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 OBJ11-J. Be wary of letting constructors throw exceptions.

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 OBJ04-J. Provide mutable classes with copy functionality to safely allow passing instances to untrusted code 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 approach 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 OBJ04-J. Provide mutable classes with copy functionality to safely allow passing instances to untrusted code.

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

Automated Detection

Tool	Version	Checker	Description
Parasoft Jtest	10.3	**SECURITY.WSC.MCNC**	Implemented

Related Guidelines

MITRE CWE	CWE-498, Cloneable Class Containing Sensitive Information CWE-491, Public `cloneable()` Method without Final (aka "Object Hijack")

Bibliography

[McGraw 1998]	"Twelve Rules for Developing More Secure Java Code"
[Wheeler 2003]	Section 10.6, "Java"