SER03-J. Do not serialize unencrypted sensitive data

Although serialization allows an object's state to be saved as a sequence of bytes and then reconstituted at a later time, it provides no mechanism to protect the serialized data. An attacker who gains access to the serialized data can use it to discover sensitive information and to determine implementation details of the objects. An attacker can also modify the serialized data in an attempt to compromise the system when the malicious data is deserialized. Consequently, sensitive data that is serialized is potentially exposed, without regard to the access qualifiers (such as the `private` keyword) that were used in the original code. Moreover, the security manager cannot guarantee the integrity of the deserialized data.

Examples of sensitive data that should never be serialized include cryptographic keys, digital certificates, and classes that may hold references to sensitive data at the time of serialization.

This rule is meant to prevent the unintentional serialization of sensitive information. SER02-J. Sign then seal objects before sending them outside a trust boundary applies to the intentional serialization of sensitive information.

Noncompliant Code Example

The data members of class `Point` are private. Assuming the coordinates are sensitive, their presence in the data stream would expose them to malicious tampering.

```
public class Point implements Serializable {
  private double x;
  private double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public Point() {
    // No-argument constructor
  }
}

public class Coordinates extends Point {
  public static void main(String[] args) {
    FileOutputStream fout = null;
    try {
      Point p = new Point(5, 2);
      fout = new FileOutputStream("point.ser");
      ObjectOutputStream oout = new ObjectOutputStream(fout);
      oout.writeObject(p);
    } catch (Throwable t) {
      // Forward to handler
    } finally {
      if (fout != null) {
        try {
          fout.close();
        } catch (IOException x) {
          // Handle error
        }
      }
    }
  }
}
```

In the absence of sensitive data, classes can be serialized by simply implementing the `java.io.Serializable` interface. By doing so, the class indicates that no security issues may result from the object's serialization. Note that any derived subclasses also inherit this interface and are consequently serializable. This approach is inappropriate for any class that contains sensitive data.

Compliant Solution

When serializing a class that contains sensitive data, programs must ensure that sensitive data is omitted from the serialized form. This includes suppressing both serialization of data members that contain sensitive data and serialization of references to nonserializable or sensitive objects.

This compliant solution both avoids the possibility of incorrect serialization and protects sensitive data members from accidental serialization by declaring the relevant members as transient so that they are omitted from the list of fields to be serialized by the default serialization mechanism.

```
public class Point implements Serializable {
 private transient double x; // Declared transient
 private transient double y; // Declared transient

 public Point(double x, double y) {
  this.x = x;
  this.y = y;
 }

 public Point() {
   // No-argument constructor
 }
}

public class Coordinates extends Point {
  public static void main(String[] args) {
    FileOutputStream fout = null;
    try {
      Point p = new Point(5,2);
      fout = new FileOutputStream("point.ser");
      ObjectOutputStream oout = new ObjectOutputStream(fout);
      oout.writeObject(p);
      oout.close();
    } catch (Exception e) {
      // Forward to handler
    } finally {
      if (fout != null) {
        try {
          fout.close();
        } catch (IOException x) {
          // Handle error
        }
      }
    }
  }
}
```

Other compliant solutions include

- Developing custom implementations of the `writeObject()`, `writeReplace()`, and `writeExternal()` methods that prevent sensitive fields from being written to the serialized stream.
- Defining the `serialPersistentFields` array field and ensuring that sensitive fields are omitted from the array (see SER00-J. Enable serialization compatibility during class evolution).

Noncompliant Code Example

Serialization can be used maliciously, for example, to return multiple instances of a singleton class object. In this noncompliant code example (based on [Bloch 2005]), a subclass `SensitiveClass` inadvertently becomes serializable because it extends the `java.lang.Number` class, which implements `Serializable`:

```
public class SensitiveClass extends Number {
  // ... Implement abstract methods, such as Number.doubleValue()â¦

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

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

  private int balance = 1000;
  protected int getBalance() {
    return balance;
  }
}

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

  // This method should not be used in production code
  static public 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);
    }
  }
}
```

See MSC07-J. Prevent multiple instantiations of singleton objects for more information about singleton classes.

Compliant Solution

Extending a class or interface that implements `Serializable` should be avoided whenever possible. For instance, a nonserializable class could contain an instance of a serializable class and delegate method calls to the serializable class.

When extension of a serializable class by an unserializable class is necessary, inappropriate serialization of the subclass can be prohibited by throwing `NotSerializableException` from custom `writeObject()`, `readObject()`, and `readObjectNoData()` methods, defined in the nonserializable subclass. These custom methods must be declared private (see SER01-J. Do not deviate from the proper signatures of serialization methods for more information).

```
class SensitiveClass extends Number {
  // ...

  private final Object writeObject(java.io.ObjectOutputStream out) throws NotSerializableException {
    throw new NotSerializableException();
  }
  private final Object readObject(java.io.ObjectInputStream in) throws NotSerializableException {
    throw new NotSerializableException();
  }
  private final Object readObjectNoData(java.io.ObjectInputStream in) throws NotSerializableException {
    throw new NotSerializableException();
  }
}
```

It is still possible for an attacker to obtain uninitialized instances of `SensitiveClass` by catching `NotSerializableException` or by using a finalizer attack (see OBJ11-J. Be wary of letting constructors throw exceptions for more information). Consequently, an unserializable class that extends a serializable class must always validate its invariants before executing any methods. That is, any object of such a class must inspect its fields, its actual type (to prevent it being a malicious subclass), and any invariants it possesses (such as being a malicious second object of a singleton class).

Exceptions

SER03-J-EX0: Sensitive data that has been properly encrypted may be serialized.

Risk Assessment

If sensitive data can be serialized, it may be transmitted over an insecure connection, stored in an insecure location, or disclosed inappropriately.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SER03-J	Medium	Likely	High	P6	L2

Automated Detection

Tool	Version	Checker	Description
Coverity	7.5	**UNSAFE_DESERIALIZATION**	Implemented
Parasoft Jtest	10.3	**SECURITY.ESD.SIF**	Implemented

Related Guidelines

MITRE CWE	CWE-499, Serializable Class Containing Sensitive Data CWE-502, Deserialization of Untrusted Data
Secure Coding Guidelines for Java SE, Version 5.0	Guideline 8-2 / SERIAL-2: Guard sensitive data during serialization

Bibliography

[Bloch 2005]	Puzzle 83, "Dyslexic monotheism"
[Bloch 2001]	Item 1, "Enforce the Singleton Property with a Private Constructor"
[Greanier 2000]	Discover the Secrets of the Java Serialization API
[Harold 1999]	
[Long 2005]	Section 2.4, "Serialization"
[Sun 2006]	Serialization Specification, A.4, Preventing Serialization of Sensitive Data