Understanding Java™ 2 Platform Security Permissions—A Practical Approach

Larry Koved
Marco Pistoia
Aaron Kershenbaum
Java Security Team
IBM T.J. Watson Research Center
Learn about the authorization features in the Java™ 2 Platform, Standard Edition (J2SE™) security model

Specifically, focus on the permission/authorization model and the Permission API

See a prototype tool which demonstrates an automated technique for determining which Permissions are required by a Java technology-based program
Learning Objectives

• As a result of this presentation, you will be able to:
  – Understand the Java 2 Platform Permission/Authorization Model
  – Define “grant” statements for authorizing code to perform restricted operations
  – Use the Java 2 Permission API
  – Understand the concept of privileged code and when/how to use it
Speaker’s Qualifications

- **Dr. Aaron Kershenbaum**
  
  is a Research Staff Member at IBM Research, and former Professor at Polytechnic University in New York.

  His current focus is security for the Java platform.

- **Mr. Larry Koved**
  
  is a Research Staff Member at IBM Research. He co-leads IBM’s security team for the Java platform.

- **Mr. Marco Pistoia**
  
  is an Advisory Software Engineer at IBM Research, working on security for the Java platform.

  He has co-authored several books on Java technology, including “Java™ 2 Network Security”.
Agenda

- Why Java™ Platform Security?
- Java 2 Platform Authorization Model
- Scenario I and II
- Permission API
- Demo
Java security assists in preventing the following types of attacks:

<table>
<thead>
<tr>
<th>Attack Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System modification</td>
<td>A program gets read/write access and makes changes to the system</td>
</tr>
<tr>
<td>Privacy invasion</td>
<td>A program gets read access and steals sensitive information</td>
</tr>
<tr>
<td>Denial of service</td>
<td>A program uses system resources without authorization</td>
</tr>
<tr>
<td>Impersonation</td>
<td>A program masquerades as the real user of the system</td>
</tr>
</tbody>
</table>
Java 2 Platform, Authorization

- Resource Protection
- SecurityManager
- Security Policy
- CodeSource
- ProtectionDomain
- SecureClassLoader
- Run-time access controls
Access to Protected External Resources

- All protected external system resources, including native libraries, are accessible only via trusted code.

Composition of a Java application environment
The installed Security Manager is only active on request.

It checks a Permission only when it is called by other system functions.
## Default SecurityManager Controls

<table>
<thead>
<tr>
<th>Package Type</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td>SocketPermission, RuntimePermission</td>
</tr>
<tr>
<td>Thread</td>
<td>RuntimePermission</td>
</tr>
<tr>
<td>File System</td>
<td>FilePermission, RuntimePermission</td>
</tr>
<tr>
<td>Operating System</td>
<td>RuntimePermission</td>
</tr>
<tr>
<td>JVM</td>
<td>RuntimePermission, PropertyPermission, AWTPermission</td>
</tr>
<tr>
<td>Packages and Classes</td>
<td>RuntimePermission</td>
</tr>
<tr>
<td>Security</td>
<td>SecurityPermission</td>
</tr>
</tbody>
</table>
The Concept of CodeSource

- Combination of a set of signers (certificates) and a CodeBase URL
- It is the basis for many authorization decisions
- By default, the Java 2 architecture uses a policy file to associate permissions with CodeSources
The Concept of ProtectionDomain

- A ProtectionDomain is an aggregation of a CodeSource and Permissions granted to the CodeSource by the Policy in effect.
- Each class loaded into the VM via a ClassLoader is assigned to a ProtectionDomain as determined by the Policy.
- Classes signed by the same keys and from the same URL are placed in the same ProtectionDomain.
- Classes that have the same Permissions but are from different CodeSources belong to different ProtectionDomains.
Function of SecureClassLoader

- SecureClassLoader assigns the appropriate ProtectionDomain to each loaded class
  1. SecureClassLoader creates the CodeSource for the class
  2. The CodeSource is used to locate, or instantiate, the ProtectionDomain for the class
  3. SecureClassLoader assists the VM in loading other classes
  4. These classes are also assigned the appropriate ProtectionDomain based on their CodeSource
The Fine-Grained Access Control Model of the Java 2 Platform

- Ability to grant specific permissions to a particular piece of code about accessing specific resources on the client, depending on the signer of the code and/or the location from which the code was loaded.
What a Policy File Looks Like

- The default Policy implementation is a flat file consisting of a number of grant entries.
- Each entry describes the permissions granted to a particular CodeSource.
- Each grant entry may contain one or more permissions.

```java
grant signedBy "mykey",
    codeBase "file:/application/*" {
    permission java.lang.RuntimePermission "queuePrintJob";
    permission java.io.FilePermission "${user.home}/${}/.profile", "read";
};
```
A protection domain can be:

- **The system domain**—all system code (Java built-in classes)
- **An application domain**—specific non-system code
- Standard extensions are part of an application domain
Important Note on the Java™ 2 Platform Permission Model

- A less powerful domain must not gain additional permissions as a result of calling or being called by a more powerful domain.

An application printing a message interacts with the system domain. The application domain must not gain additional permissions by calling the system domain.

The AWT system domain calls an applet's `paint()` method to display the applet. The effective access rights of the applet are the same as current rights enabled in the application domain.
Threads of Execution in the Java Programming Language

- A thread of execution may occur completely within a single ProtectionDomain or may involve application domain(s) and the system domain.
- Each thread in the VM contains a number of stack frames.
- Each frame contains the local variables for each method called in the current thread.

Thread stack
Run-Time Access Control Steps

- Steps necessary to verify whether or not the current thread has access to a protected resource

1. The library routine makes a call to the method `SecurityManager.checkPermission()`

2. `SecurityManager.checkPermission()` calls `AccessController.checkPermission()`

3. `AccessController.checkPermission()` walks back through the stack frames of the current thread and obtains the ProtectionDomains of all the classes on the thread’s stack

4. `AccessController.checkPermission()` verifies that all the unique ProtectionDomains on the stack have the Permission to access the protected resource
How the Algorithm Works

Class 1
Class 2
...
Class K

ProtectionDomain 1
ProtectionDomain 2
...
ProtectionDomain K

Permission to check
How to Determine the Permission Set of an Execution Thread

- The permission set of an execution thread is the intersection of the Permissions of all ProtectionDomains traversed by the executing thread

\[ \bigcap_{i=1}^{K} \bigcap_{j=1}^{M_i} Perm_{i,j} \]
Scenario I: Simple Check of the Current Thread

- The GetProperty application class
- Representation of the thread stack
The GetProperty Class

// ...

String s = System.getProperty("user.home");
System.out.println("user.home is: " + s);

// ...
Simple Check of the Current Thread

1. AccessController is in the system domain—Permission is implicitly granted
   - Proceed to the next frame on the thread stack
2. SecurityManager is in the system domain—Permission is implicitly granted
   - Proceed to the next frame on the thread stack
3. System is in the system domain—Permission is implicitly granted
   - Proceed to the next frame on the thread stack
4. GetProperty is in the application domain—Is the Permission granted?
   - If yes, then proceed to the next frame on the thread stack
   - If no, throw a SecurityException

Calling hierarchy:
- `GetProperty.main()`
- `java.lang.System.getProperty()`
- `java.lang.SecurityManager.checkPropertyAccess()`
- `java.security.AccessController.checkPermission()`
Lexical Scoping of Privilege Modification

• How privileged code works
• Why it is necessary
• Modifications in the run-time access control verification
How Privileged Code Works

- The `AccessController.doPrivileged()` method enables a piece of trusted code to temporarily enable access to more resources than are available directly to the application that called it.

![Diagram showing how privileged code works](image)
Why Privileged Code

An application is not allowed to access font files

The system utility to display a document must obtain those fonts on behalf of the user

The application is temporarily enabled to access the font files
How the Privileged Code Mechanism Is Implemented

- When a class calls doPrivileged(), an annotation is made on the stack frame of the thread, indicating that AccessController.checkPermission() must stop its Permission testing at this stack frame.

- The ProtectionDomains for the class and all the classes that it calls are checked, but the ProtectionDomains of its callers are not checked.

<table>
<thead>
<tr>
<th>Call hierarchy</th>
<th>checkPermission() testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller class – Permission testing not performed</td>
<td></td>
</tr>
<tr>
<td>Caller class – Permission testing not performed</td>
<td></td>
</tr>
<tr>
<td>Class called – Permission testing performed</td>
<td>✔</td>
</tr>
<tr>
<td>Class called – Permission testing performed</td>
<td></td>
</tr>
<tr>
<td>Class called – Permission testing performed</td>
<td></td>
</tr>
<tr>
<td>Class called – Permission testing performed</td>
<td></td>
</tr>
<tr>
<td>Class called – Permission testing performed</td>
<td></td>
</tr>
</tbody>
</table>
Scenario II: \texttt{doPrivileged()} Was Called

- The CountFile application class
- Representation of the thread stack
- AccessController optimizations
- Thread ProtectionDomain inheritance
### Scenario Graphical Representation

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Path</th>
<th>File Permission</th>
<th>doPrivileged()</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountFileCaller1</td>
<td>E:\JavaSec\accesscontrol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountFileCaller2</td>
<td>E:\JavaSec\accesscontrol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountFile1</td>
<td>E:\JavaSec\newdir</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CountFile2</td>
<td>E:\JavaSec\newdir</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
public class CountFileCaller1
{
    public static void main(String[ ] args)
    {
        try
        {
            System.out.println("Instantiating CountFile1...");
            CountFile1 cf = new CountFile1();
        }
        catch(Exception e)
        {
            System.out.println("" + e.toString());
            e.printStackTrace();
        }
    }
}
public class CountFileCaller2
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Instantiating CountFile2...");
            CountFile2 cf = new CountFile2();
            cf.countChars();
        }
        catch(Exception e)
        {
            System.out.println("" + e.toString());
            e.printStackTrace();
        }
    }
}
public class CountFile1
{
    public CountFile1()
    {
        throws FileNotFoundException
    
        try {
            AccessController.doPrivileged(
                new MyPrivilegedExceptionAction());

        } catch (PrivilegedActionException e) {

            throw (FileNotFoundException)
                e.getException();

        }
    }
}
import java.io.*;
import java.security.*;

class MyPrivilegedExceptionAction
    implements PrivilegedExceptionAction
{
    public Object run() throws FileNotFoundException
    {
        FileInputStream fis = new FileInputStream("C:\AUTOEXEC.BAT");
        try {
            int count = 0;
            while (fis.read() != -1)
                count++;
            System.out.println("Hi! We counted \" + count + \" chars.\");
        }
        catch (Exception e) {
            System.out.println("Exception \" + e);
        }
        return null;
    }
}
import java.io.*;

public class CountFile2
{
    int count = 0;
    public void countChars() throws Exception
    {
        FileInputStream fis = new FileInputStream("C:\\autoexec.bat");
        try
        {
            while (fis.read() != -1)
            {
                count++;
                System.out.println("Hi! We counted " + count + " chars.");
            }
        } catch (Exception e)
        {
            System.out.println("No characters counted");
            System.out.println("Exception caught" + e.toString());
        }
    }
}
Check of the Current Thread When doPrivileged() Was Called

Class in the boot class path – Permissions implicitly granted
Class in the application class path for which Permissions are checked
Class in the application class path calling doPrivileged() – Permissions are checked
Class in the application class path for which Permissions are not checked
AccessController Optimizations

- Remove duplicate ProtectionDomains
- Filter out the system domain every time it appears on the thread stack
- ProtectionDomain == null
- Equivalent to AllPermission
- Stop at first privileged stack frame
Threads: ProtectionDomain Inheritance

- Each new thread creates a new run-time stack
- When a program creates a new thread, the classes on the parent thread’s stack are not on the new thread
- A child thread always inherits all the ProtectionDomains of the parent thread
The Permission API

- Permission classes
- Permission hierarchy
- The implies() method
- Permission Target/Action
- How to implement a new Permission
Permission Classes

• Permission classes regulate access to restricted resources

• The java.security package provides the abstract class Permission, which is subclassed to regulate access to specific resources

• Permission classes are part of the package where they are most likely to be used
Permission Hierarchy

Permission

- All Permission
- Basic Permission
  - AWT Permission
  - Runtime Permission
- File Permission
  - Socket Permission
- Security Permission
- Property Permission
Target and Action

Permissions may have a target and an optional list of actions

- Target—Object of the Permission ("C:\AUTOEXEC.BAT")
- Action—Type of access ("read, write, execute")

<table>
<thead>
<tr>
<th>Permission</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilePermission</td>
<td>read, write, execute, delete</td>
</tr>
<tr>
<td>PropertyPermission</td>
<td>read, write</td>
</tr>
<tr>
<td>SocketPermission</td>
<td>resolve, accept, connect, listen</td>
</tr>
</tbody>
</table>
The implies() Method

- All Permission classes must implement the implies() method

- \texttt{a implies b} means that if one is granted Permission \texttt{a} then is granted also Permission \texttt{b}

```java
Permission p1 = new FilePermission("/tmp/*", "read");
Permission p2 = new FilePermission("/tmp/readme", "read");

p1.implies(p2) == true
p2.implies(p1) == false
```

- The class AllPermission implies all the rest of Permissions
How to Create Your Own Permission - MyPermission

- New Permissions can be created if the Java platform built-in Permissions are not adequate
- New Permissions generally subclass from Permission or BasicPermission:
  - BasicPermission already implements implies()
- The new Permission class must be included in the application package
- A permission entry must be added in the policy file
- Call to SecurityManager.checkPermission():

```java
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    MyPermission p = new MyPermission(("permissionTest"));
    sm.checkPermission(perm);
}
```

- Non-standard Permissions should be signed
import java.security.);

public class MyPermission extends BasicPermission  
{
   public MyPermission(String name) {
      super(name);
      System.out.println
         ("Constructor MyPermission(String name) called");
   }

   public MyPermission(String name, String actions) {
      super(name);
      System.out.println
         ("Constructor MyPermission(String name, " +
         "String actions) called");
   }

}
Testing MyPermission

// ...

SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    MyPermission p =
        new MyPermission("permTest");
    sm.checkPermission(perm);
}

// ...
Policy File Modification

- Add the following entry to one of the current policy file:

```java
grant codeBase "file:/PermDemo.jar" {
    permission MyPermission "permTest";
};
```
Session Summary

• You are now familiar with:
  – Possible security attacks
  – The Java 2 Platform Permission and Authorization Model
    • CodeSource, ProtectionDomain, SecureClassLoader, AccessControlContext, AccessController, Thread security
  – The concept of privileged code
  – The Java 2 Permission API
  – How to create your own Permissions
Call to Action: Now You Can…

• Grant Permissions to Java™ code
  – Configure your system so untrusted code does not perform unauthorized operations
  – Grant only those Permissions the code really needs (*Principle of Least Privilege*)

• Write your own Permission classes

• Write privileged code

- Authors:
  Marco Pistoia,
  Duane F. Reller,
  Deepak Gupta,
  Milind Nagnur,
  Ashok K. Ramani

- Title:
  Java™ 2 Network Security,
  Second Edition

- Publisher:
  Prentice Hall PTR

- ISBN: 0130155926
Java™ Platform Security Resources

- http://www.ibm.com/java
Demo