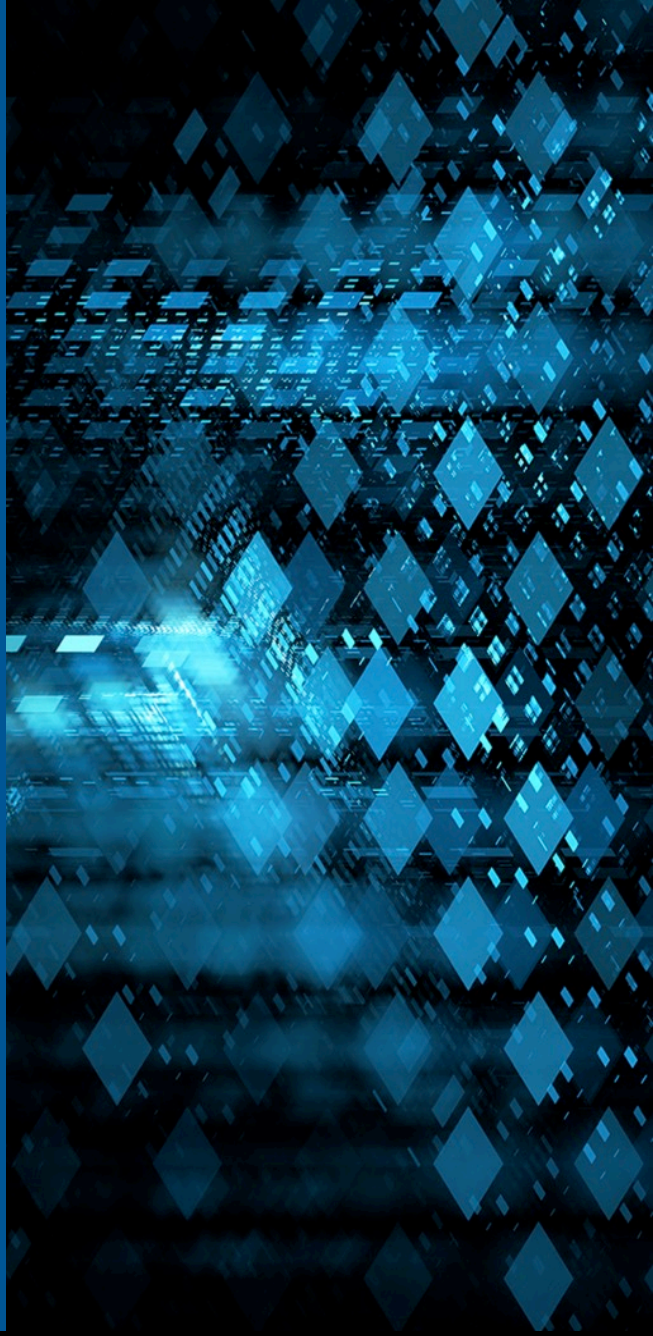
Inside the CERT Oracle Secure Coding Standard for Java

David Svoboda

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Copyright 2016 Carnegie Mellon University

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-0003997

Agenda: Inside the Java Coding Standard

Secure Coding and SCALe

SER07-J. Do not use the default serialized form for classes with implementation-defined invariants

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

OBJ03-J. Prevent heap pollution

ERR01-J. Do not allow exceptions to expose sensitive information

Summary

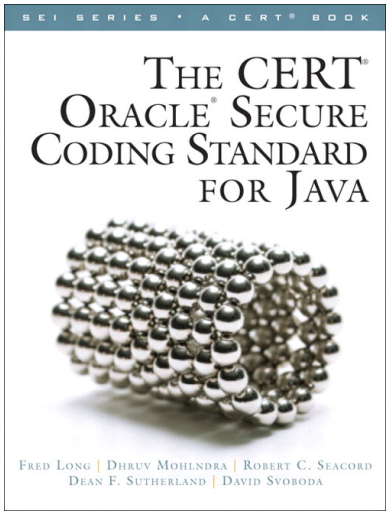
Inside the Java Coding Standard

Secure Coding and SCALe

CERT Secure Coding Standards

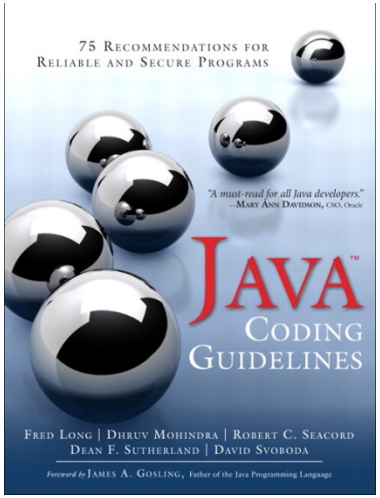
CERT Oracle Secure Coding Standard for Java

- Version 1.0 (Java 7) published in 2011
- Subset applicable to Android development
- Android Annex
- Establishes normative requirements for software systems
- Evaluated for conformance to the coding standard using the Source Code Analysis Laboratory



Java Coding Guidelines

- Published Aug 23, 2013
- Java SE 7 Platform
- Documents and warn against insecure coding practices that are not conformance issues



Secure Coding Professional Certificates

CERT | Software Engineering Institute Carnegie Mellon University

CERT Secure Coding Professional Certificates

Online Courses with Exam and Certificates for C/C++ and Java
2 Courses (Secure Software Concepts & Secure Coding) and Exam
Onsite, instructor-led courses available for groups

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

http://www.sei.cmu.edu/training/p118.cfm

SCALe 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.

Rules and Guidelines

Rules and guidelines include

- Concise but not necessarily precise title
- Precise definition of the rule
- Noncompliant code examples or antipatterns in a pink frame—do not copy and paste into your code
- Compliant solutions in a blue frame that conform with all rules and can be reused in your code
- Risk assessment (rules only)

Risk Assessment

Risk assessment is performed using failure mode, effects, and criticality analysis.

Severity—How serious are the consequences of the rule being ignored?

Value	Meaning	Examples of Vulnerability
1	low	denial-of-service attack, abnormal termination
2	medium	data integrity violation, unintentional information disclosure
3	high	run arbitrary code

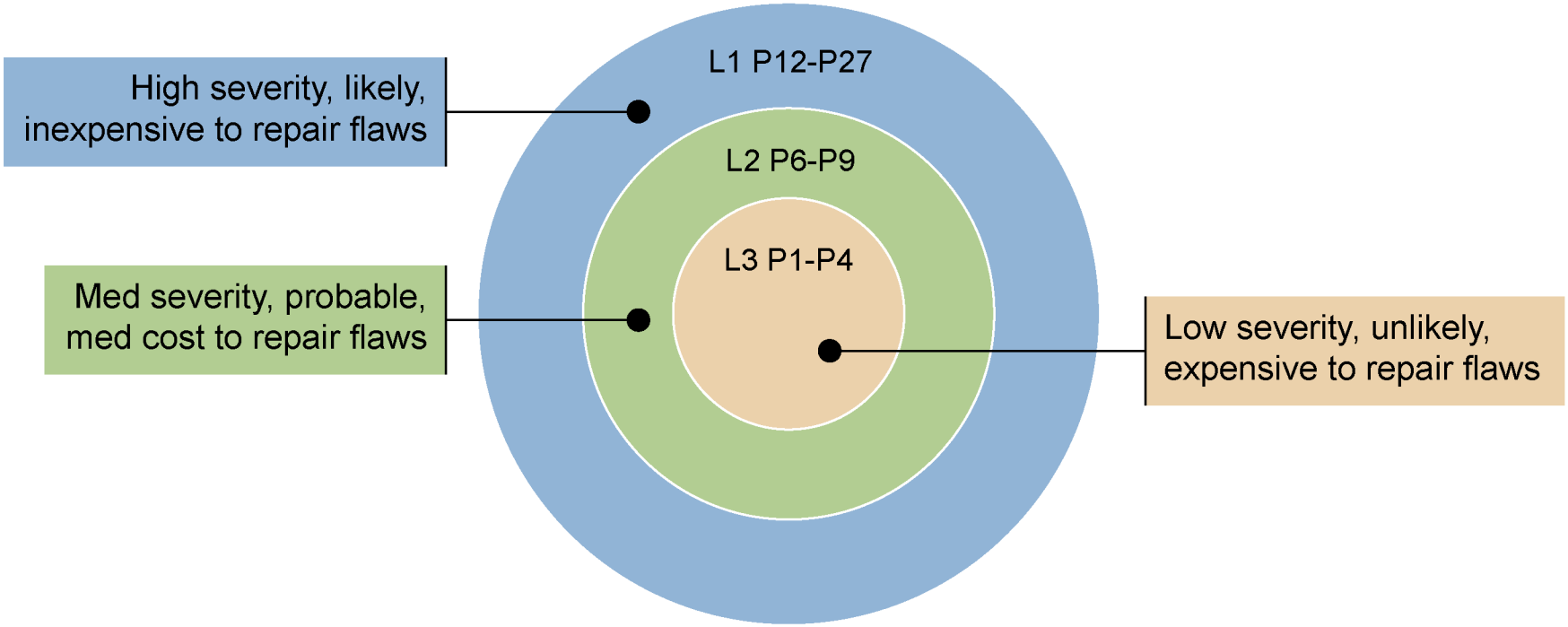
Likelihood—How likely is it that a flaw introduced by ignoring the rule can lead to an exploitable vulnerability?

Value	Meaning
1	unlikely
2	probable
3	likely

Cost—The cost of mitigating the vulnerability.

Value	Meaning	Detection	Correction
1	high	manual	manual
2	medium	automatic	manual
3	low	automatic	automatic

Priorities and Levels



Source Code Analysis Laboratory

Source Code Analysis Laboratory (SCALe)

- Consists of commercial, open source, and experimental analysis
- Is used to analyze various code bases including those from the DoD, energy delivery systems, medical devices, and more
- Provides value to the customer but is also being instrumented to research the effectiveness of coding rules and analysis

SCALe customer-focused process:

1. Customer submits source code to CERT for analysis.
2. Source is analyzed in SCALe using various analyzers.
3. Results are analyzed, validated, and summarized.
4. Detailed report of findings is provided to guide repairs.
5. The developer addresses violations and resubmits repaired code.
6. The code is reassessed to ensure all violations have been properly mitigated.
7. The certification for the product version is published in a registry of certified systems.

Government Demand

SEC. 933 of the National Defense Authorization Act for Fiscal Year 2013 requires evidence that government software development and maintenance organizations and contractors are conforming in computer software coding to approved secure coding standards of the Department during software development, upgrade, and maintenance activities, including through the use of inspection and appraisals.

The Application Security and Development Security Technical Implementation Guide (STIG)

- is being specified in the DoD acquisition programs' Request for Proposals (RFPs).
- provides security guidance for use throughout an application's development lifecycle.

Section 2.1.5, "Coding Standards," of the Application Security and Development STIG identifies the following requirement:

(APP2060.1: CAT II) "The Program Manager will ensure the development team follows a set of coding standards."

Industry Demand

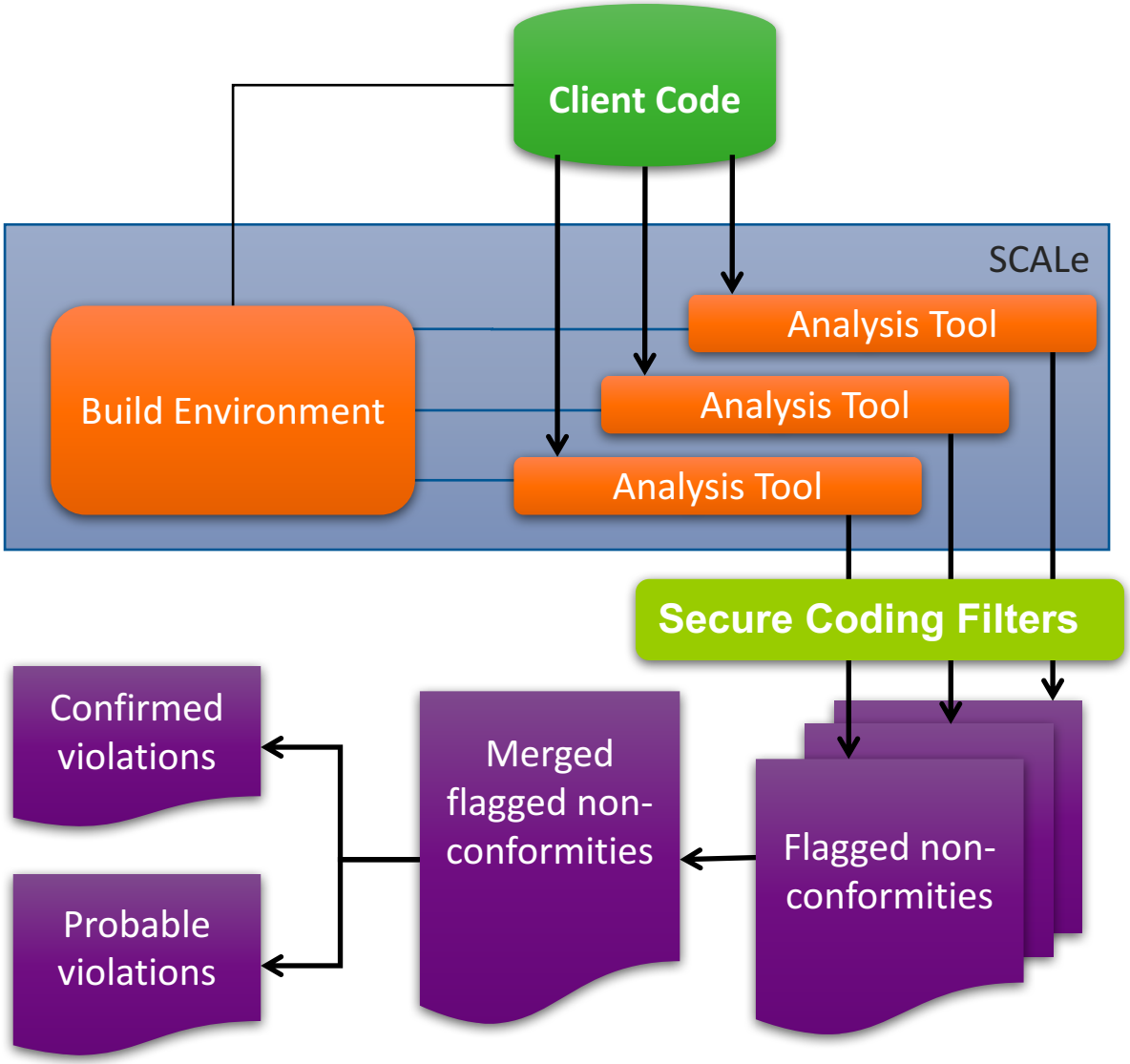
Conformance with CERT secure coding standards can represent a significant investment by a software developer, particularly when it is necessary to refactor or modernize existing software systems.

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 SCALe seal when marketing products.
- Maintain a certificate registry with the certificates of conforming systems.

Conformance Testing Process



SCALe Demo Videos

David Svoboda narrates a series of videos demonstrating the Source Code Analysis Laboratory (SCALe)

Available as a YouTube playlist:

https://www.cert.org/secure-coding/products-services/scale.cfm

https://www.youtube.com/playlist?list=PLSNIEg26NNpxMofZSX-72rxjFEUk9myk-

Rule Coverage

Java Analyzer	Version	Number Of Rules
Coverity	6.5.3	35
FindBugs	2.0.3	44
Fortify	5.1	26
Eclipse	4.3.2 (kepler)	10

Select SCALe Java Assessments

Codebase	Date	KSigLOC	Rules Violated	Diags	True	Suspect	Diags /KSigLOC
A	6/12	4.3	18	345	117	228	80.8
B	9/12	61.2	33	538	288	250	8.8
C	11/13	17.6	21	414	341	73	23.5
D	2/14	653.0	29	8,526	64	8,462	13.1
E	3/14	1.5	8	53	53	0	35.1
F	5/14	403.0	27	3114	723	2,391	7.7

Agenda: Inside the Java Coding Standard

Secure Coding and SCALe

SER07-J. Do not use the default serialized form for classes with implementation-defined invariants

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

OBJ03-J. Prevent heap pollution

ERR01-J. Do not allow exceptions to expose sensitive information

Summary

Inside the Java Coding Standard

SER07-J. Do not use the default serialized form for classes with implementation-defined invariants

Serialization Killer

Serialization can be used maliciously.

CVE-2012-0507 describes an exploit that bypassed Java's applet security sandbox and run malicious code on a remote user's machine.

- deserialized a malicious object that subverted Java's type system.
- infected more than 550,000 Macintosh computers, mostly in the United States and Canada.

Deserialization

Deserialization is analogous to object construction

- Any and all invariants enforced during object construction must also be enforced during deserialization.

Default serialization lacks any enforcement of class invariants.

Creates a new instance of the class without invoking any of the class's constructors.

- any input validation checks in constructors are bypassed.
- programs must not use the default serialized form for any class with implementation-defined invariants.

`AtomicReferenceArray<E>` Class 1

An array of object references in which elements may be updated atomically

- Introduced in Java 5
- `java.util.concurrent.atomic` package
- Parameterized type
- Implements `Serializable`
- No customized `readObject()` method

`AtomicReferenceArray<E>` Class 2

```
public class AtomicReferenceArray<E> implements
java.io.Serializable {
  private static final long serialVersionUID =
                                    -6209656149925076980L;
  private final Object[] array;
```

Prior to Java 1.7.0_02, the **`AtomicReferenceArray<>`** did not validate **`Object`** array during deserialization.

```
  private static final long arrayFieldOffset =
    unsafe.objectFieldOffset(
      AtomicReferenceArray.class.getDeclaredField("array")
    );
  private Object array[];
  // Rest of class...
  // No readObject() method, relies on default readObject
}
```

AtomicReferenceArray<E> Fix

Java 1.7.0_03 validates its array upon deserialization.

```
private void readObject(java.io.ObjectInputStream s)
  throws java.io.IOException, ClassNotFoundException {
  // Note: This must be changed if fields are added
  Object a = s.readFields().get("array", null);
  if (a == null || !a.getClass().isArray()) {
    throw new java.io.InvalidObjectException("Not an array");
  }
  if (a.getClass() != Object[].class){
    a = Arrays.copyOf(
      (Object[])a, Array.getLength(a), Object[].class
    );
  }

  unsafe.putObjectVolatile(this, arrayFieldOffset, a);
```

Compliance with OBJ06-J. Defensively copy mutable inputs and mutable internal components ensures the object remains private.

Agenda: Inside the Java Coding Standard

Secure Coding and SCALe

SER07-J. Do not use the default serialized form for classes with implementation-defined invariants

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

OBJ03-J. Prevent heap pollution

ERR01-J. Do not allow exceptions to expose sensitive information

Summary

Inside the Java Coding Standard

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

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?`**

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

Validation

```
String dir = System.getProperty("dir");
```

Ensures that command succeeds

```
if (new File(dir).isDirectory()) {
  throw new IOException("Not a directory");
}
Runtime rt = Runtime.getRuntime();
Process proc = rt.exec(
  new String[] {"sh", "-c", "ls " + dir}
);
```

Sanitization

```
String dir = System.getProperty("dir");
```

Prevents command injection but does not guarantee that the command succeeds.

```
if (dir.matches(".*\W.*") {
  throw new IOException("Invalid input");
}
Runtime rt = Runtime.getRuntime();
Process proc = rt.exec(
  new String[] {"sh", "-c", "ls " + dir}
);
```

Agenda: Inside the Java Coding Standard

Secure Coding and SCALe

SER07-J. Do not use the default serialized form for classes with implementation-defined invariants

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

OBJ03-J. Prevent heap pollution

ERR01-J. Do not allow exceptions to expose sensitive information

Summary

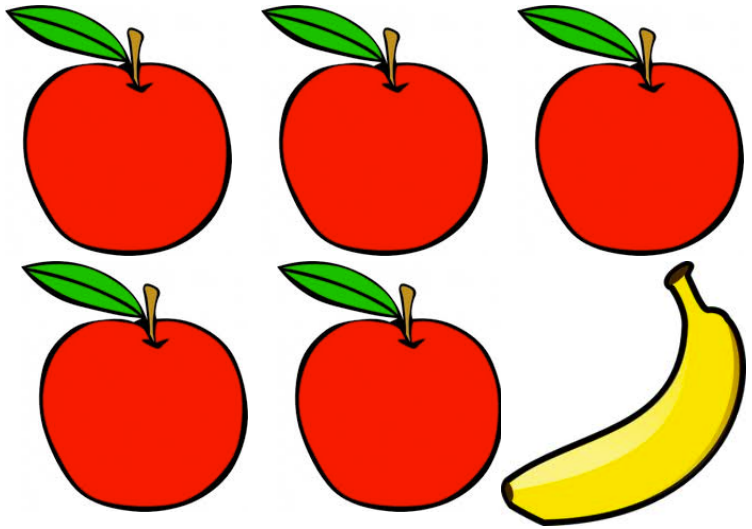
Inside the Java Coding Standard

OBJ03-J. Prevent heap pollution

Heap Pollution

"It is possible that a variable of a parameterized type will refer to an object that is not of that parameterized type. This situation is known as heap pollution."

Java Language Specification § 4.12.2.1.



Heap Pollution Example 1

```
class ListUtility {
  private static void addToList(List list, Object obj) {
    list.add(obj);
  }

  public static void main(String[] args) {
    List<String> list = new ArrayList<String> ();
    addToList(list, 42);
    System.out.println(list.get(0));
  }
}
```

Compile-time unchecked warning.

`List` is not guaranteed to refer to a subtype of its declared type (`List<String>`), but only to subclasses or subinterfaces of the declared type (e.g., `List`).

Heap Pollution Example 2

```
class ListUtility {
  private static void addToList(List list, Object obj) {
    list.add(obj);
  }


  public static void main(String[] args) {
    List<String> list = new ArrayList<String> ();
    addToList(list, 42);
    System.out.println(list.get(0));
  }
}
```

ClassCastException

The JVM lets allows an `Integer` to be stored in a `List<String>` and only throws an exception when the `Integer` is read.
Not fail-fast.

Mitigation: Avoid Raw Types

```
class ListUtility {
  private static void addToList(List<String> list,
                                String str) {
    list.add(str);
  }

  public static void main(String[] args) {
    List<String> list = new ArrayList<String> ();
    addToList(list, 42);
    System.out.println(list.get(0));
  }
}
```

No more compiler warning

Compiler error, can't add **`Integer` to `List<String>`**

Program compiles and runs correctly if we replace 42 with a string, e.g. **`"42"`**

Legacy Code

```
class ListUtility {
  private static void addToList(List list, Object obj) {
    list.add(obj);
  }
  // ...
}
```

Raw types were left in Java to prevent legacy code from breaking.

So what if we couldn't modify `addToList()`?

Mitigation: Checked Collection

```
class ListUtility {
  private static void addToList(List list, Object obj) {
    list.add(obj);
  }

  public static void main(String[] args) {
    List<String> list = new ArrayList<String> ();
    List<String> checkedList = Collections.checkedList(
        list, String.class);
    addToList(checkedList, 42);
    System.out.println(list.get(0));
  }
}
```

Compiler emits warning

ClassCastException

Arrays

```
public class PolluteArrayExample {
  public static void main(String[] args) {
    String list[] = {"foo", "bar"};
    modify(list);
  }

  public static void modify(String[] list) {
    Object[] objectArray = list;
    objectArray[1] = new Integer(42);

    for (String s : list) {
      System.out.println(s);
    }
  }
}
```

Arrays have extra runtime checks and are consequently immune to heap pollution

```
Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
```

Agenda: Inside the Java Coding Standard

Secure Coding and SCALe

SER07-J. Do not use the default serialized form for classes with implementation-defined invariants

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

OBJ03-J. Prevent heap pollution

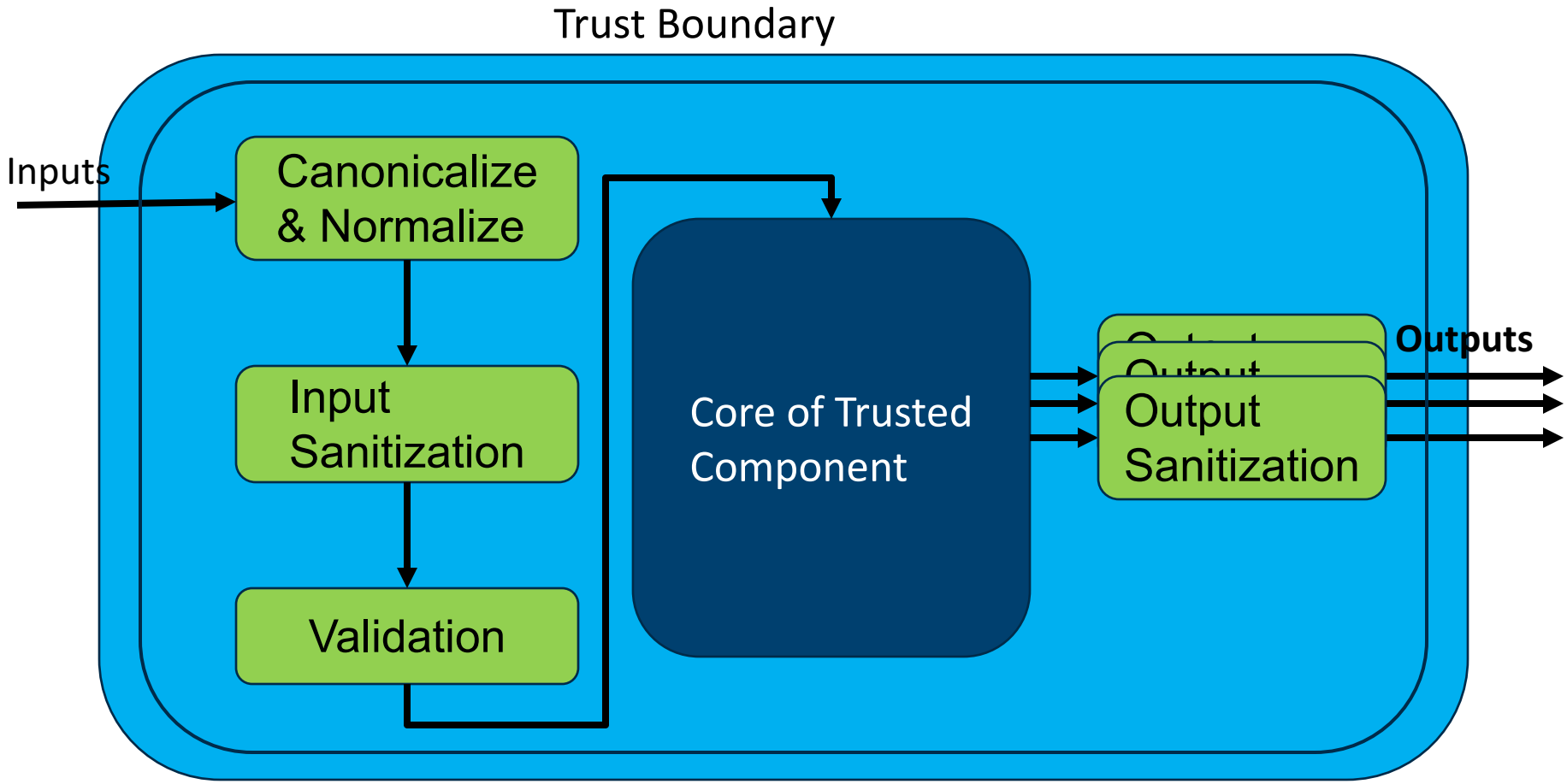
ERR01-J. Do not allow exceptions to expose sensitive information

Summary

Inside the Java Coding Standard

ERR01-J. Do not allow exceptions to expose sensitive information

Trust Boundary Summary



What is Sensitive Information?

That's for you or your organization to decide.

Information that might be sensitive:

- Database internals
- File System
- User names
- Class names
- Platform details (free memory, swap space, etc)

Information that is always sensitive

- Passwords
- Credit Cards
- Personal information (SSN, medical history, etc)

Exceptions to Sanitize

Exception Name	Description of Information Leak or Threat
`java.io.FileNotFoundException`	Underlying file system structure, user name enumeration
`java.lang.OutOfMemoryError`	DoS
`java.lang.StackOverflowError`	DoS
`java.net.BindException`	Enumeration of open ports when untrusted client can choose server port
`java.security.acl.NotOwnerException`	Owner enumeration
`java.sql.SQLException`	Database structure, user name enumeration
`java.util.ConcurrentModificationException`	May provide information about thread-unsafe code
`java.util.jar.JarException`	Underlying file system structure
`java.util.MissingResourceException`	Resource enumeration

Jetty Vulnerability

CVE-2015-2080 describes a vulnerability in the Jetty web server, versions 9.2.3 to 9.2.8,

An illegal character passed in an HTML request caused the server to respond with an error message containing the illegal character.

To provide context, the exception also contained input text that occurred before and after the illegal character.

But due to a miscalculation, the exception also received sensitive information from previous web requests, such as cookies.

What can an attacker learn from `IOException`?

```
class ExceptionExample {
  public static void main(String[] args)
      throws FileNotFoundException {
    // Linux stores a user's home directory path in
    // the environment variable $HOME, Windows in %APPDATA%
    FileInputStream fis =
        new FileInputStream(System.getenv("APPDATA")
                                          + args[0]);
  }
}
```

`IOException` contains the name of the missing file, revealing what files exist on the filesystem.

`IOException` also reveals the value of the **`%APPDATA%`** environment variable

Hiding Sensitive Data

```
class ExceptionExample {
  public static void main(String[] args)
      throws FileNotFoundException {
    // Linux stores a user's home directory path in
    // the environment variable $HOME, Windows in %APPDATA%
    try {
      FileInputStream fis =
          new FileInputStream(System.getenv("APPDATA")
                                          + args[0]);
    } catch (FileNotFoundException e) {
      // Log the exception, hidden from user
      throw new IOException("Unable to retrieve file", e);
    }
}
```

`IOException` still indicates (by its presence) what files exist on the filesystem.

Hiding Sensitive Data 2

```
class ExceptionExample {
  public static void main(String[] args)
      throws FileNotFoundException {
    // Linux stores a user's home directory path in
    // the environment variable $HOME, Windows in %APPDATA%
    try {
      FileInputStream fis =
          new FileInputStream(System.getenv("APPDATA")
                                          + args[0]);
    } catch (FileNotFoundException e) {
      // Log the exception, hidden from user
      throw new SecurityIOException(); //Extends IOException
    }
}
```

SecurityIOException still indicates (by its presence) what files exist on the filesystem.

Hiding Sensitive Data, conclusion

If you want to hide the filesystem from a user, don't let them specify a filename that affects your program's behavior.

Alternatives:

- Abort unless filename obeys restrictions, such as living in the user's home directory.
- Force user to choose between several filenames instead of providing their own.

Agenda: Inside the Java Coding Standard

Secure Coding and SCALe

SER07-J. Do not use the default serialized form for classes with implementation-defined invariants

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

OBJ03-J. Prevent heap pollution

ERR01-J. Do not allow exceptions to expose sensitive information

Summary

Inside the Java Coding Standard

Summary

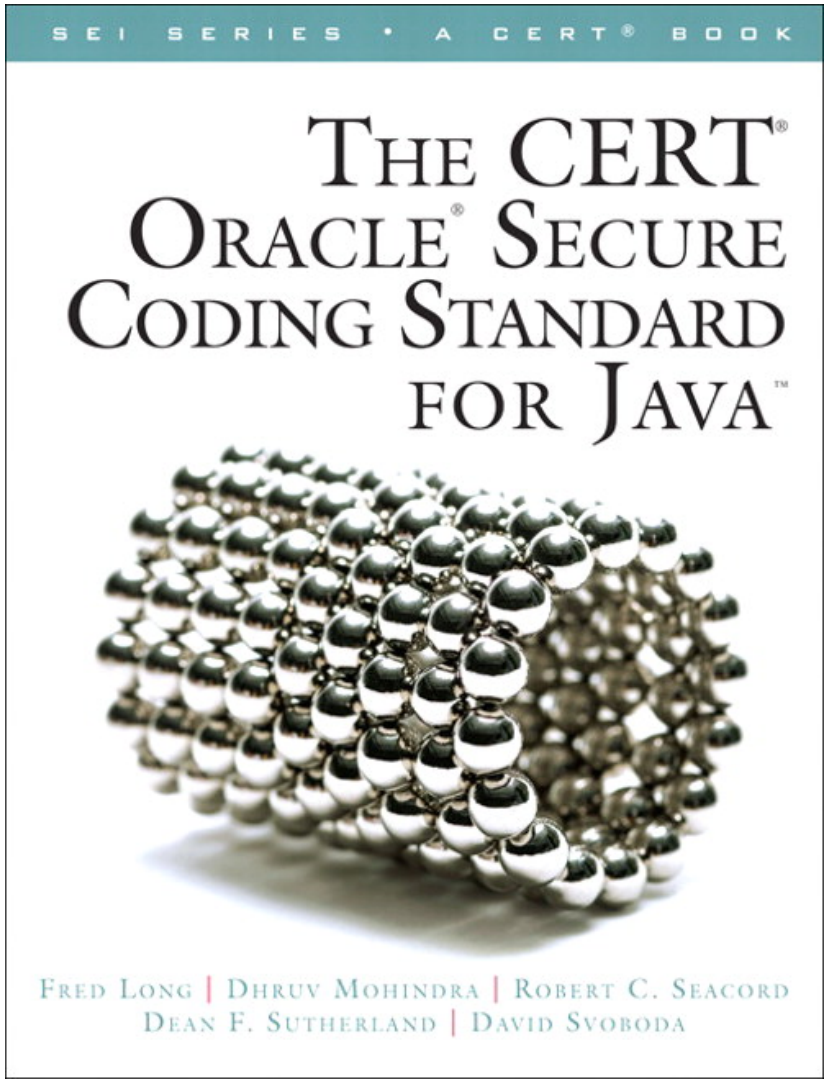
Java Coding Guidelines: Now Available Free Online

Intended primarily for software professionals working in Java Standard Edition (SE) 7 Platform environments.

We encourage the community to participate in the process by creating an account on the secure coding wiki and leaving comments or by contacting the team at secure-coding@cert.org to become an editor.

For free online access to the content of Java Coding Guidelines, visit https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

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

Inside the Java Coding Standard

The End