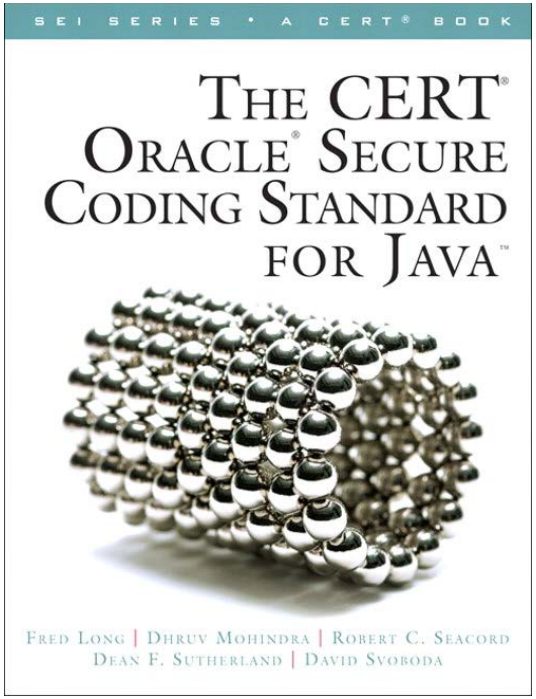
Don't Be Pwned: A Very Short Course on Secure Programming in Java

Dean F. Sutherland & Robert Seacord & David Svoboda

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.

DM-0001396

CERT Java Documentation

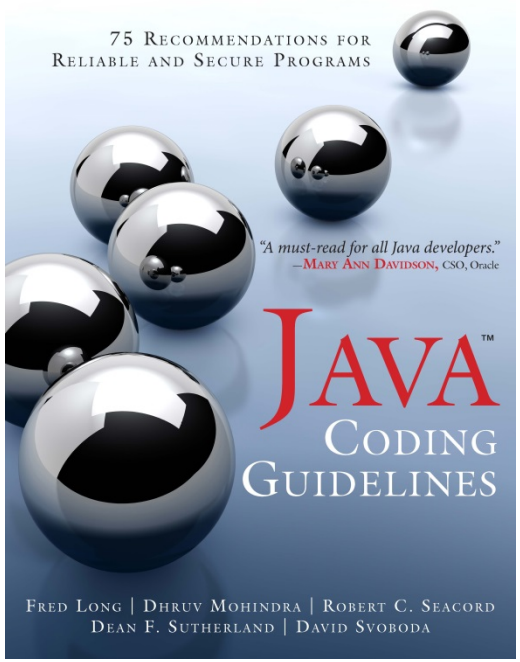


The CERT™ Oracle™ Secure Coding Standard for Java

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

Rules available online at

www.securecoding.cert.org



Java Coding Guidelines

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

Don't Be Pwned!

Many of *The CERT Oracle Secure Coding Standard for Java* and the *Java Coding Guidelines* address real exploits that have compromised Java programs in the field.

This presentation

- examines several examples that describe the core vulnerability exploited
- presents techniques for avoiding or repairing the vulnerability (including code examples)

Authors / Acknowledgements

Authors

- Dr. Fred Long is a senior lecturer and director of learning and teaching in the Department of Computer Science, Aberystwyth University in the United Kingdom.
- Dhruv Mohindra is a senior software engineer at Persistent Systems Limited, India.
- Robert Seacord leads CERT's Secure Coding Initiative.
- Dr. Dean Sutherland is a senior software security engineer at CERT.
- David Svoboda is a software security engineer at CERT.

Acknowledgements

- The many individuals who have reviewed and contributed to the CERT Oracle Secure Coding Standard for Java
- Francis Ho and Thomas Hawtin

Agenda

Secure Coding and SCALe

IDS07-J. Do not pass untrusted, unsanitized data to the **`Runtime.exec()`** method

IDS08-J. Sanitize untrusted data passed to a regex

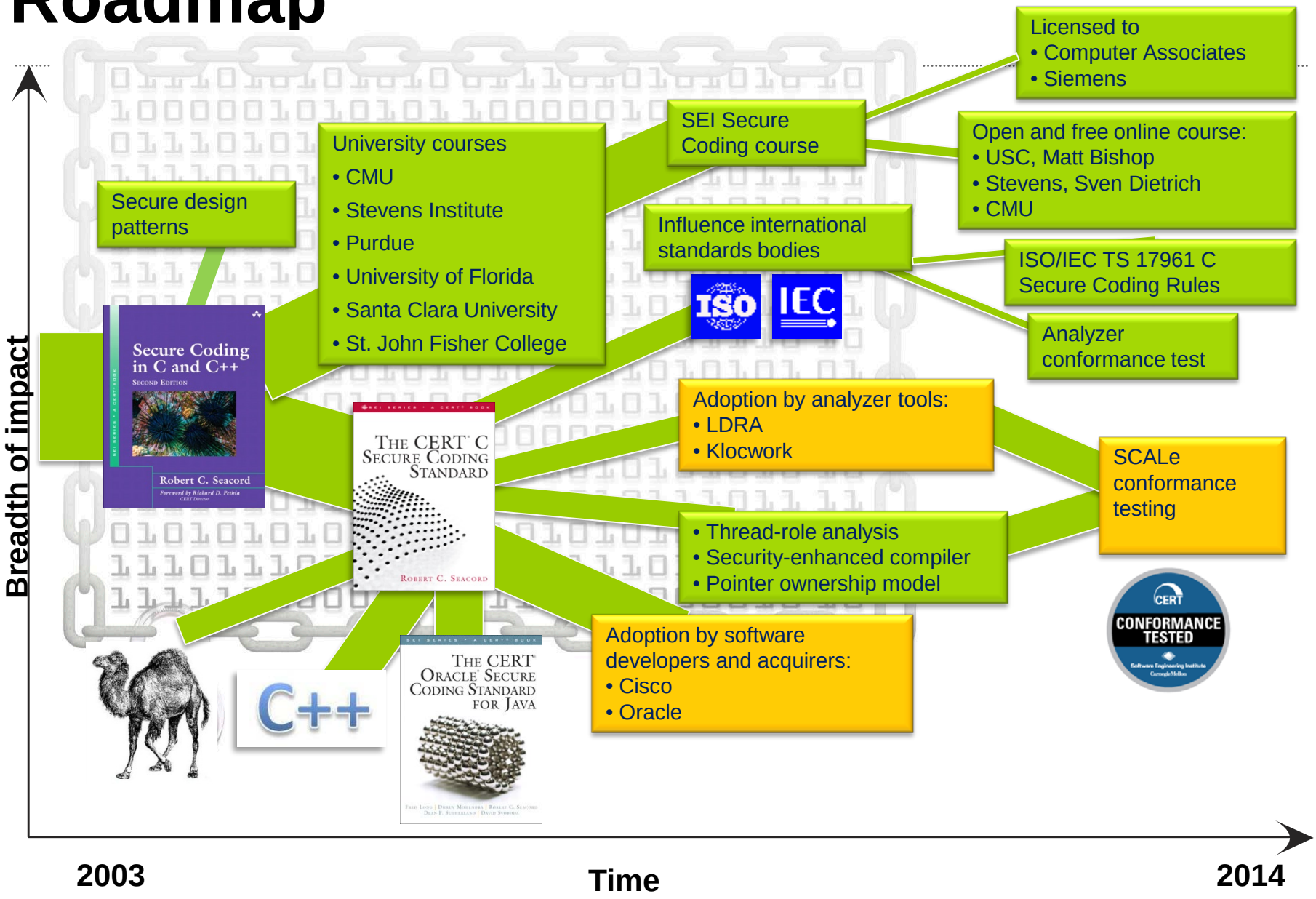
FIO13-J. Do not log sensitive information outside a trust boundary

SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields

SEC03-J. Do not load trusted classes after allowing untrusted code to load arbitrary classes

NUM05-J. Do not use denormalized numbers

Roadmap


Licensed to
• Computer Associates
• Siemens
SEI Secure Coding course
Open and free online course:
• USC, Matt Bishop
• Stevens, Sven Dietrich
• CMU
University courses
• CMU
• Stevens Institute
• Purdue
• University of Florida
• Santa Clara University
• St. John Fisher College
Secure design patterns
Influence international standards bodies
ISO
IEC
ISO/IEC TS 17961 C Secure Coding Rules
Analyzer conformance test
Secure Coding in C and C++
Second Edition
Robert C. Seacord
Adoption by analyzer tools:
• LDRA
• Klocwork
SCALe conformance testing
The CERT C Secure Coding Standard
Robert C. Seacord
• Thread-role analysis
• Security-enhanced compiler
• Pointer ownership model
CERT
CONFORMANCE TESTED
Adoption by software developers and acquirers:
• Cisco
• Oracle
C++
The CERT Oracle Secure Coding Standard for Java
Breadth of impact
2003
Time
2014

Source Code Analysis Laboratory

The CERT Source Code Analysis Laboratory (SCALe) is an operational capability for application conformance testing against one of CERT's secure coding standards.

- A detailed report of findings is provided to the customer to repair.
- After the developer has addressed these findings, the product version is certified as conforming to the standard
- The certification is published in a registry of certified systems.

Industry Demand

Conformance with CERT Secure Coding Standards can represent a significant investment by a software developer, particularly when it is necessary to refactor or modernize existing software systems.

However, it is not always possible for a software developer to benefit from this investment, because it is not always easy to market code quality.

A goal of conformance testing is to provide an incentive for industry to invest in developing conforming systems.

- perform conformance testing against CERT secure coding standards
- verify that a software system conforms with a CERT secure coding standard
- use CERT “seal” when marketing products
- maintain a certificate registry with the certificates of conforming systems

Conformance Testing

The use of secure coding standards defines a proscriptive set of rules and recommendations to which the source code can be evaluated for compliance.

For each secure coding standard, the source code is certified as provably nonconforming, conforming, or provably conforming against each guideline in the standard:

Provably nonconforming	The code is provably nonconforming if one or more violations of a rule are discovered for which no deviation has been allowed.
Conforming	The code is conforming if no violations of a rule can be identified.
Provably conforming	Finally, the code is provably conforming if the code has been verified to adhere to the rule in all possible cases.

Evaluation violations of a particular rule ends when a “provably nonconforming” violation is discovered.

Agenda

Secure Coding and SCALe

IDS07-J. Do not pass untrusted, unsanitized data to the **`Runtime.exec()`** method

IDS08-J. Sanitize untrusted data passed to a regex

FIO13-J. Do not log sensitive information outside a trust boundary

SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields

SEC03-J. Do not load trusted classes after allowing untrusted code to load arbitrary classes

NUM05-J. Do not use denormalized numbers

Trusted and Untrusted Code

Java programs can invoke, contain, or depend on

- The JVM itself
- Locally-developed code
- 3rd party code
  - Java runtimes & JDK libraries
  - Other libraries
    - Bespoke
    - Commercial OTS
    - Commonly available open-source
  - Dynamically loaded code
    - System- or user-provided plug-ins
    - System-provided or downloaded libraries
    - Malicious attacker's classes

Necessarily trusted

Both trusted & untrusted

Necessarily trusted to some extent

Both trusted & untrusted

Trust level ??

Obviously untrusted. But how can you distinguish this from other cases?

Trust Boundaries

Software often contains multiple components & libraries

Each component may operate in one or more trusted domains that are determined by

- architecture
- security policy
- required resources
- functionality

Example:

- Component A can access file-system, but lacks any network access
- Component B has general network access, but lacks access to the file-system and the secure network
- Component C can access a secure network, but lacks access to the file-system and the general network

Command Injection

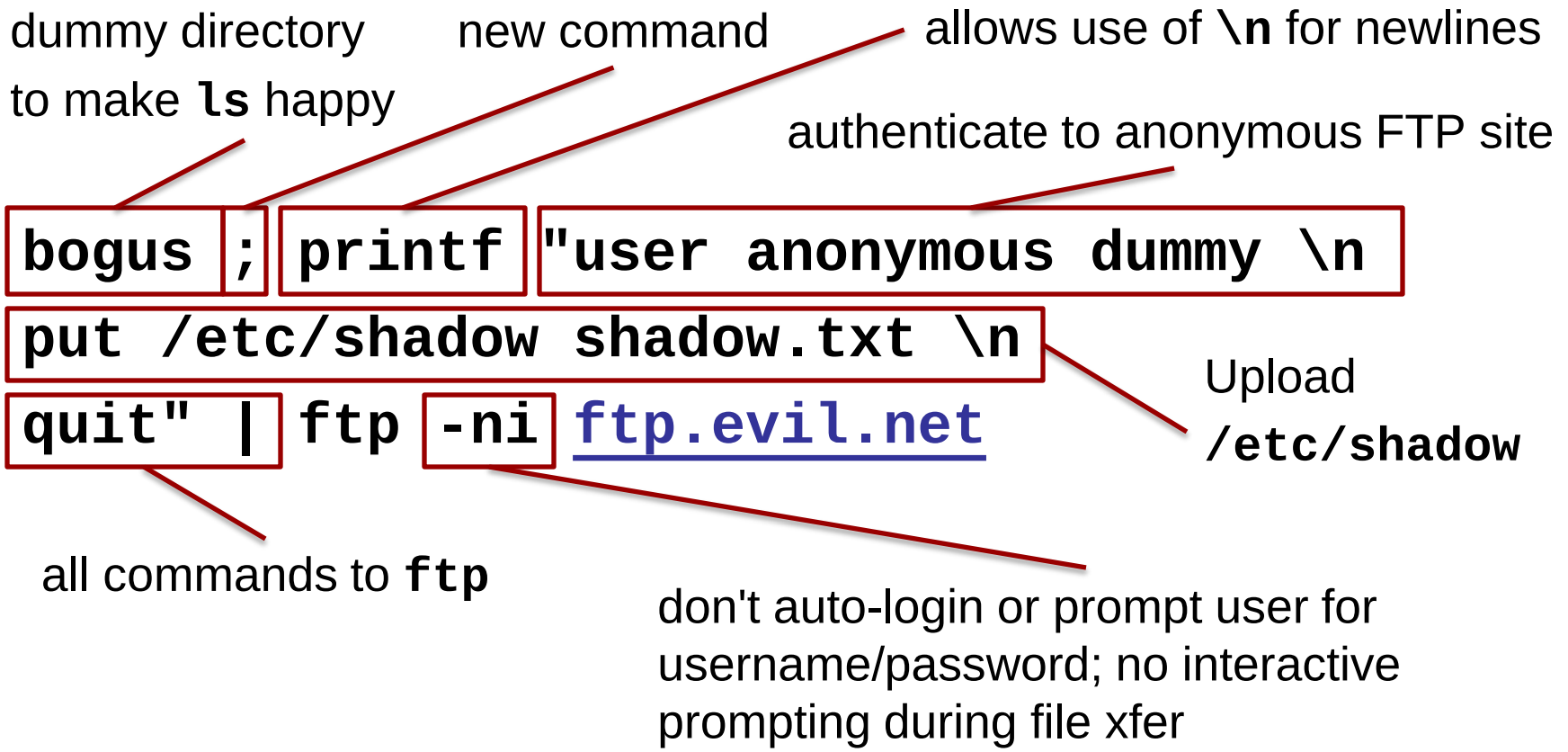
Command injection can occur when an improperly sanitized string is passed across a trust boundary. For example:

```
String dir = System.getProperty("dir");
Runtime rt = Runtime.getRuntime();
Process proc = rt.exec(
  new String[] {"sh", "-c", "ls " + dir}
);
```

This code violates rule **IDS07-J. Do not pass untrusted, unsanitized data to the Runtime.exec() method**

Attack Scenario

The program is running with root privileges and the attacker provides the following string for **`dir?`**



Key Idea: Distrustful Decomposition

Components have limited trust in each other

- Similar to compartmentalized security

Consequence: interactions between components must be managed with care

- Canonicalize, Sanitize, Normalize & Validate inputs
  - Goal: Limit potential attacks
- Sanitize outputs
  - Goal: Prevent information and capability leaks
- Addressed by many rules in the coding standard and guidelines

Key Ideas: Privilege Separation & Privilege Minimization

Privilege Separation

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

Privilege Minimization

- Privileges are *disabled* most of the time
- Privileges are enabled exactly and only when required
- Consequences:
  - Reduces amount of privileged code
    - Easier to get it right
    - Reduces cost of review
  - Temporally limits certain attack opportunities

Trust Boundaries Guidelines

How much do you trust your network?

- If you answered “less than I trust my program,” you just found another trust boundary

Ask the same question for

- File system
- System operators and administrators
- Various kinds of users
- Input data from various sources
- Log files
- Cloud service providers
- *Etc.*

Validation & Sanitization

Programs must take steps to ensure that any data that crosses a trust boundary is both

- Appropriate
- Non-malicious

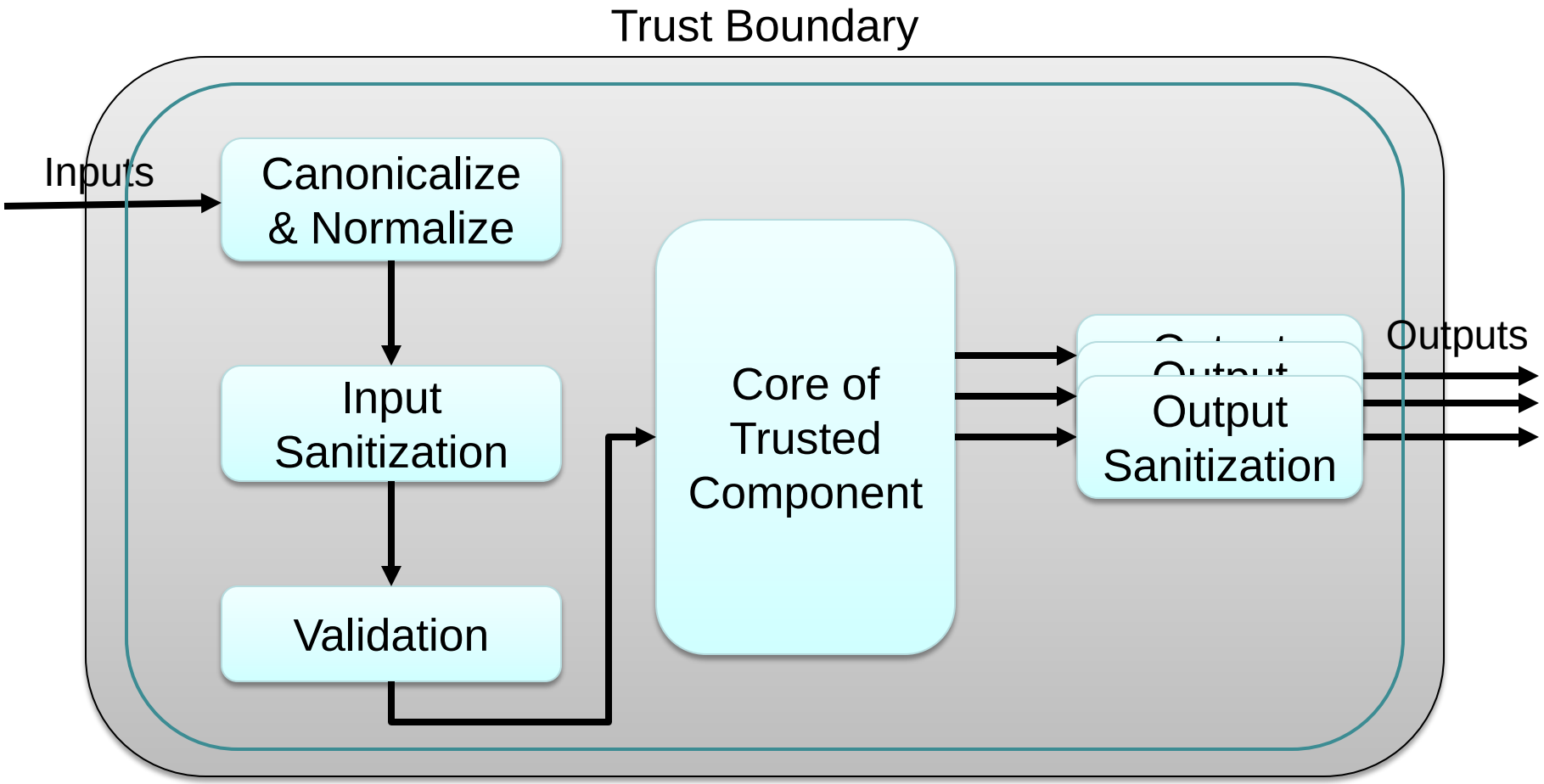
This can include appropriate

- Canonicalization & normalization
- Input Sanitization
- Validation

These steps must be taken in *exactly* that order

- Although steps may be omitted when appropriate

Trusted Component



Agenda

Secure Coding and SCALe

IDS07-J. Do not pass untrusted, unsanitized data to the **`Runtime.exec()`** method

IDS08-J. Sanitize untrusted data passed to a regex

FIO13-J. Do not log sensitive information outside a trust boundary

SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields

SEC03-J. Do not load trusted classes after allowing untrusted code to load arbitrary classes

NUM05-J. Do not use denormalized numbers

Regular Expressions

Regular expressions are widely used to match strings of text

For example, the POSIX **`grep`** utility supports regular expressions for finding patterns in the specified text

The **`java.util.regex`** package provides the

- **`Pattern`** class that encapsulates a compiled representation of a regular expression
- **`Matcher`** class that uses a **`Pattern`** to perform matching operations on a **`CharSequence`**

Problem Description

Suppose a system log file contains messages output by various system processes

Some processes produce public messages and some processes produce sensitive messages marked “private”

```
10:47:03 private[423] Successful logout  name: usr1 ssn: 111223333
10:47:04 public[48964] Failed to resolve network service
10:47:04 public[1] (public.message[49367]) Exited with exit code: 255
10:47:43 private[423] Successful login  name: usr2 ssn: 444556666
10:48:08 public[48964] Backup failed with error: 19
```

Goals:

- Permit user to search the log file for interesting messages
- Prevent user from seeing any private messages

(Insecure) Solution

Periodically loads the log file into memory

Permits the user to provide the **<SEARCHTEXT>** that becomes part of the following regex:

```
(.*? +public\[\d+\] +.*<SEARCHTEXT>.*)
```

Vulnerability

However, if an attacker can substitute any string for **<SEARCHTEXT>**, he can perform a regex injection with the following text:

```
.*)|(.*
```

When injected into the regex, the regex becomes:

```
(.*? +public\[\d+\] +.*.*)|(.*.*)
```

This regex will match any line in the log file, including the private ones

Exploit

CVE-2005-1949 provides a real world example of an exploit involving **`eping`** (which executes the **`ping`** command)

It used a regex to check that the user supplied input was a valid IP address

Unfortunately, the regex was incorrectly written and allowed a command injection so that a user could cause **`eping`** to execute an arbitrary command

Secure Solution

A secure solution is to filter out non-alphanumeric characters (except space and single quote) from the search string, which prevents regex injection

Another method of mitigating this vulnerability is to filter out the sensitive information prior to matching

Such a solution would require the filtering to be performed every time the log file is loaded into memory, incurring extra complexity and a performance penalty.

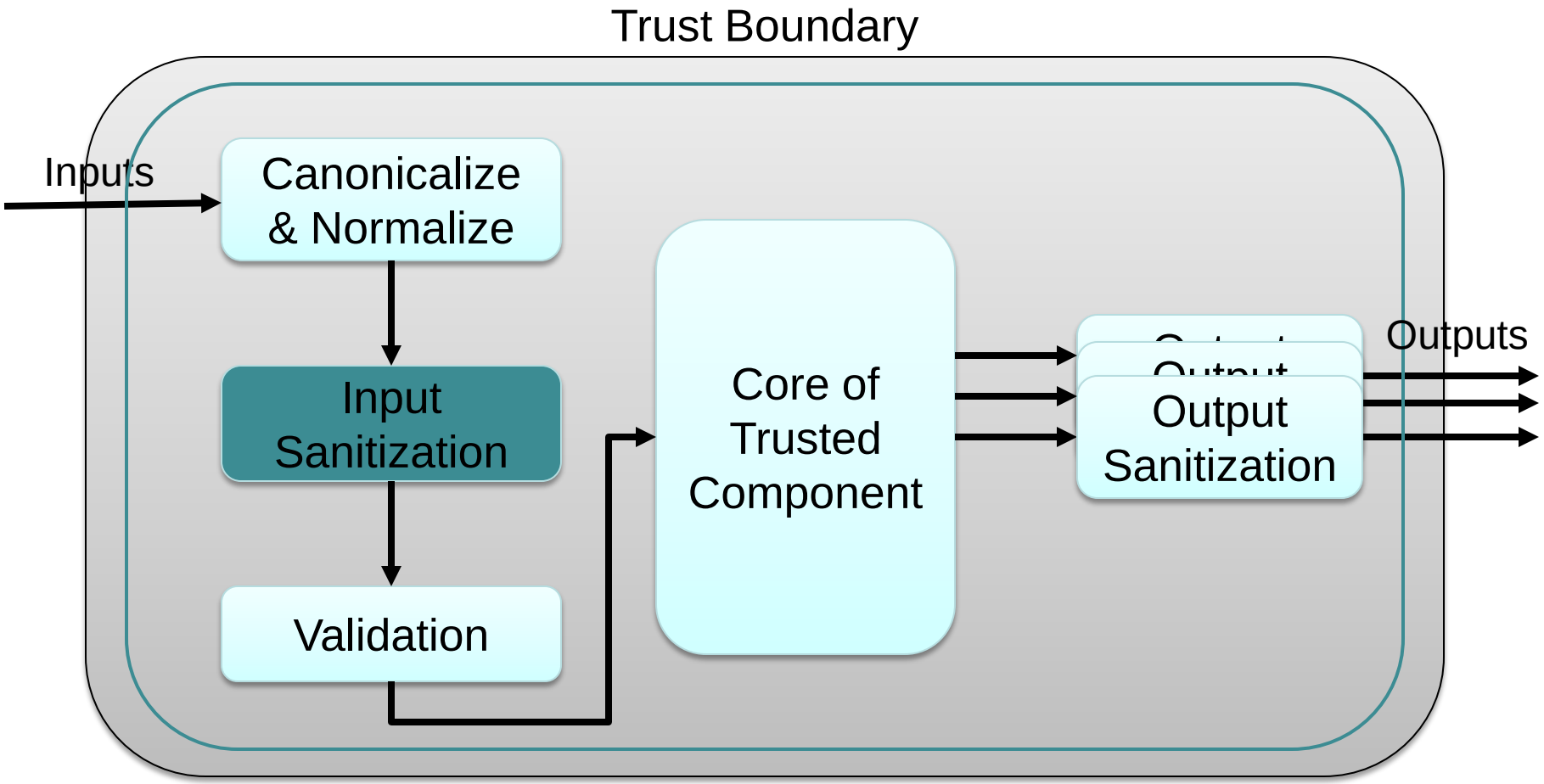
Conformance to *The CERT Oracle Secure Coding Standard for Java* rule **IDS08-J. Sanitize untrusted data passed to a regex** eliminates this vulnerability and prevents this exploit.

Secure Solution (Whitelisting)

```
String search = /* untrusted search string */
StringBuilder sb = new StringBuilder();
for (int i = 0; i < search.length(); ++i) {
  char ch = search.charAt(i);
  if (Character.isLetterOrDigit(ch) ||
      ch == ' ' ||
      ch == '\'') {
    sb.append(ch);
}}
search = sb.toString();

// Construct regex dynamically from user string
String regex =
       "(.*? +public\\[\\d+\\] +.*" + search + ".*)";
// ...
```

Trusted Component



Agenda

Secure Coding and SCALe

IDS07-J. Do not pass untrusted, unsanitized data to the **`Runtime.exec()`** method

IDS08-J. Sanitize untrusted data passed to a regex

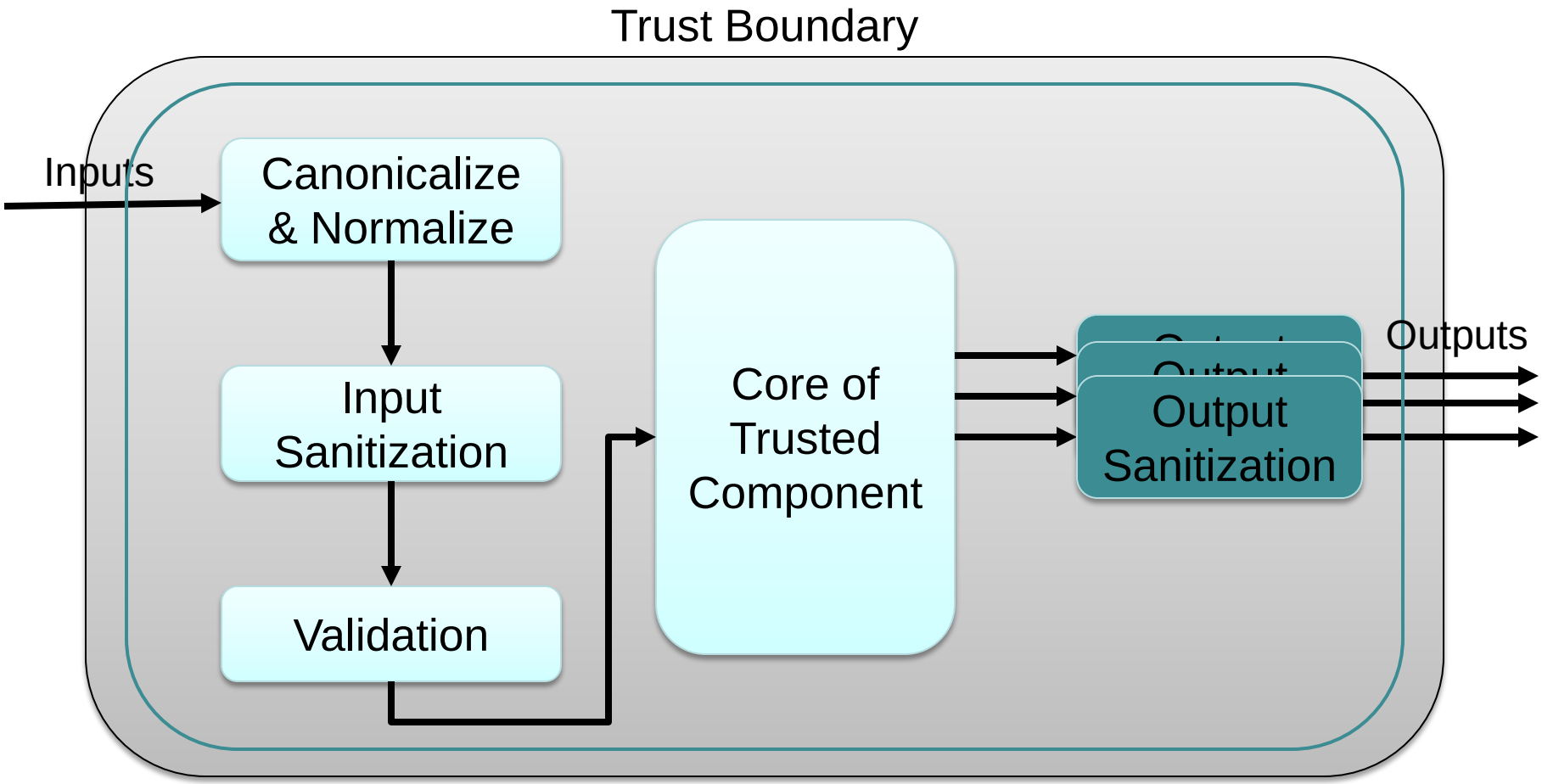
FIO13-J. Do not log sensitive information outside a trust boundary

SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields

SEC03-J. Do not load trusted classes after allowing untrusted code to load arbitrary classes

NUM05-J. Do not use denormalized numbers

Trusted Component



Logging Vulnerability

In exploit **CVE-2005-2990**, **`AuthInfo.java`** in the **`LineControl`** Java Client (**`jlc`**) before 0.8.1 stored sensitive information such as user passwords in log files.

Clearly, this is a *Bad Thing*™ — the sensitive information may become available to untrusted parties who may access the log files

This vulnerability is addressed by *The CERT Oracle Secure Coding Standard for Java* rule FIO13-J. Do not log sensitive information outside a trust boundary

Logging Sensitive Data

Logging is essential for

- debugging
- incident response
- collecting forensic evidence

Nevertheless, logging sensitive data raises many concerns, including

- the privacy of the stakeholders
- limitations imposed by the law on the collection of personal information
- the potential for data exposure by insiders

- Sensitive information includes, but is not limited to
  - IP addresses
  - user names and passwords
  - email addresses
  - credit card numbers
  - any personally identifiable information such as social security numbers

Many countries prohibit or restrict collection of personal data; others permit retention of personal data only when held in an anonymized form. Consequently, logs must not contain sensitive data, particularly when prohibited by law.

Vulnerable Code

```
public void logRemoteIPAddress(String name) {
  Logger logger = Logger.getLogger("com.organization.Log");
  InetAddress machine = null;
  try {
    machine = InetAddress.getByName(name);
  } catch (UnknownHostException e) {
    Exception e = MyExceptionReporter.handle(e);
  } catch (SecurityException e) {
    Exception e = MyExceptionReporter.handle(e);
    logger.severe(name + "," + machine.getHostAddress() +
                  "," + e.toString());
  }
}
```

Server logs the IP address of the remote client in the event of a security exception. This data can be misused, for example, to build a profile of a user's browsing habits.

Secure Solution

```
public void logRemoteIPAddress(String name) {
  Logger logger = Logger.getLogger("com.organization.Log");
  InetAddress machine = null;
  try {
    machine = InetAddress.getByName(name);
  } catch (UnknownHostException e) {
    Exception e = MyExceptionReporter.handle(e);
  } catch (SecurityException e) {
    Exception e = MyExceptionReporter.handle(e);
  }
}
```

Does not log security exceptions except for the logging implicitly performed by **`MyExceptionReporter`**

Logging Sensitive Information

Log messages with sensitive information should not be printed to the console display for security reasons

The **`java.util.logging.Logger`** class supports different logging levels that can be used for classifying such information

These are **`FINEST`**, **`FINER`**, **`FINE`**, **`CONFIG`**, **`INFO`**, **`WARNING`**, and **`SEVERE`**

By default, the **`INFO`**, **`WARNING`**, and **`SEVERE`** levels print the message to the console, which is accessible to end users and system administrators

Sensitive information that must be recorded in log files but not displayed on the console should be logged at the **`FINEST`** level, for example.

Logging Sensitive Information

Log messages with sensitive information should not be printed to the console display for security reasons

The **`java.util.logging.Logger`** class supports different log ng such information

These are **F** **WARNING**, and **SEVER**

Important Note

Ensure that attackers cannot modify the default logging level filters

By default, the **`INFO`**, **`WARNING`**, and **`SEVERE`** levels print the message to the console, which is accessible by end users and system administrators

Sensitive information that must be recorded in log files but not displayed on the console should be logged at the **`FINEST`** level, for example.

Agenda

Secure Coding and SCALe

IDS07-J. Do not pass untrusted, unsanitized data to the **`Runtime.exec()`** method

IDS08-J. Sanitize untrusted data passed to a regex

FIO13-J. Do not log sensitive information outside a trust boundary

SEC05-J. Do … e accessibility of classes, meth… or fields

This is the January 2013 0-Day Exploit

SEC03-J. Do not load trusted classes after allowing untrusted code to load arbitrary classes

NUM05-J. Do not use denormalized numbers

January 2013 0-Day Exploit

Three-step exploit

1. Get handles for classes Context and GeneratedClassLoader from sun.org.mozilla.javascript.internal package
   - Note: Applet security manager is configured to forbid access to sun.* packages
2. Use those classes to create a trusted class loader
   - Note: Applet security manager is configured to reject all attempts to create class loaders…
   - …*except* those that originate from trusted framework code
3. Use class loader to load arbitrary malicious code
   - Carried as a byte-stream payload
   - Sets security manager to null
     - Consequence: malicious code now able to do *anything it wants*

January 2013 0-Day, Step 1

Gaining class handles:

- Ordinary reflection won't work
  - Security manager blocks the attempt
- Use com.sun.jmx.mbeanserver.MBeanInstantiator.findClass() to do it for them
  - *Should be* blocked by the security manager
  - Works anyway

MBeanInstantiator vulnerability

```
public Class<?> findClass(String className, ClassLoader loader) throws ReflectionException {
  return loadClass(className,loader);
}

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;
}
```

findClass calls loadClass

loadClass loads *any* pre-existing class without making a SecurityManager check

MBeanInstantiator vulnerability (2)

MBeanInstantiator is intended to be a hidden part of the Java Beans implementation

- So untrusted callers shouldn't be an issue

But:

- JmxBeanServer exposes the MBeanInstantiator through a public getter method
  - Consequence: Attacker now able to load any pre-existing library class, including those "hidden" in the sun.* package hierarchy

January 2013 0-Day, Step 2

Exploit code needs to build a trusted class loader

- Can’t just call methods from GeneratedClassLoader
  - Security manager would terminate the process
- Can’t use ordinary reflection to call methods from GeneratedClassLoader
  - Security manager would terminate the process

java.lang.invoke.MethodHandles **vulnerability**

Three step process:

1. Use previous vulnerability to get references to classes …javascript.internal.Context (in c1) and …javascript.internal.GeneratedClassLoader (in c2)
2. Use ordinary MethodHandles methods to
   i. Get a MethodHandle for the findConstructor method
   ii. Use findConstructor to get a MethodHandle for the …internal.Context constructor
   iii. Invoke the constructor to create an internal Context
3. Use the Context to construct the class loader
   - Permitted because the Context is trusted

java.lang.invoke.MethodHandles **vulnerability**

```
MethodHandles.Lookup public_lookup =
MethodHandles.publicLookup();
MethodType mh =
   MethodType.methodType(MethodHandle.class,
                         Class.class,
                         new Class[] { MethodType.class });
MethodHandle findConstructor_mh =
    public_lookup.findVirtual(MethodHandles.Lookup.class,
                              "findConstructor", mh);
MethodType mh1 = MethodType.methodType(Void.TYPE);
MethodHandle context_constructor_mh =
   (MethodHandle) findConstructor_mh.invokeWithArguments(
       new Object[] { public_lookup, c1, mh1 });
Object js_context =
      context_constructor_mh.invokeWithArguments(new Object[0]);
```

Wait a minute…

Why on earth are they using reflection to invoke MethodHandles methods…

…that use reflection to invoke the desired methods?

Implementation of reflection library

- Must prevent untrusted callers from invoking sensitive methods, but…
- Frequently delegates responsibility to other reflection implementation methods

Consequence: Must discover “effective caller” while ignoring trusted code that is part of the implementation

- Perform security manager check on “effective caller”

Finding the “Effective Caller”

Examine the call stack

- Each reflection method knows “how many stack frames” above itself the effective caller *should* be found
- Computation of “how many stack frames” should ignore frames belonging to delegated reflection library methods
  - This allows successful delegation inside reflection library
- Permit operation only when effective caller has permission to perform operation
  - Achieve this by performing security manager check on the effective caller

The Underlying Flaw

When adding the Invoke framework, implementers forgot to add its methods to the group that should be ignored

- Consequence: Reflection operations erroneously permitted
  - Security manager check from reflection library sees invokeWithArguments and wrongly decides it's the effective caller.
  - invokeWithArguments is and should be a trusted caller
  - Trusted callers are—and *should be*—permitted to perform unsafe operations

This violates rule SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields

Agenda

Secure Coding and SCALe

IDS07-J. Do not pass untrusted, unsanitized data to the **`Runtime.exec()`** method

IDS08-J. Sanitize untrusted data passed to a regex

FIO13-J. Do not log sensitive information outside a trust boundary

SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields

SEC03-J. Do not load trusted classes after allowing untrusted code to load arbitrary classes

NUM05-J. Do not use denormalized numbers

Class loaders and loading on-demand

Some programs allow untrusted code to load classes

- Often a valuable feature

Example:

- Tomcat supports a flag **`useContextClassLoader`** that indicates whether to use the **`WebappClassLoader`** (untrusted), or the ordinary trusted class loaders.
- Both **`WebappClassLoader`** and the classes it loads are untrusted
  - **`WebappClassLoader`** loads whatever classes are requested by web applications running on Tomcat

On-Demand class loading

Many JVMs deliberately defer class loading until a class is requested

- Smaller average memory footprint
- Avoids wasting time loading unneeded classes

What happens when we mix trusted and untrusted class loaders?

Exploit: Tomcat (pre 6.0.20)

Creating a Digester could invoke an incorrect or malicious XML parser

In **`org.apache.catalina.startup.ContextConfig`**

```
protected static Digester webDigester = null;
// …
if (webDigester == null) {
  webDigester = createWebDigester();
}
```

Getting a Digester$_1$

In class **DigesterFactory**:

```
// This method exists in the class DigesterFactory and is
// called by ContextConfig.createWebXmlDigester(),
// which is in turn called by ContextConfig.createWebDigester()
public static Digester newDigester(boolean xmlValidation,
                                   boolean xmlNamespaceAware,
                                   RuleSet rule) {
  Digester digester = new Digester();
  // ...
  digester.setUseContextClassLoader(true);
  // ...
  return digester;
}
```

The result produced by this method ends up in **webDigester** (seen on the previous slide)

Getting a Digester$_2$

Digesters use the **`useContextClassLoader`** flag to decide which **`ClassLoader`** to use:

```
public ClassLoader getClassLoader() {
  // ...
  if (this.useContextClassLoader) {
    // Uses the context class loader which was
    // previously set to the WebappClassLoader
    ClassLoader classLoader =
     Thread.currentThread().getContextClassLoader();
  }
  return classloader;
}
```

Getting a Digester$_3$

To process a **`web.xml`** file (among others), some code will call **`Digester.getParser()`** which calls:

```
public SAXParserFactory getFactory() {
  if (factory == null) {
    factory = SAXParserFactory.newInstance();
    // Uses WebappClassLoader to load the
    // SAXParserFactory class
    // ...
  }
  return (factory);
}
```

This all looks fine.

Where's the exploit?

Tomcat exploit (pre 6.0.20)

Suppose that an *earlier* web application loads a *malicious* **`SAXParserFactory`** class

The method on the previous slide would find and use the malicious factory

- Even though it expected to get the trusted system version

Consequence:

- Local users can read *or modify* (1) **`web.xml`**, (2) **`context.xml`**, or (3) **`tld`** files of arbitrary web applications

More generally, this class of vulnerability allows attacker to provide *malicious versions of trusted classes (a.k.a., Trojans)*

Simple Solution

Load all trusted classes before loading any untrusted classes

In Tomcat, change creation of **`WebDigester`**:

```
protected static final Digester webDigester = init();
protected Digester init() {
  Digester digester = createWebDigester();
  // Context Classloader turned off at init
  digester.getParser();
  return digester;
}
```

When using complex classloader trees (as in OSGI modules), apply simple solution in *each* relevant classloader context

Agenda

Secure Coding and SCALe

IDS07-J. Do not pass untrusted, unsanitized data to the **`Runtime.exec()`** method

IDS08-J. Sanitize untrusted data passed to a regex

FIO13-J. Do not log sensitive information outside a trust boundary

SEC05-J. Do not use reflection to increase accessibility of classes, methods, or fields

SEC03-J. Do not load trusted classes after allowing untrusted code to load arbitrary classes

NUM05-J. Do not use denormalized numbers

Denormalized Numbers 1

What does the following code print? (Assume FP-strict mode execution)

```
float x = 1/3.0f;
System.out.println("Original    : " + x);
x = x * 7e-45f;
System.out.println("Denormalized: " + x);
x = x / 7e-45f;
System.out.println("Restored    : " + x);
```

Output is:

```
Original     : 0.33333334
Denormalized : 2.8E-45
Restored     : 0.4
```

Denormalized Numbers 2

Mantissa of a **`float`** has 23 bits

Normal values have an implicit leading 1 bit; high-order mantissa bit is 1.

- For example $1.10110110011001011011100 * 2^{48}$ would be a normal float value.

When value is very close to 0 (*e.g.*, $<2^{-126}$ for a **`float`**)

- High order bit of mantissa cannot be a 1
  - For this reason, leading 0 bits in mantissa
- Number is said to be *denormalized*

Leading 0 bits no longer function as significant bits of precision

- They become notionally part of the exponent, rather than of the mantissa
- Loss of precision is *guaranteed*

Denormalized Numbers 3

What happens when you execute the following:

```
double d = Double.parseDouble("2.2250738585072012e-308");
```

Prior to Java 1.6 update 24 (or Java 1.5 update 28, or Java 1.4.2_29):

- Denial of service!
- This code caused an infinite loop in `Double.parseDouble`!
- Implementers failed to consider the lost precision in denormalized numbers, so the algorithm they used never converged

The implementation has been fixed, and now produces the correct value.

Detect Denormalized Numbers

```
strictfp
public static boolean isDenormalized(float val) {
  if (val == 0) {
    return false;
  }
  if ((val > -Float.MIN_NORMAL) &&
      (val < Float.MIN_NORMAL)) {
    return true;
  }
  return false;
}
```

Risks and Exceptions

Risks

- Loss of precision can lead to incorrect results
- Print representation can be unexpected (leading 0)
- Loss of precision can break numeric algorithms (non-convergence can cause DoS via infinite loop or wildly incorrect answers, depending on the algorithm)

Exceptions

- Denormalized numbers are acceptable when suitable numeric analysis shows that all relevant accuracy and behavioral requirements are preserved.

The CERT Oracle Secure Coding Standard for Java rule NUM05-J. Do not use denormalized numbers further describes this problem.

Summary

When Java was first designed, dealing with security was a key component

In the years since then, all of the various standard libraries, frameworks, and containers that have been built have had to deal with security too

The mere presence of the facilities, however, is insufficient to ensure security automatically

A set of standard practices has evolved over the years

The CERT® Oracle® Secure Coding Standard for Java™ is a compendium of these practices

For More Information

Visit CERT® web sites:

http://www.cert.org/secure-coding/

https://www.securecoding.cert.org/

Contact Presenters:

David Svoboda

svoboda@cert.org

(412) 268-3965

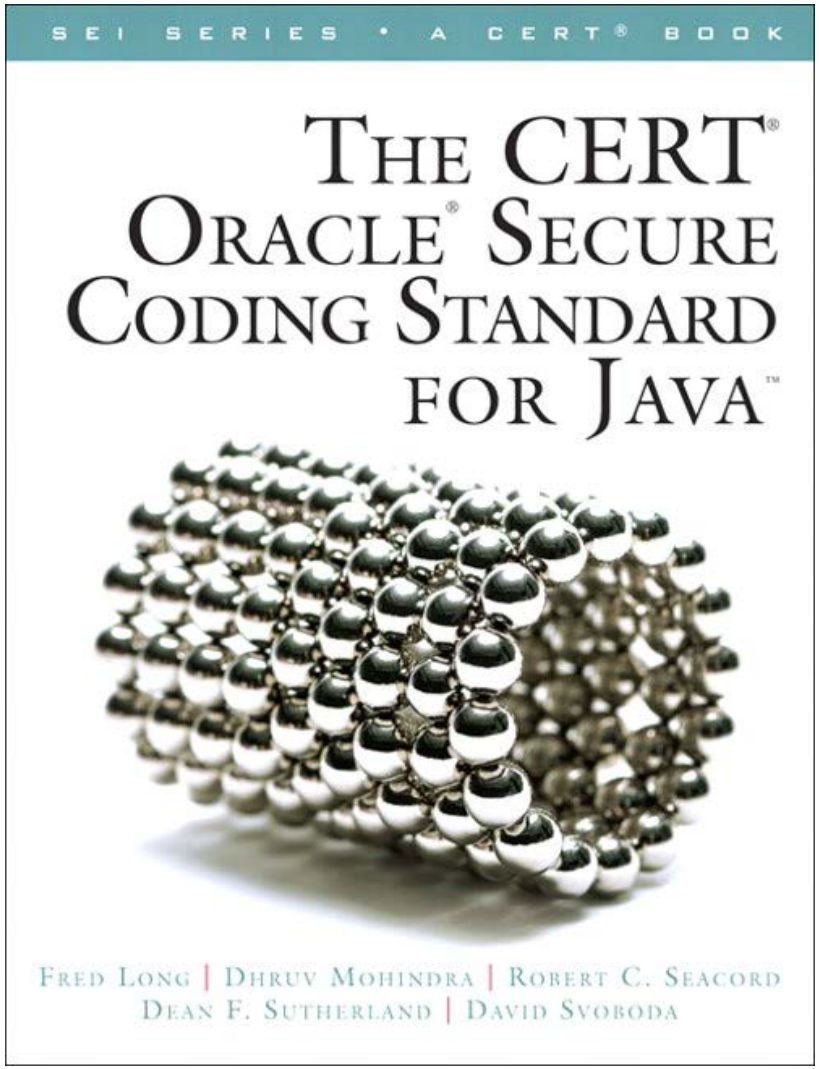
Contact CERT:

Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue

Pittsburgh PA 15213-3890



References

[Bloch 2008] Bloch, Joshua. *Effective Java*, 2nd edition. Addison Wesley, 2008.

[JLS 2005] Gosling, James; Joy, Bill; Steele, Guy; & Bracha, Gilad. *Java Language Specification*, 3rd edition. Prentice Hall, The Java Series. The Java Language Specification, 2005.

[Tutorials 2008] The Java Tutorials, Sun Microsystems, Inc., 2008.