

## 6 A Custom Policy

This chapter describes an example of implementing a custom `java.security.Policy`. The `Policy` we'll develop is actually composed of several parts: a `CompositePolicy` that delegates to any number of “sub-`Policy`” implementations; a database-backed `Policy` that retrieves permission grants from a database; and the default file-based `Policy`. Aggregating a `Policy` like this allows for more flexibility and code reuse. As with previous chapters, we'll use a small class with a `main()` method to demonstrate using the custom `Policy`.

Our custom `Policy` takes one large short cut to simplify the example: `Permission` checks are effectively only applied when a `Subject` is logged in, allowing most code to execute with all permissions enabled. With that disclaimer aside, let's jump into the code.

### 6.1 The “Main” Application

The simple application we use initializes the custom `Policy` and `SecurityManager`, creates a test user, and then does a simple `Permission` check for reading a temporary file. The first check will fail because the required `java.io.FilePermission` hasn't been added to any of the `Subject`'s `Principals`. Before doing a second check, the application associates the needed `FilePermission` to one of the `Subject`'s `Principals`. After the `FilePermission` has been added, the call to `SecurityManager.checkPermission()` returns successfully.

Below is the code used to run the example:

```
package chp06;

import java.io.FilePermission;
import java.security.Policy;
import java.security.PrivilegedAction;
import java.util.ArrayList;
import java.util.List;

import javax.security.auth.Subject;

import util.id.Id;

public class Main {

    static public void main(String[] args) throws Exception {
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
AuthHelper authHelper = new AuthHelper();

try {
    Policy defaultPolicy = Policy.getPolicy();
    DbPolicy dbPolicy = new DbPolicy();
    List policies = new ArrayList(2);
    policies.add(defaultPolicy);
    policies.add(dbPolicy);
    CompositePolicy p = new CompositePolicy(policies);
    Policy.setPolicy(p);

    System.setSecurityManager(new SecurityManager());
    authHelper.createTestUser("testuser", "testpassword");
    authHelper.loginTestUser();
    Subject subject = authHelper.getSubject();

    final FilePermission filePerm = new FilePermission(
        "/tmp/test", "read");

    boolean allowed = true;
    try {
        Subject.doAsPrivileged(subject, new PrivilegedAction() {

            public Object run() {
                SecurityManager sm = System.getSecurityManager();
                if (sm != null) {
                    sm.checkPermission(filePerm);
                }
                return null;
            }

        }, null);
    } catch (SecurityException e) {
        allowed = false;
    }

    if (allowed) {
        System.out.println("Subject can read file /tmp/test");
    } else {
        System.out.println("Subject cannot read file /tmp/test");
    }

    Id principalId = authHelper.getUserGrp().getId();
    PermissionService.addPermission(principalId, Id.create(),
```



```
        filePerm);
    System.out.println("Added " + filePerm + " to Subject.");

    allowed = true;
    try {
        Subject.doAsPrivileged(subject, new PrivilegedAction() {

            public Object run() {
                SecurityManager sm = System.getSecurityManager();
                if (sm != null) {
                    sm.checkPermission(filePerm);
                }
                return null;
            }

        }, null);
    } catch (SecurityException e) {
        allowed = false;
    }

    if (allowed) {
        System.out.println("Subject can read file /tmp/test");
    } else {
        System.out.println("Subject cannot read file /tmp/test");
    }
} finally {
    if (authHelper != null) {
        authHelper.cleanUp();
    }
}
}
```

To run the above, change directories to the root of this book's project and execute the command `ant run-chp06`. The output will include:

```
Subject cannot read file /tmp/test
Added (java.io.FilePermission /tmp/test read) to Subject.
Subject can read file /tmp/test
```

### *6.1.1 Initializing the Custom Policy*

As its name suggests, the `CompositePolicy` is an implementation of the Composite pattern: a `CompositePolicy` delegates its behavior to the 0-to-n `Policies` it holds, and



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

instances of `CompositePolicy` can be used wherever a `java.security.Policy` can be used.

The `CompositePolicy` instance we use is composed of two `Policys`: the default J2SE `Policy`<sup>1</sup>, obtained by calling the static method `Policy.getPolicy()` and our custom `chp06.DbPolicy`. The default `Policy` performs `Permission` assignments and checks for legacy code, and also assigns several basic permissions to every security context. For example, the default `Policy` grants the permission to access many simple, low-risk properties. Rather than re-implement this behavior, we just re-use the default `Policy` in our `CompositePolicy`. The `DbPolicy`, discussed below in section 6.3, is backed by a database, and provides us with a more dynamic runtime way of doing authorization checks.

Once the `CompositePolicy` and its sub-`Policies` are created, the application enables the authorization by calling two static set methods: one method to set our `CompositePolicy` as the VM-wide `Policy`, `Policy.setPolicy()`, and another method to enable the default `java.lang.SecurityManager`, `System.setSecurityManager()`.

The `CompositePolicy` is covered in more detail below, section 6.2.

### 6.1.2 Creating and Authenticating the Test User

The helper class `chp06.AuthHelper` is used to create, authenticate, and then cleanup the `Subject` and its `Principals` that are used in the example. As the focus of this chapter is authorization, the code isn't excerpted here.

### 6.1.3 Checking Permissions

To check if the `Subject` has been authorized to read the file `/tmp/test`, the code first creates a `java.io.FilePermission` instance describing the target and action, creates a security context based on the `Subject`'s authorizations, and then uses the `SecurityManager.checkPermission()`. If the `FilePermission` has been granted, `checkPermission()` silently succeeds, otherwise it throws a `SecurityException`.

The `Subject.doAsPrivileged()` is used to limit the security context to only the `Permissions` granted to the `Subject`'s `Principals`. As discussed in section 5.7 of the previous chapter, the alternative is to use the `Subject.doAs()` method, which instead combines the `Subject`'s `Principals` with the each `ProtectionDomain` in the execution stack, namely the `ProtectionDomain` that represents `chp06.Main`.

## 6.2 The CompositePolicy

The purpose of `CompositePolicy` is to allow any number of `Policys` to be in effect at one time. As with any `Policy` implementation, the `CompositePolicy` implements the three methods: `getPermissions(CodeSource)`, `getPermissions(ProtectionDomain)`, and the `implies(ProtectionDomain, Permission)`. The first two methods combine the `Permissions` of returned by the aggregated into one `java.security.Permissions` object,

---

<sup>1</sup> Implemented by the class `sun.security.providers.PolicyFile`, and, by default, backed by the file `<JAVA_HOME>/lib/security/java.policy`.



which is returned. The `implies()` method returns if at least one of the aggregated Policies returns true from their `implies()` method; that is, for a Permission to be granted to a ProtectionDomain, at least one of its aggregate Policies must return true from its `implies()` method.

The code for CompositePolicy is below:

```
package chp06;

import java.security.CodeSource;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Permissions;
import java.security.Policy;
import java.security.ProtectionDomain;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CompositePolicy
    extends Policy {

    private List policies = Collections.EMPTY_LIST;

    public CompositePolicy(List policies) {
        this.policies = new ArrayList(policies);
    }

    public PermissionCollection getPermissions(ProtectionDomain domain) {
        Permissions perms = new Permissions();
        for (Iterator itr = policies.iterator(); itr.hasNext();) {
            Policy p = (Policy) itr.next();
            PermissionCollection permCol = p.getPermissions(domain);
            for (Enumeration en = permCol.elements(); en
                .hasMoreElements();) {
                Permission p1 = (Permission) en.nextElement();
                perms.add(p1);
            }
        }
        return perms;
    }

    public boolean implies(final ProtectionDomain domain,
        final Permission permission) {
        for (Iterator itr = policies.iterator(); itr.hasNext();) {
            Policy p = (Policy) itr.next();
            if (p.implies(domain, permission)) {
                return true;
            }
        }
    }
}
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
    }  
  }  
  
  return false;  
}  
  
public PermissionCollection getPermissions(CodeSource codesource) {  
    Permissions perms = new Permissions();  
    for (Iterator itr = policies.iterator(); itr.hasNext();) {  
        Policy p = (Policy) itr.next();  
        PermissionCollection permsCol = p.getPermissions(codesource);  
        for (Enumeration en = permsCol.elements(); en  
            .hasMoreElements();) {  
            Permission p1 = (Permission) en.nextElement();  
            perms.add(p1);  
        }  
    }  
    return perms;  
}  
  
public void refresh() {  
    for (Iterator itr = policies.iterator(); itr.hasNext();) {  
        Policy p = (Policy) itr.next();  
        p.refresh();  
    }  
}  
}
```

## 6.3 DbPolicy

The `chp06.DbPolicy` relies on the `UserGroupPrincipal` class introduced in chapter 4, although it supports a wide variety of `Permission` implementations, including its own `DbPermission`. `DbPolicy` implements the three `Policy` methods, `getPermissions(CodeSource)`, `getPermissions(ProtectionDomain)`, and `implies(ProtectionDomain, Permission)`. As mentioned in the overview, `DbPolicy` takes a shortcut by only providing authorization services for Subjects: if the security context does not include a Subject, the `DbPolicy` grants all permissions. Though this helps illustrate the inter-workings of JAAS, it can be extremely dangerous depending on your security requirements.

First, we'll look at the code. The rest of this section will then go over different methods in `DbPolicy`.

```
package chp06;  
  
import java.security.AccessController;  
import java.security.AllPermission;  
import java.security.CodeSource;  
import java.security.Permission;
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5  
License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
import java.security.PermissionCollection;
import java.security.Permissions;
import java.security.Policy;
import java.security.Principal;
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.security.ProtectionDomain;
import java.sql.SQLException;
import java.util.Arrays;
import java.util.Enumeration;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;

import chp04.UserGroupPrincipal;

public class DbPolicy
    extends Policy {

    public PermissionCollection getPermissions(CodeSource codesource) {
        // others may add to this, so use heterogeneous Permissions
        Permissions perms = new Permissions();
        perms.add(new AllPermission());
        return perms;
    }

    public PermissionCollection getPermissions(
        final ProtectionDomain domain) {
        final Permissions permissions = new Permissions();

        // Look up permissions
        final Set principalIds = new HashSet();
        Principal[] principals = domain.getPrincipals();
        if (principals != null && principals.length > 0) {
            for (int i = 0; i < principals.length; i++) {
                Principal p = principals[i];
                if (p instanceof UserGroupPrincipal) {
                    UserGroupPrincipal userGroup = (UserGroupPrincipal) p;
                    principalIds.add(userGroup.getId());
                }
            }
        }
        if (!principalIds.isEmpty()) {
            try {
                List perms = (List) AccessController
                    .doPrivileged(new PrivilegedExceptionAction() {

                        public Object run() throws SQLException {
                            return PermissionService
                                .findPermissions(principalIds);
                        }
                    });
            } catch (Exception e) {
                // ...
            }
        }
    }
}
```



```
        }
    });
    for (Iterator itr = perms.iterator(); itr.hasNext();) {
        Permission perm = (Permission) itr.next();
        permissions.add(perm);
    }
} catch (PrivilegedActionException e) {
    // Log
}
}
} else if (domain.getCodeSource() != null) {

    PermissionCollection codeSrcPerms = getPermissions(domain
        .getCodeSource());
    for (Enumeration en = codeSrcPerms.elements(); en
        .hasMoreElements();) {
        Permission p = (Permission) en.nextElement();
        permissions.add(p);
    }
}

return permissions;
}

public boolean implies(final ProtectionDomain domain,
    final Permission permission) {
    if (permission.getName().equals("/tmp/test.tx")) {
        int i = 0;
    }
    PermissionCollection perms = getPermissions(domain);

    boolean implies = perms.implies(permission);

    return implies;
}

public void refresh() {
    // does nothing for DB.
}
}
```

### 6.3.1 *getPermissions(ProtectionDomain)*

Most of the behavior of DbPolicy is done by `getPermissions(ProtectionDomain)`. The other `getPermissions(CodeSource)` method, as noted above, grants all permissions to any `CodeSource`, while the `implies` method simply uses the `PermissionCollection` returned by `getPermissions(ProtectionDomain)` to perform authorization.

`getPermissions(ProtectionDomain)` first checks if a Subject is logged in, by getting the Principals associated with the passed in `ProtectionDomain`. If there are no



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>



Principals, we assume that there is no Subject logged in<sup>2</sup>. In such cases, `getPermissions(ProtectionDomain)` delegates to `getPermissions(CodeSource)`, which grants all permissions.

If a Subject with Principals is logged in, `getPermissions(ProtectionDomain)` first gathers all of the Subject's `DbUserGroupPrincipal` Id's, and then uses `PermissionService` to retrieve the associated Permissions. The union of all the Permissions is returned in a `java.security.Permissions` instance, which allows us to collect different types of Permissions together.

### *Privileged Code*

Also, notice that the call to `PermissionService.findPermissions()` is done in a privileged code block. The database code used by `PermissionService.findPermission()` requires that certain `java.io.SocketPermissions` be granted to the current security context. The only Permission we've granted to the Subject, however, is the ability to read the temporary file `/tmp/test`. The Subject does not have the required `SocketPermissions` to connect to the database.

To get around this, enabling the `DbPolicy` to access the database regardless of which Subject is logged in two conditions must be satisfied:

1. The `ProtectionDomain` that represents `chp06.DbPolicy` must be granted the appropriate `SocketPermissions`.
2. A Privileged code block must be created to execute the sensitive database code in. This Privileged block will exclude the Subject's Principals and, thus, Permission restrictions, from the security context.

Because of the shortcut we've taken, any code that executes without a Subject is granted `java.security.AllPermission`, granting *all* Permissions, including the `SocketPermissions` we need. So, by creating a Privileged code block around the call the `PermissionService.findPermission()`, we exclude considering the Subject in the authorization checks, thus, allowing the needed `SocketPermission`. See section 5.5.2 in chapter 5 for more discussion on using privileged blocks.

### *6.3.2 UserGroupPrincipals Only*

The `DbPolicy` works only with `UserGroupPrincipals` to support the database schema it relies on. In the database, each Principal is assigned a unique Id. The Principal interface doesn't guarantee that the Principal's name will be unique in any context, so a different field must be used for a unique identifier, provided by `UserGroupPrincipal`'s `Id` field.

Limiting the Principals that a Policy supports in this way would be too constrictive if the `DbPolicy` were the only Policy that could be used at one time. But, since we provide a `CompositePolicy`, other Policies that do not have this limitation can co-exist, allowing other types of Principals to be used and supported by other Policies. For example, the use

---

<sup>2</sup> This does leave a loophole in which a Subject *with no* Principals is granted all Permissions.



of the SDK's default `Policy` allows our application to support `Principals` other than `DbUserGroupPrincipal`, albeit through the standard flat-file.

### 6.3.3 The Database and *PermissionService*

The `PermissionService` provides the data access layer between the `DbPolicy` and the database. As with the `DbLoginModule`, pure JDBC calls are used to simplify the example. Using JDBC directly like this isn't required, of course. In production systems, it may be appropriate for you to use more featureful data-access libraries like Hibernate, Spring, or EJBs.

`PermissionService` provides methods for adding `Permissions` to `Principals`, looking up the `Permissions` granted to `Principals`, and for removing `Permissions` and their association to `Principals`.

#### *Database Schema*

Each of `PermissionService`'s static methods relies, of course, on the permission schema in the database. The schema includes the tables from chapter XXX [authentication chapter], and the tables below:

```
CREATE TABLE permission
(
  id varchar(64) NOT NULL,
  permissionClass varchar(255) NOT NULL,
  name varchar(64),
  actions varchar(255),
  PRIMARY KEY (id )
);

CREATE TABLE principal_permission
(
  principal_id varchar(64) NOT NULL,
  permission_id varchar(64) NOT NULL
);
```

The `permission` table is responsible for storing each `Permission` assigned to a `Principal`. Each entry must have an `Id` and a fully qualified class name. The `name` and `actions` columns are optional. The `principal_permission` table is a tie table used to associate `Permissions` to `Principals`.

#### *PermissionService Implementation*

The code for `PermissionService` is listed below. Most of the code is simple JDBC code that does the grunt work of creating, reading, and updating the `Permissions` stored in the database. The primary point of interest are the attempts to reflectively create the loaded `Permissions`, marked by code annotations.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
package chp06;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.security.Permission;
import java.security.Principal;
import java.security.UnresolvedPermission;
import java.security.cert.Certificate;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;

import util.db.DbService;
import util.id.Id;

public class PermissionServiceNoLogging {

    static private final Certificate[] EMPTY_CERTS = new Certificate[0];

    static private final Class[] ZERO_ARGS = {};

    static private final Object[] ZERO_OBJS = {};

    static private final Class[] ONE_STRING_ARG = { String.class };

    static private final Class[] TWO_STRING_ARGS = { String.class,
        String.class };

    static public void removePermission(Id id) throws SQLException {
        removePermissions(Collections.singleton(id));
    }

    static public void removePermissions(Set ids) throws SQLException {
        Connection conn = null;
        try {
            conn = DbService.getInstance().getConnection();
            String sql = "DELETE FROM principal_permission WHERE
permission_id = ?";
            PreparedStatement tiePstmt = conn.prepareStatement(sql);
            PreparedStatement permPstmt = conn
                .prepareStatement("DELETE FROM permission WHERE id= ?");
            for (Iterator itr = ids.iterator(); itr.hasNext();) {
                // HSQLDB doesn't support addBatch() :(
                Id id = (Id) itr.next();
```



```
        tiePstmt.setString(1, id.getId());
        permPstmt.setString(1, id.getId());
        tiePstmt.executeUpdate();
        permPstmt.executeUpdate();
    }
} finally {
    if (conn != null) {
        conn.close();
    }
}
}

static public List findPermissions(Set principalIds)
    throws SQLException {
    // HSQLDB doesn't allow batching, so we have to do a call per id
    List permissions = new ArrayList();
    for (Iterator itr = principalIds.iterator(); itr.hasNext();) {
        Id principalId = (Id) itr.next();
        permissions.addAll(findPermissions(principalId));
    }
    return permissions;
}

static public List findPermissions(Id principalId)
    throws SQLException {
    List perms = new ArrayList();
    Connection conn = null;
    try {
        conn = DbService.getInstance().getConnection();
        String sql = "SELECT permission.id id, "
            + "permission.permissionClass clazz, permission.name name, "
            + "permission.actions actions "
            + "FROM principal_permission, permission "
            + "WHERE principal_permission.principal_id=? "
            + "AND permission.id=principal_permission.permission_id ";

        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, principalId.getId());
        ResultSet rs = pstmt.executeQuery();
        while (rs.next()) {
            String idStr = rs.getString("id");
            Id id = Id.create(idStr);
            String clazzStr = rs.getString("clazz");
            String name = rs.getString("name");
            String actions = rs.getString("actions");

            Permission perm = null;
            // make class
            Class clazz = null;

            perm = createPermission(id, clazzStr, name, actions);
        }
    }
}
```



```
        if (perm != null) {
            perms.add(perm);
        } else {
            continue;
        }
    }
} finally {
    if (conn != null) {
        conn.close();
    }
}
return perms;
}

private static Permission createPermission(Id id,
    String clazzStr, String name, String actions) {
    Permission perm = null;
    Class clazz = null;
    try {
        clazz = Class.forName(clazzStr);
    } catch (ClassNotFoundException e) {
        // deal with below
    }

    if (clazz == null) {
        perm = new UnresolvedPermission(clazzStr, name, actions,
            EMPTY_CERTS);

    } else if (clazz.equals(DbPermission.class)) {
        perm = new DbPermission(id, name, actions);
    } else if (Permission.class.isAssignableFrom(clazz)) {
        try {
            if (name == null && actions == null) {
                Constructor con = clazz.getConstructor(ZERO_ARGS); #1
                perm = (Permission) con.newInstance(ZERO_OBJS);
            } else if (actions == null) {
                Constructor con = clazz.getConstructor(ONE_STRING_ARG); #1
                perm = (Permission) con
                    .newInstance(new String[] { name });
            }
            // BasicPermission types
        } else if (name != null && actions != null) {
            Constructor con = clazz.getConstructor(TWO_STRING_ARGS); #1
            perm = (Permission) con.newInstance(new String[] { name,
                actions });
        }
    } catch (Exception e) {
        // Log
    }
}
return perm;
}
```



```

    }

    static public void addPermission(Id principalId,
        DbPermission dbPermission) throws SQLException {
        addPermission(principalId, dbPermission.getId(), dbPermission);
    }

    static public void addPermission(Id principalId, Id permissionId,
        Permission permission) throws SQLException {
        Connection conn = null;
        try {
            conn = DbService.getInstance().getConnection();
            PreparedStatement pstmt = conn
                .prepareStatement("INSERT INTO permission VALUES (?, ?, ?,
?)" );
            pstmt.setString(1, permissionId.getId());
            pstmt.setString(2, permission.getClass().getName());
            pstmt.setString(3, permission.getName());
            pstmt.setString(4, permission.getActions());
            pstmt.executeUpdate();

            PreparedStatement pstmt2 = conn
                .prepareStatement("INSERT INTO principal_permission VALUES
(?, ?)" );
            pstmt2.setString(1, principalId.getId());
            pstmt2.setString(2, permissionId.getId());
            pstmt2.executeUpdate();
        } finally {
            if (conn != null) {
                conn.close();
            }
        }
    }
}

```

(annotation) <#1 the `findPermissions()` method uses reflection extensively to create the `Permission` instances retrieved from the database. Because `Permission` instances are not required to have a default, no argument constructors, each instance must be reflectively created by attempting to acquire the `java.lang.reflect.Constructor` needed, as dictated by the presence or absence of the name and actions attributes. If an appropriate `Constructor` cannot be found, or an error occurs using it, the `Permission` is skipped, avoiding the “throwing out the baby with the bathwater” effect where one bad apple ruins the whole barrel. For a much more detailed discussion of reflection, see the Forman’s *Java Reflection in Action*.>

## Summary

This chapter demonstrated integrating JAAS authorization functionality with a database. To accomplish this goal, first we created a `CompositePolicy` class that allowed us to use multiple `Policies` at the same time. Next, we created a custom `Policy` implementation that was backed by a database rather than a flat file. Using a database instead of the flat files allows



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

your application to more easily specify Permissions at runtime and provides an easier way to maintain all of the Permission grants in your system than flat files.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5  
License: <http://creativecommons.org/licenses/by-nc/2.5/>