Java^{TM} Access Control Mechanisms*

Anne Anderson

October 8, 2001

Version 1.5

1 Introduction

This document describes the mechanisms available in J2SE^{TM} software as of version 1.4 for doing access control. It describes the way in which AccessControlExceptions are generated, the classes that are involved, and the various authorization checking models that are supported for application writers. It also describes some ways in which the Java^{TM} access control mechanisms have been extended.

2 Different mind-set

Java technology has historically had a different take on access control from what most security people are used to. We are used to asking "what do we trust this person to do?" Java technology started out being concerned primarily with "what do we trust this code to do?"

This paranoia about code trust is due to Java technology's use in applets that are downloaded from remote locations. Allowing such code to execute on your machine without proper controls is like inviting a stranger into your home and saying "Do whatever you want."

With the rise in distributed Java applications and the use of Java technology in multi-user environments, applications often need to make access control decisions based on who is making a particular request. The Java^{TM} Authentication and Authorization Service (JAAS), introduced in Java 2 version 1.3, provided for the first time an authenticated way of determining what person was executing code. The JAAS mechanisms have been integrated into J2SE version 1.4, so Java applications can now control what a particular person (or other entity) is allowed to do.

Both types of approaches will be discussed in this document.

3 Basic Authorization Example: `AccessTest.java`

To get started, here is a simple application that can exercise the basic Java access control mechanisms. The application simply writes the letter "`A`" to a file.

3.1 AccessTest.java

```
import java.io.FileOutputStream;

public class AccessTest {
    public static void main(String[] args) {
        try {
            FileOutputStream fos = new FileOutputStream("/home/aha/out", false);
            fos.write('A');
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

3.2 What happens when we run AccessTest?

If we run `AccessTest` normally, the character "`A`" is written to file `/home/aha/out`, and we get no exception.

```
$ java AccessTest
$ more /home/aha/out
A
$
```

If, however, we run `AccessTest` with a "`SecurityManager`" (explained below in Section 4.4), then we get an `AccessControlException` as follows.

```
$ java -Djava.security.manager AccessTest
java.security.AccessControlException: access denied
  (java.io.FilePermission /home/aha/out write)
    at java.security.AccessControlContext.checkPermission
      (AccessControlContext.java:273)
    at java.security.AccessController.checkPermission
      (AccessController.java:404)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:545)
    at java.lang.SecurityManager.checkWrite(SecurityManager.java:978)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:172)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:105)
    at AccessTest.main(AccessTest.java:6)
```

Yet again, if I create a file called "`/home/aha/.java.policy`" and containing the following text:

```
grant codeBase "file:${user.dir}/*" {
    permission java.io.FilePermission "/home/aha/*", "write";
};
```

and if I then run `AccessTest` again with a `SecurityManager`, the program executes normally without any exception.

```
$ java -Djava.security.manager AccessTest
$ more /home/aha/out
A
$
```

The following sections attempt to explain what causes these behaviors.

3.3 java.io.FileOutputStream

Here is the code in `java.io.FileOutputStream` that was invoked when `AccessTest` tried to create the new `FileOutputStream`[1]:

```
public FileOutputStream(String name, boolean append)
    throws FileNotFoundException
{
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
       security.checkPermission(new FilePermission(name, ‘‘write’’));
    }
    fd = new FileDescriptor();
    if (append) {
       openAppend(name);
    } else {
      open(name);
    }
}
```

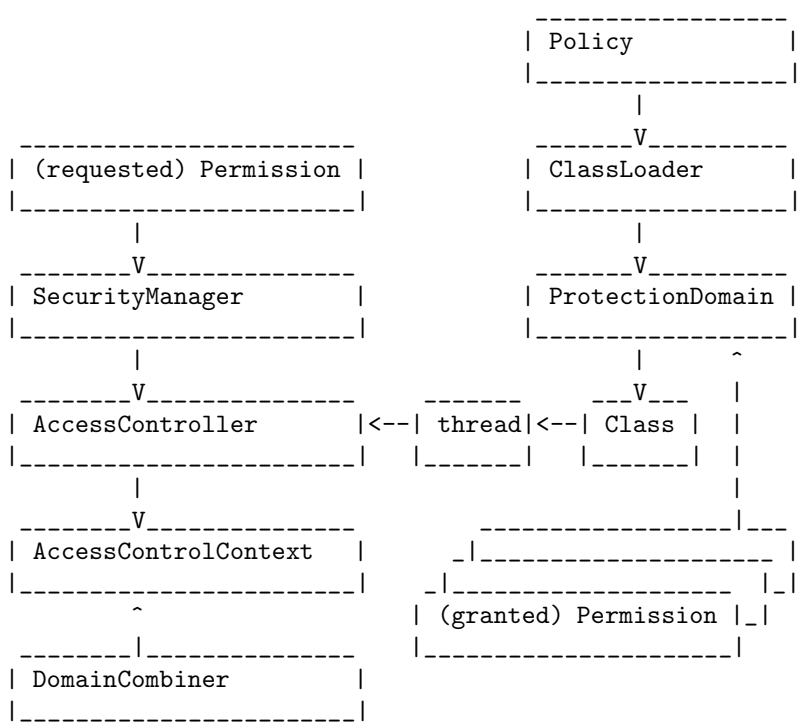
All access to resources in the Java platform is done through the Java API. The `FileOutputStream` example above is typical Java API code for performing access to a resource. The Java API has been carefully written to check with the "`SecurityManager`", if one is installed, on each request by application code to access a resource. The task of the `SecurityManager` is to see if the current thread of execution has permission to perform the requested resource access.

One exception is where the Java API supports extensibility by allowing the user to supply implementations of classes in the Java API that perform resource access. In such cases, it is the provider of the implementation that is responsible for making appropriate access control checks. Such extensible classes include: `java.awt.Toolkit, java.security.Provider`, and `java.net.Socket`. It is very important to use implementations for such classes that make the same types of checks as the standard Java API classes.

4 How is an `AccessControlException` Generated?

Several components work together to set up the structures that allow `AccessControlException`s to be generated.

[1] This example was modified slightly to use the preferred `SecurityManager checkPermission` method rather than the old-style `SecurityManager checkWrite` method.



4.1 java.security.Permission

In the Java platform, sub-classes of `Permission` are used to represent both requested and granted access permissions.

For example, a `FilePermission` takes two `String` parameters: `target` and `action`. It uses the `target` to represent a path to a file (or to a directory subtree), while `action` is used to represent specific actions that are permitted or requested with respect to that path.

The following constructor creates a set of permissions that includes the rights to either read or write to any file in the directory subtree under "`/home/aha/`".

```
new FilePermission("/home/aha/*", "read,write");
```

The following constructor creates a permission to write to the specific file "`/home/aha/out`".

```
new FilePermission("/home/aha/out", "write");
```

Another `Permission` sub-class, `RuntimePermission`, uses just a `target`. The following constructor creates a permission to set a new `SecurityManager`.

```
new RuntimePermission("setSecurityManager");
```

Each `Permission` has an "`implies`" method that is used to check a requested permission against a granted permission. Each `Permission` sub-class knows how to determine whether a specific requested `Permission` is implied by (i.e. is a subset of) a possibly more general granted `Permission`.

In the case of `FilePermission`, the "`implies`" method must recognize that "`*`" in the `target` is a "wild-card" matching any path that has the prefix "`/home/aha/`". It must also recognize that "`write`" is a subset of the set of actions "`read,write`".

Multiple `Permission` objects of the same class type are stored in a `PermissionCollection` object. Multiple `PermissionCollection` objects, each which may contain a different class type collection, are stored in a `Permissions` object (which is actually a sub-class of `PermissionCollection`).

The Permission class "`AllPermission`" is a shortcut for indicating that all permissions are granted.

4.2 `java.security.ProtectionDomain`

When a `ClassLoader` loads a class, it associates the class with a set of permissions granted to that class, encapsulated in a "`java.security.ProtectionDomain`". These granted permissions are obtained from the system Policy object. This is how a `ClassLoader` sets up `ProtectionDomain`s:

1. Find location (URL) where class bytes are located.
2. Read bytes for the class itself.
3. Read any `Certificate[]` associated with the class bytes (via a signed JAR file or custom mechanism).
4. Verify any signatures on class bytes using the `Certificate[]`. The certificates used as trust anchors for this verification are located in `file:${java.home}/lib/security/cacerts`.
5. Verify that all `Certificate[]` associated with the class bytes match the `Certificate[]` associated with any other class already loaded from this package.
6. If class is from a package in `java.*`, then create the class and associate it with an implicit `System ProtectionDomain` that has `AllPermission`. If the class is not from a package in `java.*`, then continue.
7. If class is from a package that is already associated with a `ProtectionDomain`, create the class, with a reference to the existing `ProtectionDomain`. Otherwise, continue.
8. Create a `CodeSource` object encapsulating the location from which the code bytes were loaded and any certificates associated with them: `CodeSource = new CodeSource(URL, Certificate[])`.
9. Generate a heterogeneous PermissionCollection populated with statically bound permissions granted to this `CodeSource`. The source of these could be the `ClassLoader` or the `Policy` class.
10. Create a new `ProtectionDomain` encapsulating the `CodeSource` and the associated permissions: `ProtectionDomain = new ProtectionDomain(CodeSource, PermissionCollection)`.
11. Create the new class, with a pointer to the new `ProtectionDomain`.

4.3 `java.security.Policy`

The `ClassLoader` may invoke `Policy.getPermissions(CodeSource)` to obtain the set of permissions granted to code loaded from a particular location. `Policy` is an abstract class. There can be only one `Policy` object in effect at a time, but the object can be replaced by code with sufficient permissions using the static method `Policy.setPolicy(Policy)`.

A `Policy` implementation must have some way of associating permissions with locations from which class code has been obtained, and possibly also with certificates with which class code has been signed.

The default `Policy` implementation is `sun.security.provider.PolicyFile`. `PolicyFile` reads one or more text files that map permissions to locations and signers. This is done at the time `PolicyFile` is created

and initialized. PolicyFile creates sets of granted Permission objects representing the permissions granted in the files. See Appendix A for more details on policy files.

By default, the file ${JAVA_HOME}/jre/lib/security/java.security contains the locations from which PolicyFile will read policy files. The file locations are in properties named "policy.url.*". The default locations for policy files are file:${java.home}/lib/security/java.policy and file:${user.home}/.java.policy.

The default policy file(s) can be supplemented for a particular application by using the following option on the command line:

```
-Djava.security.policy[=<URL to policy file>]
```

If the java.security file permits it, default policy file(s) can also be overridden for a particular application by using "==" rather than "=" as follows:

```
-Djava.security.policy[==<URL to policy file>]
```

When Policy.getPermissions(CodeSource) is invoked, PolicyFile examines its sets of permissions for those associated with the specified CodeSource and returns them.

PolicyFile reads from the policy file(s) at startup and when the refresh() method is invoked. Changes to the policy file(s) after the application or applet has started running do not affect the permissions of existing ProtectionDomains, however. Other Policy providers may behave differently.

4.4 java.lang.SecurityManager

The SecurityManager is the gateway for all permissions checking. It contains a large number of methods having names starting with "check", for example "checkWrite(String file)". Historically, SecurityManager managed all the permissions itself, and many applications provided custom implementations of SecurityManager. For this reason, all access control checking must go through the SecurityManager interfaces for backwards compatibility.

In current versions of the Java platform, however, all the "check<something> methods create an appropriate Permission object to represent the requested access and use it to call a single method: checkPermission(Permission). This method invokes AccessController.checkPermission(Permission).

There is only one SecurityManager in effect at a time, although code with sufficient permissions can install a new SecurityManager.

By default, Java applications do not run with a SecurityManager, but Java applets do. The SecurityManager can be specified on the command line using the following option:

```
-Djava.security.manager[=<classname>]
```

It may also be set within the application by invoking:

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

4.5 java.security.AccessController

AccessController.checkPermission(Permission) takes a snapshot of the ProtectionDomains associated with the classes in the current thread of execution. This snapshot is called an "AccessControlContext". The requested Permission is tested against the Permissions in each ProtectionDomain. Each ProtectionDomain in the AccessControlContext must have the requested Permission.

Permission checking is very expensive, although the implementation has taken great pains to make it as efficient as possible. In many cases ProtectionDomains are merged or otherwise optimized with respect to a particular AccessControlContext. The PermissionCollections used by ProtectionDomains contain multiple PermissionCollections, each containing only Permission objects of the same class type. When ProtectionDomain.implies(Permission) is invoked, only the Permissions of the same type as the requested Permission need to be examined.

In the AccessTest example, PolicyFile read permissions from the default policy file, which contains no FilePermissions at all. Therefore, the ProtectionDomain associated with AccessTest.run() does not contain any FilePermissions, and so the request to write is denied.

```
THREAD STACK                ProtectionDomain        PermissionCollection
=======================     ================        ====================
FileOutputStream.<init> -> System Domain        -> AllPermission
AccessTest.run          -> /home/aha/bin Domain -> default Permissions
AccessTest.main         -> /home/aha/bin Domain -> default Permissions
```

4.6 java.security.DomainCombiner

A DomainCombiner is a way to dynamically update the ProtectionDomains associated with an AccessControlContext.

A DomainCombiner can be associated with an AccessControlContext via one of the constructors. This AccessControlContext is then bound to the current execution thread by using AccessController.doPrivileged(..., context). Subsequent calls to AccessController.getContext or AccessController.checkPermission cause the method DomainCombiner.combine to be invoked.

DomainCombiner.combine typically merges the ProtectionDomains from one context with those from another, but can perform more complex operations. For example, a DomainCombiner is used to associate a Subject identity with the ProtectionDomains of a particular thread of execution when Subject.doAs(Subject) is called. See Section 5.4 for more information.

5 Authorization Check Models In J2SE Version 1.4

The basic authorization security mechanisms have now been described, along with the most common way in which they are used.

This section describes additional ways of using the authorization mechanisms provided in J2SE version 1.4.

5.1 Access based on ProtectionDomains of current thread

This is the model already described above. A class, such as FileOutputStream, that is invoked to perform access to some protected resource calls SystemManager.check<something>() or SystemManager.checkPermission(Permission). This causes the SystemManager to invoke the AccessController to check the Permissions for each ProtectionDomain for each class in the current thread of execution. If any class in the current thread fails to have the necessary Permission, an AccessControlException is generated.

```
THREAD STACK
=======================
AccessControlContext.checkPermission(FilePermission())
```

```
----V(AccessControlContext)V----
AccessController.checkPermission(new FilePermission())
SecurityManager.checkWrite("/home/aha/out")
FileOutputStream.<init>
AccessTest.run
AccessTest.main
----^(AccessControlContext)^----
```

5.2 Access based on `ProtectionDomain` of `doPrivileged` caller

The `AccessController` has a method called `doPrivileged()`. When a method invokes `doPrivileged`, it causes subsequent `AccessController.checkPermission(Permission)` calls to create an `AccessControlContext` that contains only the `ProtectionDomain`s associated with the class that called `doPrivileged` or those above it in the stack of the executing thread.

```
THREAD STACK
==============================
AccessControlContext.checkPermission(SecurityPermission("getProperty.keystore.type"))
----V(AccessControlContext)V----
AccessController.checkPermission(SecurityPermission("getProperty.keystore.type"))
SecurityManager.checkPermission(new SecurityPermission("getProperty.keystore.type"))
Security.getProperty("keystore.type")
AccessController.doPrivileged()
KeyStore.getDefaultType()
----^(AccessControlContext)^----
Application.run
Application.main
```

This is used where a class is trusted to perform a dangerous action "safely" and "appropriately".

For example, applications are not allowed to read system properties at will, since some will contain privileged information. The `java.security.KeyStore` class, however, is trusted to access properties safely. This class calls `AccessController.doPrivileged` before trying to access properties it needs. In the example above, the `KeyStore` class reads a property that it wants to make available to users: "keystore.type", which is the default keystore type.

Another example is a `ClassLoader`. Whenever a method invokes a method in a previously unreferenced class, the `ClassLoader` is invoked to create the new class. The `ClassLoader` has to perform many actions that ordinary code is not trusted to do (such as getting policy, instantiating new classes, creating and assigning `ProtectionDomain`s). The `ClassLoader`, however, is trusted to do these actions safely and appropriately so that system security is not compromised, so `ClassLoader` calls "`AccessController.doPrivileged()`" prior to performing those actions.

Note that "`doPrivileged`" does not give a class any permissions it does not already have. `ClassLoader` and `Keystore` are part of the trusted `java.*` namespace, and thus have all permissions already. An application class that calls "`doPrivileged`" will be able to exercise only those permissions that are granted to it by the `Policy` that is in effect.

5.3 Access based on `ProtectionDomain`s of another thread

When a server thread performs actions on behalf of another thread, the server thread needs to evaluate the action in the context of the requesting thread, not its own context.

The requesting thread calls `AccessController.getContext()` to create a copy of its own `AccessControlContext`. It then passes that to the server thread along with the requested action.

The server thread uses this `AccessControlContext` to call `AccessControlContext.checkPermission()` to check the requested permissions.

```
Requester Thread Stack
==============================
accessControlContextA = AccessController.getContext()
----V(accessControlContextA)V----
ClassMethodC
ClassMethodB
ClassMethodA
----^(accessControlContextA)^----

Server Thread Stack
==============================
accessControlContextA.checkPermission(Permission)
----V(AccessControlContext)V----
ClassMethodE
ClassMethodD
----^(AccessControlContext)^----
```

The requested `Permission` must be granted in all `ProtectionDomains` in `accessControlContextA` and in `ClassMethodE` and in `ClassMethodD`.

5.4 Access based on `Subject` identity

`javax.security.auth.Subject` represents a grouping of related information for a single entity, such as a person. Such information includes the `Subject`'s identities as well as its security-related attributes (passwords, cryptographic keys, etc.).

`javax.security.auth.login.LoginContext()` (which is similar to Pluggable Authentication Modules PAM) can create a `Subject` that contains one or more authenticated identities, represented as instances of `java.security.Principal`. This `Subject` can be associated with a thread's `AccessControlContext` by using the static method `Subject.doAs(Subject, PrivilegedAction)` or `Subject.doAsPrivileged(Subject, PrivilegedAction[, AccessControlContext])`. Once associated with a thread's `AccessControlContext`, all `ProtectionDomains` for that thread will contain the `Principal[]` associated with the `Subject` (due to internal use of a special `SubjectDomainCombiner`).

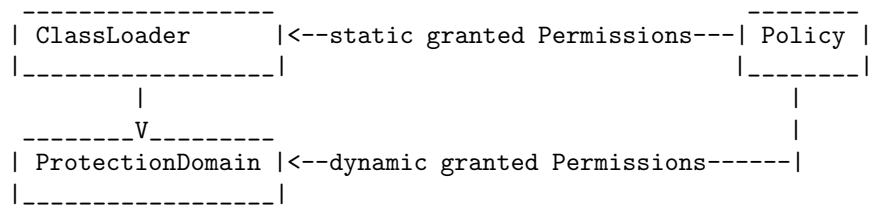
In J2SE version 1.4, a policy file can specify permissions for particular principals, such that the specified principals must be in the ProtectionDomains of the AccessControlContext being checked for access permissions. The syntax for a policy file `grant entry` that includes `Subject`-based permissions is in Appendix A.

```
THREAD STACK
============
AccessControlContext.checkPermission()
----V(AccessControlContext)V----
----V(AccessControlContext + SubjectA)V----
AccessController.checkPermission()
SecurityManager.checkWrite()
ClassMethodC
ClassMethodB
----^(AccessControlContext + SubjectA)^----
Subject.doAs(SubjectA)
ClassMethodA
----^(AccessControlContext)^----
```

5.5 Static and dynamic policy

In J2SE version 1.4, there is a new `ProtectionDomain` constructor that takes a `ClassLoader` and `Principal[]` in addition to the usual `CodeSource` and `Permissions`.

If a `ProtectionDomain` is created using this new constructor, then the `ProtectionDomain.implies` method checks with `Policy` each time it is called for any permissions to use in addition to the static `Permissions` with which the `ProtectionDomain` was created.



The reference implementation of `Policy` – `PolicyFile` – does NOT re-read the policy file each time. It is possible, however, to create a custom `Policy` implementation that does obtain fresh policy information on each call.

5.6 `GuardedObject`

A `GuardedObject` encapsulates a target object and a `Guard` object, such that access to the target object is possible only if the `Guard` object allows it. Once an object is encapsulated by a `GuardedObject`, access to that object is controlled by the `GuardedObject.getObject()` method, which invokes the "`checkGuard`" method on the `Guard` object that is guarding access. If access is not allowed, an exception is thrown.

```
Server Thread Stack
============================
guardedObjectA = new GuardedObject(Object, Guard)
ClassMethodE
ClassMethodD

Requester Thread Stack
============================
----V(AccessControlContext)V----
Guard.checkGuard()
Object = guardedObjectA.getObject()
ClassMethodC
ClassMethodB
ClassMethodA
----^(AccessControlContext)^----
```

This would be used much like the "Access Based on ProtectionDomains of Another Thread" model above, in cases where the caller is unable to provide its own `AccessControlContext` (or other information needed) to the resource provider for various reasons. It might also be used where more complex types of access checks need to be performed than are supported by the `AccessController`, or where the resource is not aware of access control checks.

`Permission` implements the `Guard` interface, so a `Permission` can be used as the `Guard` for an object.

5.7 `java.security.SignedObject`

A `SignedObject` contains another serializable Object and a signature. The class contains a "`verify`" method that takes a public key that should correspond to the private key used to sign the object.

A `SignedObject` is used to authenticate objects and to protect their integrity. It can provide an unmodifiable token (or more accurately, a token where any modification can be detected), protect objects stored outside the Java runtime, or be nested to create a logical sequence of signatures for purposes of authorization and delegation.

While this class is not an authorization mechanism in itself, it can be used as part of authorization.

6 Extending the Java Security Model

6.1 Open Services Gateway Initiative (OSGi) dynamic permissions

The Core Platform Expert Group of the Open Services Gateway Initiative (OSGi) consortium (http://www.osgi.org) needed to be able to change the permissions associated with code running at various service gateways. This change needed to be specified remotely and without having to reload the code at any of the service gateways. Due to footprint constraints (the deployment platform must be something small like PJava), they were not able to leverage the dynamic permissions design implemented in J2SE version 1.4.

The Java 2 Security Model is too restrictive for an environment such as the OSGi service platform where multiple services from different vendors may be downloaded on demand, without a corresponding entry in the system policy file. Also, the permissions associated with a service's bundle of downloaded code may need to change over time, for example, when the bundle gets updated or when its configuration changes.

The next release of the OSGi specification (due to be released at the beginning of October 2001) defines a PermissionAdmin service that provides a standard interface for assigning Java 2 permissions to bundles based on their location (codebase). This allows for permissions to be set up before a bundle is installed on an OSGi service platform, and for the permissions to be modified at any time during the bundle's lifecycle. Updates to the permissions become effective immediately and persist across restarts of the OSGi service platform.

Java Embedded Server (JES), SUN's implementation of the OSGi standard, implements a special permission collection ("PermissionCollectionWrapper") that is used by the bundle classloader to construct a bundle's protection domain. PermissionCollectionWrapper encapsulates another permission collection that contains the real permissions granted to the bundle, and delegates all method calls (including "implies" - which is not final!) to it.

Every time when a bundle's permissions change (by calling PermissionAdmin.setPermissions), its new permissions are injected into the PermissionCollectionWrapper and will be used during the next permission check that involves the bundle's classes.

6.2 SPKIPolicy

Two researchers at Helsinki University of Technology published a paper at NDSS '99 [5] in which they described an implementation based on JDK1.2 that used Simple Public Key Infrastructure (SPKI) certificates [2] rather than a static configuration file. Their goal was both to provide dynamic policy and to distribute policy across machines.

They implemented `SPKICertificate` as a sub-class of the abstract `Certificate` class. They wrote a `Policy` sub-class called `SPKIPolicy`. `SPKIPolicy` derives the `PermissionCollection` for each `ProtectionDomain` from valid SPKI certificate chains. The chains are rooted in a key with alias "`self`" in a local KeyStore. `SPKIPolicy` returns a new sub-class of `PermissionCollection` called "`DynamicPermissions`". `AccessController.implies()` invokes these new `DynamicPermissions`. `DynamicPermissions` asks `SPKIPolicy` for the requested `Permission`.

SPKI certificates are attached to class files in a JAR file, just as regular certificates are attached to signed JAR files. These certificates contain the set of `Permission`s that the implementer of the class knows the class will need in order to run successfully. Standard `ClassLoader`s already retrieve certificates attached to JAR class files and store them in the `ProtectionDomain`.

A new JAR file verifier had to be written, since the SPKI certificates can't be used to verify a signature on the class. Instead, these certificates contain the set of `Permission`s needed by the class. These certificates are encapsulated in the `ProtectionDomain` associated with each class.

They expressed Java permissions in SPKI certificates as follows:

```
(tag (java-permission (type FilePermission)
                      (target /home/aha/)
                      (action "write"))
```

They parsed these permissions to create appropriate instances of Java `Permission` objects just as `PolicyFile` parses a policy file and creates appropriate `Permission`s.

When a `Permission` is verified at runtime for a class, the certificate chain is evaluated to see if the needed `Permission`s are actually granted at that time. `Permission`s from certificates along the chain are intersected to obtain the resulting certificate.

The researchers stored the SPKI certificates required to build complete chains in a local filestore as a proof-of-concept. The `SPKIPolicy` class knows how to look up certificates needed to complete a chain. They published another paper suggesting storing the certificates in the DNS directory. Tuomas Aura [1] has analysed several different algorithms for SPKI chain reduction.

A Policy File Syntax

The syntax for a policy file to be read by `sun.security.provider.PolicyFile` follows.

```
[keystore "some_keystore_url", "keystore_type";]
<grant entries>
```

The "`keystore`" entry is used for obtaining public keys for verifying any signed classes specified in the policy file. It must be used if there are any "`signedBy`" clauses in the policy file.

A.1 `grant` entry

The syntax for a "`grant entry`" follows.

```
grant [signedBy "<signer_names>",]
    [codeBase "<URL>",]
    [principal [<principal_class_name>] "<principal_name>",]
    [principal [<principal_class_name>] "<principal_name>",]
    ...
{
   <permission entries>
};
```

If the "`signedBy`" clause is present in the header of the `grant entry`, then the specified permissions are granted only to those `ProtectionDomain`s whose classes were signed by the specified signers.

If the "`codeBase`" clause is present, then the permissions are granted only to those `ProtectionDomain`s whose classes were obtained from the specified URL.

If "`principal`" clauses are present, then the permissions are granted only to an `AccessControlContext` associated with a `Subject` containing the specified authenticated `Principal[]`s.

If the <`principal_class_name`> ''<`principal_name`>'' pair is a single quoted string, it is treated as a `KeyStore` alias. The keystore is queried for an X509 Certificate associated with the alias. If found, the `principal_class` is automatically treated as a `javax.security.auth.x500.X500Principal`, and the `principal_name` is treated as the subject distinguished name from the certificate. If the certificate is not found, the entire `grant` entry is ignored.

A.2 `permission` entry

The syntax for a "`permission entry`" follows.

```
permission <permission_class_name>
       "<target_name>"[, "<action>",]
       [signedBy "<signer_names>"];
```

Typically, a "`permission`" clause is converted to a `Permission` object as follows:

```
new <permission_class_name>("<target_name>", "<action>");
```

Depending on the `Permission` class type, the "`target_name`" may include wild cards such that multiple targets match that specified. Likewise, the "`action`" string may contain a list of permitted actions, any one of which will match that specified.

If the "`signedBy`" clause is present in a `permission entry`, then the "`permission_class_name`" must be a class signed by the indicated "`signer_names`".

References

[1] Tuomas Aura. Comparison of graph-search algorithms for authorization verification in delegation networks. In *Proceedings of 2nd Nordic Workshop on Secure Computer Systems*, http://saturn.hut.fi/html/staff/Publications/papers/aura/aura-nordsec97.ps, 1997.

[2] C. Ellison, G. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. Experimental RFC 2693, IETF, http://www.ietf.org/rfc/rfc2693.txt, Sep 1999.

[3] Li Gong. *Java^{TM} 2 Platform Security Architecture*. Sun Microsystems, Inc., http://java.sun.com/j2se/1.4/docs/guide/security/spec/security-spec.doc.html, 1.0 edition, Oct 1998.

[4] Li Gong. *Inside Java^{TM} 2 Platform Security: Architecture, API Design, and Implementation*. The Java^{TM} Series. Addison-Wesley, 1999.

[5] Pekka Nikander and Jonna Partanen. Distributed policy management for JDK 1.2. In *Proceedings of the 1999 Network and Distributed System Security Symposium*, http://www.isoc.org/isoc/conferences/ndss/99/proceedings/papers/nikander.pdf, Feb 1999.

[6] Scott Oaks. *Java Security*. JAVA Series. O'Reilly, May 1998.

[7] Sun Microsystems, Inc. API for privileged blocks. Technical report, Sun Microsystems, Inc., http://java.sun.com/j2se/1.4/docs/guide/security/, Sep 1998.

[8] Sun Microsystems, Inc. Default policy implementation and policy file syntax. Technical report, Sun Microsystems, Inc., http://java.sun.com/j2se/1.4/docs/guide/security/, Oct 1998.

[9] Sun Microsystems, Inc. Permissions in the Java^{TM} 2 SDK. Technical report, Sun Microsystems, Inc., http://java.sun.com/j2se/1.4/docs/guide/security/, Oct 1998.

[10] Sun Microsystems, Inc. Security managers and the Java^{TM} 2 SDK. Technical report, Sun Microsystems, Inc., http://java.sun.com/j2se/1.4/docs/guide/security/, Oct 1998.