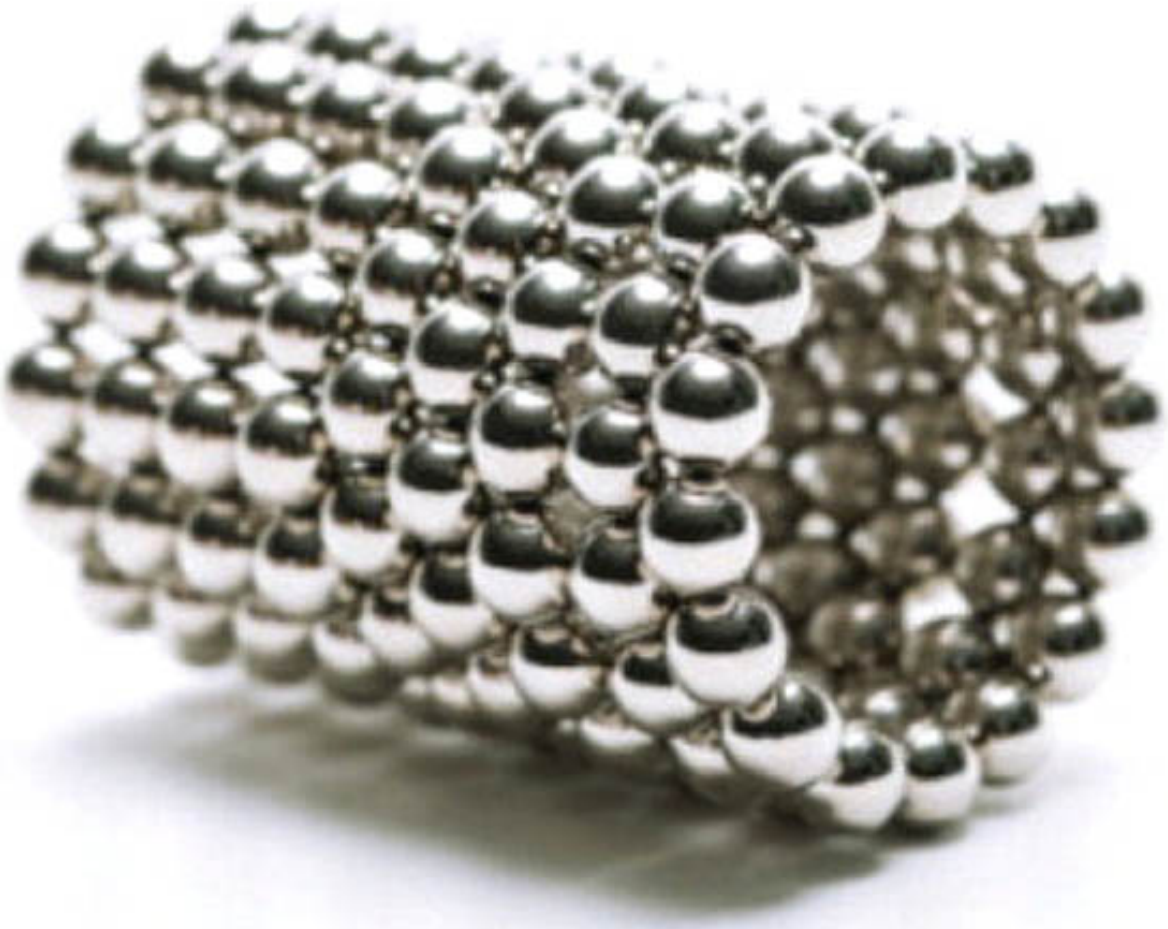
SEI SERIES • A CERT® BOOK

The CERT® Oracle® Secure Coding Standard for Java



Fred Long | Dhruv Mohlndra | Robert C. Seacord
Dean F. Sutherland | David Svoboda

The CERT® Oracle® Secure Coding Standard for Java™

```
// java.util.Collection is an interface
public void copyInterfaceInput(Collection<String> collection) {
  doLogic(collection.clone());
}
```

Compliant Solution

This compliant solution protects against potential malicious overriding by creating a new instance of the nonfinal mutable input, using the expected class rather than the class of the potentially malicious argument. The newly created instance can be forwarded to any code capable of modifying it.

```
public void copyInterfaceInput(Collection<String> collection) {
  // Convert input to trusted implementation
  collection = new ArrayList(collection);
  doLogic(collection);
}
```

Some objects appear to be immutable because they have no mutator methods. For example, the `java.lang.CharSequence` interface describes an immutable sequence of characters. Note, however, that a variable of type `CharSequence` is a reference to an underlying object of some other class that implements the `CharSequence` interface; that other class may be mutable. When the underlying object changes, the `CharSequence` changes. Essentially, the `CharSequence` interface omits methods that would permit object mutation *through that interface* but lacks any guarantee of true immutability. Such objects must still be defensively copied before use. For the case of the `CharSequence` interface, one permissible approach is to obtain an immutable copy of the characters by using the `toString()` method. Mutable fields should not be stored in static variables. When there is no other alternative, create defensive copies of the fields to avoid exposing them to untrusted code.

Risk Assessment

Failing to create a copy of a mutable input may result in a TOCTOU vulnerability or expose internal mutable components to untrusted code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OBJ06-J	medium	probable	high	P4	L3

Related Guidelines

Secure Coding Guidelines for the Java Programming Language, Version 3.0	Guideline 2-2. Create copies of mutable outputs

Bibliography

[Bloch 2008] Item 39. Make defensive copies when needed

[Pugh 2009] Returning References to Internal Mutable State

OBJ07-J. Sensitive classes must not let themselves be copied

Classes containing private, confidential, or otherwise sensitive data are best not copied. If a class is not meant to be copied, then failing to define copy mechanisms, such as a copy constructor, is insufficient to prevent copying.

Java's object cloning mechanism allows an attacker to manufacture new instances of a class by copying the memory images of existing objects rather than by executing the class's constructor. Often this is an unacceptable way of creating new objects. An attacker can misuse the clone feature to manufacture multiple instances of a singleton class, create thread-safety issues by subclassing and cloning the subclass, bypass security checks within the constructor, and violate the invariants of critical data.

Classes that have security checks in their constructors must beware of finalization attacks, as explained in rule OBJ11-J.

Classes that are not sensitive but maintain other invariants must be sensitive to the possibility of malicious subclasses accessing or manipulating their data and possibly invalidating their invariants. See rule OBJ04-J for more information.

Noncompliant Code Example

This noncompliant code example defines class `SensitiveClass`, which contains a character array used to hold a file name, along with a `Boolean` *shared* variable, initialized to `false`. This data is not meant to be copied; consequently, `SensitiveClass` lacks a copy constructor.

```
class SensitiveClass {
  private char[] filename;
  private Boolean shared = false;

  SensitiveClass(String filename) {
    this.filename = filename.toCharArray();
  }

  final void replace() {
    if (!shared) {
      for(int i = 0; i < filename.length; i++) {
        filename[i]= 'x'; }
   }
 }
```

```
  final String get() {
    if (!shared) {
      shared = true;
      return String.valueOf(filename);
    } else {
      throw new IllegalStateException("Failed to get instance");
    }
  }

  final void printFilename() {
    System.out.println(String.valueOf(filename));
  }
}
```

When a client requests a `String` instance by invoking the `get()` method, the `shared` flag is set. To maintain the array's consistency with the returned `String` object, operations that can modify the array are subsequently prohibited. As a result, the `replace()` method designed to replace all elements of the array with an `x` cannot execute normally when the flag is set. Java's cloning feature provides a way to circumvent this constraint even though `SensitiveClass` does not implement the `Cloneable` interface.

This class can be exploited by a malicious class, shown in the following noncompliant code example, that subclasses the nonfinal `SensitiveClass` and provides a `public clone()` method.

```
class MaliciousSubclass extends SensitiveClass implements Cloneable {
  protected MaliciousSubclass(String filename) {
    super(filename);
  }

  @Override public MaliciousSubclass clone() {
  // Well-behaved clone() method
    MaliciousSubclass s = null;
    try {
      s = (MaliciousSubclass)super.clone();
    } catch(Exception e) {
      System.out.println("not cloneable");
    }
    return s;
  }

  public static void main(String[] args) {
    MaliciousSubclass ms1 = new MaliciousSubclass(“file.txt”);
```

```
    MaliciousSubclass ms2 = ms1.clone(); // Creates a copy
    String s = ms1.get();  // Returns filename
    System.out.println(s); // Filename is "file.txt"
    ms2.replace();         // Replaces all characters with 'x'
    // Both ms1.get() and ms2.get() will subsequently
    // return filename = 'xxxxxxxx'
    ms1.printFilename();   // Filename becomes 'xxxxxxxx'
    ms2.printFilename();   // Filename becomes 'xxxxxxxx'
  }
}
```

The malicious class creates an instance `ms1` and produces a second instance `ms2` by cloning the first. It then obtains a new `filename` by invoking the `get()` method on the first instance. At this point, the `shared` flag is set to `true`. Because the second instance `ms2` does not have its shared flag set to `true`, it is possible to alter the first instance `ms1` using the `replace()` method. This obviates any security efforts and severely violates the class's invariants.

Compliant Solution (Final Class)

The easiest way to prevent malicious subclasses is to declare `SensitiveClass` to be final.

```
final class SensitiveClass {
  // ...
}
```

Compliant Solution (Final `clone()`)

Sensitive classes should neither implement the `Cloneable` interface nor provide a copy constructor. Sensitive classes that extend from a superclass that implements `Cloneable` (and are cloneable as a result) must provide a `clone()` method that throws a `CloneNotSupportedException`. This exception must be caught and handled by the client code. A sensitive class that does not implement `Cloneable` must also follow this advice because it inherits the `clone()` method from `Object`. The class can prevent subclasses from being made cloneable by defining a `final clone()` method that always fails.

```
class SensitiveClass {
  // ...
  public final SensitiveClass clone()
                              throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}
```

This class fails to prevent malicious subclasses but does protect the data in `SensitiveClass`. Its methods are protected by being declared final. For more information on handling malicious subclasses, see rule OBJ04-J.

Risk Assessment

Failure to make sensitive classes noncopyable can permit violations of class invariants and provide malicious subclasses with the opportunity to exploit the code to create new instances of objects, even in the presence of the default security manager (in the absence of custom security checks).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OBJ07-J	medium	probable	medium	P8	L2

Bibliography

[McGraw 1998] Twelve rules for developing more secure Java code

[MITRE 2009] CWE-498. Cloneable class containing sensitive information; CWE-491. Public `cloneable()` method without final (aka "object hijack")

[Wheeler 2003] 10.6, Java

■ OBJ08-J. Do not expose private members of an outer class from within a nested class

A nested class is any class whose declaration occurs within the body of another class or interface [JLS 2005]. The use of a nested class is error prone unless the semantics are well understood. A common notion is that only the nested class may access the contents of the outer class. Not only does the nested class have access to the private fields of the outer class, the same fields can be accessed by any other class within the package when the nested class is declared public or if it contains public methods or constructors. As a result, the nested class must not expose the private members of the outer class to external classes or packages.

According to the *Java Language Specification*, §8.3, "Field Declarations" [JLS 2005]:

> Note that a private field of a superclass might be accessible to a subclass (for example, if both classes are members of the same class). Nevertheless, a private field is never inherited by a subclass.

Noncompliant Code Example

This noncompliant code example exposes the private `(x,y)` coordinates through the `getPoint()` method of the inner class. Consequently, the `AnotherClass` class that belongs to the same package can also access the coordinates.

```
class Coordinates {
  private int x;
  private int y;

  public class Point {
    public void getPoint() {
      System.out.println("(" + x + "," + y + ")");
    }
  }
}

class AnotherClass {
  public static void main(String[] args) {
    Coordinates c = new Coordinates();
    Coordinates.Point p = c.new Point();
    p.getPoint();
  }
}
```

Compliant Solution

Use the private access specifier to hide the inner class and all contained methods and constructors.

```
class Coordinates {
  private int x;
  private int y;

  private class Point {
    private void getPoint() {
      System.out.println("(" + x + "," + y + ")");
    }
  }
}

class AnotherClass {
  public static void main(String[] args) {
    Coordinates c = new Coordinates();
    Coordinates.Point p = c.new Point();   // fails to compile
    p.getPoint();
  }
}
```

Compilation of `AnotherClass` now results in a compilation error because the class attempts to access a private nested class.