefully protected so that only authorized users can access them. Services must also be able to handle unexpected input so they cannot be subverted.

For example, one of the most common security holes in server software is insufficient protection of memory buffers. An unprotected buffer will overflow when the input is unexpectedly long, and overwrite other memory locations. Hackers have used such carelessness to break into the most tightly secured machines. This, too, is a lesson still being learnt daily.

New vulnerabilities are being discovered every week in some of the world's most popular server software. For up to date information and security bulletins, see the website of the Computer Emergency Response Team (CERT, http://www.cert.org).

Java Security

In Java, Sun has addressed security issues from the very start:

- The Java language ensures that array access never goes out of bounds. Any attempt to access memory beyond the array boundaries results in an `ArrayIndexOutOfBoundsException`. This prevents buffer overflows and underflows.
- The pointers and unions that lie at the root of so many bugs in C and C++ are not supported. Together with garbage collection, this prevents uncontrolled memory access.
- Casts are always checked and throw a `ClassCastException` if illegal. That way, code can never get around security features by casting an object to an incompatible type.
- Before Java bytecode (that is, a `.class` file) is loaded as a class, the virtual machine runs a bytecode verifier to ensure that the code is valid. This makes it impossible to construct invalid bytecode that could cause the JVM to behave in unexpected ways.
- The language does not contain any low-level constructs that would allow direct hardware access. All access has to go through the Java libraries, with the sole exception of **Java Native Interface (JNI)** calls.
- The libraries enforce security by blocking Java programs from accessing the system unless they are allowed to do so. Any attempt to perform a prohibited operation will cause a `SecurityException` to be thrown.
- Finally, there is extensive support for cryptography. Cryptography is essential to data protection and authentication. We will return to it in Chapter 14.

These features make Java arguably the best language to implement Internet-secure applications, although it is important to remember that there may still be security bugs in the JVM implementation itself. If we want to safely use Java networking to connect to services on the net, or to implement services made available over the net, we need to be familiar with its security support.
Broadly speaking, there are two types of Java programs. **Applications** need to be installed by the user before they can be used. An application is typically purchased or downloaded from a trusted source. It takes an explicit action to install and run it. Traditionally, the user implicitly trusts an application with full access to the machine; later in this chapter, we will see how Java 2 refines this by allowing the user to confer specific levels of trust using **certificates**.

Applets, on the other hand, are primarily Java programs embedded within web pages. They can be used to make a page more interactive, or to access a server resource such as a database without the delay associated with server-side processing. They execute automatically when you view the web page that contains the applet, unless Java is disabled on the user's browser. The user does not know whether a web page contains an applet and if the applet can be trusted. Like executable content within an e-mail, an applet should be treated as if it contained malicious code.

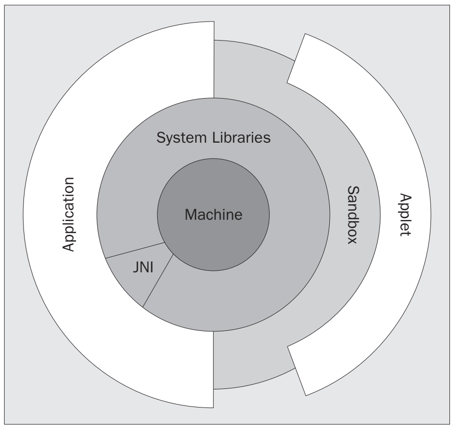
The basic Java security model directly reflects this fundamental difference between applications and applets. Applications get full access to all system resources, while applets run in a protected environment known as a **sandbox**.

The Sandbox Model

We have seen that a Java applet in a web page is downloaded from the web and executed without explicit permission from the user, provided that the user had enabled Java on their browser. In fact, the user may not even realize that the web page uses an applet. There is no reason why it could not contain malicious code, such as a virus, a function to delete all the files on the hard drive, or a program which tracks all the passwords typed in and sends the results to some e-mail address. The only safe thing to do is to treat the applet as **untrusted code** and deny it access to all system resources that could potentially be abused. This is known as the **sandbox model**. Basically, the applet is put in a virtual container, separate from the rest of the system, where it can cause no harm.

The Basic Sandbox

The sandbox model has been with Java from the first production release, version 1.0:



While applications get full access to the libraries and the machine, the Java 1.0 sandbox subjects an applet to a large number of restrictions. Every facility that could allow the applet to damage data, access confidential information, or otherwise compromise system security, is blocked from access.

Before we discuss these restrictions in more detail, let's examine how it works with an example. The following code implements a simple applet performing some simple privileged operations. The class has a `main()` method, so you can run it as an application and compare results.

```
//SecurityApp.java

package com.wrox.security;

import java.awt.*;
import java.awt.event.*;

public class SecurityApp extends java.applet.Applet {

  private SecurityTestWindow window;

  public void init() {
    window = new SecurityTestWindow("Applet Security Test");
    window.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        window.hide();
      }
    });
    window.showTestResults();
  }

  public void destroy() {
    window.dispose();
  }
```

This is a small applet. Its only purpose is to create the `SecurityTestWindow`, which runs a number of security tests and displays their results. In addition, there is a static `main()` method to allow the tests to be run from an application:

```
  public static void main(String[] args) {
    SecurityTestWindow window =
      new SecurityTestWindow("Application Security Test");
    window.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    window.showTestResults();
  }
}
```

The `SecurityTestWindow` class itself attempts to perform a selection of three security-sensitive operations: it lists a file from the root directory, opens a network connection to the Wrox site and retrieves the length of its homepage, and finally displays the location of your home directory. All three operations are security-sensitive and generally forbidden to untrusted code.

```
//SecurityTestWindow.java

package com.wrox.security;

import java.awt.*;
import java.io.File;
import java.net.URLConnection;

class SecurityTestWindow extends Frame {

  private TextArea textArea;

  public SecurityTestWindow(String title) {
    super(title);

    textArea = new TextArea();
    textArea.setEditable(false);
    add(textArea);

    pack();
    show();
  }

  public void showTestResults() {
    println("Running tests...");
    try {
      fileTest("/");
    } catch (Exception e) {
      println(e);
    }
    try {
      socketTest("http://www.wrox.com");
    } catch (Exception e) {
      println(e);
    }
    try {
      systemTest("user.home");
    } catch (Exception e) {
      println(e);
    }
    println("Done.");
  }

  private void fileTest(String path) {
    String[] files = new java.io.File(path).list();
    if (files.length > 0) {
      println("First file in " + path + " is " + files[0]);
    }
  }

  private void socketTest(String url) throws java.io.IOException {
    URLConnection connection = new java.net.URL(url).openConnection();
    connection.connect();
    println(url + " is " + connection.getContentLength() + " bytes");
    connection.getInputStream().close();
  }

  private void systemTest(String property) {
    println(property + " has value " + System.getProperty(property));
  }
```

```
    private void println(Object toPrint) {
      textArea.append(toPrint + "\n");
    }
  }
```

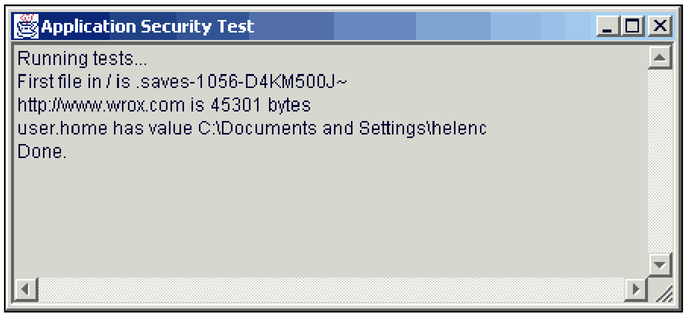
It is not necessary to have anything in particular in your classpath. After compilation, the code can be run as an application:

```
C:\Beg_Java_Networking\Ch07>set CLASSPATH=.;

C:\Beg_Java_Networking\Ch07>javac com.wrox.security.SecurityApp.java

C:\Beg_Java_Networking\Ch07>java com.wrox.security.SecurityApp
```

As an application has full access to the system, you should be seeing something like the following window:



```
Running tests...
First file in / is .saves-1056-D4KM500J~
http://www.wrox.com is 45301 bytes
user.home has value C:\Documents and Settings\helenc
Done.
```

If you are running the application from behind a firewall, you may find that instead of returning the number of bytes from the connection, an `UnknownHostException` is thrown. In such an instance, the information being returned is effectively hidden by the firewall.

It is possible to run a workaround on a proxy server or another machine that is not behind this firewall, providing you know the IP address for the proxy you will use. If you yourself do not have this information, contact your system administrator who can advise you further.

For the purposes of this example, we shall say that the proxy IP address is `123.456.7.89`. The port will be `99`. The workaround explicitly names the IP address and port that the application will run against on the command line, and it looks like this:

```
C:\Beg_Java_Networking\Ch07>java -Dhttp.proxyHost=123.456.7.89 -Dhttp.proxyPort=99
SecurityApp
```

Obviously, your address and port will be different, but when included on the command line as in the example above, this command should run successfully.

To view the program as a browser applet, we need a small HTML file:

```
  <HTML>
     <HEAD><TITLE>Applet Security Example</TITLE></HEAD>
     <BODY>
```

```
         <HR WIDTH="100%">
         <H3>Applet Security Example</H3>
         <HR WIDTH="100%">
         <APPLET
            code="com.wrox.security.SecurityApp.class"
            archive="com.wrox.security.SecurityDemo.jar" width=0 height=0></APPLET>
      </BODY>
   </HTML>
```

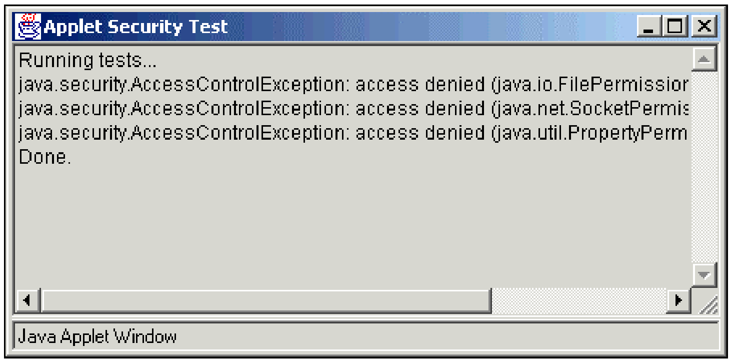
After saving this file as `SecurityApp.html` in the same directory as the Java source, open it in a browser or the `appletviewer` tool:

```
C:\Beg_Java_Networking\Ch07>appletviewer SecurityApp.html
```

If you are operating behind a firewall, the syntax for this command is as follows:

```
C:\Beg_Java_Networking\Ch07>appletviewer -J"-Dhttp.proxyHost=123.456.7.89" -J"-Dhttp
.proxyPort=99" SecurityApp.html
```

A window like the one below should appear.



```
Running tests...
java.security.AccessControlException: access denied (java.io.FilePermissior
java.security.AccessControlException: access denied (java.net.SocketPermis
java.security.AccessControlException: access denied (java.util.PropertyPerm
Done.
```

All three tests failed with a security `AccessControlException` or a subclass of it, depending on the browser. Had this demo been loaded from the Wrox web site, the download from www.wrox.com would have succeeded, as an applet can always access the host it was loaded from.

In the `fileTest()`, `socketTest()` and `systemTest()` methods of `SecurityTestWindow`, the program attempts to do things which could potentially be abused by malicious code. For that reason, they are forbidden to applets and throw a `java.lang.SecurityException` (or, as in this case, a subclass). The forbidden operations that `SecurityApp` tries to accomplish are as follows:

- **Read a directory**: Untrusted code cannot read from or write to the file system in any way. This applies to both files and directories, as even checking for the existence of a file can reveal sensitive information such as the presence of software with known security holes.
- **Access the Internet**: The networking operations that can be performed are strictly limited. These restrictions ensure that network communications are only possible within the machine the code was loaded from.

Untrusted code can only create network connections to the computer it was loaded from, and server sockets created by it cannot accept connections from anywhere else. It cannot create server sockets on the privileged ports (port `1023` or lower); on UNIX, this would even apply to the root use. It cannot create multicast sockets, and none of the network object factories – `SocketImplFactory`, `URLStreamHandlerFactory` or `ContentHandlerFactory` – may be created or registered. This prevents an applet from creating or accessing unauthorized services.

- **Read a system property**: Only a few selected properties, typically `java.version`, `java.vendor`, `java.vendor.url`, `java.class.version`, `os.name`, `os.arch`, `os.version`, `file.separator`, `path.separator` and `line.separator`, can be retrieved using `System.getProperty()`. `System.setProperties()` would allow the modification of system properties and cannot be used at all.

  Many other methods of the `Runtime` and `System` classes are also security-sensitive. Untrusted code may not access the `exit()` methods, as these would stop the JVM. The `Runtime.exec()` methods which run external commands are likewise inaccessible. The ability to run commands would allow untrusted code to perform just about any operation.

  The `load()` and `loadLibrary()` methods cannot be accessed either. Allowing these would enable malicious code to gain direct access to the machine using JNI.

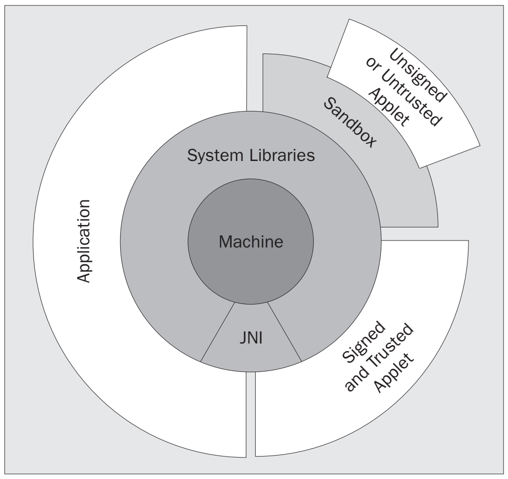
These are only some of the sandbox restrictions. They partly depend on the applet container (for example, a web browser or the Sun `AppletViewer`), but in addition to what has been mentioned above you can typically expect the following:

- The system clipboard and event queue cannot be accessed. Printing is not possible.
- Threads can be accessed or created only within the thread group in which the code is running.
- From the libraries installed on the client machine, only `java.*` classes can be directly used. No `ClassLoader` can be instantiated or used.
- Access to the `java.security` package is limited. In particular, security properties are inaccessible and untrusted code cannot create or register its own `SecurityManager`, as this would enable it to circumvent the security restrictions. `SecurityManager` will be discussed later in this chapter.
- The applet gets its own **namespace**, so it cannot gain access to classes loaded by other Java applications running in the same JVM.

Together, these restrictions ensure that untrusted applet code can safely be run. The flip side is that a lot of useful things have become impossible. For example, if you would want to create a word processing applet capable of saving its files to the user's hard drive, that would be impossible. This problem was addressed in version 1.1 of the JDK.

Escaping the Sandbox

Confining all applets to the sandbox means that many types of applications are impossible to realize for applet writers. Some applets simply need to be able to escape in order to perform their task. To support this, JDK 1.1 introduced **signed applets**.



Signed applets are ordinary applets in a JAR archive with a cryptographically secure **signature**. The signature both identifies the code's author or publisher, and indicates that they vouch for the integrity of the code. It is almost impossible for a third party to forge the publisher's signature or to change existing code without invalidating the signature. This makes it safe for a user to indicate that applets from a specific publisher should be **trusted**. The JVM can then give the applet full access rights. An applet that is both signed by its author and trusted by the user is allowed to do everything an application can do.

It is also possible to obtain a **trusted certificate** from a **certificate authority** such as VeriSign (http://www.verisign.com/) or Thawte (http://www.thawte.com/). A certificate contains your signature's **public key** and information about your identity, signed by the authority. Browsers are generally pre-configured to trust certificates issued by such authorities. Users can set up their trust by simply acknowledging a dialog box, provided you sign with a certificate signed by a recognized authority. The level of security granted to the applet will also depend on the security settings of the browser, such as the "zones" that can be set within Internet Explorer.

Unfortunately, Microsoft Internet Explorer does not recognize signed JAR archives as such, but needs to have its code packaged and signed in **Windows Cabinet (CAB)** files. On the other hand, Netscape uses its own JAR signing tools and a proprietary security API. This fragmentation of Java applet support is one of the things that make applets less viable in an Internet environment. Signed applets can still be useful in the more controlled and uniform world of intranet applications, but for the Internet world server-side technologies or Java Web Start are more viable.

Certificates and cryptography in general will be discussed in more detail in Chapter 14.

Code Signing

To allow the applet described in the previous section to escape the sandbox, we need to sign it. This procedure is best illustrated by a simple example. Assuming you are using JDK 1.2 or better, the steps are the following. First, in order to sign code it needs to be packaged in a JAR archive. Save the following text as `C:\Beg_Java_Networking\Ch07\SecurityApp.manifest`, or alternatively this file is in the code download for this chapter:

```
Manifest-Version: 1.0
Main-Class: com.wrox.security.SecurityApp
```

This is the manifest to be used in creating the JAR:

```
C:\Beg_Java_Networking\Ch07>jar cfm SecurityApp.jar SecurityApp.manifest
com\wrox\security\Security*.class
```

This creates a file called `SecurityApp.jar` containing the security demo classes. Before we can sign it, we first need to generate a keystore database with our signature key, which we will give the alias signkey.

If you are using Windows, you need to ensure that the JDK's `bin` directory is in your path. The tool will prompt you for the information it needs, such as passwords for both the `SecurityApp.keystore` file and the generated signature key, and information about your identity. The contents of the square brackets are the current or default settings.

```
C:\Beg_Java_Networking\Ch07>keytool -genkey -keystore SecurityApp.keystore -alias signkey
-keyalg rsa

Enter keystore password:  bronte
What is your first and last name?
  [Unknown]:  Ellen Callaghan
What is the name of your organizational unit?
  [Unknown]:  Mathe Group
What is the name of your organization?
  [Unknown]:  Breuil Holdings
What is the name of your City or Locality?
  [Unknown]:  Angouleme
What is the name of your State or Province?
  [Unknown]:  France
What is the two-letter country code for this unit?
  [Unknown]:  FR
Is <CN=Ellen Callaghan, OU=Mathe Group, O=Breuil Holdings, L=Angouleme, ST=France, C=FR>
correct?
  [no]:  yes

Enter key password for <signkey>
        (RETURN if same as keystore password):  bronte

C:\Beg_Java_Networking\Ch07>
```

Answer the prompts and the tool will generate a **key pair** (signature) for your named signkey. The keytool will be described more fully later in this chapter.

At this point, we could let the tool generate a certificate request, send that off to an authority, and import the certificate they will return. Everyone would then be able to trust our signed applet. Unfortunately, that would probably cost rather more time and money than you'd be willing to spend on an example like this, so we will continue without an "official" certificate.

At this stage we are ready to sign the code with our signature.

```
C:\Beg_Java_Networking\Ch07>jarsigner -keystore SecurityApp.keystore SecurityApp.jar signkey
```

When asked for a passphrase, merely re-enter your password. Browsers do not yet trust the signed applet – after all, anyone can sign an applet. To achieve this, we would either have to get a certificate signed by a **certificate authority (CA)** that the browser has been set up to trust, or import our own certificate into the browser as a trusted root certificate, so we effectively become our own CA – at least, that is how it would work in an ideal world. (We will discuss CAs in more detail below.) As mentioned above, Internet Explorer and Netscape are incompatible in their approach to code signing.

Rather than deal with all this, we can for the moment use the JDK applet viewer tool and create a security policy file for the signed code. Save the following as `SecurityApp.policy1` (it is also included in the code download for this chapter):

```
keystore "file:SecurityApp.keystore";

grant SignedBy "signkey" {

   permission java.security.AllPermission;
};
```

The applet viewer can now be run with the following prompt:

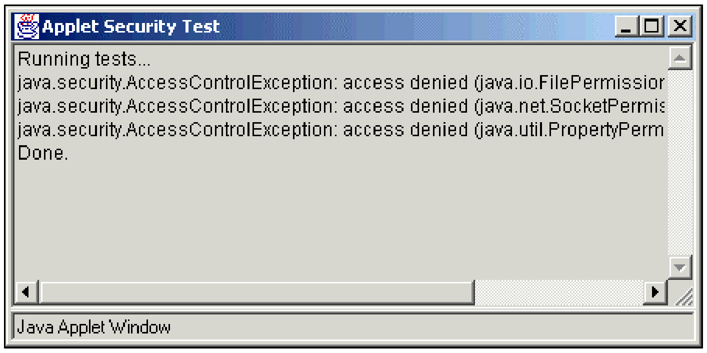
```
C:\Beg_Java_Networking\Ch07>appletviewer -J"-Djava.security.policy=SecurityApp.policy1" SecurityApp.html
```

Or if you are behind a firewall:

```
C:\Beg_Java_Networking\Ch07>appletviewer -J"-Dhttp.proxyHost=123.456.7.89" -J"-Dhttp.proxyPort=99" -J"-Djava.security.policy=SecurityApp.policy1" SecurityApp.html
```

where `proxyHost` and `proxyPort` are equal to the proxy server and port that your system is configured to.

The policy allows applets signed by signkey to execute outside the sandbox, and the applet should run as if it were an application installed on the machine.



```
Running tests...
java.security.AccessControlException: access denied (java.io.FilePermissior
java.security.AccessControlException: access denied (java.net.SocketPermis
java.security.AccessControlException: access denied (java.util.PropertyPerm
Done.
```

By signing the `SecurityApp.jar` file we have achieved two things:

- We have established the authorship of the archive. The archive is signed with the **private key** of the key pair generated by keytool. This signature can be verified with the other, **public key**. We can then freely give the public key away so others can verify that we really signed the archive. As long as the private key is kept secret, it is impossible for anyone else to create an archive that will verify against our public key.

- ❑ We have protected the archive against tampering. The signature includes a checksum (digest) of all the archive contents. Any modification of the classes or other files will change the checksum and invalidate the signature.

Security is based on trust, so don't just believe it – try it! Change one of the classes, recompile and update the archive, then run the applet viewer again. Rename your keystore file and generate a new key pair with the same name. Convince yourself that it really works.

The applet viewer uses the Java **security policy file** to check whether applets signed with our specific signature should be trusted. Few people use Sun's applet viewer, of course. And you will generally not want to force all of your users to set up a policy file or reconfigure their browser just so they can use your applet anyway. So, how are they to know that you are indeed who you say you are, and that you can be trusted?

With this issue in mind, browsers generally trust applets signed with certificates issued by one of a list of common **certificate authorities (CA)**. These authorities will check your identity before they issue you with a certificate signed by them. Because the certificate is signed, it is unforgeable and tamper-proof. Should your code be found to be doing something unsavory, you can be easily tracked down using the information from the certificate you used to sign the code with.

When we come to explore secure web sites (HTTPS, SSL), we will see that virtually the same considerations apply there. Browsers will only accept an SSL connection from the server without complaint if it uses a certificate signed by a reputable certificate authority.

Java Web Start

Slow downloads and incompatible implementations have all but killed off Java applets. Our discussion would therefore not be complete without at least mentioning the **Java Web Start** plugin. This is a comparatively new technology to deploy Java applications from a web server. With a minimum of user interaction, it features good security, incremental downloads and updates.

Although Java Web Start uses a sandbox model similar to the applet sandbox, applications can gain limited, controlled access to the local disk. Upon acceptance of the certificate by the user, signed applications can get unrestricted access without the cross-browser compatibility problems that affect applets. The price to pay is that browsers do not support it by default; the user will have to install a plugin first. The plugin is also included in version 1.4 of the JDK.

The Command Line Security Tools

We have already been using two of the security-related tools shipped with JDK 1.2 and higher: `keytool` and `jarsigner`. It is worth taking a slightly closer look at each. We will not discuss every command line option in minute detail, but you can find more information in the JDK documentation itself or from http://java.sun.com/j2se/1.3/docs/index.html.

Keytool

The **keytool** will manage a small database with keys and trusted certificates. These are used to sign code and to verify signatures:

- **Private key**: this half of a key pair can be used to sign a JAR file. The private key needs to be kept secret, so that we will be the only one able to sign code and other files.
- **Public key**: the public key of a key pair is necessary to verify the signature. If we only know someone's public key, it is currently impossible to discover the corresponding private key. For this reason it is safe to spread the public key far and wide so that others can authenticate our code.
- **Trusted certificate:** This contains your identity, your public key, and some other information, signed by a trusted certificate authority. Such certificates remove the need of disseminating your public key before your code can be trusted. If you sign using the certificate, the other party will know that the authority vouches for the authenticity of your public key.

Using the keytool, it is possible to generate a **key pair** of public and private keys using the `-genkey` option. We've already seen this in operation in the previous example. The key's alias defaults to `mykey`, but you can specify an alias with the `-alias` option:

```
keytool -genkey -alias signkey -keyalg rsa
```

The `-keyalg` option tells the tool to use generate a key pair for the given encryption algorithm. The default is DSA, but most browsers support only RSA key pairs. Other options include setting a limited validity period (`-validity`), the key size (`-keysize`), and a password for the key in addition to the keystore password (`-keypass`). The password will protect any usage of the private key.

It is also possible to generate the **certificate request** an authority needs before they can give you a certificate, using `-certreq -file certificate_file`:

```
keytool -certreq -alias signkey -file codeCertReq.cer
```

Please refer to the individual certificate authorities' guidelines on submitting a certificate request.

The following command will display the contents of a certificate file:

```
keytool -printcert -file fred.cer
```

If you wish to import certificates from people you trust, once you have firmly convinced yourself that the certificate is genuinely theirs, this is possible using the `-import` option:

```
keytool -import -alias fredsCertificate -file fred.cer
```

You can also import the certificate signed and returned to you by a certificate authority:

```
keytool -import -trustcacerts -alias fredsCertificate -file codeCertReq.cer
```

The `-trustcacerts` option tells the keytool to trust certificates from the `${java.home}/lib/security/cacerts` keystore file, which contains the keys of certificate authorities. If you use an authority other than VeriSign, you will probably need to import their certificate in this keystore first. Without the `-trustcacerts` option, the `cacerts` keystore is ignored.

Should you wish to export your certificate or public key so that someone else can import it in his or her key database, the command is as follows:

```
keytool -export -alias signkey -file peter.cer
```

If you want to change any details in your certificate without generating a new key pair, this is possible though self-signing certificates with `-selfcert`. The keytool documentation contains a step-by-step description of the procedure.

It is also possible to perform common maintenance tasks with the tool, such as displaying all keystore entries or a particular alias using the `-list` option, deleting a keystore entry using `-delete -keystore` *`keystorename`*`.keystore` and then deleting entries using `-delete -alias` *`alias`*, cloning one with `-keyclone -alias` *`alias`* `-dest` *`destination_alias`*, or changing the keystore and entry passwords using `-storepasswd` and `-keypasswd`.

There are a number of options common to virtually all keytool commands. The most important are the `-keystore` option to specify a keystore file other than the default `${user.home}/.keystore; -alias` to specify a key or certificate alias different from `mykey`; and `-file` to read the input from, or write output to, a file.

The keystore file is by default stored in the Java Key Store (JKS) format, a Sun proprietary file format. You can provide your own keystore implementation to change this, or use the alternative keystores provided by some Java extensions. The keytool will work with any file-based keystore, but not with implementations that load their information from a database or the network. The other security tools in the JDK, `jarsigner` and `policytool`, have no such restriction.

Jarsigner

Signing JAR archives is done using the `jarsigner` tool. Once an archive has been signed, its integrity and authenticity can be verified by the Java classloader. A signed archive cannot be modified without invalidating the signature. The most important usage options are the following:

```
jarsigner -keystore mystore -storepass mystorePassword -keypass signkeyPassword -
signedjar MySignedJarFile.jar MyJarFile.jar signkey
```

This command would sign `MyJarFile.jar` using the alias `myCodeSigningKey` from the keystore located in the `mystore` file. The signed archive will be written to `MyJarFile.jar`. You can omit the passwords if you wish; the tool will prompt for them. The keystore can be omitted too if you are using the default keystore file.

The tool can also be used to check the validity of signatures:

```
jarsigner -verify -keystore mystore MySignedJarFile.jar
```

The public key of the signature or certificate authority needs to be in your keystore file for verification to work. Multiple people can sign an archive simply by running the `jarsigner` tool multiple times.

Browser Compatibility

It has been mentioned before that Microsoft Internet Explorer does, unfortunately, not recognize signed JAR archives. To create an applet for use with IE, you will need to download the Microsoft SDK for Java from http://www.microsoft.com/java/sdk/ and use their archiving and signing tools. For a truly cross-browser applet, you will need to package it in both JAR and the CAB archives and use Internet Explorer's `CABBASE` applet parameter:

```
<HTML>
   <HEAD><TITLE>Applet Security Example</TITLE></HEAD>
   <BODY>
      <HR WIDTH="100%">
      <H3>Applet Security Example</H3>
      <HR WIDTH="100%">
      <APPLET
         CODE="/com/wrox/security/SecurityApp.class"
         ARCHIVE="/come/wrox/security/SecurityApp.jar" WIDTH=0 HEIGHT=0>
         <PARAM NAME="CABBASE" VALUE="SecurityApp.cab">
      </APPLET>
   </BODY>
</HTML>
```

This HTML will load and verify `SecurityApp.cab` in IE and `SecurityApp.jar` in most other browsers.

Although Netscape supports JAR archives, it introduces its own incompatibilities by requiring the use of the Netscape signing tools available from http://developer.netscape.com/software/signedobj/jarpack.html, and a proprietary **Capabilities API** to perform privileged operations.

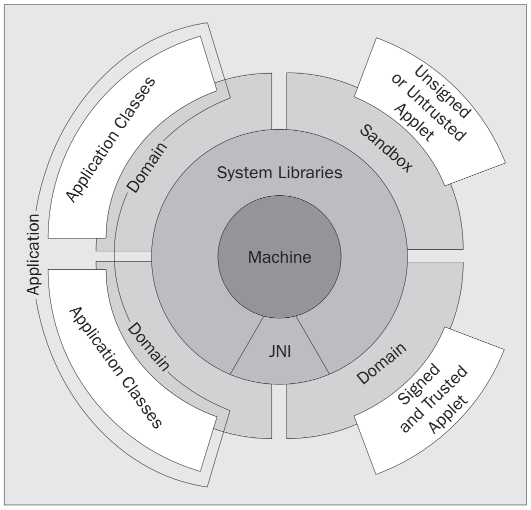
This turns writing cross-browser code to escape the sandbox into a developer's nightmare. You could require your users to download and install the Sun Java 2 plugin, but if they have to install a plugin anyway, Java Web Start (discussed above) is likely to be the best option.

The Java 2 Model

The sandbox model has two important limitations. First, it is very coarse-grained: either you are inside the sandbox and you cannot access the system in a meaningful way at all, or you are outside of it and free to do completely as you please. There is no middle way. If, for example, you are an Application Service Provider hosting a number of your customers' web applications on a single application server, you cannot set up security so that each application can only access the directories assigned to that customer. The **principle of least privilege** means that, if you are concerned about security, you give a program exactly those privileges it needs to perform its function. The sandbox does not support this model.

The second limitation of the sandbox is that it is only concerned with applet security. In practice, there are many other situations where you may need security. The example above clearly illustrates this. To give another example, an Enterprise JavaBean (EJB) may not create threads. Yet without a better security model, an EJB container will actually not be able to enforce this.

In the Java 2 platform (JDK 1.2 and higher), the security model was completely overhauled to address these problems. The new model offers fine-grained, highly configurable access control to applications as well as applets. Separate permissions can be set up for each major API, such as file access, networking, and AWT/Swing.



In the Java 2 security model, every Java class (including both applets and applications) runs inside a **protection domain**. Each domain has a set of **permissions** associated with it. For example, classes in a specific domain might be able to access the hard disk but not be allowed any network connections. Yet another domain might allow read-only access to the disk and limited networking.

A Java class is assigned to exactly one protection domain based both on where it was loaded from, and who signed it. There is little that sets applets apart from applications anymore; the only difference is that applets default to strict security settings, while applications don't. Such defaults can easily be changed. Finally, it is worth noting that an application may consist of classes from more than one protection domain; this is quite common in secure applications where access to sensitive resources is granted only through libraries.

At the lowest level are the Java libraries, running inside the **system domain**. The system domain always enjoys full privileges. For each privileged operation that the libraries implement, they check with the **security manager** to see if the protection domains of the calling code have permission to perform that operation, and throw a `SecurityException` if they don't. It follows that the security system is critically dependent on the fact that all machine access goes through the libraries – in other words, if you grant code access to native code libraries (JNI) you no longer have any serious security in place.

Looking back at the two examples of protection domains given above – one with full hard disk access but no networking privileges, one with disk read access and networking – a potential problem comes up. There is no reason why a Java object from one domain cannot call an object in another domain. In fact, this happens all the time when Java code makes use of the system libraries. It is important that restricted code cannot get more permissions simply by calling more privileged code, or by arranging to receive a callback from such code. The security manager has to examine the call chain and work out the *combined* permissions of all protection domains involved in the call. If any of the domains do not have enough permissions to perform the operation, a `SecurityException` will be thrown.

As we have seen, every class gets assigned to one, and only one, protection domain. This is done by the classloader as the class is loaded. To construct protection domains and decide which domain a class belongs to, the Java `SecureClassLoader` normally uses a configuration file called the **security policy file**. Before going into details about the Java classes that implement the Java 2 security model, we will first take a look at the policy file to get a better high-level feel for what the model actually can do.

The Security Policy File

The security policy file controls the permissions granted to code coming from one or more code sources. In fact, we already constructed a simple policy for the applet-signing example that simply granted all possible permissions to the applet, which is the Java 2 way of emulating the JDK 1.1 sandbox model.

A policy file consists of a `keystore` entry, which tells the policy manager where to find the public keys for signers or certificate authorities, and one or more `grant` entries. Each `grant` entry specifies permissions for code based on who signed the code, where the code is located, or a combination of both. With the latest JDK, version 1.4, you can even specify permissions based on who executes the code.

In everyday life, you will probably prefer working with the JDK's `policytool` application. The policy tool gives you a simple graphical front-end to the policy file settings, so you don't have to remember all the system permission names, targets and actions. However, in the examples below, we will edit policy files directly because this offers valuable insights into the way the Java 2 security model operates.

Anatomy of a Policy File

The example below illustrates a policy file assigning three different levels of trust. This file demonstrates most of the features you may find in security property files. We will go through the entries line by line:

```
// The keystore file is "myapp.keystore" in the "keystores" directory

keystore "keystores/myapp.keystore"

// give everyone access to the temporary directory
grant {
   permission java.io.FilePermission "C:\\tmp\\*", "read,write,delete";
};

// code from the company applets directory get limited permissions
grant CodeBase "http://www.mycompany.com/applets/*" {
   permission java.io.FilePermission "<<ALL_FILES>>", "read";
   permission java.io.FilePermission "${user.home}${/}-",
              "write,delete,execute";
   permission java.io.SocketPermission "*.mycompany.com", "connect";
   permission com.mycompany.application.CustomPermission, SignedBy "signkey";
};

/* code from anywhere on the company website, which is also signed by *
 * both my key and the authorization key, gets full permissions       */
grant CodeBase "http://www.mycompany.com/-",
SignedBy "signkey,authorizationkey" {
   permission java.security.AllPermission;
};
```

The keywords appearing in this file, such as `keystore`, `grant`, `codeBase`, `signedBy`, and `permission`, are all case-insensitive. The permission names represent case-sensitive class names.

```
keystore "keystores/myapp.keystore"
```

This is the location of the keystore file that stores the public keys of signers, necessary to authenticate signed code. It is given relative to the location of the policy file itself; in this case, it is a file called `java` in the `keystores` subdirectory of wherever the policy is located. If the policy file was retrieved from a web server, the keystore file will be retrieved from the same server. If you want, it is also possible to specify an absolute keystore location, for instance `http://www.mycompany.com/keystores/myapp.keystore`.

These three lines form a simple `grant` entry that allows all code to read, write and delete files in the temporary directory.

```
grant {
   permission java.io.FilePermission "C:\\tmp\\*", "read,write,delete";
};
```

Running executable code from this directory is not permitted because permission to `execute` has not been granted. Because neither a signer nor a codebase is given for this grant (which would be specified between the grant keyword and the open curly brace), it applies to all code irrespective of where it came from or who signed it, or indeed of whether it is signed at all.

Note how the temporary directory `c:\tmp\` is specified – the first argument to the `FilePermission`, its **target**, is not a URL but a filesystem path. Because of this, it needs to use system-specific notation (Windows, in this case). Backslashes need to be escaped as in ordinary Java strings. The second argument, its **action**, tells what the permissible actions on the target are; in this case reading, writing, and deleting files.

The next grant specifies a code base:

```
grant CodeBase "http://www.mycompany.com/applets/*" {
```

This grant is only valid for code loaded from the http://www.mycompany.com/applets/ web site directory. Code loaded from any subdirectories is *not* included. You can denote a directory and all its subdirectories by replacing the star by a dash, "-". The code base can be any URL, including `file:URL`s for locations on the local hard disk.

```
permission java.io.FilePermission "<<ALL_FILES>>", "read";
```

This permission entry grants full read access. The string `<<ALL_FILES>>` is a special constant denoting all files on the system. The next line is also a file permission entry:

```
permission java.io.FilePermission "${user.home}${/}-",
              "write,delete,execute";
```

Full access is granted to files in the user's home directory and all its subdirectories. This entry uses **property expansion**: `${user.home}` is replaced by the value of the `user.home` Java system property. If you are familiar with UNIX shell variables you will no doubt recognize the syntax. The `${/}` is a special property which will be expanded to the system-specific path separator, that is, the backslash in Windows and the forward slash in UNIX. This helps to make property files a bit more portable.

The action omits read permission, because read permission has already been granted to all files on the system in the previous line. When a single permission occurs multiple times like this, the privileges granted are those of the permissions combined. Put differently, permissions are combined using an `OR` operation.

```
   permission java.io.SocketPermission "*.mycompany.com", "connect";
```

In the next line, connections are possible to any host within the mycompany.com domain, and all ports on those hosts, but the code is not allowed to operate as a server (this would require the `listen` and `accept` actions).

```
   permission com.mycompany.application.CustomPermission SignedBy "signkey";
   };
```

The Java 2 security framework is extensible – we can create our own permissions and grant them in the policy file. Every permission name is actually the name of the Java class that implements it. In this case, a custom permission implemented by `com.mycompany.application.CustomPermission` is granted to code loaded from the company web site *and* signed with the `signkey` private key. We will take a closer look at implementing our own permissions later on in this chapter.

The next grant shows that it is possible to require multiple signatures and combine them with a code base:

```
grant CodeBase "http://www.mycompany.com/-",
SignedBy "signkey,authorizationkey" {
   permission java.security.AllPermission;
};
```

This code states that code loaded from anywhere on the company web site that is signed by both keys `signkey` *and* `authorizationkey`, gets all permissions. `java.security.AllPermission` is a special permission which implies all other permissions.

When the Java classloader loads a new Java class, it compares the signers of the class and the location it was loaded from with the `grant` entries in the policy file. The permissions granted by all matching entries are combined into the class's protection domain. Every combination of code base and signers (and, as of JDK 1.4, principals) corresponds to a domain. Whenever the class attempts to perform a privileged operation such as network access, the security manager examines the call chain to trace the path taken through the code. For each protection domain involved, it verifies that the domain actually has permission to perform this operation. If not, a `SecurityException` is thrown.

In JDK 1.4 policy files, you can specify not only `CodeBase` and `SignedBy` in your grant statements, but also `Principal classname "principalname"`, enabling you to assign permissions to code based on who executed it (the `principal`).

```
grant principal javax.security.auth.x500.X500Principal "cn=Joe" {
   permission java.net.SocketPermission "www.wrox.com:80", "connect";
}
```

This grant gives code run by Joe access to the www.wrox.com web site, provided he has been properly authenticated and the application was written to support principal-based permissions. If the principal has an entry in your keystore, there is a useful shorthand whereby you omit the class and simply specify the user's keystore alias:

```
grant principal "Joe" {
   permission java.net.SocketPermission "www.wrox.com:80", "connect";
}
```

Authentication and principals will be fully discussed in Chapter 14.

A Policy for the Security Application

Having seen a number of different ways to determine exactly what code gets granted permissions, and quite a few of those permissions themselves, it is useful to see how this can be applied to the policy file from the security demo applet.

```
keystore "file:SecurityApp.keystore";

grant SignedBy "signkey" {
  permission java.io.FilePermission "/", "read";
  permission java.net.SocketPermission "www.wrox.com", "connect";
  permission java.util.PropertyPermission "user.home", "read";
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
};
```

This policy file grants the applet precisely those permissions it needs to operate the way a normal application would – no more, no less – in the spirit of the principle of least privilege. Assuming the policy is saved as `SecurityApp.policy2`, running

```
C:\Beg_Java_Networking\Ch07>appletviewer -J"-Djava.security.policy=SecurityApp.policy2"
SecurityApp.html
```

should remove any difference between applet and application mode.

If you are operating behind a firewall, then the command prompt is as follows:

```
C:\Beg_Java_Networking\Ch07>appletviewer -J"-Dhttp.proxyHost=123.456.7.89" -J"-
Dhttp.proxyPort=99" -J"-DtrustProxy=true" --J"-Djava.security.policy=SecurityApp.policy2"
SecurityApp.html
```

where, once again, the `proxyHost` and `proxyPort` values are equal to the proxy IP address and its port that is not behind your firewall. In addition, there is a new instruction here: -`DtrustProxy=true`. Without this, an `AccessControlException` is thrown as the `SocketPermission` needs to use DNS lookup to reconcile proxy-returned data with the hostname-based permission it has. The `trustProxy` property tells it to trust the proxy for DNS lookup.

You might want to try removing a few permissions, or changing the permissions' actions, to verify that an exception gets thrown. Or add a `CodeBase` to the `grant` entry, for example:

```
grant SignedBy "signkey", CodeBase "file:C:/Program Files/-" {
```

This will cause the code to be granted permissions only if it was loaded from the `Program Files` directory, or (thanks to the dash) a subdirectory underneath it. You can find a description of the most important permissions, their targets and actions in the table a few pages onwards.

The Java 2 security model does not only work for applets, but for any Java code including applications. To enable application-mode security for the above `SecurityApp`, we need to do two things:

- Because applications do not normally run with security restrictions, we have to set the `java.security.manager` system property to indicate that the application needs to be run with a security manager. Alternatively, the application can install its own security manager using

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

- The security manager needs to be told that, rather than the system policy file, we want to use our own security policy. Setting the `java.security.policy` system property will cause an application to be run under the specified policy file.

To execute the demo as an application with a security policy, run:

```
C:\Beg_Java_Networking\Ch07>java -Djava.security.manager
-Djava.security.policy=SecurityApp.policy2 -jar SecurityDemo.jar
```

Again, if you are behind a firewall, the prompt is:

```
C:\Beg_Java_Networking\Ch07>java -Dhttp.proxyHost=123.456.7.89 -Dhttp.proxyPort=99 -
DtrustProxy=true -Djava.security.manager -Djava.security.policy=SecurityApp.policy2 -jar
SecurityApp.jar
```

It is worthwhile to experiment with the various settings and become comfortable with them. Setting up a well thought out security policy file can make networked applications more secure and help reduce the impact if your server is compromised.

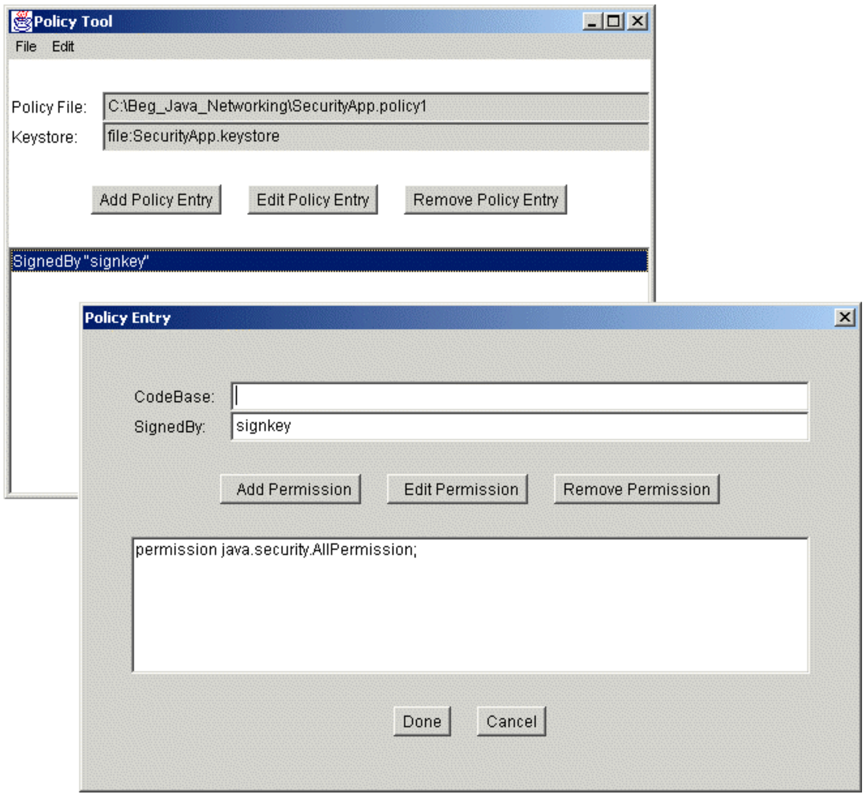
Policytool

The `policytool` is a simple graphical front-end to create and maintain policy files. The greatest advantage is that it knows about all the standard permissions, their targets, and their actions, which saves quite a bit of browsing through the documentation. Take, for example, the policy constructed in the security policies example.

```
keystore "file:SecurityApp.keystore";

grant SignedBy "signkey" {
   permission java.io.FilePermission "/", "read";
   permission java.net.SocketPermission "www.wrox.com", "connect";
   permission java.util.PropertyPermission "user.home", "read";
   permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
};
```

The following picture shows its contents when edited by the policytool:



Policy File: C:\Beg_Java_Networking\SecurityApp.policy1
Keystore: file:SecurityApp.keystore

SignedBy "signkey"

CodeBase:
SignedBy: signkey

```
permission java.security.AllPermission;
```

Filename and keystore are shown in the main screen. A slightly non-intuitive aspect is that you can't just overtype the location of the keystore file on the main screen, but have to set it using Change KeyStore in the Edit menu. Every `grant` policy entry is shown in the list underneath. Pressing Add or Edit brings up the policy entry window.

In the policy entry window, you can set the code base and signers the `grant` applies to, and add, edit, or remove the individual permissions inside it. If you have worked through the discussion of policy files, the tool should be straightforward to use.

Types of Permissions

The table below lists some of the most important permissions and targets. You can find an exhaustive listing in your JDK directory at `docs/guide/security/permissions.html`, or consult the Javadoc documentation for the various `java.security.Permission` classes. We will talk more about actions and targets in the section on *Defining a Custom Permission*:

Permission	**Target**	**Action**	**Description**
`java.security.AllPermission`	None	None	Grants every other permission, including user-defined permissions.

Permission	Target	Action	Description
`java.awt.AWT Permission`	`AccessClipboard`	None	Posts to and retrieves from the AWT clipboard.
	`AccessEvent Queue`	None	Grants access to the AWT event queue.
	`ListenToAll AWTEvents`	None	Installs an arbitrary event listener. This could allow malicious code to read keyboard input.
	`ShowWindow WithoutWarning Banner`	None	Removes the applet warning from windows.
`java.io.File Permission`	`File pattern, or <<ALL FILES>>`	`read` `write` `execute` `delete`	Grants permission for the writing, execution and deletion of files. The file pattern may contain the star (`*`) denoting all files in the directory, or the dash (`-`) for all files and subdirectories. Other than on UNIX systems, execute permission is meaningless for directories, and the `delete` option is not applicable to UNIX. `<<ALL FILES>>` gives access to all files. Code *always* has `read` access to files and subdirectories in the directory it was loaded from.
`java.net.Net Permission`	`SetDefault Authentication`	None	Sets the authentication method when a proxy or HTTP server asks for authentication.
	`RequestPassword Authentication`	None	Asks the registered authenticator for a password.
	`SpecifyStream Handler`	None	Grants ability to specify a stream handler when constructing an URL.
`java.util. Property Permission`	`Property pattern`	`read` `write`	Grants permission to read and write system properties. The last character of the property pattern may be a star (`*`), for example, `java.*`.

Table continued on following page

Permission	**Target**	**Action**	**Description**
`java.lang.RuntimePermission`	`ExitVM`	None	Exits the Java virtual machine.
	`SetFactory`	None	Sets socket factories and URL stream handlers.
	`SetIO`	None	Modifies `System.in`, `System.out` and `System.err`.
	`GetProtectionDomain`	None	Gives access to policy information for a code domain.
	`LoadLibrary`	None	Allows access to native libraries (JNI).
	`AccessDeclaredMembers`	None	Accesses reflection information for a class.
	`QueuePrintJob`	None	Initiates a print job.
`java.net.SocketPermission`	`Host:port pattern`	`accept` `connect` `listen` `resolve`	Accepts connections from a host, connects to a host, listens on a port (as in `ServerSocket`; this only makes sense for `localhost`), and provides DNS and service lookups. `resolve` is implied if any of the other three is present. In the target pattern, the host may be either a hostname or an IP address. If it is a hostname, a single wildcard may be included in the leftmost position, for instance `*.wrox.com`. The port may be a single port or a port range, for example, `hostname:1024-2048`.

System and User Policy Files

In the examples discussed so far, a -Djava.security.policy argument was passed to the Java runtime telling it which policy file to use. Two more policies are always loaded: the **system security policy** located in the JDK directory under `/lib/security/java.policy`, and the **user security policy,** `.java.policy`, located in the user's home directory. (In Windows, the default home directory is the parent directory of `My Documents`.)

These default locations are, in turn, determined by the values of some system properties:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

If you want, you can redefine these properties, or load additional policy files by defining new properties along the same lines: `policy.url.3`, and so forth. The file `${java.home}/lib/security/java.security`, where `${java.home}` is the location of the Java runtime environment, specifies the default values for these system properties.

Defining a Custom Permission

We have seen that the permissions in the security policy file are actually class names; unsurprisingly, the Java security framework uses these classes to represent the permissions at runtime. All of them directly or indirectly extend `java.security.Permission`. To create a new permission, all you have to do is subclass it.

Permission and BasicPermission

Let's first examine how the `Permission` class works. It is an abstract class with the following attributes:

- A name, returned by `getName()`; in the policy file, this was referred to as the *target* of the permission. For example, the name of a `java.io.FilePermission` is a filename pattern.
- Zero or more **actions** returned by `getActions()`, for example `read` or `execute` to indicate permission to read or execute the files matching the pattern.

You will recognize these properties from the policy file settings discussed in the previous sections. In addition, there are two more abstract methods that help the access controller to query permissions:

- An `implies(Permission)` method returns a Boolean indicating whether the argument is implied by the permission object. Taking `java.net.SocketPermission` as an example: `resolve` permission for `www.wrox.com:80` is implied by a `connect` permission for `*.wrox.com:1-1023`. The security code uses this method to check permissions and simplify permission collections.
- The `newPermissionCollection()` factory method is used by the security code to create collections holding multiple permissions of the same class. That way, a number of permissions such as `SocketPermissions` can be conveniently bundled and checked as a whole.

To create your own permission class, you need to subclass `Permission` and provide implementations for at least the `equals()`, `hashCode()`, `implies()` and `getActions()` methods.

There are a few things to watch out for when creating your own permissions:

- The `equals(Object)` and `hashCode()` contracts, as set out in the documentation of `java.lang.Object`, should be satisfied so that `Permissions` can be put in a hashtable.
- It should correctly implement `implies(Permission)` or the security manager will get all confused about whether a required permission is covered by the granted permissions or not.

- The action list returned by `getActions()` should be in *canonical form*, meaning that the same string should be returned regardless of the order in which the actions where listed in the security properties file.
- If a set of permissions can *between them* imply a permission – even if no single permission in the set explicitly implies it completely by itself – you will need to provide your own implementation of `PermissionCollection`. We will go into this in more detail when we discuss permission collections. If you don't need to implement your own `PermissionCollection`, the `newPermissionCollection()`method should simply return `null`.

If you are creating a permission that recognizes wildcard patterns in the name (target), you can probably save yourself implementing the `implies()` method. The `BasicPermission` class provides an implementation of `implies()` that recognizes wildcards in the target name, so that the name `my.permission.*` implies `my.permission.special`.

Checking the Permission

Defining a permission is only one side of the coin. Presumably, some privileged operation is being implemented where we should check that the calling code has that permission. This is done using the security manager, implemented by the `java.lang.SecurityManager` class. This class can seem daunting at first because of its bewildering array of methods, all checking for specific privileges. They actually date back to the old sandbox model. In Java 2, all of them delegate the checking to a single method, `checkPermission(Permission)`.

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
   Permission toCheck = new CheckPasswordPermission(arguments);
   security.checkPermission(toCheck);
}
```

First, the currently installed security manager is retrieved. If the code is not running under a security manager, this returns `null`, so we should remember to check for this. A `Permission` object representing the privileges we need is constructed, and `checkPermission()` is called. This method does not return a value, but simply throws a `SecurityException` if any of the calling code does not have enough privileges.

In fact, the default `SecurityManager` does in turn delegate the `checkPermission()` call to `java.security.AccessController.checkPermission(Permission)`. The `AccessController` has a number of static methods that actually implement the permission checks and a couple of other facilities, which we will examine below.

Performing the Privileged Operation

When we build code to perform a privileged function, it is vital that the client code does not have any access to the resources we need. After all, anyone can run a Java decompiler against our classes, figure out what they do, implement the functionality themselves and circumvent any security we want to impose. We could try to keep our classes secret, of course, but that still doesn't prevent a determined attacker from reverse engineering the logic. **Security through obscurity is very bad security**. In fact, openness tends to encourage security because it exposes your algorithms to peer review, and prevents you from lazily relying upon your algorithm being "secret".

This immediately runs us into problems. Suppose for a moment that we need to access a database over a network. If the client code calling our function does not have network access to the database machine, then neither have we. Remember, the security model requires that *all* the code in the call chain has the required privileges. Clearly we need a way to perform a **privileged operation** subject to the protection domain of our class only, disregarding any code further up the chain.

The `java.security.AccessController` class provides a number of static methods:

- `checkPermission(Permission)` checks whether all code in the call chain has sufficient privileges to perform the operation specified by the permission, as described previously.
- A number of `doPrivileged(PrivilegedAction,...)` methods give the ability to execute a section of code without being restricted by the protection domains of the calling code.
- `getContext()` returns an `AccessControlContext`, which essentially is a snapshot of the current permissions, looking at all the code in the call chain. This can be useful if the actual permission checking needs to happen in another thread, where the current call chain is not available.

Clearly, the `doPrivileged(PrivilegedAction)` method provides exactly the facility we need. The privileged code is encapsulated in an object implementing the `PrivilegedAction` interface. This interface is virtually identical to `Runnable`, except for the fact that the `run()` method is not defined as `void` but returns an `Object`. In the simplest cases, you can instantiate an inner class:

```
AccessController.doPrivileged(new PrivilegedAction() {
   public Object run() {
      // start of privileged code
      doDatabaseAccess();
      return null; // no return value
      // end of privileged code
});
```

For actions that need to throw exceptions we can use `PrivilegedExceptionAction`. We will take a closer look at this class after an example.

Putting It All Together

To see the entire process in action, we will use a password-checking library example. It will read its passwords from a simple, unencrypted password file. This file should be unreadable to normal code so that a client program will only have access to it through the library. The library will never directly return a user's password, but simply `true` or `false` to indicate whether the password is correct. The password verification process itself will be regarded as a privileged operation, subject to a `CheckPasswordPermission`. Finally, a simple command line application is written to allow some basic testing of the library.

The first ingredient is the password-checking permission. The password library will need to be signed, so its classes need to live in a separate `password` package.

```
//CheckPasswordPermission.java

package com.wrox.password;

import java.security.*;
```

```
public class CheckPasswordPermission extends Permission {

  public CheckPasswordPermission() {
    super("<CheckPasswordPermission>");
  }

  public CheckPasswordPermission(String name, String actions) {
    this();
  }

  public boolean equals(Object obj) {
    return obj instanceof CheckPasswordPermission;
  }

  public String getActions() {
    return "";
  }

  public boolean implies(Permission p) {
    return equals(p);
  }

  public int hashCode() {
    return CheckPasswordPermission.class.hashCode();
  }
}
```

There is little to this class except very basic implementations for the abstract methods in `Permission`. The actual password-checking code is a bit more involved. The `CheckPassword` class living in the password package is responsible for performing the actual check:

```
//CheckPassword.java

package com.wrox.password;

import java.io.*;
import java.net.*;
import java.security.*;

public class CheckPassword {
```

The following is little more than a `doPrivileged()` wrapper around the second method. Ensuring that the caller has `CheckPasswordPermission` takes just a few lines of code. Note that the Boolean value returned by `privilegedCheck()` needs to be "boxed" inside a Boolean object because `PrivilegedAction.run()` cannot return a primitive type.

```
  public boolean check(final String username, final String password,
                       final String passwordLocation) {

    // First verify that the callers have permission to check passwords
    SecurityManager security = System.getSecurityManager();
```

```
      if (security != null) {
        security.checkPermission(new CheckPasswordPermission());
      }

      // Then check the password in a privileged code block
      Boolean passwordOk =
        (Boolean) AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() {

          // start of privileged code
          try {
            boolean ok = privilegedCheck(username, password,
                                         passwordLocation);
            return new Boolean(ok);
          } catch (IOException e) {
            return Boolean.FALSE;
          }
          // end of privileged code
        }
      });

      return passwordOk.booleanValue();
    }
```

The meat of the password checker is the `privilegedCheck()` method. It opens the password file from any URL, and attempts to find the given username in it. Once it has found the user, it compares the password to the one passed into the function. We would normally make this method private, but it is declared public here so that the test program can call it for demonstration purposes.

```
    public boolean privilegedCheck(String username, String password,
                                   String passwordLocation) throws IOException {
      String toSearch = username + ":";
      String passwordFound = null;
      URL passwordUrl = new URL(passwordLocation);
      BufferedReader passwords =
        new BufferedReader(new InputStreamReader(passwordUrl.openStream()));

      try {
        while (true) {
          String line = passwords.readLine();
          if (line == null) {
            break;

          }
          if (line.startsWith(toSearch)) {
            passwordFound = line.substring(toSearch.length());
            break;
          }
        }
      }
      finally {
        passwords.close();
      }
      return passwordFound != null && password.equals(passwordFound);
    }
  }
```

The `CheckPassword` and `CheckPasswordPermission` classes make up the password-checking library. After compilation, they need to be packaged in a JAR and signed so they can be given exclusive permission to read the password file:

```
C:\Beg_Java_Networking\Ch07>javac com\wrox\password\*.java
```

```
C:\Beg_Java_Networking\Ch07>jar cf CheckPassword.jar com\wrox\password\*.class
```

```
C:\Beg_Java_Networking\Ch07>jarsigner -keystore SecurityApp.keystore CheckPassword.jar
signkey
```

We will assume the archive is signed with the `signkey` signature generated for the security application above. The `CheckPassword` class needs to access a password file, consisting of a list of *`username`*`:`*`password`* combinations, one per line. For example:

```
bill:clinton
joe:secret
```

In this example, the name of the file should be `password.txt`. The use of the library can be demonstrated with a simple command line client called `CheckPasswordApp`:

```
import com.wrox.password.*;

public class CheckPasswordApp {

  public static void main(String args[]) {
    if (args.length == 2) {
      try {

        // install a security manager if none has been installed yet
        if (System.getSecurityManager() == null) {
          System.setSecurityManager(new SecurityManager());
        }

        // check the password
        CheckPassword checker = new CheckPassword();
        System.out.println(checker.check(args[0], args[1],
                                         "file:password.txt"));
      } catch (Exception e) {
        System.out.println(e);
      }
    } else {
      System.out.println("Arguments: username password");
    }
  }
}
```

After compilation, the application needs to be packaged in a JAR with the following manifest file, saved as `CheckPasswordApp.manifest`:

```
Manifest-Version: 1.0
Main-Class: CheckPasswordApp
Class-Path: CheckPassword.jar
```

This tells the Java interpreter that the password application needs to use classes from `CheckPassword.jar`.

Next, we compile the class and create `CheckPasswordApp.jar`:

```
C:\Beg_Java_Networking\Ch07>javac CheckPasswordApp.java
```

```
C:\Beg_Java_Networking\Ch07>jar cfm CheckPasswordApp.jar CheckPasswordApp.manifest
CheckPasswordApp.class
```

`CheckPasswordApp` has a `CheckPasswordPermission` but is otherwise unprivileged code, so there is no need to sign this JAR. To set up the permission, and allow the password library free access to the password file, a security policy needs to be set up:

```
//CheckPassword.policy

keystore "file:SecurityApp.keystore";

// give the password library permission to read the password file
grant CodeBase "file:CheckPassword.jar", SignedBy "signkey" {
   permission java.io.FilePermission "password.txt", "read";
};

/* give password-checking permission only to locally installed code, but *
 * not to anything loaded over the internet                              */
grant CodeBase "file:/-" {
   permission com.wrox.password.CheckPasswordPermission;
};
```

Assuming this policy file is saved as `CheckPassword.policy`, the demonstration application can be run by typing:

```
C:\Beg_Java_Networking\Ch07>java -Djava.security.policy=CheckPassword.policy –jar
CheckPasswordApp.jar joe secret
```

The application should respond with

```
true
```

to indicate that joe does indeed have the password secret. It is worth experimenting with a couple of user names and passwords to convince yourself that it really works.

The fact that normal, unprivileged code really does not have the ability to directly read the password file can be verified by changing `CheckPasswordApp` and replacing the call to `check()` by a call to `privilegedCheck()`:

```
            CheckPassword checker = new CheckPassword();
            System.out.println(checker.privilegedCheck(
               args[0], args[1], "password.txt"));
         }
```

Upon recompiling and running the example, an `AccessControlException` is now thrown because the `CheckPasswordApp` class is not allowed to access the password file. The same exception is thrown when the `CheckPasswordPermission` is commented out of the policy file so that the application is no longer allowed to access the password-checking library.

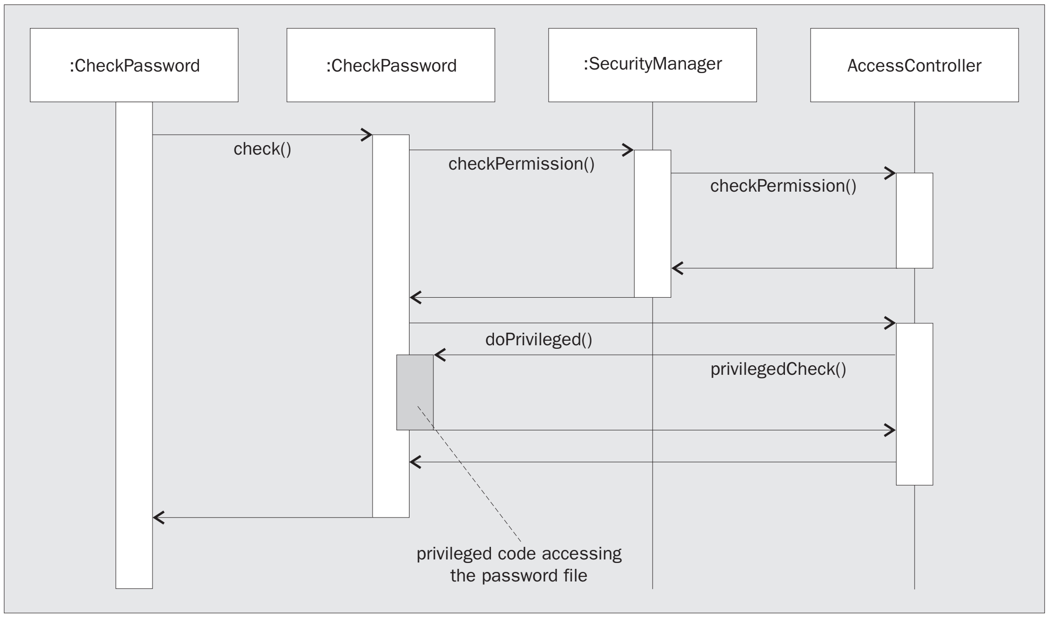
You may want to experiment some more with the policy file settings. For example, to retrieve the password file from the password.mycompany.com web server, the first `grant` in the policy file would need to be modified to:

```
// give the password library permission to read the password file
grant CodeBase "file:CheckPassword.jar", SignedBy "signkey" {
  permission java.net.SocketPermission "password.mycompany.com:80",
                                      "connect";
};
```

SecurityManager, AccessController and All That

To understand the roles of protection domains, the security manager, and the `AccessController` class, we can stick with the example and track its progress through the internals of the Java security system. When the classloader loads the classes from `CheckPassword.jar` and `CheckPasswordApp.jar`, it assigns them to two different protection domains, both characterized by their code base and signers.

Predictably, these protection domains are represented by `ProtectionDomain` objects. The protection domain of `CheckPassword.jar` has the `FilePermisson` necessary to access the password file, while both domains are granted a `CheckPasswordPermission`.



The sequence diagram above illustrates what is happening in the example. The vertical "swimlanes" represent the objects; time flows from top to bottom. The arrows represent the method calls taking place. The gray area is where the program is executing the `privilegedAction()` (itself not drawn, because it is little more than a call to `privilegedCheck()`). In this area, it is subject only to the privileges of `CheckPassword`'s protection domain and therefore able to read the password file.

The first thing `CheckPassword.check()` does is call the security manager to verify that the calling code does have password-checking permission. If you want this check to happen regardless whether there is a security manager installed or not, you can also call `AccessController.checkPermission()` directly.

When the main class calls `CheckPermission.check()`, the code would not normally be allowed to open and read the password file. The security manager would find `CheckPermissionApp()` in the call chain, which does not have the necessary `FilePermisson`. This is why replacing the call to `check()` by one to `privilegedCheck()` led to an `AccessControlException`.

For this reason, the `AccessController.doPrivileged()` method is used to execute the `privilegedCheck()` in the protection domain of `CheckPassword.jar`, disregarding all domains further up in the call chain. However, if the code inside a `PrivilegedAction` would make further calls to classes in other protection domains, the permissions of those domains *would* be imposed.

Finally, it is worth noting at this point that `CheckPassword` cannot reside in the same package as `CheckPasswordApp`. When a class is loaded from a signed archive, the classloader demands that *all* of the classes in that package are signed with the same key. This prevents malicious code from compromising security by redefining classes in the package.

Throwing Exceptions in Privileged Operations

The password-checking implementation given above has one problem: it does not handle exceptions very well. If there is a problem reading the password file, for example because it doesn't exist or because there are errors on the disk or network, the `check()` method simply returns `false`. It would be much better to propagate the exception to the calling code, and fortunately the security framework provides a way of doing this.

The `PrivilegedExceptionAction` interface is just like `PrivilegedException`, except that the `run()` method is defined to throw `Exception`. The `doPrivileged()` method will wrap the exception inside a `PrivilegedActionException` and throw that. In your code, you can catch the exception in `CheckPassword.java`, recover the original exception, and re-throw it:

```
public boolean check(final String username, final String password,
                     final String passwordLocation)
throws IOException {
   // First verify that the callers have permission to check passwords
   SecurityManager security = System.getSecurityManager();
   if (security != null) {
      security.checkPermission(new CheckPasswordPermission());
   }

   // Then check the password in a privileged code block
   try {
      Boolean passwordOk = (Boolean)
         AccessController.doPrivileged(new PrivilegedExceptionAction() {
```

```
                  public Object run() throws IOException {
                     // start of privileged code
                     boolean ok = privilegedCheck(username, password,
                                                  passwordLocation);
                     return new Boolean(ok);
                     // end of privileged code
                  }
               });

            return passwordOk.booleanValue();
         }
         catch(PrivilegedActionException e) {
            throw (IOException) e.getException();
         }
      }
```

This technique does require you to keep careful track of the exceptions that can be thrown by your `PrivilegedExceptionAction`. If you modify your code to throw a new type of exception and forget to cater for it in the final catch clause, the compiler will not warn you; a `ClassCastException` will be thrown at runtime. Because this is in exception-handling code that can be difficult to test, it is easy for this bug to slip through unnoticed.

Guarding Access from Other Threads

Sometimes, checking if the calling code has the right permissions isn't quite as easy as calling the security manager. Take another look at the relevant code from `CheckPassword`:

```
         // First verify that the callers have permission to check passwords
         SecurityManager security = System.getSecurityManager();
         if (security != null) {
            security.checkPermission(new CheckPasswordPermission());
         }
```

To determine if the caller has `CheckPasswordPermission`, it examines all the protection domains involved in the call chain. But what if `CheckPassword` was a system service running in a different thread? The relevant call chain would not be available in this thread and it would be impossible to determine its domains. The security check would have to be delayed until *after* the privileged operation has completed and returned its result, when we are back in the calling thread and its protection domain information is at our disposal.

To give another example, the library could be made available to other applications as a remote service with an RMI interface. The password checking service would have no information about the permissions available to the client code, yet it needs this information to determine whether the client is allowed to check passwords.

To achieve this, we return the result in a **guarded object**. The `java.security.GuardedObject` class acts as a "safe" in which the result of the call is locked away. When the client code attempts to retrieve the result by calling `getObject()`, the guarded object calls the **guard**, which then performs the necessary security checks first. `Guard` is a simple interface with a single `checkGuard(Object)` method. Your security check has to implement this interface.

Sticking with the password-checking library as an example, this is how you implement a simple guard:

```
import java.security.*;

public class CheckPasswordGuard implements Guard {

   public void checkGuard(Object toGuard) throws SecurityException {
      SecurityManager security = System.getSecurityManager();
      if (security != null) {
         security.checkPermission(new CheckPasswordPermission());
      }
   }
}
```

With this guard, returning a guarded object with the Boolean result of the password check takes just a single line of code:

```
return new GuardedObject(passwordOk, new CheckPasswordGuard());
```

The permissions are now checked only when the calling code attempts to retrieve the result from the `GuardedObject`.

In fact, if you only have a single permission to check you do not even need to create your own `Guard` as the `Permission` class already implements it! The implementation is basically identical to that of `CheckPasswordGuard` above. This class can therefore be scrapped and the code simplified even further to:

```
return new GuardedObject(passwordOk, new CheckPasswordPermission());
```

Of course, the password-checking library does not actually need guarded objects as everything is handled in a single thread, but in network services that typically make heavy use of multi-threading or remote method calls you may well encounter this technique.

Permission Hierarchies

If you need to implement multiple permission classes in your application, you could of course make all of them subclasses of `java.security.Permission`. It is recommended, however, that in those cases you insert your own base class in the hierarchy:

```
package com.wrox.password;

public abstract class PasswordLibraryPermission
extends java.security.Permission {
   // insert whatever part of the implementation can be shared by your subclasses
}
```

It is then possible to subclass your permission classes off that. You could also extend `BasicPermission` or any other permission class if that is more convenient.

Permission Collections

In most of the code we write, we aren't interested in the individual permissions granted to code. Instead, we have something we want to do, create a `Permission` that encapsulates the privileges we need to do it, and are only interested in the question whether the permissions associated with a protection domain imply the `Permission` we need. In other words, in most cases we are only interested in the permissions as a group. An important part of the security API comes down to grouping and querying `Permission` collections.

PermissionCollection

The `Permission` class declares a factory method `newPermissionCollection()` which produces a collection object for its own permission type. What is so special about this collection? Why not a simple `Vector` or `ArrayList`?

The abstract class `PermissionCollection` is what the name suggests: a collection of `Permission` objects. Unlike a `Vector` or `ArrayList`, it has an `implies()` method that determines whether a given permission is granted by the `Permissions` in the collection. Permission collections are, in general, permission-specific, exactly because of this `implies()` method. Consider for example the following fragment of a security policy file:

```
    permission java.io.FilePermission "<<ALL_FILES>>", "read";
    permission java.io.FilePermission "${user.home}${/}-",
               "write,delete,execute";
```

What if we need to determine whether we can read from and write to files in the directory `/home/jack/`?

```
  permissonCollection.implies(
      new java.io.FilePermission("/home/jack/*", "read,write"));
```

If Jack is indeed the current user, this should return `true`. Although neither permission in the policy file actually implies it by itself, between them they *do* imply the permission.

It is the job of `PermissionCollection.implies(Permission)` to figure this out. There is no simple generic algorithm to do this; the implementation depends on the way the targets and actions of a particular type of permission are organized. That is why these collections are necessarily permission-specific and **homogeneous**, containing only a single class of permission. When you are creating a `Permission` with non-trivial implication rules, you need to implement a `PermissionCollection` to be returned by the `newPermissionCollection()` method of your permission class.

Permissions

Homogeneous collections alone are not enough, though. Looking at a protection domain, the set of permissions it encapsulates is **heterogeneous**, consisting of many different types of permissions. How to reconcile this with the argument that a collection is permission-specific?

The answer is that a heterogeneous `PermissionCollection` must not contain any `Permission` objects directly – it would be extremely difficult to write a generic `implies()` method – but only further, homogeneous `PermissionCollections`. You would no longer need any logic to figure out if permissions interact to grant a given `Permission`, because that can be left completely to the individual homogeneous sub-collections.

The `java.security.Permissions` class implements such a heterogeneous permission collection. When you add a permission to it, it will put it in the appropriate homogeneous subcollection, instantiating this collection if necessary.

Summary

In this chapter, we thoroughly explored the Java security model. We discussed:

- ❑ Why security is important, especially when running executable content from over the network, or when offering services to a network
- ❑ The "sandbox" security model and the difference between applets and applications, both in terms of trustworthiness and in terms of the privileges granted to them by the JDK
- ❑ Packaging and signing code using secure cryptographic signatures
- ❑ How the code base and signers of Java code are mapped to protection domains and permissions using the Java security policy file in the Java 2 security model
- ❑ Constructing a policy file, and the meaning of the most important permissions supported by the Java library
- ❑ Securely implementing your own privileged operations by defining your own `Permission`, checking that the calling code has this permission, and accessing the protected resource using `AccessController.doPrivileged()`
- ❑ How exceptions thrown in privileged code can be propagated to the calling code
- ❑ How security can be enforced using guarded objects if a privileged service does not execute in the caller's thread
- ❑ Coding a permission collection when your `Permissons` imply each other in nontrivial ways
- ❑ What the JDK security tools do

Armed with this knowledge, we are better able to implement networked services in a secure way.