A JavaGeeks.com
White Paper

Ted Neward
http://www.javageeks.com/~tneward
tneward@javageeks.com
15July2001

java.security.Policy

When "java.policy" Just Isn't Good Enough

Abstract

Java 2's security system is a complex, pluggable architecture that allows for Java programmers to participate in the default process, or to replace the java.policy-based implementation altogether in favor of something else. In fact, Sun favors this latter approach, urging developers to implement a customized Policy implementation more suitable to their business' needs.

In this paper, we will examine the details of how a new Policy implementation is built, and provide readers with the knowledge necessary to "roll your own" Java Security Policy implementations.

This paper assumes you are familiar, at least in concept, with some of the Java2 Security architecture.

Problem discussion

History. Bear with me through a slight history lesson about the Java Security mechanism.

The Java Platform has, from its very inception, been very sensitive to the idea of security. This is true both in the sense of preventing malicious attacks along the lines of the infamous buffer-overrun attack so common in CERT alerts[1], as well as the idea of restricting access to sensitive operations to untrusted code.

This latter idea, first used in the applet architecture used by both the major browsers, came to be known as the "applet sandbox". It severely restricted applets' abilities to either affect the machine they were executing on, or to obtain sensitive information about that same machine. As applets became more useful within developers' lives, it became apparent that it was necessary to give developers the ability to allow applets to "escape" the sandbox. Users are accustomed to executable code manipulating things on the local machine, and couldn't understand why an applet couldn't do the same. It was a natural expectation that an applet should be able to store user preferences, for example, on the local disk. However, only code the user trusted should have such permissions-all other applets needed to be kept within the sandbox.

Within the JDK 1.1 architecture, applets could be "signed" with a digital key that allowed the applet (and only that applet) to escape the sandbox and have additional permissions, such as the ability to write to the local disk or open socket connections to other hosts. This was known as the ability to have "signed applets", and both major browsers supported it[2].

The Java Runtime API has always had the notion of security laced throughout the API, so supporting security within the API has been simple. Any time the Java team decided the Runtime needed to provide a facility that was considered sensitive, it required a check to the java.lang.SecurityManager:

```
SecurityManager sm = System.getSecurityManager();
if (sm != null)
    sm.checkRead();
```

This sort of code is omnipresent within the Runtime API, and for the JDK 1.0 and 1.1 releases, served Java well. If the current SecurityManager (if there is one) disallows the caller permission to, in the above code, open a file, then a java.security.SecurityException is thrown and the call terminates. If the caller has such permission, the call to checkRead silently completes and the rest of the method is allowed to execute. Each sensitive operation requires its own checkXXX method, implemented on the base SecurityManager class, and SecurityManager itself is abstract, requiring security implementors to provide a concrete SecurityManager-extending class instance to provide security policy.

As Li Gong writes, however, "the need to support flexible and fine-grained access-control security policies, with extensibility and scalability, called for a new and improved security architecture. The result is JDK 1.2."[3] He goes on to describe the architecture of this new-and-improved security system:

This new architecture uses a security policy to decide which individual access permissions are granted to running code. These permissions are based on the code's characteristics, for example where the code is coming from, whether it is digitally signed, and if so by whom. Later, attempts to access protected resources will invoke security checks that will compare the granted permissions with the permissions needed for the attempted access. If the former includes the latter, access will be permitted; otherwise, access will be denied.

--Inside the Java 2 Security Platform, p. 37

One of the key problems with the 1.1 architecture was that of extensibility--as long as SecurityManager itself contained all the possible sensitive operations within its API, then all was well. However, the Sun security team (led by Li Gong) realized that various systems may want to introduce customized security checks, and doing so under a 1.1 architecture required one of two possible roads:

1. Brute-force the API and shoehorn permissions checks into existing API calls. For example, an enterprise system might subvert the checkRead call into not only asking whether the code has the ability to open a file, but also to open a connection to the database. The key problem with this approach is that now each API call takes on dual meaning, and the code is not inherently clear as to which meaning is being used ("Hey, Joe, is this call to checkRead because I want to open a file, or because I want to display a dialog?" "How should I know?")
2. Introduce new API calls onto a custom SecurityManager-extending derived class, and explicitly cast the call from System.getSecurityManager to the customized type before attempting to check the new customized permission. Unfortunately, this carries with it a serious question that could never be answered satisfactorily-what should the code do in case the instance returned from System.getSecurityManager isn't, in fact, the custom SecurityManager expected? As strange as it might seem at first, it's not uncommon for code to be executing within an environment in which a strange SecurityManager already has control: servlet engines, EJB servers, and other such container/component environments (think Jini).

java.security.Policy. Sun's solution was to make the SecurityManager class concrete, and no longer require security implementors to create their own customized SecurityManager class. The SecurityManager now delegates all calls to the checkPermission call, which expects an appropriate Permission object, describing the permission sought by the caller. The security system compares the permission against the permissions allowed by this CodeSource[4], and if the code has appropriate permissions, the call is allowed.

Effectively, what this does is separate two vitally important functions into two separate systems, both of which were formerly intertwined inside of SecurityManager. One is the enforcement of a security policy, which SecurityManager retains as its principal function in Java2. The other is the establishment of that security policy in the first place. In JDK 1.1 VMs, the SecurityManager not only enforced policy, via the various check calls, but through those calls, also implicitly established policy. To change the policy required changing the SecurityManager-extending-class' code.

Under the Java2 system, the establishment of security policy is now handled by a new class, the java.security.Policy class. To be more specific, it is now handled by a derivative of the java.security.Policy class. The Singleton (see [gof] for details) Policy instance is called to establish the various Permissions allowed for a particular CodeSource, and how this Policy instance establishes those permissions is entirely an implementation detail. From the JDK 1.3 javadocs,

This is an abstract class for representing the system security policy for a Java application environment (specifying which permissions are available for code from various sources). That is, the security policy is represented by a Policy subclass providing an implementation of the abstract methods in this Policy class.

--jdk1.3/docs/api/java/security/Policy.html

In short, the Policy implementation (that is, an instance of a class that extends java.security.Policy) is responsible for the establishment of PermissionCollection-to-CodeSource mappings[5].

An abstract class is all well and good, but without a default implementation, it accomplishes exactly nothing. As expected, the Sun JDK ships with a default Policy implementation, and it is this implementation that is responsible for the java.policy file as we know (and love) it today.

sun.security.provider.PolicyFile. In lieu of another implementation, the Sun JVM uses the sun.security.provider.PolicyFile class as its Policy instance. The PolicyFile implementation parses the java.policy file in the ${java.home}/lib/security directory[6] and uses that to direct the association of Permissions to code. It is possible to point the PolicyFile implementation to use some other file, or to use an amalgamation of multiple files; in fact, by default two files will be consulted-the java.policy file mentioned above, as well as a ".java.policy" (note the leading dot) file in the user's home directory. This is controlled by the "policy.url.n" entries in the "java.security" file, which lives in the same directory as the java.policy file. The location of this file can also be controlled at the command-line, via the "-Djava.security.policy=..." property parameter.

The format of this policy file is documented, both in the Java Security docs that come as part of the standard JDK documentation bundle, as well as in [1] and [2] . Briefly put, the java.policy file consists of one or more "grant" blocks, describing the permissions assigned to a particular CodeSource:

```
grant {
    java.net.SocketPermission "localhost:1024-", "listen";
};
grant codeBase "file:/${user.home}" {
    java.util.PropertyPermission "user.name" "read, write";
};
grant signedBy "fred_wesley" {
    java.lang.RuntimePermission "stopThread";
};
```

At its simplest, the "grant" block can appear with no decorations, indicating that this block applies to any and all code in the VM. A "grant" block associated with a URL applies these permissions to any and all code coming from that URL. A "grant" block that has a "signedBy" attribute on it applies these permissions to any code digitally signed under that name. In the example above, the first permission set applies to all code, the second only to code loaded from the user's home directory, and the third to any code signed by "fred_wesley".

Permission blocks are cumulative; if code signed by "fred_wesley" is loaded from the user's home directory (that is, the .jar file lives in the user's home directory), then that code has all three sets of permissions. That is, code signed by "fred_wesley" that comes out of my "user.home" directory has permission to listen on ports 1024 and above (the java.net.SocketPermission permission), to look up the value of the "user.home" system property (the java.util.PropertyPermission permission), and to call Thread.stop() on Threads (the java.lang.RuntimePermission permission).

Criticisms of PolicyFile. As it stands, the PolicyFile implementation serves its role as "plain vanilla Policy implementation" well enough for casual use. For use in client-side applications, it's more than enough. However, the PolicyFile implementation breaks down in a couple of places:

1. Permissions can only be associated by CodeSource. The PolicyFile syntax doesn't permit any kind of role-based permissions; all permissions are assigned based on CodeSource. The truth is, most Java "security" needs will be more along the lines of a role-based permissions system (that is, each user has a role, and roles have certain permissions available to them), than a code-based one. Unfortunately, even though the PolicyFile syntax supports the "signedBy" attribute, that doesn't really offer the same level of control as a real role-based system.
2. The policy file(s) are parsed once, then cached and never refreshed. Modifying the java.policy file on disk doesn't trigger a refresh() call on the Policy instance, which means that the JVM itself won't know about the modifications to the security policy until it is restarted (or explicitly told to reload via an explicit Policy.getPolicy().refresh() call). While suitable for JVMs that are frequently restarted, this has disastrous consequences for long-running Java processes (like servlet engines and/or EJB servers).
3. The policy file lives on disk, in plain-text format. An entire suite of disquieting thoughts come along with this. The entire security policy is laid out in plain text, thus allowing anyone with access to the

filesystem to examine (and possibly modify) the security policy at will. In addition, because few filesystems offer strong concurrent modification support, it's not clear exactly what will happen if I modify the policy file in the middle of JVM startup.

All of this is not to say that the default implementation is not a strong or solid one. It simply doesn't serve the purposes to which we are putting Java these days: long-running server-side processes. In a Five-Nines[7] environment, bringing the EJB server down just to allow the new security policy to take effect is not acceptable. In addition, leaving the security policy implementation available for any to see is a downright security risk; ideally, it should at least be marginally encrypted somehow[8]. And a centralized system for supporting role-based security is just too tempting to blow off.

Solution discussion

Custom Policy implementations. The solution, of course, is to get your hands dirty and start writing your own Policy implementation.

In theory, creating a customized Policy implementation is a simple three-step process:

1. *Extend java.security.Policy.* This is pretty simple.
2. *Override Policy.getPermissions().* This method will be called every time the security architecture needs the PermissionCollection associated with a particular CodeSource. You just have to write the code to hand back the right (by your own definition) set of permissions.
3. *Override Policy.refresh().* This method is available to clients to force the Policy implementation to refresh its Permission settings. In your implementation, you should reset whatever internal flags you keep in order to force re-analysis of the policy settings.

At its heart, this is all there is to it. In practice, however, it's a bit trickier. This is due partly to the fact that each and every Permissions object isn't consulted every time a security check is made. In point of fact, ultimately, the call to `AccessController.checkPermission` finds the ProtectionDomain for each stack frame on the stack[9] and calls the `implies` method on the ProtectionDomain. The ProtectionDomain's `implies` method in turn calls into its PermissionCollection's `implies` method.

This is where things can get a bit tricky. Because certain PermissionCollection classes can (and do) attempt to take shortcuts during the `implies` method, it's not guaranteed that your Permission objects' `implies` methods will be invoked. For example, consider the `implies` method from the java.security.Permissions [10] class:

```
public boolean implies(Permission permission) {
    PermissionCollection pc = getPermissionCollection(permission);
    if (allPermission != null && allPermission.implies(permission))
        return true;
    else
        return pc.implies(permission);
}
private PermissionCollection getPermissionCollection(Permission p) {
    Class c = p.getClass();
    PermissionCollection pc = null;
    synchronized (perms) {
        pc = (PermissionCollection) perms.get(c);
        //check for unresolved permissions
        if (pc == null) {
            pc = getUnresolvedPermissions(p);
            // if still null, create a new collection
            if (pc == null) {
                pc = p.newPermissionCollection();
                // still no PermissionCollection?
                // We'll give them a PermissionsHash.
                if (pc == null)
```

```
                    pc = new PermissionsHash();
                }
            }
            perms.put(c, pc);
        }
        return pc;
    }
```

java.security.Permissions

In the Permissions class, each different type of Permission lives in its own PermissionCollection instance. (The `perms` Hashtable is keyed off the Class object for the Permissions stored in the PermissionCollection stored in the table.) This means that if you create a custom Permission class that intends to represent a "composite" permission (like java.security.AllPermission does), then this composite permission class is going to live in its own PermissionCollection instance. Then, when a FilePermission is checked, only FilePermission-holding PermissionCollections will be asked, and the PermissionCollection containing your composite permission is never consulted. This is why, if you look at the source for the java.security.AllPermission class, it creates its own, custom, PermissionCollection class from the `newPermissionCollection` method[11].

In a way, this design conceptually sacrifices the intent of the system (Permissions determining access control) on the altar of performance. In practical terms, though, it's a fairly safe optimization-aside from AllPermission, most Permission classes will be atomic permission concepts, and won't need to cross PermissionCollection boundaries.

What this means to us as potential Policy implementors, however, is that we're going to need to be aware of this PermissionCollection implementation when we start handing back PermissionCollection instnaces from the `getPermissions` method[12].

Giving your Policy implementation control. Just creating the Policy implementation class isn't the end of it, however-you need some way for your Policy implementation instance to replace the Singleton PolicyFile instance, as well. Java2 provides two ways in which your Policy instance can "take over" for the default Policy instance: at JVM startup time, and via a call to the static `Policy.setPolicy` method.

Of the two, the easier approach is to use the static `Policy.setPolicy` method. This is a secure operation (which means that if the current Policy doesn't allow for it-it's a SecurityPermission("setPolicy") check) that replaces the current Singleton Policy instance with the one passed in. From that point forward, any calls to `Policy.getPolicy` will return your Policy instance and not the default Sun PolicyFile instance.

The drawback to using the `setPolicy` approach is that it requires user code to execute it-but Policy decisions need to be made long before the VM can get to the point of calling that user code. This means that, at startup, the VM will use the default PolicyFile instance, and use the "java.policy" file(s) for policy decisions, at least until your `Policy.setPolicy` executes. This means that not all policy decisions are being made by your Policy instance, and that policy decisions are now decentralized between two places: your implementation, and the default implementation.

In order to centralize all policy decisions within the hands of our Policy instance, we need to get our Policy implementation hooked up from the very start of the VM. Fortunately, the Sun Java2 architecture provides for this, via the "java.security" file.

According to the Sun documentation, the only step required to become the from-startup Policy implementation is to change a single line in the "java.security" properties file. Changing "policy.provider=sun.security.provider.PolicyFile" to a value matching that of your own Policy implementation (for example, "com.javageeks.security.MyPolicy", which is listed below) does the trick. At least, it would appear to do so.

What Sun fails to document is that if the Policy implementation is to be successfully loaded, it must be available to the bootstrap ClassLoader for loading[13]. [5] describes ways in which to work with the bootstrap ClassLoader, but for now, simply create a "classes" directory under the "jre" directory and place the Policy implementation there. Now the bootstrap ClassLoader will pick up the MyPolicy implementation, and all is well-MyPolicy will be consulted for PermissionCollections, not the standard Sun PolicyFile implementation.

Remember, the Java Security model has now entered a new dimension when you do this-no longer will the "java.policy" file have any effect on the security policy in effect for this VM. That decision will be entirely up to your Policy implementation, and your Policy alone[14].

A first Policy implementation: MyPolicy. Take a look at the following code, a simple(?) implementation of a Policy-extending class:

```
package com.javageeks.security;
import java.security.*;
import java.util.*;
/**
 *
 */
class MyPermission extends BasicPermission
{
    static boolean allow;
    public MyPermission()
    {
        super("<<custom permission>>");
    }
    public boolean implies(Permission p)
    {
        System.out.println("<<" + getClass() + ".implies(" + p +
            ") called>>");
        // Special-case permissions always on
        //
        if (p instanceof SecurityPermission)
        {
            return true;
        }
        return allow;
    }
}
/**
 *
 */
class MyPermissionCollection extends PermissionCollection
{
    ArrayList perms = new ArrayList();
    public void add(Permission p)
    {
        perms.add(p);
    }
    public boolean implies(Permission p)
    {
        System.out.println("<<PermissionCollection.implies(" + p +
            ") called>>");
        for (Iterator i = perms.iterator(); i.hasNext(); )
        {
            if (((Permission)i.next()).implies(p))
            {
                return true;
            }
        }
        return false;
    }
    public Enumeration elements()
    {
        return null;
    }
    public void setReadOnly()
    {
    }
    public boolean isReadOnly()
    {
        return false;
    }
    public String toString()
    {
        return super.toString();
    }
}
/**
 *
 */
public class MyPolicy extends Policy
{
    private static PermissionCollection perms;
    static
    {
        System.out.println("<<MyPolicy loaded>>");
    }
    public MyPolicy()
    {
        System.out.println("<<MyPolicy constructed>>");
        if (perms == null)
        {
            perms = new MyPermissionCollection();
            perms.add(new MyPermission());
        }
    }
    /**
     * Evaluates the global policy and returns a
     * PermissionCollection object specifying the set of
     * permissions allowed for code from the specified
     * code source.
     *
     * @param codesource the CodeSource associated with the caller.
     * This encapsulates the original location of the code (where the code
     * came from) and the public key(s) of its signer.
     *
     * @return the set of permissions allowed for code from
     * codesource according to the policy.
     *
     * @exception java.lang.SecurityException if the current thread does
     *            not have permission to call getPermissions
     *            on the policy object.
     */
    public PermissionCollection getPermissions(CodeSource codesource)
    {
        System.out.println("<<getPermissions called for '" + codesource +
            "'>>");
        return perms;
    }
    /**
     * Refreshes/reloads the policy configuration. The behavior of this
     * method depends on the implementation. For example, calling
     * refresh on a file-based policy will cause the file to
     * be re-read.
     *
     * @exception java.lang.SecurityException if the current thread does
     *            not have permission to refresh this Policy object.
     */
    public void refresh()
    {
        System.out.println("<<refresh called>>");
        MyPermission.allow = true;
    }
}
```

MyPolicy.java

To see how it works, let's also take a look at a simple test driver to exercise the Policy and see if it all works:

```
import java.io.*;
public class Driver
{
    /**
     * Test driver
     */
    public static void main (String args[])
        throws Exception
    {
        try
        {
            // Now try to do something
            //
            FileInputStream fis = new FileInputStream("test.file");
            int ch;
            while ((ch = fis.read()) != -1)
            {
                System.out.write(ch);
            }
            System.out.println("");
        }
        catch (SecurityException secEx)
        {
            secEx.printStackTrace();
        }
        // Force a refresh, which for our policy causes everything
        // to be allowed from now on.
        //
        java.security.Policy.getPolicy().refresh();
        try
        {
            // Now try to do something
            //
            FileInputStream fis = new FileInputStream("test.file");
            int ch;
            while ((ch = fis.read()) != -1)
            {
                System.out.write(ch);
            }
            System.out.println("");
        }
        catch (SecurityException secEx)
        {
            secEx.printStackTrace();
        }
    }
}
```

Driver.java

The Driver code is a simple two-step process: attempt to open a file (which, inside the FileInputStream constructor, generates a call to `SecurityManager.checkRead`, which in turn means a `checkPermission` call with a java.io.FilePermission object), then call `refresh` on the Policy instance (which, in the case of the MyPolicy policy, flips the "allow" static from false to true) and try again. Running the code looks like this:

```
 C:\ >run Driver
 C:\ >jre\bin\java -classpath classes -Djava.security.manager Driver
 <<MyPolicy loaded>>
 <<MyPolicy constructed>>
 <<getPermissions called for '(file:/C:/Projects/Papers/JavaPolicy/code/classes/ <no
certificates>)'>>
 <<PermissionCollection.implies((java.io.FilePermission test.file read)) called>>
 <<class com.javageeks.security.MyPermission.implies((java.io.FilePermission test.file read))
called>>
 java.security.AccessControlException: access denied (java.io.FilePermission test.file read)
        at java.security.AccessControlContext.checkPermission(AccessControlContext.java:272)
        at java.security.AccessController.checkPermission(AccessController.java:399)
        at java.lang.SecurityManager.checkPermission(SecurityManager.java:545)
        at java.lang.SecurityManager.checkRead(SecurityManager.java:890)
        at java.io.FileInputStream.<init>(FileInputStream.java:61)
        at Driver.main(Driver.java:15)
 <<PermissionCollection.implies((java.security.SecurityPermission getPolicy)) called>>
 <<class com.javageeks.security.MyPermission.implies((java.security.SecurityPermission
getPolicy)) called&gt;&gt;
 <<refresh called>>
 <<PermissionCollection.implies((java.io.FilePermission test.file read)) called>>
 <<class com.javageeks.security.MyPermission.implies((java.io.FilePermission test.file read))
called>>
 This is a test
 C:\ >
```

(Note that, as usual, the "-Djava.security.manager" property must be set in order for the calls to `SecurityManager.checkRead` to take place; creating your own Policy implementation doesn't change this.) Looking at the output, we can see the MyPolicy class is loaded, then an instance is constructed. When the Driver class is first loaded, MyPolicy's `getPermissions` is called for the CodeSource representing the local directory (in this case, "C:/Projects/Papers/JavaPolicy/code/classes/") with no certificates. MyPolicy responds by returning its internal instance of MyPermissionCollection, which contains one MyPermission object. That MyPermission object, when its `implies` method is invoked, returns either true or false, based on the value of the static `allow` field. When MyPolicy's `refresh` method is invoked, it changes the value of the `MyPermission.allow` field to true, thereby allowing calls to go through. As we can see, from the output, after the call to `refresh`, the second attempt by the Driver class to open the test.file file is successful.

A couple of implementation notes must be highlighted here. To begin, this implementation makes use of a customized PermissionCollection class (MyPermissionCollection) as well as a custom Permission class (MyPermission) for simplicity's sake. A production-quality system would use the standard Permission classes (java.io.FilePermission, java.net.SocketPermission, and so on) and, most likely, the java.security.Permissions class to store them[15]. Secondly, a `refresh` call normally would force a complete reparse/reload/re-whatever of the security policy, rather than affect the current policy in force.

A second Policy implementation: PolicyRoleFile. The MyPolicy implementation above doesn't really do anything exciting-all-or-nothing kinds of permissions aren't particularly useful in a real-world scenario, and simply writing diagnostic material to the console doesn't typically get users or customers very excited. Instead, let's create a (simplified) real-world Policy implementation for solving a common problem: application role-based security permissions.

Within many systems, it's common to have different types of users. Most commonly, users when grouped fall into one of three categories: "user", "administrator", and "guest". The "user" group is the most common scenario-these are the users for whom the application was originally intended. The "administrator" group is those users responsible for the smooth operation of the system. The "guest" group is those users who have no ability to do anything but tour the system and "visit" for a while, without being able to damage it in any way[16].

Each of these different groups needs to have a unique security policy attached to them: admins need the ability to arbitrarily change data, regardless of business rules[17]. This is not an operation we normally want to grant to "average users", and certainly not to guests within the system. One way to provide this kind of capability is to simply hand passwords to the database to the admins, and to have a separate database for "guests" to play around in. This approach works, although it lacks a certain elegance. What's more, it lacks the ability to restrict the admins, as well-we don't always want the admins to have complete control over the entire database, which giving them the passwords does. Ideally, we'd really prefer to have a fine-grained approach to security within the system.

Fortunately, Java Security offers another way without requiring a ton of coding on the programmers' parts. We can create a custom Policy implementation that takes roles into account when allowing or disallowing certain operations to take place; what's more, we can hook it into the normal Java Policy mechanism without too much trouble.

There are a couple of ways to accomplish this within the Java2 Security architecture. One approach would be to create a series of username-driven Permission classes, which returned true or false based on whatever the current user.name value was:

```
/**
 * (Fictional example; no code for this exists in the sample code)
 */
public class DatabasePermission extends Permission
{
    public DatabasePermission(String action, String target)
    { . . . }
    public boolean implies(Permission p)
    {
        if (p instanceof DatabasePermission)
        {
            String username = UserManager.getUsername();
            String role = UserManager.getRole(username);
            if (role.equals("user") || role.equals("admin"))
                return true;
        }
        return false;
    }
}
```

where UserManager is a static class[18] used to retrieve the current user's username and the role to which they belong (user, admin, guest, and so on).

This approach carries a couple of drawbacks. For starters, this example hardcodes all of the possible roles into the Permissions class, making it awkward and difficult to add a new role or remove one. Secondly, the Permissions classes are now doing some of the same work the SecurityManager and/or AccessController classes are supposed to be doing-enforcing policy, not simply deciding it. Lastly, this approach works only as long as no "standard" Permissions, like java.io.FilePermission or java.net.SocketPermission, are considered. Because we can't change the Java Runtime API[19], we can't arbitrarily modify the FilePermission or SocketPermission or any other Runtime API Permission classes.

In the ideal, we'd like to have different permission sets active (incuding both custom and standard Permission types), based on the username[20]. We can accomplish this by creating a custom Policy implementation to:

1. Obtain the current user's username and/or role. This could be as simple as popping up a dialog box, or as complex as obtaining the current OS-specific value (such as looking up the value of an OS-specific environment variable).
2. Look up the permissions allowed for this given user. One such approach would be to continue the format of the "java.policy" mechanism, so that a given java.policy file could look something like the following:

   ```
   grant user "Fred" {
       permission java.io.FilePermission "/usr/fred" "read, write";
   };
   grant user "Don" {
       permission java.security.AllPermission;
   };
   ```

   (This particular syntax isn't the most ideal, nor does it solve any of the problems described in the earlier criticism of the PolicyFile syntax except one: user-based security permissions[21].)
3. Once the Permission set for this user is obtained, then return a PermissionCollection instance (most likely a java.security.Permissions instance) that contains the new Permissions for the specific user.

One of the key advantages this particular idea has is that the usual PolicyFile implementation would remain intact, so that java.policy files of the form:

```
grant codebase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};
grant signedBy "duke" codebase "http://www.javasoft.com/*" {
    permission java.security.AllPermission;
};
grant user "fred_wesley" {
    permission java.security.AllPermission;
};
```

would still be possible. (This particular policy file would grant AllPermission to code loaded from the Extensions directory, to code signed by "duke" that was downloaded from the Sun website, and to code executed by the user "fred_wesley".)

In short, this implementation would duplicate the Java Authorization and Authentication Service, or JAAS, that ships as part of JDK 1.4 and is available as a standalone download from Sun for JDK 1.3. This implementation is still viable, however, and may be simpler to understand at the beginning than JAAS. (JAAS, however, I believe is more powerful and ultimately offers a better path for developers to tread.)

Other Policy implementation ideas. Other possible ideas for Policy implementations include, but certainly aren't limited to:

- *An XML-based version of PolicyFile.* This one is actually rather simple to contemplate-instead of using Sun's PolicyFile syntax to describe the security policy, use XML (and an XML Parser to parse it) instead. This approach doesn't necessarily solve any of the problems listed earlier with the standard PolicyFile implementation (that is, it's still text, it's on the filesystem, and so on), but it does allow us to tie a neat buzzword into the topic.[22]
- *An RDBMS Policy.* Storing policy within an RDBMS offers several advantages, including security-RDBMS vendors spend a lot more time thinking about ways to prevent unauthorized intruders from accessing the data than most programmers do, so why not make use of that? In addition, by structuring the SQL correctly, role-based policies can be established by looking up the user's role (where the username is obtained either from the command-line or environment variable, perhaps) and finding the associated permissions from there.
- *An encrypted version of PolicyFile.* Take the standard PolicyFile implementation (or the XMLPolicyFile version discussed above) and run it through a standard decrypt when parsing the policy file. This way, though the policy file exists on disk, it remains opaque to unauthorized viewers.

It's also reasonable to consider implementations that combine one or more of the ideas described above. The fact is, writing a custom Policy implementation is not an impossible task, nor one that needs to be viewed with trepidation.

Consequences

As with any aspect of this industry, each decision made carries with it risks and consequences that have to be weighed carefully. Creating custom Policy implementations is certainly no exception:

- *Java2 Security learning curve.* It's not feasible to create a custom Policy implementation without a good understanding of CodeSources, Permissions, ProtectionDomains, and ClassLoaders. If any of these subjects remain unfamiliar, then trying to create a custom Policy implementation will make zero sense whatsoever. Fortunately, there are a number of good resources available (see the Bibliography) to help understand the subject better.
- *Security risks.* I freely admit that I'm not a "security expert", not in the traditional sense of the term. This means that any custom Policy implementation I write could potentially have security holes within it I haven't considered, and could be vulnerable to attack. This carries its own advantage, too-if it's a custom Policy implementation, attacks on the standard Policy implementation won't work, either.
- *Difficulty of debugging and/or diagnosing.* During development of the Policy implementation class, it can be extraordinarily tricky to debug and/or diagnose problems. Trying to run under a standard debugger can be difficult, if only because most Java debuggers assume that debugging will begin after JVM startup, and will not allow breakpoints to be set inside your Policy implementation. For the most part, the best debug/diagnostic tool is "good ol' `System.out.println()`".
- *Potential decentralized policy management.* If you make use of the `Policy.setPolicy()` approach to set up your security policy implementation, then your security policy is spread out across two locations: the java.policy file, and wherever your policy is specified (database, XML file, whatever). This means two places system administrators must go in order to modify the security policy, and that means twice as many opportunities for things "to just go wrong".

The consequences would seem to be an impassable obstacle standing between a would-be Policy implementor and success. The truth of the matter, as always, is a murkier thing; although not impossible, creating your own Policy class is not something to be attempted trivially or "just because it looks cool on a resume". Working within the standard Java Policy mechanism (that is, the java.policy-file-based approach) is itself a very flexible system and can yield some powerful results. Only consider creating your own Policy implementation when the java.policy file fails you for some reason (some of which were discussed earlier).

Summary

In this paper, the means by which a new Policy implementation is created was explored. To do that, it was necessary to understand (briefly) how the Policy is consulted for the set of permissions granted to a particular CodeSource, and how these Permissions are considered by the VM when attempting security checks within the heart of the permissions system.

The blunt truth of the matter is a common refrain: "With power, comes complexity". Creating your own Policy implementation gives a Java coder an incredible amount of control over the VM and the security policy it obeys; the tradeoff, as can be seen, is a much more complicated JRE arrangement.

Notes

[1] As cited in [1] , footnote 2, "A recent study concluded that about 50 percent of all CERT-issued alerts are due in part to buffer-overflow errors". (p. 22)
[2] Naturally, however, each one did it in a different manner. <sigh>
[3] From [1] , pg. 37
[4] See the Glossary for a quick summary of CodeSource, Permission, ProtectionDomain and Policy classes, or see [1] or the Java Security documentation in the JDK 1.2/1.3 documentation bundle for in-depth details.
[5] To be quite specific about it, it's actually SecureClassLoader (from java.security), the parent to URLClassLoader, that establishes the permissions for a particular class, at load-time. SecureClassLoader, in its defineClass method, obtains a PermissionCollection instance by calling the Policy instance's getPermissions method. It then takes the returned PermissionCollection instance and creates a ProtectionDomain instance around it (and the CodeSource). This ProtectionDomain instance is later what will be used to find the permissions associated with the code when the AccessController.checkPermission method is called.
[6] On my Win98 system, for example, this translates to "C:\jdk1.3\jre\lib\security\java.policy".
[7] For those unfamiliar with the term, "Five-Nines" refers to the goal of having the production servers available 99.999% of the year. Over 365.25 days, that translates roughly into five minutes of down time, scheduled or not, per year. That doesn't leave a lot of room for mistakes.
[8] Certainly, it could be argued that if Mallory (see [4]) has access to the filesystem, modifications to the java.policy file are the least of your concerns. That only holds so long as the "keys to the kingdom" lie in environments that aren't driven by Java-as more and more Java-based systems assume key roles within the enterprise, this assumption becomes less and less stable.
[9] Walking you through the code from the start of the `checkPermission` call to this point introduces a lot more complexity than is really necessary; you can either walk it through yourself using the source or your favorite Java debugger, or else just take my word for it.
[10] Note that this code comes from the JDK 1.4.0 implementation; in 1.2, the `Permissions` class used the Double-Checked Locking pattern, which has since been proven unsafe within Java.
[11] If you examine the implementation of the Permissions class in more detail, you'll also see that the Permissions class has an optimization explicitly designed to take care of the AllPermission scenario.
[12] This one bit me, personally, when trying to come up with the MyPolicy example implementation you'll see in a bit. Being aware of it makes all the difference.
[13] At first, it would seem like this is an obvious bug on Sun's part. Thinking about it for a second, however, yields some interesting insights about mucking around with non-standard implementations in Java. The AppClassLoader (the ClassLoader responsible for loading code from the CLASSPATH) isn't available to load code at the time the Policy implementation is first needed (that is, the first time a SecureClassLoader-extending class instance looks to the Policy for a PermissionCollection). Therefore, relying on the AppClassLoader (or the ExtClassLoader, for that matter) is unacceptable-and the only other ClassLoader available is the bootstrap loader. (In all honesty, this is why the MyPolicy implementation I describe below has a static initializer block to print out a message when the class gets loaded-I spent two days trying to figure out why MyPolicy wasn't being used.)
[14] It's always possible, of course, to create an instance of the sun.security.provider.PolicyFile class and use it from within your own Policy class (a la Decorator-see [3]), but the fact remains that the ultimate decision still rests in the hands of your Policy instance.
[15] Again, it's only when you need to support composite Permission classes (like java.security.AllPermission) that customized PermissionCollection classes need to come into play.
[16] By entering bogus data, for example. "guest" roles are typically reserved for demos of an online system, or for product demos, and so on.
[17] Usually, they need this ability because there are still a few leftover bugs in the system. It's not

uncommon for that privilege to be needed for other purposes, but by far, the number one need for this capability is to find "lost" data, correct mistyped data, and so on, many (if not most) of which arise from programmer bugs.

[18] That is, a class with all methods declared `static`; in effect, a Singleton.

[19] OK, we could change it, if we really wanted to (see [5] for details), but it's definitely not high on my list of recommended solutions to the problem.

[20] Actually, usually it will be based on the user's given role (usually "guest", "admin", or "user", most likely).

[21] In all likelihood, you'll grant Permissions to "roles", and then describe users and being one (or more) roles, but this syntax serves to demonstrate the concept, which is all I'm looking to do here.

[22] In fact, Stu Halloway of DevelopMentor has already written such an implementation, called JSeX, Java Security in XML, and is available from his website at http://staff.develop.com/halloway

Glossary

CodeSource. (java.security.CodeSource): A class representing the origination point of code executing within the VM. From the javadocs, "This class extends the concept of a [applet's] codebase to encapsulate not only the location (URL) but also the certificate(s) that were used to verify signed code originating from that location." Effectively, as its name implies, this class wraps around the source of some code, be it .jar file, http URL, or some other location describing the code's location. If any certificates are present from the code's source, those certificates are present in the CodeSource object representing that code. CodeSources will be unique, one for each location/URL.

Mallory. Name referring to a fictitious individual bent on performing malicious actions within the system. From .

Permission. (java.security.Permission): A class encapsulating a request for a sensitive operation. From the javadocs, "Abstract class for representing access to a system resource. All permissions have a name (whose interpretation depends on the subclass), as well as abstract functions for defining the semantics of the particular Permission subclass." Each Permission-extending class represents the control of a sensitive operation, such as the opening of a socket (java.net.SocketPermission), opening access to a file for either read or write (java.io.FilePermission), and so on. The Permission-extending class itself is responsible for the implementation of the implies(), equals(), getActions(), and hashCode() methods.

PermissionCollection. (java.security.PermissionCollection): A class encapsulating a collection of Permission objects (which is almost word-for-word what the javadocs say). In essence, this class is a shorthand way to check a group of Permissions without having to call the implies method on each individual Permission object (assuming the PermissionCollection object implements its own implies method that way). The Sun JDK ships with one PermissionCollection-extending class, java.security.Permissions, and several other Permission classes (such as java.security.AllPermission) have their own anonymous PermissionCollection-extending classes.

Policy. (java.security.Policy): The class responsible for representing the security policy for this particular VM in Java. From the javadocs, "This is an abstract class for representing the system security policy for a Java application environment (specifying which permissions are available for code from various sources). That is, the security policy is represented by a Policy subclass providing an implementation of the abstract methods in this Policy class." In practical terms, the Policy class is responsible for establishing the PermissionCollections for each CodeSource given to it, via the getPermissions method.

ProtectionDomain. (java.security.ProtectionDomain): From the javadocs, "This ProtectionDomain class encapulates the characteristics of a domain, which encloses a set of classes whose instances are granted the same set of permissions. In addition to a set of permissions, a domain is comprised of a CodeSource, which is a set of PublicKeys together with a codebase (in the form of a URL). Thus, classes signed by the same keys and from the same URL are placed in the same domain. Classes that have the same permissions but are from different code sources belong to different domains. A class belongs to one and only one ProtectionDomain." A ProtectionDomain, simply put, is a { CodeSource, Permissions } tuple. ProtectionDomains will usually, but not always, correspond on a one-to-one basis with the "grant" entries in a java.policy file.

Bibliography

- **[1]** *Inside Java2 Security*, by Li Gong. Addison-Wesley 1999 (ISBN: No clue)
- **[2]** *Inside the Java2 Virtual Machine*, by Bill Venners. IDG Books 2000 (ISBN: No clue)
- **[gof]** *Design Patterns*, by Erich Gamma, Richard Helm, Richard Johnson, John Vlissides. Addison-Wesley 1995 (ISBN: No clue)
- **[Schneier]** *Applied Cryptography*, by Bruce Schneier. John Wiley & Sons 1996 (ISBN: No clue)

Copyright

Colophon

This paper was written using a standard text editor, captured in a custom XML format, and rendered to PDF using the Apache Xalan XSLT engine and the Apache FOP-0.17 XSL:FO engine. For information on the process, contact the author at tneward@javageeks.com. Revision $Revision: 1.1 $, using whitePaper.xsl revision $Revision: 1.1 $.