The Java Security Architecture: How? and Why?

David Svoboda

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

CERT® is a registered mark of Carnegie Mellon University.

DM-0004009
The Java Security Architecture: How? and Why?

Introduction

The Security Manager
Policy
Permissions
Confused Deputy Problem
doPrivileged()
Reduced Security Checks
Summary
The Java Security Architecture

Introduction
CERT Java Documentation

The CERT™ Oracle™ Secure Coding Standard for Java
by Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda

Rules and guidelines available online at www.securecoding.cert.org

Java Coding Guidelines
by Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda
Privilege System

Integrated with a larger system

Delegation of authority

Java privilege system

Grants different privileges to different code segments in the same program

Other examples:

• UNIX privileges and permissions
• Windows NT-based privileges
• Android Permission System
Design: Privilege Separation

Privilege Separation

• Each component possesses the minimum privileges required for it to function
• Consequence: component cannot perform other privileged operations
  - Limits impact of errors and of successful attacks

SEC50-J. Avoid granting excess privileges
Design: Privilege Minimization

Privilege Minimization

• Privileges are *disabled* most of the time
• Privileges are enabled only when required
• Consequences:
  - Reduces amount of privileged code
    • Simplifies the privileged code & makes it easier to secure
    • Reduces cost of review
  - Temporally limits certain attack opportunities

SEC51-J. Minimize privileged code

SEC53-J. Define custom security permissions for fine-grained security
Design: Distrustful Decomposition

Distrustful Decomposition

- Components have limited trust in each other
  - Similar to compartmentalized security
- Consequence: Must manage interactions between differently privileged components with care
  - Canonicalize, sanitize, normalize, and validate inputs
    - Goal: Limit potential attacks
  - Sanitize outputs
    - Goal: Prevent information and capability leaks

A method with certain privileges may be invoked by another method that lacks those privileges. Should the first method proceed?
Usage

Java’s privilege model is used in

- Applets
- Java Web Start (JWS) applets
- Servlets
  - Tomcat
  - Jetty
- Application servers
  - WebSphere
  - Jboss/WildFly

In Java’s privilege model

- Execution of untrusted code is permitted
- Untrusted code unaware of restrictions
  - Doesn’t need to know Security API
Cast of Characters

- Policy
- Protection Domains
  - Code Source
  - Permissions
  - URL
  - Certificates
- ClassLoader (abstract)
  - SecureClassLoader (contains CodeSource)
  - URLClassLoader (contains URL)
  - Other class loaders
- Access Control Context
- Access Controller
- Security Manager

Package `java.security`.

Package `java.lang`
The Java Security Architecture: How? and Why?

Introduction

The Security Manager

Policy

Permissions

Confused Deputy Problem

doPrivileged()

Reduced Security Checks

Summary
SecurityManager

Class in java.lang

Public interface to Java’s security model
Enforces a security policy
Provides many check*() methods

Each check*() method checks to see if the calling program is permitted to perform some action.

If permitted, check*() returns silently
Otherwise, throws a SecurityException
System.SecurityManager 1

Static field in the java.lang.System class

Indicates the SecurityManager that is currently in effect

Any SecurityManager object that is not the “system security manager” is ignored

Can be unset (null)

Managed by static getter/setter methods:
• System.getSecurityManager()
• System.setSecurityManager(SecurityManager s)
System.SecurityManager 2

Applets run with the default system security manager

Applications can be run with no security manager

```
java App.java
```

But they can be explicitly run with the default security manager

```
java -Djava.security.manager App.java
```

or a custom security manager

```
java -Djava.security.manager=MySecMgr \ App.java
```
Any method that implements privileged operations should first make sure its calling program is permitted to execute these operations, using the security manager’s check methods:

```java
System.getSecurityManager().check*();
```

Don’t forget to check the system security manager for null first!

Most methods assume that if system security manager is null, all operations are permitted.
Example: `java.io.FileInputStream`

```java
class FileInputStream {
    public FileInputStream(File file) throws FileNotFoundException {
        String name = (file != null ? file.getPath() : null);
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
            security.checkRead(name);
        }
        if (name == null) {
            throw new NullPointerException();
        }
        fd = new FileDescriptor();
        fd.incrementAndGetUseCount();
        open(name);
    }
}
```

Permitted if no system security manager present

Security check before open
Sensitive Operations

• Open a file
• Open a network socket
• Create a new window
• Read a system property
• Write a system property
• Change or remove the system security manager
• Load native libraries
• Load new Java code
• Access classes in certain packages (eg `sun.*`)

To define a new sensitive operation, just add a new security manager check!
The Java Security Architecture: How? and Why?

Introduction
The Security Manager
Policy
Permissions
Confused Deputy Problem
doPrivileged()
Reduced Security Checks
Summary
The Java Security Architecture

Policy
Policy 1

Indicates what a program is allowed to do

Enforced by the security manager

Only one policy object in effect; it is returned by

```java
java.security.Policy.getPolicy()
```
Policy 2

All applets and applications run with the default policy, which is very restrictive

The policy is ignored, however, if no security manager is installed

An application can be run with a custom policy:

```
java -Djava.security.manager
   -Djava.security.policy=my.policy
   Application.java
```
// Standard extensions get all permissions by default
grant codeBase "file:${{java.ext.dirs}}/*" {  
    permission java.security.AllPermission;
};
grant codeBase "file:/usr/lib/jvm/java-7-openjdk-common/jre/lib/ext/*" {  
    permission java.security.AllPermission;
};
...

grant {
...
    // allows anyone to listen on un-privileged ports
    permission java.net.SocketPermission "localhost:1024-", "listen";

    // "standard" properties that can be read by anyone
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
...
};

Grants all permissions to all paths containing core Java libraries and extensions

Some other properties that all code can read:
- os.version
- file.separator
- path.separator
- line.separator
Default Policy

Permissions that the default policy did NOT grant (except to core libraries):

- Access to the filesystem
- Open a network socket on a privileged port (<1024)
- Access certain system properties
  - `java.class.path`
  - `java.home`
  - `user.dir`
  - `user.home`
  - `user.name`
- Change or remove the system security manager
- Load new Java code
- Access classes in certain packages (e.g., `sun.*`)
Applet Policy

Remote applets can do the following:
- Open a network socket to their origin host (e.g., *phone home*)
- Access public methods of other active applets

But they can’t do the following:
- Access the filesystem
- Open a network socket anywhere besides their origin host
- Load native libraries
- Create a `ClassLoader`

Local applets have fewer restrictions
Policy Contents

- Policy
  - Protection
    - Domains
      - Code
        - Source
      - Permissions
        - URL
        - Certificates
ProtectionDomain

Used to partition the components of a program into differing levels of security

A policy contains a set of protection domains

Each protection domain contains
- Code source
- Permissions
CodeSource

Used in a protection domain (which is part of a security policy) to indicate where code originates

A code source contains

- URL indicating where the code originated
- List of certificates indicating who vouches for the code
  - Could be empty
Class Loaders

Responsible for loading all classes needed by the program

All class loaders inherit from `java.lang.ClassLoader`

Every object can access its class using

```
Object.getClass()
```

Every class can access its class loader using

```
Class.getClassLoader()
```

Since every class loader is itself a class, it has its own class loader, so class loaders have a “loading tree”

Class loaders also have an inheritance tree with `java.lang.ClassLoader` at the root
Class Loader Inheritance

- **ClassLoader** (abstract)
- **SecureClassLoader** (contains `CodeSource`)
- **URLClassLoader** (contains `URL`)
- Other class loaders
Class Loaders

Application and applet class loaders inherit from URLClassLoader

So each class loader can associate a class with a CodeSource and consequently with the Permissions associated with that class by the security policy.
Putting the Pieces Together

To check if a method has permission to do something:

1. Get its associated class
2. Get that class’s class loader
3. Get the Permissions that the class loader associated with the class
4. If the requested permission isn’t listed, throw a security exception

OK, but how do we figure this out?
The Java Security Architecture: How? and Why?

Introduction
The Security Manager
Policy
Permissions
Confused Deputy Problem
doPrivileged()
Reduced Security Checks
Summary
The Java Security Architecture

Permissions
FilePermission

Stores an absolute path to file or directory that permissions apply to

<table>
<thead>
<tr>
<th>Special String</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/*</td>
<td>All files in that directory</td>
</tr>
<tr>
<td>/*-</td>
<td>All files in that directory and all subdirectories</td>
</tr>
<tr>
<td>&lt;&lt;ALL FILES&gt;&gt;</td>
<td>All files</td>
</tr>
</tbody>
</table>
## FilePermission

Also indicates which permissions are granted

<table>
<thead>
<tr>
<th>Permission</th>
<th>Meaning</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>May read path</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>May write to path</td>
<td></td>
</tr>
<tr>
<td>execute</td>
<td>May execute program in path</td>
<td><code>Runtime.exec()</code></td>
</tr>
<tr>
<td>delete</td>
<td>May delete path</td>
<td><code>File.delete()</code></td>
</tr>
<tr>
<td>readlink</td>
<td>May follow symbolic link</td>
<td><code>FileSystemProvider.readSymbolicLink()</code></td>
</tr>
</tbody>
</table>
Permission Implication

One permission can imply another:

```java
boolean Permission.implies(Permission p)
```

For instance,

```java
java.security.FilePermission \\
"/home/*", "read,write"
```

implies

```java
java.security.FilePermission \\
"/home/.login", "read"
```
Permission Guard

Every permission object supports the `java.security.Guard` interface which provides one method:

```java
void checkGuard(Object object)
```

Determines whether or not to allow access to the guarded object. Returns silently if access is allowed. Otherwise, throws a `SecurityException`
The Java Security Architecture: How? and Why?

Introduction
The Security Manager
Policy
Permissions
Confused Deputy Problem
doPrivileged()
Reduced Security Checks
Summary
The Java Security Architecture

Confused Deputy Problem
Privileges Can Vary per Class

If $a$ and $b$ are objects of the same class, they will always have the same privileges.

But if they are different classes, they may have differing privileges:
- even if $a$ is a subclass of $b$
- even if they are in the same package
- in the same JVM

Object privileges are determined by their classes’ `CodeSource`.

Classes in the Java core library have full privileges.
Privilege Security Issues

*Privilege escalation vulnerability*

Restricted code manages to execute code in an unrestricted (privileged) context

Less privileged methods can invoke more privileged methods

More privileged methods can invoke less privileged methods unknowingly:

- Unprivileged subclasses
- Interfaces
  - Callbacks
  - Event handlers
Q: If class A is unprivileged and class B is privileged, how do we make sure that class A doesn’t trick class B into doing something privileged on A’s behalf?
Confused Deputy Problem 2

A: Require that all callers are privileged before proceeding.
Mitigating Confused Deputy

For a sensitive operation to proceed, every method on the call stack must be allowed to do it.

This stops unprivileged classes from “hiding” behind privileged classes when trying to do something malicious.

Enables privileged classes to publish sensitive methods, because the security check will prevent unprivileged classes from using them.

Sensitive methods can “take care of themselves”

Encourages Distrustful Decomposition

OK but is there a way to perform sensitive operations safely?
AccessControlContext 1

For a sensitive operation to proceed, every method on the call stack must be allowed to do it.

This class embodies the permissions that are allowed for the current method, as well as every calling method.

This is the “intersection” of the privileges of every class in the call stack.

```java
void checkPermission(Permission perm)
```

If the access control context contains the given permission, returns silently. If not, throws an `AccessControlException`.
AccessControlContext 2

For a sensitive operation to proceed, every method on the call stack must be allowed to do it.

Every Thread also has a private inheritedAccessControlContext field, which contains the context it was created in.

The AccessController can access it using this method:

```java
static native AccessControlContext getInheritedAccessControlContext();
```

So the context is preserved not only across method invocations but also across thread creation.
For a sensitive operation to proceed, **every** method on the call stack must be allowed to do it

```java
void checkPermission(Permission perm)
```

If the access control context contains the given permission, returns silently

If not, throws an `AccessControlException`

This call creates an `AccessControlContext` object from the current stack:

```java
AccessControlContext acc = AccessController.getContext();
```
AccessController.checkPermission()

```java
public static void checkPermission(Permission perm)
    throws AccessControlException
{
    ...
    if (perm == null) {
        throw new NullPointerException("permission can't be null");
    }

    AccessControlContext stack = getStackAccessControlContext();
    // if context is null, we had privileged system code on the stack
    if (stack == null) {
        // ...lots of debug code...
        return;
    }

    AccessControlContext acc = stack.optimize();
    acc.checkPermission(perm);
}
```

This method is private, static, and native
AccessController

`java.security.AccessController`

Actual enforcer of Java’s security model

`java.lang.SecurityManager` is an “ambassador”

Most `SecurityManager` methods simply delegate their work to `AccessController` methods
SecurityManager Methods

```java
public void checkRead(FileDescriptor fd) {
    if (fd == null) {
        throw new NullPointerException(
            "file descriptor can't be null");
    }
    checkPermission(
        new RuntimePermission("readFileDescriptor"));
}

public void checkPermission(Permission perm) {
    java.security.AccessController.checkPermission(perm);
}

public Object getSecurityContext() {
    return AccessController.getContext();
}
```

This actually returns an AccessControlContext
## AccessController methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getContext()</code></td>
<td>Returns the context (e.g., permissions) for the current stack</td>
</tr>
<tr>
<td><code>checkPermission()</code></td>
<td>Validates that the current stack has the given permission</td>
</tr>
<tr>
<td><code>doPrivileged()</code></td>
<td>Executes a privileged action</td>
</tr>
<tr>
<td><code>doPrivilegedWithCombiner()</code></td>
<td>Executes a privileged action</td>
</tr>
</tbody>
</table>
The Java Security Architecture: How? and Why?

Introduction

The Security Manager

Policy

Permissions

Confused Deputy Problem

doPrivileged()

Reduced Security Checks

Summary
The Java Security Architecture

doPrivileged()
AccessController.doPrivileged()

Executes a block of code with “elevated” privileges

Java’s analogue to UNIX’s setuid feature… sort of

Specifically instructs AccessController to not check the stack beyond the current method

Does check immediate caller, but no higher

This prevents untrusted code from executing malicious code inside a doPrivileged() block
AccessController.doPrivileged()

Permission perm;

Object f() {
    AccessController.checkPermission(perm);
    return g();
}

Object g() {
    AccessController.checkPermission(perm);
    return AccessController.doPrivileged(
        new PrivilegedAction<Object>() {
            public Object run() {
                return h();
            }
        });
}

Object h() {
    AccessController.checkPermission(perm);
    ...
}
doPrivileged() Features

Always returns an object; the return type is a generic parameter of the `PrivilegedAction` interface

- Use the `Void` type for blocks that don’t return anything

Privileged code must not throw a checked exception, because `PrivilegedAction.run()` has no `throws` declaration

- Use a `PrivilegedExceptionAction` to run an action that can throw an exception

Can take an extra `AccessControllerContext` indicating an arbitrary context to limit items

- Analogous to Unix `setuid`-non-root (sort of)

If no context given, analogous to UNIX `setuid`-root (sort of)
Other Contexts

Permission perm;
AccessControlContext context = ...

Object f() {
    AccessController.checkPermission(perm);
    return g();
}

Checks permissions of f()

Checks permissions of g() and f()

Object g() {
    AccessController.checkPermission(perm);
    return AccessController.doPrivileged(
        new PrivilegedAction<Object>() {
            public Object run() {
                return h();
            }
        }, context);
}

Checks permissions of h(), g() and context

Object h() {
    AccessController.checkPermission(perm);
    ...
}
doPrivileged() Security

doPrivileged() can’t be used by unprivileged code to gain privileges

It can be used by privileged code to ignore the restrictions imposed by unprivileged code that called the privileged code

So privileged methods that invoke doPrivileged() code blocks can be subject to the “confused deputy” problem
doPrivileged() Guidelines

Guideline 9-3: Safely invoke
`java.security.AccessController.doPrivileged`

Guideline 9-4: Know how to restrict privileges through
`doPrivileged`

Guideline 9-7: Understand how thread construction
transfers context

**SEC00-J.** Do not allow privileged blocks to leak sensitive
information across a trust boundary

**SEC01-J.** Do not allow tainted variables in privileged blocks

**SEC51-J.** Minimize privileged code
The Java Security Architecture: How? and Why?

Introduction
The Security Manager
Policy
Permissions
Confused Deputy Problem
doPrivileged()
Reduced Security Checks
Summary
The Java Security Architecture

Reduced Security Checks
Reduced Security Checks 1

Some core methods use reduced security checks

Instead of checking the permissions for all callers in the call stack, they check the permissions only for the immediate caller

Any method that invokes one of these methods may be vulnerable to “confused deputy”

SEC52-J. Do not expose methods that use reduced-security checks to untrusted code
Reduced Security Checks 2

Guideline 9-10: Be aware of standard APIs that perform Java language access checks against the immediate caller

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Class.newInstance</td>
</tr>
<tr>
<td>java.lang.reflect.Constructor.newInstance</td>
</tr>
<tr>
<td>java.lang.reflect.Field.get*</td>
</tr>
<tr>
<td>java.lang.reflect.Field.set*</td>
</tr>
<tr>
<td>java.lang.reflect.Method.invoke</td>
</tr>
<tr>
<td>java.util.concurrent.atomic.AtomicIntegerFieldUpdater.newUpdater</td>
</tr>
<tr>
<td>java.util.concurrent.atomic.AtomicLongFieldUpdater.newUpdater</td>
</tr>
<tr>
<td>java.util.concurrent.atomic.AtomicReferenceFieldUpdater.newUpdater</td>
</tr>
</tbody>
</table>
Reduced Security Checks 3

Guideline 9-9: Safely invoke standard APIs that perform tasks using the immediate caller’s class loader instance

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Class.forName</td>
</tr>
<tr>
<td>java.lang.Package.getPackage(s)</td>
</tr>
<tr>
<td>java.lang.Runtime.load</td>
</tr>
<tr>
<td>java.lang.Runtime.loadLibrary</td>
</tr>
<tr>
<td>java.lang.System.load</td>
</tr>
<tr>
<td>java.lang.System.loadLibrary</td>
</tr>
<tr>
<td>java.sql.DriverManager.getConnection</td>
</tr>
<tr>
<td>java.sql.DriverManager.getDriver(s)</td>
</tr>
<tr>
<td>java.sql.DriverManager.deregisterDriver</td>
</tr>
<tr>
<td>java.util.ResourceBundle.getBundle</td>
</tr>
</tbody>
</table>
Reduced Security Checks 4

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Class.getClassLoader</td>
</tr>
<tr>
<td>java.lang.Class.getClasses</td>
</tr>
<tr>
<td>java.lang.Class.getField(s)</td>
</tr>
<tr>
<td>java.lang.Class.getMethod(s)</td>
</tr>
<tr>
<td>java.lang.Class.getConstructor(s)</td>
</tr>
<tr>
<td>java.lang.Class.getDeclaredClasses</td>
</tr>
<tr>
<td>java.lang.Class.getDeclaredField(s)</td>
</tr>
<tr>
<td>java.lang.Class.getDeclaredMethod(s)</td>
</tr>
<tr>
<td>java.lang.Class.getDeclaredConstructor(s)</td>
</tr>
<tr>
<td>java.lang.ClassLoader.getParent</td>
</tr>
<tr>
<td>java.lang.ClassLoader.getSystemClassLoader</td>
</tr>
<tr>
<td>java.lang.Thread.getContextClassLoader</td>
</tr>
</tbody>
</table>

Guideline 9-8: Safely invoke standard APIs that bypass SecurityManager checks depending on the immediate caller’s class loader
The Java Security Architecture: How? and Why?

Introduction
The Security Manager
Policy
Permissions
Confused Deputy Problem
doPrivileged()
Reduced Security Checks
Summary
The Java Security Architecture

Summary
Summary 1

Java’s security architecture is designed to be:
- Extendable
- Modular
- Behind-the-scenes

Encourages the use of these secure design patterns:
- Privilege separation
- Privilege minimization
- Distrustful decomposition
Summary 2

Security architecture is **NOT** designed to be

- Modifiable
- Familiar

  - Analogies with UNIX privileges or setuid are *very* tenuous

Watch out for

- `doPrivileged()`
- Methods that use reduced security checks
The End

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213