Introduction to the Capabilities Classes

Recycled and Recyclable Paper

The Team:
Engineering:
Marketing:
Publications:
Quality Assurance:
Technical Support:

Version 1.0

©Netscape Communications Corporation 1997
All Rights Reserved

Printed in USA
99 98 97 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

Introduction to the Capabilities Classes

This document is intended for Java developers who are writing applets or libraries that need access to local system resources, such as read/write file access or the ability to establish an arbitrary network connection.

This document introduces the basic Capabilities classes and their most important methods. It does not include reference information for the Capabilities classes. For the complete Capabilities reference, seehttp://developer.netscape.com/library/documentation/signedobj/javadoc/Package-netscape_security.html.

Before you read this document, you should have a basic understanding of digital signatures, certificates, and JAR archive files. You should also be familiar with the object-signing tools described at http://developer.netscape.com/software/signedobj/jarpack.html. These tools allow you to create a JAR archive and digitally sign the files it contains. You should also read "Netscape Object Signing: Establishing Trust for Downloaded Software" (http://developer.netscape.com/library/documentation/signedobj/trust/index.htm), which introduces the concepts you need to understand before you read this document.

Overview of the Capabilities Classes

The Capabilities classes add facilities to and refine the control provided by the standard Java `SecurityManager` class. Java developers can use these classes to exercise fine-grained control over an applet's activities beyond the "sandbox"—the Java term for the carefully defined limits within which Java applets must otherwise operate.

The Capabilities classes take advantage of Communicator client software's ability to

- validate digital signatures associated with files in a JAR archive
- maintain a list of signing certificates and the kinds of access the user (or a system administrator) has decided to allow or disallow for entities represented by those certificates

All access-control decisions boil down to who is allowed to do what. In the Capabilities model, a **principal** represents the "who," a **target** represents the "what," and the **privileges** associated with a principal represent the authorization (or denial of authorization) for a principal to access a specific target.

A principal consists of an instance of class `Principal` and typically represents a signing certificate**.** A target consists of an instance of class `Target` and typically represents one or more system resources, such as reading files stored on a local disk or sending email on your behalf.

The Capabilities classes make it possible to determine whether any given principal is allowed to access the local system resources represented by a given target. The answer is expressed by an instance of class `Privilege`, which states whether access is allowed and, if so, for how long.

In addition to recognizing signing certificates as principals, Communicator can recognize codebases as principals. From a user's point of view, this is of interest primarily for system administrators using the Netscape Administration Toolkit provided with Communicator Pro, because it allows them to establish codebases within the intranet that can be treated as principals with certain capabilities. From a developer's point of view, treating codebases as principals is an essential part of developing code that uses the Capabilities classes, because it allows you to test unsigned code.

Communicator doesn't automatically recognize codebases as principals. To enable this preference, add the following line to the file `prefs.js` (or Netscape Preferences on Mac OS):

```
user_pref("signed.applets.codebase_principal_support", true);
```

When you next launch Communicator, it will allow you to add codebases to its list of principals.

Running a Signed Applet

Consider what happens when a user clicks a URL that downloads a JAR archive containing signed Java class files. For the purposes of this example, assume that the Communicator client has the signer's certificate in its list of accepted signing certificates. Communicator first checks the digital signatures in the JAR archive to validate the identity of the entity whose certificate was used to sign the files (the principal) and to make sure the content of the files hasn't changed since they were signed. If these operations are successful, Communicator keeps track of the information for later use and the applet starts to run.

It's important to remember that an applet signed by a known principal is not automatically allowed access to the targets for which it has been authorized. Instead, the applet must explicitly request the privileges it needs. An applet that is not signed at all runs "inside the sandbox" as applets always have and does not have extended access to system resources.

The **Privilege Manager**, with the aid of the Communicator client, keeps track of which principals are allowed to access which targets at any given time. Signed applets use an instance of class `PrivilegeManager` to request privileges for particular targets. Communicator associates each principal with a list of targets for which privileges may or may not be permitted. If the client's list of principals and their permitted targets includes the applet's principal and the requested target, the Privilege Manager grants the privilege without interrupting the user. If the applet's principal has not previously been granted or denied a privilege for the requested target, the client displays a dialog box asking the user whether to grant that privilege to that principal.

Granting Versus Enabling Privileges

The Privilege Manager enforces a distinction between **granting** a privilege and **enabling** a privilege. An applet that has been granted a privilege has a potential power that is not yet turned on. An applet that successfully enables a previously granted privilege has the privilege's power available and ready to use.

A privilege can be granted only if the client software has previously associated that principal with a privilege for that target or if the user explicitly grants that privilege when the applet requests it. In both cases, after the applet's first successful request for the privilege, that privilege is both granted and enabled.

However, although the privilege typically remains granted for the rest of the time the applet is running, it is enabled only for the duration of the method call in which the privilege is requested. Subsequent method calls that require the same kind of access must therefore request the privilege again. In those cases, the privilege has already been granted (so no dialog box, for example, would have to be displayed), and the request simply enables the privilege—that is, allows the applet to access the target.

Fine-Grained Access Control

In effect, after its principal has been granted some privileges, an applet can turn those privileges on, use them, and turn them off again, without interrupting the user's activities.

This is an important distinction between the Java-based Capabilities model and other schemes for allowing signed code to access local system resources. Instead of requiring users to allow code either unlimited access or no access at all, the Capabilities model allows the user to control the degree of access. This approach permits a fine-grained continuum of access privileges from relatively innocuous operations, such as read/write access to a single file, to full-blown powers over many different aspects of the local system.

From a programmer's point of view, the Capabilities model limits execution of code with potentially dangerous powers in two other ways:

- **Lexical limits.** The design of the Capabilities classes makes it possible to enable access only for those lines of code that actually require it.

- **Temporal limits.** Once it is has enabled specific privileges, an individual method retains those privileges only while it is running. After the method returns, access is no longer possible.

These limits on the execution of code with enabled privileges not only reduce opportunities for accidental or malicious damage to the user's system, but also reduce the number of lines of code that you must scrutinize for possible security problems.

Basic Capabilities Operations

This section introduces a few of the methods defined by class `PrivilegeManager` and class `Target`. These are the only interfaces that most Java developers need to obtain extended access, under the user's control, to local system resources.

The easiest way to enable your privilege for a given system target is to pass the name of the target to the static `PrivilegeManager` method `enablePrivilege()`. This method takes a string describing a standard system target, creates an instance of the target, and enables its privileges. For example, this call enables terminal emulation privileges:

```
PrivilegeManager.enablePrivilege("UniversalTerminalEmulator");
```

For a list of the names of standard system targets and some of the Java calls that require access to them, see http://developer.netscape.com/library/documentation/signedobj/targets/index.htm.

When you enable a privilege, it is stored as an annotation on the stack frame for the method from which you enabled it. When the method that called `enablePrivilege()` returns, the annotation is automatically removed. Thus, if you write a subroutine specifically to enable your privileges, those privileges disappear when that subroutine returns and your code can no longer use them. Instead, you should either enable privileges within the same method where you intend to use them or enable them in a method that calls another method where the privileges are used.

When your Java code attempts to access a restricted system target, Communicator checks for the appropriate annotation on the stack frame for the currently executing method. If it doesn't find the annotation there, it looks through any stack frames earlier on the call stack until it finds the appropriate annotation. If it can't find the annotation that corresponds to the specified target, it displays a dialog to get the user's approval for access to the requested target. If the user decides to grant the request, Communicator adds the appropriate annotation to the stack frame; if not, Communicator generates a `ForbiddenTargetException`.

In general, the placement of your calls to `enablePrivilege()` determines how long potentially dangerous privileges are enabled. Depending on how you structure your program, you can restrict potentially dangerous access privileges to just a few lines of code, to multiple ranges of code, or to almost all of your program (though the last is not normally recommended). You must take responsibility for determining where in your program to enable privileges in a way that best protects the user from malicious or accidental access to system resources.

In many cases it's easiest to enable privileges right before you need them. For an example of this approach, see Enabling Right Before You Need a Privilege.

In other cases it makes more sense to enable privileges in your initialization method, which gets any potential dialog box out of the way, then enable them again in each place that you need them. For an example of this approach, see Enabling During Initialization.

If one of your own public methods requires access privileges, you should call `enablePrivilege()` right before you call the public method, rather than calling it from within the method. A call to `enablePrivilege()` inside a public method makes it possible for someone else's code to call your method and make it do something you did not intend. For an example of how to deal with public methods that require access privileges, see Preventing Subversion of Public Helper Methods

In addition to `enablePrivilege()`, there are two other Privilege Manager methods that you can use to restrict enabled privileges even further: `revertPrivilege()` and `disablePrivilege()`.

Use `revertPrivilege()` to turn off specified privileges after they have been turned on with a call to `enablePrivilege()`. Calling `revertPrivilege()` simply removes the stack annotation for the method that made the call. Annotations earlier on the call stack aren't affected.

For example, suppose privileges for a target have been granted for a session, then enabled within a particular method. If you then pass the same target to `revertPrivilege()` later in the same method, those privileges are still granted, but they are no longer enabled for that method from that point on. To enable them, you need to call `enablePrivilege()` again. You can use `enablePrivilege()` and `revertPrivilege()` within the same method to bracket just a few lines of code that actually perform access operations. For an example of the use of `revertPrivilege()`, see Enabling Then Reverting.

The `disablePrivilege()` method places an annotation on the caller's stack frame that forbids access to the specified target. This has the effect of shadowing a positive annotation for the same target earlier on the call stack. After you call `disablePrivilege()`, the privileges in question are not unavailable, and Communicator returns a `ForbiddenTargetException` if the method attempts to access the target that has been disabled, without giving the user any choice in the matter. A subsequent call to `revertPrivilege()` in the same method removes the negative annotation, making it possible to grant and enable privileges again or to take advantage of annotations earlier on the call stack. Alternatively, a subsequent call to `enablePrivilege()` in the same method replaces the negative annotation with the appropriate positive annotation (assuming the call succeeds and the user doesn't explicitly deny access). For an example of the use of `disablePrivilege()`, see Enabling Then Disabling.

Class `PrivilegeManager` defines several versions of the method `checkPrivilegeGranted()`, which allows you to find out whether a privilege has been granted before you attempt to enable it. This can be useful when you have used the `Target` methods `findTarget()` and `registerTarget()` to define your own target. You can use `checkPrivilegeGranted()`to check whether you have been granted privileges before attempting to access your target. For an example of the use of `checkPrivilegeGranted()`, `findTarget()` and `registerTarget()`, see Creating a User Target and Checking Its Privileges.

Class `PrivilegeManager` also defines several versions of `enablePrivilege()` that can take different combinations of parameters. For example, if your applet is signed with several different signing certificates and therefore has several different principals, it may be useful to specify which principal's privileges you want to enable.

For the complete Capabilities reference documentation, see http://developer.netscape.com/library/documentation/signedobj/javadoc/Package-netscape_security.html.

Using the Capabilities Classes

The sections that follow illustrate the following uses of the Capabilities classes:

- Enabling Right Before You Need a Privilege
- Enabling During Initialization
- Preventing Subversion of Public Helper Methods
- Enabling Then Reverting
- Enabling Then Disabling
- Creating a User Target and Checking Its Privileges

The code samples described in this document are all designed to run within an IFC application called `SecSampleHarness`, which provides a wrapper and a user interface for the samples. This arrangement allows the individual samples to remain uncluttered while still providing a meaningful interface for demonstration purposes. The `SecSampleHarness` application can load any available class (including test classes you write yourself) whose name is typed into the Sample Class Name field, provided that the class implements the `netscape.sample.security.RunnableSample` interface.

To run any of the samples discussed in this document from within the `SecSampleHarness` application, see http://developer.netscape.com/library/documentation/signedobj/secsampleharness/SecSampleHarness.html.

Enabling Right Before You Need a Privilege

The simplest way to enable a privilege is to do so right before you need to use it. For example, some programs occasionally need to read properties of the local system that are normally kept private, such as the user name and current directory. The simplest way to do this is to enable the target `UniversalPropertyRead` right before the code that actually reads the desired property. Listing 0.1 shows how to do this.

Listing 0.1 Enabling a privilege right before you need it

```
// SimplePrivSample.java

package netscape.sample.security;

import netscape.security.PrivilegeManager;
import java.io.*;

public class SimplePrivSample
   implements RunnableSample
{
   public void init(PrintStream ps) { // not used in this example
   }

   public void run(PrintStream ps) {
      ps.println("Trying to acquire permission to read system
                  properties...");
      try {
         PrivilegeManager.enablePrivilege("UniversalPropertyRead");
         ps.println("\tSuccess!");
      } catch (netscape.security.ForbiddenTargetException e) {
         ps.println("\tFailed! Permission to read system properties
            denied by user.");
      } catch (Exception e) {
         ps.println("\tFailed! Unknown exception while enabling
            privilege.");
         e.printStackTrace(ps);
      }

      ps.println();
      String property = "user.home";
      ps.println("Trying to get system property (" + property + ")...");
      try {
         String propertyValue = System.getProperty(property);
         ps.println("\tSuccess!");
         ps.println("\t" + property + " = " + propertyValue);
      } catch (netscape.security.AppletSecurityException e) {
         ps.println("\tFailed! Security Violation.");
         e.printStackTrace(ps);
      } catch (Exception e) {
         ps.println("\tFailed! Unkndown exception while accessing
                     property.");
         e.printStackTrace(ps);
      }
   }

   public void finish(PrintStream ps) { //not used in this example
   }
}
```

To run Listing 0.1, go tohttp://developer.netscape.com/library/documentation/signedobj/secsampleharness/SecSampleHarness.html, choose `netscape.sample.security.SimplePrivSample` from the pop-up menu, and click Run.

Enabling During Initialization

In many cases it may be more convenient to enable privileges in your initialization method—thus getting any dialog boxes out of the way—then enable privileges again in each place that you need them. Listing 0.2 shows an example of this approach.

Listing 0.2 Enabling privileges in an initialization method

```
// InitPrivSample.java

package netscape.sample.security;

import netscape.security.PrivilegeManager;
import java.io.*;

public class InitPrivSample
   implements RunnableSample
{
   booleanhasPrivilege = false;

   public void init(PrintStream ps) {
      ps.println("Trying to acquire permission to read system
                  properties...");
      try {
         PrivilegeManager.enablePrivilege("UniversalPropertyRead");
         hasPrivilege = true;
         ps.println("\tSuccess! Privilege is enabled.");
      } catch (netscape.security.ForbiddenTargetException e) {
         ps.println("\tFailed! Permission to read system properties
            denied by user.");
      } catch (Exception e) {
         ps.println("\tFailed! Unknown exception while enabling
            privilege.");
         e.printStackTrace(ps);
      }

      ps.println();
   }

   public void run(PrintStream ps) {
      if(hasPrivilege == false) {
         // if hasPrivilege is false, then you know that the privilege
         // either has not been granted or is not available.
         ps.println("Necessary privileges have not been granted.
            Aborting.");
         return;
      }
      // If you get this far, then you know that the privilege is
      // available and granted. Go ahead and activate it.
      PrivilegeManager.enablePrivilege("UniversalPropertyRead");

      String property = "user.home";
      ps.println("Trying to get system property (" + property + ")...");
      try {
         String propertyValue = System.getProperty(property);
         ps.println("\tSuccess!");
         ps.println("\t" + property + " = " + propertyValue);
      } catch (netscape.security.AppletSecurityException e) {
         ps.println("\tFailed! Security Violation.");
         e.printStackTrace(ps);
      } catch (Exception e) {
         ps.println("\tFailed! Unkndown exception while accessing
            property.");
         e.printStackTrace(ps);
      }
   }

   public void finish(PrintStream ps) { // not used in this example
   }
}
```

To run Listing 0.2, go to http://developer.netscape.com/library/documentation/signedobj/secsampleharness/SecSampleHarness.html, choose `netscape.sample.security.InitPrivSample` from the pop-up menu, and click Run.

Preventing Subversion of Public Helper Methods

In general, it's a good idea to use private methods wherever possible to prevent subversion of your code. If you need to use a public helper method that requires access privileges, however, you should call `enablePrivilege()` right before you call the public method, and *not* from within the public method itself. If instead you call `enablePrivilege()` within the same public method that exercises the enabled privileges, you effectively export your code's special powers to any other code that calls that public method.

To guard against potential attacks that subvert your code's privileges in this way, you should always enable privileges for public methods as shown in Listing 0.3.

Listing 0.3 Preventing subversion of public helper methods

```
// PrivWrapperSample.java

package netscape.sample.security;

import netscape.security.PrivilegeManager;
import java.io.*;

public class PrivWrapperSample
   implements RunnableSample
{
   booleanhasPrivilege = false;

   public void init(PrintStream ps) {
      ps.println("Trying to acquire permission to read system
            properties...");
      try {
         PrivilegeManager.enablePrivilege("UniversalPropertyRead");
         hasPrivilege = true;
         ps.println("\tSuccess! Privilege is enabled.");
      } catch (netscape.security.ForbiddenTargetException e) {
         ps.println("\tFailed! Permission to read system properties
            denied by user.");
      } catch (Exception e) {
         ps.println("\tFailed! Unknown exception while enabling
            privilege.");
         e.printStackTrace(ps);
      }

      ps.println();
   }

   public void run(PrintStream ps) {
      if(hasPrivilege == false) {
         // if hasPrivilege is false, then you know that the privilege
         // either has not been granted or is not available.
         ps.println("Necessary privileges have not been granted.
            Aborting.");
         return;
      }
      String property = "user.home";

      ps.println("Attempting to get property without calling
         enablePrivilege();");
      String propertyValue = getSysProperty(property, ps);

      ps.println();
      ps.println("Now enabling privilege and trying again.");
      PrivilegeManager.enablePrivilege("UniversalPropertyRead");
      propertyValue = getSysProperty(property, ps);
   }

   public void finish(PrintStream ps) { //not used in this example
   }

   // this is the public helper method that exercises previously enabled
   // privileges for UniversalPropertyRead
   public String getSysProperty(String property, PrintStream ps) {
      ps.println("Trying to get system property (" + property + ")...");
      try {
         String propertyValue = System.getProperty(property);
         ps.println("\tSuccess!");
         ps.println("\t" + property + " = " + propertyValue);
         return propertyValue;
      } catch (netscape.security.AppletSecurityException e) {
         ps.println("\tFailed! Security Violation.");
         e.printStackTrace(ps);
      } catch (Exception e) {
         ps.println("\tFailed! Unkndown exception while accessing
            property.");
         e.printStackTrace(ps);
      }
      return null;
   }
}
```

Listing 0.3 also demonstrates how this approach prevents direct access to privileges via a public method. In the example, a call to the public method `getSysProperty()` fails unless it is preceded by a call to `enablePrivilege`, which enables the necessary privileges.

To run Listing 0.3, go to http://developer.netscape.com/library/documentation/signedobj/secsampleharness/SecSampleHarness.html, choose `netscape.sample.security.InitPrivSample` from the pop-up menu, and click Run.

Listing 0.4 demonstrates the *wrong* way to enable privileges in a public method.

Listing 0.4 A foolish and dangerous way to use public helper methods

```
// DumbSecurityPracticeSample.java
package netscape.sample.BADsecurity;

import netscape.security.PrivilegeManager;
import java.io.*;
import netscape.sample.security.RunnableSample;

public class DumbSecurityPracticeSample
   implements RunnableSample
{
   public void init(PrintStream ps) {
   }

   public void run(PrintStream ps) {
      String property = "user.home";

      String propertyValue = reallyStupidGetSysProperty(property, ps);
   }

   public void finish(PrintStream ps) {
   }

   // The reallyStupidGetSysProperty() method demonstrates bad
   // programming practices. This method enables privileges and calls
   // protected system functions but provides no protection against
   // malicious Java code (such as the EvilUnsecuritySample class shown
   // inListing 0.5). DO NOT expose protected functions in public
   // methods to potential subversion in this manner. See Listing 0.3
   // for an approach that avoids this problem.

   public String reallyStupidGetSysProperty(String property,
         PrintStream ps) {
      ps.println("Trying to acquire permission to read system
                  properties...");
      try {
         PrivilegeManager.enablePrivilege("UniversalPropertyRead");
         ps.println("\tSuccess! Privilege is enabled.");
      } catch (netscape.security.ForbiddenTargetException e) {
         ps.println("\tFailed! Permission to read system properties
                     denied by user.");
      } catch (Exception e) {
         ps.println("\tFailed! Unknown exception while enabling
                     privilege.");
         e.printStackTrace(ps);
      }
      ps.println();

      ps.println("Trying to get system property (" + property + ")...");
      try {
         String propertyValue = System.getProperty(property);
         ps.println("\tSuccess!");
         ps.println("\t" + property + " = " + propertyValue);
         return propertyValue;
      } catch (netscape.security.AppletSecurityException e) {
         ps.println("\tFailed! Security Violation.");
         e.printStackTrace(ps);
      } catch (Exception e) {
         ps.println("\tFailed! Unkndown exception while accessing
            property.");
         e.printStackTrace(ps);
      }
      return null;
   }
}
```

Listing 0.5 shows how the dangerous code in Listing 0.4 can be subverted by malicious code.

Listing 0.5 A malicious class that subverts the `DumbSecurityPracticeSample` class shown in Listing 0.4

```
// EvilUnsecuritySample.java
package netscape.sample.BADsecurity;

import java.io.*;
import netscape.sample.security.RunnableSample;

public class EvilUnsecuritySample
   implements RunnableSample
{
   public void init(PrintStream ps) {
   }

   public void run(PrintStream ps) {
      String property = "user.home";

      DumbSecurityPracticeSample dummy = new
                                    DumbSecurityPracticeSample();
      String propertyValue =
                     dummy.reallyStupidGetSysProperty(property, ps);
   }

   public void finish(PrintStream ps) {
   }

}
```

Listing 0.4 and Listing 0.5 cannot be run like the other `SecSampleHarness` samples because they are bad examples and thus cannnot be signed or endorsed in any way by Netscape.

Enabling Then Reverting

Listing 0.6 demonstrates how `revertPrivilege()` works.

Listing 0.6 Removing an annotation from the call stack with `revertPrivilege()`

```
// RevertPrivSample.java

package netscape.sample.security;

import netscape.security.PrivilegeManager;
import java.io.*;

public class RevertPrivSample
   implements RunnableSample
{
   booleanhasPrivilege = false;

   public void init(PrintStream ps) {
      ps.println("Trying to acquire permission to read system
                  properties...");
      try {
         PrivilegeManager.enablePrivilege("UniversalPropertyRead");
         hasPrivilege = true;
         ps.println("\tSuccess! Privilege is enabled.");
      } catch (netscape.security.ForbiddenTargetException e) {
         ps.println("\tFailed! Permission to read system properties
                     denied by user.");
      } catch (Exception e) {
         ps.println("\tFailed! Unknown exception while enabling
                     privilege.");
         e.printStackTrace(ps);
      }

      ps.println();
   }

   public void run(PrintStream ps) {
      if(hasPrivilege == false) {
         // if hasPrivilege is false, then you know that the privilege
         // either has not been granted or is not available.
         ps.println("Necessary privileges have not been granted.
                     Aborting.");
```

```
            return;
         }
         String property = "user.home";

         ps.println("Attempting to get property without calling
            enablePrivilege();");
         String propertyValue = getSysProperty(property, ps);

         ps.println();
         ps.println("Now enabling privilege and trying again.");
         PrivilegeManager.enablePrivilege("UniversalPropertyRead");
         propertyValue = getSysProperty(property, ps);
      }

      public void finish(PrintStream ps) {
      }

      public String getSysProperty(String property, PrintStream ps) {
         ps.print("Enabling Privilege within getSysProperty() method...");
         PrivilegeManager.enablePrivilege("UniversalPropertyRead");
         ps.println("Success!");
         ps.print("Reverting privilege...");

         PrivilegeManager.revertPrivilege("UniversalPropertyRead");
         ps.println("Success! Privileges have been restored to the calling
                     state.");

         // now that privileges have been restored to the calling state,
         // the attempt to get a system property fails unless privileges
         // have been enabled before the call to getSysProperty()
         ps.println("Trying to get system property (" + property + ")...");
         try {
            String propertyValue = System.getProperty(property);
            ps.println("\tSuccess!");
            ps.println("\t" + property + " = " + propertyValue);
            return propertyValue;
         } catch (netscape.security.AppletSecurityException e) {
            ps.println("\tFailed! Security Violation.");
            e.printStackTrace(ps);
         } catch (Exception e) {
            ps.println("\tFailed! Unkndown exception while accessing
               property.");
            e.printStackTrace(ps);
         }
         return null;
      }
   }
```

In Listing 0.6, the call to `enablePrivilege()` at the beginning of the `getSystemProperty()` method places an annotation on the call stack, which is almost immediately removed by the subsequent call to revert`Privilege()`. As the sample demonstrates, this returns privileges to the state at the beginning of `getSystemProperty()`, before the call to `enablePrivilege()` was made.

To run Listing 0.6, go to http://developer.netscape.com/library/documentation/signedobj/secsampleharness/SecSampleHarness.html, choose `netscape.sample.security.RevertPrivSample` from the pop-up menu, and click Run.

Enabling Then Disabling

Listing 0.7 demonstrates how `disablePrivilege()` works. This is almost identical to Listing 0.3 except for the call to `disablePrivilege()`. Unlike `revertPrivilege()`, `disablePrivilege()` places a negative annotation on the call stack. This has the effect of shadowing a positive annotation for the same target earlier on the stack. Thus, `disablePrivilege()` prevents access to the specified target without regard for the privilege state before the method in which `disablePrivilege()` appears is called.

Listing 0.7 Adding a negative annotation to the call stack with `disablePrivilege()`

```
// DisablePrivSample.java

package netscape.sample.security;

import netscape.security.PrivilegeManager;
import java.io.*;

public class DisablePrivSample
   implements RunnableSample
{
   booleanhasPrivilege = false;

   public void init(PrintStream ps) {
      ps.println("Trying to acquire permission to read system
                properties...");
      try {
         PrivilegeManager.enablePrivilege("UniversalPropertyRead");
         hasPrivilege = true;
         ps.println("\tSuccess! Privilege is enabled.");
      } catch (netscape.security.ForbiddenTargetException e) {
         ps.println("\tFailed! Permission to read system properties
                     denied by user.");
      } catch (Exception e) {
         ps.println("\tFailed! Unknown exception while enabling
                     privilege.");
         e.printStackTrace(ps);
      }

      ps.println();
   }

   public void run(PrintStream ps) {
      if(hasPrivilege == false) {
         // if hasPrivilege is false, then you know that the privilege
         // either has not been granted or is not available
         ps.println("Necessary privileges have not been granted.
                     Aborting.");
         return;
      }
      String property = "user.home";

      ps.println("Enabling privileges and calling getSysProperty.");
      PrivilegeManager.enablePrivilege("UniversalPropertyRead");
      String propertyValue = getSysProperty(property, ps);

      // up to this point, this sample is virtually identical to
      // Listing 0.3; but now it uses disablePrivilege to place a
      // negative annotation on the call stack
      ps.println();
      ps.println("Now disabling privilege and trying again.");
      PrivilegeManager.disablePrivilege("UniversalPropertyRead");

      // after the privilege has been disabled, a call to
      // getSysProperty fails
      propertyValue = getSysProperty(property, ps);
   }

   public void finish(PrintStream ps) {
   }

   public String getSysProperty(String property, PrintStream ps) {
      ps.println("Trying to get system property (" + property + ")...");
      try {
         String propertyValue = System.getProperty(property);
         ps.println("\tSuccess!");
         ps.println("\t" + property + " = " + propertyValue);
         return propertyValue;
      } catch (netscape.security.AppletSecurityException e) {
         ps.println("\tFailed! Security Violation.");
         e.printStackTrace(ps);
      } catch (Exception e) {
```

```
            ps.println("\tFailed! Unkndown exception while accessing
               property.");
            e.printStackTrace(ps);
         }
         return null;
      }
   }
```

To run Listing 0.7, go to http://developer.netscape.com/library/documentation/signedobj/secsampleharness/SecSampleHarness.html, choose `netscape.sample.security.DisablePrivSample` from the pop-up menu, and click Run.

Creating a User Target and Checking Its Privileges

These samples demonstrate how to create your own target. Listing 0.8 is identical to Listing 0.1 , except that it uses a slightly different form of `enablePrivilege()`. The form of `enablePrivilege()` used in Listing 0.1 takes a string and assumes that the principal that owns the target identified by the string is the System principal; the form used in Listing 0.8 takes a target object. Listing 0.8 first uses `findTarget()` to get a specified target object owned by the System principal, then passes that target object to `enablePrivilege()`.

The lines in Listing 0.8 that differ from Listing 0.1 are shown here in boldface..

Listing 0.8 Specifying the principal explicitly

```
// ExplicitPrincipalSample.java

package netscape.sample.security;

import netscape.security.PrivilegeManager;
import netscape.security.Target;
import netscape.security.Principal;
import java.io.*;

public class ExplicitPrincipalSample
   implements RunnableSample
{
   Target      propReadTarget = null;
   PrincipalsysPrincipal = null;
   PrivilegeManagerprivMgr = null;
   public void init(PrintStream ps) {
      ps.println("Initializing PrivilegeManager and System
         Principal.");
      privMgr =PrivilegeManager.getPrivilegeManager();
      sysPrincipal =PrivilegeManager.getSystemPrincipal();
   }

   public void run(PrintStream ps) {
      ps.println("Trying to acquire permission to read system
                  properties...");
      try {
         propReadTarget =
            Target.findTarget("UniversalPropertyRead", sysPrincipal);
         privMgr.enablePrivilege(propReadTarget);
         ps.println("\tSuccess!");
      } catch (netscape.security.ForbiddenTargetException e) {
         ps.println("\tFailed! Permission to read system properties
                     denied by user.");
      } catch (Exception e) {
         ps.println("\tFailed! Unknown exception while enabling
                     privilege.");
         e.printStackTrace(ps);
      }

      ps.println();
      String property = "user.home";
      ps.println("Trying to get system property (" + property + ")...");
      try {
         String propertyValue = System.getProperty(property);
         ps.println("\tSuccess!");
         ps.println("\t" + property + " = " + propertyValue);
      } catch (netscape.security.AppletSecurityException e) {
         ps.println("\tFailed! Security Violation.");
         e.printStackTrace(ps);
      } catch (Exception e) {
         ps.println("\tFailed! Unkndown exception while accessing
                     property.");
         e.printStackTrace(ps);
      }
   }

   public void finish(PrintStream ps) { //not used in this example
   }
}
```

To run Listing 0.8, go to http://developer.netscape.com/library/documentation/signedobj/secsampleharness/SecSampleHarness.html, choose `netscape.sample.security.ExplicitPrivilegeSample` from the pop-up menu, and click Run.

Listing 0.9 shows how you can create a target that belongs to your own principal, enable that target, and check to make sure it's enabled before accessing it. You can create your own targets to protect unique resources owned by your code.

Listing 0.9 Creating, enabling, and checking for a user target

```
// UserTargetSample.java

package netscape.sample.security;

import netscape.security.PrivilegeManager;
import netscape.security.Target;
import netscape.security.Principal;
import netscape.security.UserTarget;
import netscape.security.UserDialogHelper;

import java.io.*;

public class UserTargetSample
   implements RunnableSample
{
   UserTarget        myUserTarget = null;
   Principal         myPrincipal = null;
   PrivilegeManager  privMgr = null;

   public void init(PrintStream ps) {
      ps.println("Initializing a UserTarget...");
      privMgr =      PrivilegeManager.getPrivilegeManager();
      myPrincipal =  PrivilegeManager.getMyPrincipals() [0];

      // Does my target already exist?
      myUserTarget = (UserTarget)Target.findTarget("MyTarget",
myPrincipal);
      if(myUserTarget == null) {
         ps.println("MyTarget does not exist, creating it.");
         myUserTarget = new UserTarget("MyTarget", myPrincipal,
                           UserDialogHelper.targetRiskLow(),
                           UserDialogHelper.targetRiskColorLow(),
                           "Manipulating a developer-defined target",
                           "");
         myUserTarget.registerTarget();
      } else {
         ps.println("MyTarget already exists.");
      }
   }

   public void run(PrintStream ps) {
      ps.println("Trying to acquire permission for user target...");
      try {
         privMgr.enablePrivilege(myUserTarget);
         ps.println("\tSuccess!");
      } catch (netscape.security.ForbiddenTargetException e) {
         ps.println("\tFailed! Permission to read system properties 
denied by user.");
      } catch (Exception e) {
         ps.println("\tFailed! Unknown exception while enabling 
privilege.");
         e.printStackTrace(ps);
      }
      ps.println();

      ps.println("Checking MyTarget privileges...");
      try {
         privMgr.checkPrivilegeEnabled(myUserTarget);
         ps.println("\tSuccess!");
      } catch (netscape.security.AppletSecurityException e) {
         ps.println("\tFailed! Security Violation.");
         e.printStackTrace(ps);
      } catch (Exception e) {
         ps.println("\tFailed! Unkndown exception while accessing 
property.");
         e.printStackTrace(ps);
      }
   }

   public void finish(PrintStream ps) {
   }
}
```

The `checkPrivilegeEnabled()` method in Listing 0.9 would not normally be called by the method that enables the privilege. The `checkPrivilegeEnabled()` method provides a way of checking whether privileges have been enabled for a particular target before you attempt to access it.