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

Serialization can be used maliciously, for example, to violate the intended invariants of a class. Deserialization is equivalent to object construction; consequently, all invariants enforced during object construction must also be enforced during deserialization. The default serialized form lacks any enforcement of class invariants; consequently, programs must not use the default serialized form for any class with implementation-defined invariants.

The deserialization process creates a new instance of the class without invoking any of the class's constructors. Consequently, any input validation checks in constructors are bypassed. Moreover, transient and static fields may fail to reflect their true values because such fields are bypassed during the serialization procedure and consequently cannot be restored from the object stream. As a result, any class that has transient fields or that performs validation checks in its constructors must also perform similar validation checks when being deserialized.

Validating deserialized objects establishes that the object state is within defined limits and ensures that all transient and static fields have *secure* values. However, fields that are declared final with a constant value will always be restored to the same constant value after deserialization. For example, the value of the field `private transient final n = 42` after deserialization will be 42 rather than 0. Deserialization produces default values for all other cases.

Noncompliant Code Example (Singleton)

In this noncompliant code example [Bloch 2005], a class with singleton semantics uses the default serialized form, which fails to enforce any implementation-defined invariants. Consequently, malicious code can create a second instance even though the class should have only a single instance. For purposes of this example, we assume that the class contains only nonsensitive data.

```
public class NumberData extends Number {
  // ... Implement abstract Number methods, like Number.doubleValue()...

  private static final NumberData INSTANCE = new NumberData ();
  public static NumberData getInstance() {
    return INSTANCE;
  }

  private NumberData() {
    // Perform security checks and parameter validation
  }

  protected int printData() {
    int data = 1000;
    // Print data
    return data;
  }
}

class Malicious {
  public static void main(String[] args) {
    NumberData sc = (NumberData) deepCopy(NumberData.getInstance());
    // Prints false; indicates new instance
    System.out.println(sc == NumberData.getInstance());
    System.out.println("Balance = " + sc.printData());
  }

  // This method should not be used in production code
  public static Object deepCopy(Object obj) {
    try {
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      new ObjectOutputStream(bos).writeObject(obj);
      ByteArrayInputStream bin =
          new ByteArrayInputStream(bos.toByteArray());
      return new ObjectInputStream(bin).readObject();
    } catch (Exception e) {
      throw new IllegalArgumentException(e);
    }
  }
}
```

Compliant Solution

This compliant solution adds a custom `readResolve()` method that replaces the deserialized instance with a reference to the appropriate singleton from the current execution. More complicated cases may also require custom `writeObject()` or `readObject()` methods in addition to (or instead of) the custom `readResolve()` method.

```
public class NumberData extends Number {
  // ...
  protected final Object readResolve() throws NotSerializableException {
    return INSTANCE;
  }
}
```

More information on singleton classes is available in MSC07-J. Prevent multiple instantiations of singleton objects.

Noncompliant Code Example

This noncompliant code example uses a custom-defined `readObject()` method but fails to perform input validation after deserialization. The design of the system requires the maximum ticket number of any lottery ticket to be 20,000 and the minimum ticket number be greater than 0. However, an attacker can manipulate the serialized array to generate a different number on deserialization. Such a number could be greater than 20,000 or could be 0 or negative.

```
public class Lottery implements Serializable {
  private int ticket = 1;
  private SecureRandom draw = new SecureRandom();

  public Lottery(int ticket) {
    this.ticket = (int) (Math.abs(ticket % 20000) + 1);
  }

  public int getTicket() {
    return this.ticket;
  }

  public int roll() {
    this.ticket = (int) ((Math.abs(draw.nextInt()) % 20000) + 1);
    return this.ticket;
  }

  public static void main(String[] args) {
    Lottery l = new Lottery(2);
    for (int i = 0; i < 10; i++) {
      l.roll();
      System.out.println(l.getTicket());
    }
  }

  private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
  }
}
```

Compliant Solution

Any input validation performed in the constructors must also be implemented wherever an object can be deserialized. This compliant solution performs field-by-field validation by reading all fields of the object using the `readFields()` method and `ObjectInputStream.GetField` constructor. The value for each field must be fully validated *before* it is assigned to the object under construction. For more complicated invariants, validating before assignment may require reading multiple field values into local variables to enable checks that depend on combinations of field values.

```
public final class Lottery implements Serializable {
  // ...
  private synchronized void readObject(java.io.ObjectInputStream s)
                       throws IOException, ClassNotFoundException {
    ObjectInputStream.GetField fields = s.readFields();
    int ticket = fields.get("ticket", 0);
    if (ticket > 20000 || ticket <= 0) {
      throw new InvalidObjectException("Not in range!");
    }
    // Validate draw
    this.ticket = ticket;
  }
}
```

Note that the class must be declared final to prevent a malicious subclass from carrying out a finalizer attack (see OBJ11-J. Be wary of letting constructors throw exceptions for more information about finalizer attacks). For extendable classes, an acceptable alternative is to use a flag that indicates whether the instance is safe for use. The flag can be set after validation and must be checked in every method before any operation is performed.

Additionally, any transient or static fields must be explicitly set to an appropriate value within `readObject()`.

Note that this compliant solution is insufficient to protect sensitive data (see SER03-J. Do not serialize unencrypted sensitive data for additional information).

Compliant Solution (Transient)

This compliant solution marks the fields as transient, so they are not serialized. The `readObject()` method initializes them using the `roll()` method. This class need not be final because its fields are private and cannot be tampered with by subclasses, and its methods have been declared final to prevent subclasses from overriding and ignoring them.

```
public class Lottery implements Serializable {
  private transient int ticket = 1;
  private transient SecureRandom draw = new SecureRandom();

  public Lottery(int ticket) {
    this.ticket = (int) (Math.abs(ticket % 20000) + 1);
  }

  public final int getTicket() {
    return this.ticket;
  }

  public final int roll() {
    this.ticket = (int) ((Math.abs(draw.nextInt()) % 20000) + 1);
    return this.ticket;
  }

  public static void main(String[] args) {
    Lottery l = new Lottery(2);
    for (int i = 0; i < 10; i++) {
      l.roll();
      System.out.println(l.getTicket());
    }
  }

  private void readObject(ObjectInputStream in)
          throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    this.draw = new SecureRandom();
    roll();
  }
}
```

Compliant Solution (Nonserializable)

This compliant solution simply does not mark the `Lottery` class serializable:

```
public final class Lottery {
  // ...
}
```

Noncompliant Code Example (`AtomicReferenceArray<>`)

CVE-2012-0507 describes an exploit that managed to bypass Java's applet security sandbox and run malicious code on a remote user's machine. The exploit deserialized a malicious object that subverted Java's type system. The malicious object was an array of two objects. The second object, of type `AtomicReferenceArray<>`, contained an array, but this array was also the first object. This data structure could not be created without serialization because the array referenced by `AtomicReferenceArray<>` should be private. However, whereas the first object was an array of `objects` of type `Help` (which inherited from `ClassLoader`), the `AtomicReferenceArray<>`'s internal array type was `Object`, enabling the malicious code to use `AtomicReferenceArray.set(ClassLoader)` to create an object that was subsequently interpreted as being of type `Help` object with no cast necessary. A cast would have caught this type mismatch. This exploit allowed attackers to create their own `ClassLoader` object, which is forbidden by the applet security manager.

This exploit worked because, in Java versions prior to 1.7.0_02, the object of type `AtomicReferenceArray<>` performed no validation on its internal array.

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

  // Rest of class...
  // No readObject() method, relies on default readObject
}
```

Compliant Solution (`AtomicReferenceArray<>`)

This exploit was mitigated in Java 1.7.0_03 by having the object of type `AtomicReferenceArray<>` validate its array upon deserialization. The `readObject()` method inspects the array contents, and if the array is of the wrong type, it makes a defensive copy of the array, foiling the exploit. This technique is recommended by OBJ06-J. Defensively copy mutable inputs and mutable internal components.

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

  // Rest of class...

  /**
   * Reconstitutes the instance from a stream (that is, deserializes it).
   * @param s the stream
   */
  private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    // Note: This must be changed if any additional fields are defined
    Object a = s.readFields().get("array", null);
    if (a == null || !a.getClass().isArray())
      throw new java.io.InvalidObjectException("Not array type");
    if (a.getClass() != Object[].class)
      a = Arrays.copyOf((Object[])a, Array.getLength(a), Object[].class);
    unsafe.putObjectVolatile(this, arrayFieldOffset, a);
  }
}
```

Risk Assessment

Using the default serialized form for any class with implementation-defined invariants may result in the malicious tampering of class invariants.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SER07-J	Medium	Probable	High	P4	L3

Automated Detection

Tool	Version	Checker	Description

Coverity	7.5	**UNSAFE_DESERIALIZATION**	Implemented
Parasoft Jtest	10.3	**SERIAL.RRSC**	Implemented

Related Guidelines

MITRE CWE	CWE-502, "Deserialization of Untrusted Data"
Secure Coding Guidelines for Java SE, Version 5.0	Guideline 8-3 / SERIAL-3: View deserialization the same as object construction

Bibliography

[API 2014]	Class `Object` Class `Hashtable`
[Bloch 2008]	Item 75, "Consider Using a Custom Serialized Form"
[Greanier 2000]	
[Harold 2006]	Chapter 11, "Object Serialization"
[Hawtin 2008]	Antipattern 8, "Believing Deserialisation Is Unrelated to Construction"
[Rapid7 2014]	Metasploit: Java AtomicReferenceArray Type Violation Vulnerability