CHAPTER 7

Customizing the Security Architecture

The office of government is not to confer happiness,
but to give men opportunity to work out happiness for themselves.
—William Ellery Channing

This chapter demonstrates ways to augment the security architecture. We explain how to develop custom implementations of the various security classes that support either extensibility or substitution mechanisms. We also describe the mechanics of implementing a custom `Permission` class, extending the functionality of the `SecurityManager` class, implementing a custom `Policy` provider, and implementing a `DomainCombiner` interface.

7.1 Creating New Permission Types

Recall from Section 5.1 that J2SDK 1.2 introduced a new hierarchy of typed and parameterized access permissions, rooted by an abstract class, `java.security.Permission`. Other permissions are subclassed from either the `Permission` class or one of its subclasses and appear in relevant packages. For example, the `FilePermission` permission representing file system access is located in the `java.io` package. Other permission classes are

- `java.net.SocketPermission` for access to network resources
- `java.lang.RuntimePermission` for access to runtime system resources, such as class loaders and threads

- `java.lang.PropertyPermission` for access to system properties
- `java.awt.AWTPermission` for access to windowing resources

As this list illustrates, accesses to controlled resources, including properties and packages, are represented by the permission classes.

Applications are free to add new categories of permissions. However, it is essential that, apart from official releases, no one extend the permissions that are built into the SDK, either by adding new functionality or by introducing additional keywords into a class such as `java.lang.RuntimePermission`. Refraining from doing this maintains compatibility.

When creating a new permission, it is advisable also to declare the permission to be `final`. The rule of thumb is that if the permission will be granted in a security policy, it is probably best to declare it `final`. However, at times it may be necessary to create a class hierarchy for your custom permission. If this is the case, a couple of design heuristics are worth mentioning. First, if the abstract, or base, class of your permission or permission collection has a concrete implementation of the `implies` method, it is recommended that the `implies` method take the type of the permissions into consideration. For example, the `implies` method of the `BasicPermission` class has the following logic, which is similar to that of the `BasicPermissionCollection` class:

```
public boolean implies(Permission permission) {
    if (! (permission instanceof BasicPermission))
            return false;
    BasicPermission bp = (BasicPermission) permission;
    if (bp.getClass() != this.getClass())
            return false;
    ...
}
```

Pay particular attention to the second `if` statement, which enforces the type equality heuristic. Without this, the implementation may be exposed to a subtle security hole whereby the subclass may be able to interact in malicious ways with the superclass's implication checking.

The second design heuristic addresses whether a custom `PermissionCollection` class should be implemented. The general principle is that if the permission has complex processing semantics for either its name or the actions it specifies, it is usually necessary to create a custom `PermissionCollection` class. The Java security architecture specifically enables this by first delegating to the permission collection object for processing of a requisite permission when making a policy decision.

Perhaps these guidelines are best shown by an example. Suppose that you are an application developer from company MyPVR and want to create a customized permission to control access to the channel programming features of a personal video recorder. Further suppose that only three actions are to be controlled for any given channel of programming: the ability to "view", to "preview", and to "record". The first question is, can you use an existing Permission object, such as the BasicPermission class, or do you need a custom permission class? Given the need to be able to control access based on the three actions, the BasicPermission class will not suffice. Therefore, a custom class needs to be designed.

Next, you must make sure that the implies method, among other methods, is correctly implemented. If you decide to support more elaborate channel-naming syntax for PVRPermissions, such as 1–10:13–20 or *, you may need to implement a custom permission collection class, PVRPermissionCollection. This custom class would be responsible for parsing the names and ensuring the proper semantics. Additionally, it may be necessary to perform the implies logic entirely within the implementation supplied by the permission collection. An example of when this is necessary is when the actions of a permission can be granted separately but tested in combination. For example, you may have two separate grants of a PVRPermission specified by the security policy: one for "view" and one for "preview", yet the resource management code tests for them in combination, perhaps as an optimization. That is, in order to be able to "preview" a channel, one must also have the permission to "view" the channel.

Here are parts of the code for sample com.mypvr.PVRPermission and com.mypvr.PVRPermissionCollection classes:

```
public final class com.mypvr.PVRPermission
                extends java.security.BasicPermission
                implements java.io.Serializable {

    /* view channel */
    private final static int VIEW       = 0x1;
    /* preview a channel */
    private final static int PREVIEW    = 0x2;
    /* record a channel */
    private final static int RECORD     = 0x4;
    /* all actions */
    private final static int ALL        = VIEW|PREVIEW|RECORD;
    /* the channel number */
    private transient String channel;
    /* the actions mask */
    private transient int actionMask;
```

```
    public PVRPermission(String channel, String actions) {
        super(channel, actions);
        this.channel = channel;
        this.actionMask = getMask(actions) // parse actions
    }
    ...
    /* for completeness we implement implies but given the usage
      pattern the real work will be done in the permission collection
    */
    public boolean implies(Permission p) {
        if (!(p instanceof PVRPermission))
            return false;
        PVRPermission that = (PVRPermission) p;

        if (this.channel.equals(that.channel) &&
            this.actions.equals(that.actions))
            return true;
        return false;
    }

    public PermissionCollection newPermissionCollection() {
        return new PVRPermissionCollection();
    }
}

final class PVRPermissionCollection extends PermissionCollection
        implements java.io.Serializable {
    private Vector permissions;

    public PVRPermissionCollection() {
        permissions = new Vector();
    }
    ...
    public boolean implies(Permission permission) {
        if (! (permission instanceof PVRPermission))
                return false;
        PVRPermission np = (PVRPermission) permission;
        int desired = np.getMask();
        int effective = 0;
        int needed = desired;
        Enumeration e = permissions.elements();
```

```
            while (e.hasMoreElements()) {
                PVRPermission x = (PVRPermission) e.nextElement();
                if (x.channel.equals(np.channel)) {
                  if ((needed & x.getMask()) != 0)  {
                    effective |=  x.getMask();
                    if ((effective & desired) == desired)
                        return true;
                    needed = (desired ^ effective);
                  }
                }
            }
            return false;
        }
    }
```

Next, you want the application's resource management code, when checking whether an access should be granted, to call SecurityManager's checkPermission method, using an instance of com.mypvr.PVRPermission as the parameter

```
public void previewChannel(int channel) {
  // in order to preview the channel you also need to be able
  // to view the channel
  com.mypvr.PVRPermission tvperm =
         new com.mypvr.PVRPermission(Integer.toString(channel),
                                     "view,preview");
  SecurityManager security = System.getSecurityManager();
  if (security != null) {
    security.checkPermission(tvperm);
  }
...
}
```

Finally, to grant this permission to applications and applets, you need to enter appropriate entries into the security policy. How to configure the policy is discussed in detail in Section 12.5. Basically, you put the string representation of this permission in the policy file so that this permission can be automatically configured for each domain granted the permission. An example of the policy file entry specifying permission for the user "Duke" to watch channel 5 is as follows, which grants to any code considered to be executed by "Duke" the privilege to view and preview channel 5:

```
grant principal javax.security.auth.x500.X500Principal "cn=Duke"{
   permission com.mypvr.PVRPermission "5", "view";
   permission com.mypvr.PVRPermission "5", "preview";
}
```

To exercise the built-in access control algorithm, our code would typically invoke a permission check by directly calling the `checkPermission` method of the `SecurityManager` class, as shown. Generally, it is best to start up the access control machinery by calling the `SecurityManager.checkPermission` method as demonstrated. The default implementation of `SecurityManager.checkPermission` delegates to the `AccessController.` However, should a custom `SecurityManager` class be installed, there is no guarantee that the `AccessController.checkPermission` method will ever be invoked. Details of these classes are provided in Chapter 6, and the question of when to use `AccessController` versus `SecurityManager` is discussed in Section 6.4.9.

7.2 Customizing Security Policy

The security policy is first processed by the `Policy` object and then is enforced by the `SecurityManager` or `AccessController`, so customizing any of these classes would customize the behavior of the security policy. Beginning with J2SE 1.4, security policy decisions are lazily evaluated; prior to J2SE 1.4, security policy decisions were in effect statically computed in advance of enforcement. This section provides general descriptions of the various ways the policy enforcement and decision machinery can be augmented to supply specialized behavior. We first describe the extension points of the `SecurityManager` and then give guidance in implementing a custom `Policy` provider.

7.2.1 Customizing Security Policy Enforcement

As a first example, suppose that you want to allow file access only during office hours: 9 A.M. to 5 P.M. That is, during office hours, the security policy decides who can access what files. Outside of office hours, no one can access any file, no matter what the security policy says. To achieve this, you can implement a `TimeOfDaySecurityManager` class, as follows:

```
public class TimeOfDaySecurityManager extends SecurityManager {
   public void checkPermission(Permission perm) {
      if (perm instanceof FilePermission) {
         Date d = new Date();
         int i = d.getHours();
```

```
            if ((i >= 9) && (i < 17))
               super.checkPermission(perm);
            else
              throw new SecurityException("Outside of office hours");
         } else super.checkPermission(perm);
      }
   }
```

The TimeOfDaySecurityManager checkPermission method checks whether the permission to be checked is a FilePermission. If it is, checkPermission computes the current time. If the time is within office hours, checkPermission invokes the checkPermission method from TimeOfDaySecurityManager's SecurityManager superclass to check the security policy. Otherwise, checkPermission throws a security exception. An application that wishes to enforce the given office hour restriction should install this TimeOfDaySecurityManager in place of the built-in SecurityManager. A SecurityManager is installed by calling the java.lang.System.setSecurityManager method, as described in Section 6.1.2.

The next example concerns the need to keep a record of resource accesses that were granted or denied, for audit purposes later. Suppose that you design a simple AuditSecurityManager class as follows:

```
public class AuditSecurityManager extends SecurityManager {

    private static java.util.logging.Logger logger =
        java.util.logging.Logger.getLogger("AuditSecurityManager");

    public void checkPermission(Permission perm) {
        logger.log(java.util.logging.Level.INFO, perm.toString());
        super.checkPermission(perm);
    }
}
```

This class assumes that you also have an instance of the java.util.logging.Logger class, whose log method records permission checks in a safe place. In this example, the log method simply records the fact that a particular permission was checked. A variation would be to enter the audit record after checkPermission and specify a different logging level, contingent on the result of the access control decision. If the checkPermission call succeeds, the log method is called right after the super.checkPermission call, with a level of INFO. If the call fails, a SecurityException is thrown, and the AuditSecurityManager checkPermission catches the SecurityException, calls the

log method with a level of WARNING to indicate the failure, and then rethrows the exception, as follows:

```
public class AuditSecurityManager extends SecurityManager {
   . . .
   public void checkPermission(Permission perm) {
      try {
         super.checkPermission(perm);
         logger.log(java.util.logging.Level.INFO, perm.toString());
      } catch (SecurityException e) {
         logger.log(java.util.logging.Level.WARNING,
            perm.toString());
         throw e;
      }
   }
}
```

To implement complex security policies, you potentially need to spend more effort in the design. For example, if you wanted to enforce a multilevel security policy, you would first have to create sensitivity labels for each object.[1] The JVM would also have to keep track of the interaction between objects and might have to change object labels dynamically, as in a High-Watermark model. Then the SecurityManager's checkPermission method would need to base its decision on the labels of the objects involved in the current thread of execution. As another example, to implement a Chinese Wall, or separation-of-duty, model, the JVM would need not only to monitor object interaction but also to keep a history of it. Much research and experimentation is needed in this area.

7.2.2 Customizing Security Policy Decisions

The wide variety of reasons for designing and implementing a custom Policy provider run the gamut from the need to support a specialized policy expression language to the need to support a custom policy store. An example might be a Policy provider that specifies policy according to the syntax and processing rules of KeyNote [14]. We introduced the role and structure of the Policy class in Section 5.4. In this section, we illustrate quintessential and sometimes subtle design details necessary to implement a Policy provider correctly.

[1] This could be done perhaps most conveniently by adding a security level attribute to the base class, the Object class, but that would be a very significant change.

Locating Security Policy

When we designed the Java security policy interface, we recognized that it would be nearly impossible to specify adequately the storage and access requirements of security policy or the language by which security policy was expressed. Therefore, we incorporated two key abstractions in our design. The first is the notion of a pluggable provider, which is a standardized mechanism fully documented in Section 12.3.3. The second, location independence of the policy store, is not part of the Java platform specification, yet most platform vendors have adopted Sun's model. Our approach to referencing the policy store is to leverage the inherent generality and location independence Uniform Resource Locators (URLs) provide.

Usually, the deployment of the Java runtime is managed by a system administrator. At deployment time, the administrator has the latitude of changing the configuration of the Java runtime to point to the location of the security policy data. This change can be statically reflected in the security properties file, or it can be specified when the runtime is started, by specifying a URL value for the `java.security.policy` system property. This is described in much greater detail in Section 12.5. For purposes of this discussion, the salient point is that it is imperative for a `Policy` provider implementation to follow the directions of the deployer when locating and accessing the policy data. That is, a proper implementation will follow the hints given by the values for the `policy.url.n` properties in the security properties file, as well as the mechanism to override or augment security policy via the system property `java.security.policy`.

The first thing to consider is whether the deployment permits the override of the security policy location via the `java.security.policy` system property. This is captured in the `policy.allowSystemProperty` security property:

```
if ("true".equalsIgnoreCase
        (Security.getProperty("policy.allowSystemProperty"))) {
    // process the override URL.
    ...
} else {
    ...
}
```

Next, assuming that the location can be overridden by the `java.security.policy` system property, determine whether the supplied URL is to be used in conjunction with the statically configured URLs or whether it is to be considered as the sole source of policy data. The former is indicated by preceding the specified location with an = sign, whereas the latter is indicated by preceding the specified location with a double equals (==). For example, specifying the following

system property as a subpart of the command line invoking the Java runtime indicates that the policy at the given URL should be the only source of policy data:

```
-Djava.security.policy==https://policy.example.com/security.policy
```

Bootstrapping Security Policy

When we designed the `Policy` provider architecture, one of our design goals was to enable the installation of third-party security policy implementations. We also wanted to give deployers flexibility in installing and configuring the Java runtime environment. Section 12.4 details configuring and installing provider packages. As described in Chapter 12, the security `Policy` provider class can be statically configured to be the default provider. One advantage of statically configuring the default security provider is to avoid having to implement application code that dynamically installs the `Policy` provider by invoking the `setPolicy` method of the `Policy` class.

Generally, system or infrastructure software of this caliber is installed as an optional package [101]. However, by installing the `Policy` provider as an installed extension, the implementation of the `Policy` provider is faced with a chicken-and-egg problem. And by statically configuring the systemwide security policy class, the Java runtime is confronted with a chicken-and-egg problem of its own. Let's consider the runtime's problem first.

The Java runtime must be able to install a security `Policy` provider that is not part of the system domain. However, code that is outside the system domain is subject to security checks by the `SecurityManager` or `AccessController`. Therefore, how can the runtime enforce policy while in the process of installing the `Policy` provider? It is possible for the Java runtime to detect this conundrum by being selective as to which class loader is used to install the `Policy` provider. Once this recursion is detected, it is possible to bootstrap the installation of the configured `Policy` provider by relying on the Java platform vendor's default `Policy` implementation class. Once the configured provider is loaded, it can be installed as the systemwide `Policy` provider. The deployer may need to realize that the platform vendor's default policy must be configured to grant the third-party `Policy` provider sufficient permissions to be bootstrapped in this manner. This approach is by no means foolproof, given that the third-party implementation may trigger class loads and subsequent security checks of utility classes outside the system domain or its own protection domain.

The second circularity problem exists regardless of whether the security `Policy` provider is configured statically or installed dynamically. Again because the `Policy` provider is not part of the system domain, it is subject to security checks. Because access control decisions are passing through the `Policy` implementation

via either the `getPermissions` method or the `implies` method, the `Policy` class's protection domain will be under scrutiny whenever it triggers a security check.

A rather rudimentary solution a `Policy` implementation can use is to cache a reference to its own protection domain within its constructor. Then whenever its `getPermissions` or `implies` method is invoked, the `Policy` implementation can treat its protection domain specially and assume it to have sufficient permission to access the resource. In other words, it is probably fair to assume that the `Policy` implementation has been granted `AllPermission`. Here is some sample code from the `Policy` implementation class:

```
// Domain of this provider
private ProtectionDomain providerDomain;

public CustomPolicy() {
    ...
    final Object p = this;
    providerDomain = (ProtectionDomain)
        AccessController.doPrivileged(
            new java.security.PrivilegedAction() {
                public Object run() {
                    return p.getClass().getProtectionDomain();
                }
    });
}
...

public boolean implies(ProtectionDomain pd, Permission p) {
    ...
    if (providerDomain == pd) {
        return true;
    }
    ...
}

public PermissionCollection getPermissions(ProtectionDomain domain)
    {
    Permissions perms = new Permissions();
    ...
```

```
        if (providerDomain == domain) {
            perms.add(new java.security.AllPermission());
        }
        ...
    }
```

Some words of caution: If the `Policy` implementation is packaged with less trusted code and both are in the same protection domain, the less trusted code will be accorded the same permissions as the `Policy` implementation. Also, a call to the `refresh` method should be careful not to drop the cached protection domain.

Spanning Permissions

In Section 5.6, we introduced the merits of dynamic policy and alluded to the spanning permission problem. We also described this issue in Section 7.1. Essentially, the issue can be stated as follows: A `Policy` provider must be able to accommodate a policy decision query through its `implies` interface to determine whether a requisite permission is implied by the given protection domain. This must be derived even when the actions of the requisite permission span the permission collection encapsulated by the `ProtectionDomain` and `Policy` objects.

In our sample `PVRPermissionCollection` implementation, we made special provisions for the advent of this when the actions of the requisite permission were specified in separate `grant` statements. The same problem exists for the `Policy` provider, as the class loader may assign permissions to the `ProtectionDomain` of a class, and the provisioning of the security policy may also specify permissions of the same type and target but for different actions. The simplest approach is for the `Policy` provider to merge the permissions from the protection domain with the permissions it deems are granted by the policy for the given `ProtectionDomain` into a single `PermissionCollection`. The obvious place to implement this merge is in the `getPermissions` method of the `Policy` class.

7.3 Customizing the Access Control Context

When we first introduced the `DomainCombiner` in Section 6.3, we described the API and the relationship of a `DomainCombiner` to the access control machinery. The remainder of this chapter describes possible uses for a `DomainCombiner` and implementation strategies. We may use the term combiner as a shorthand for `DomainCombiner`.

Two steps must be taken to insert a combiner into the access control machinery of the Java runtime environment. The first is to construct an `AccessControlContext` with an instance of a `DomainCombiner`. The second step is to bind the

security context with the execution context. This is accomplished by supplying the AccessControlContext to the appropriate AccessController doPrivileged method.

A real-world application of a DomainCombiner is the javax.security.auth.SubjectDomainCombiner. This particular implementation encapsulates an instance of a javax.security.auth.Subject. As described in Section 8.4.1, a Subject encapsulates a set of (ostensibly authenticated) principals. The role of the SubjectDomainCombiner is to augment the ProtectionDomains of the current execution context with the principal information so that security policy can be based on who is running the code.

The combine method of a bound DomainCombiner is invoked as a result of a call to either the checkPermission or the getContext AccessController method. In the absence of a DomainCombiner, the AccessController optimizes the AccessControlContext object. When a DomainCombiner is present, it is up to the implementation of the installed combiner to perform any optimizations on the AccessControlContext returned from the combine method. That said, another possible application for a DomainCombiner is one that implements special optimizations. For example, suppose that the DomainCombiner encapsulated principal information analogous to the SubjectDomainCombiner. Additionally, the custom combiner makes special provisions for the administrative principal such that it is accorded AllPermission. Such a combiner can make a significant optimization to the AccessControlContext when it detects that it is executing as the administrative principal. One possibility is for the combiner to return an AccessControlContext with a single ProtectionDomain; in that case, the PermissionCollection encapsulated within the ProtectionDomain would include an instance of the AllPermission permission.