SER08-J. Minimize privileges before deserializing from a privileged context

Unrestricted deserializing from a privileged context allows an attacker to supply crafted input that, upon deserialization, can yield objects that the attacker would otherwise lack permissions to construct. One example is the construction of a sensitive object such as a custom class loader. Consequently, avoid deserializing from a privileged context. When deserializing requires privileges, programs must strip all permissions other than the minimum set required for the intended usage.

Noncompliant Code Example (CVE-2008-5353: `ZoneInfo`)

CVE-2008-5353 describes a Java vulnerability discovered in August 2008 by Sami Koivu [CVE 2008]. Julien Tinnes subsequently wrote an exploit that allowed arbitrary code execution on multiple platforms running vulnerable versions of Java. The problem resulted from deserializing untrusted input from within a privileged context. The vulnerability involves the `sun.util.Calendar.ZoneInfo` class, which, being serializable, is deserialized by the `readObject()` method of the `ObjectInputStream` class.

The default security model of an applet does not allow access to `sun.util.calendar.ZoneInfo` because applets cannot be permitted to invoke any method from any class within the `sun` package. As a result, prior to JDK 1.6 u11, the acceptable method for an unsigned applet to deserialize a `ZoneInfo` object was to execute the call from a privileged context, such as a `doPrivileged()` block. A vulnerability results because there is no guaranteed method of knowing whether the serialized stream contains a bona fide `ZoneInfo` object rather than a malicious serializable class. The vulnerable code casts the malicious object to the `ZoneInfo` type, which typically causes a `ClassCastException` if the actual deserialized class is not a `ZoneInfo` object. This exception, however, is of little consequence because it is possible to store a reference to the newly created object in a static context so that the garbage collector cannot act upon it.

A nonserializable class can be extended and its subclass can be made serializable. Also, a subclass automatically becomes serializable if it derives from a serializable class. During deserialization of the subclass, the Java Virtual Machine (JVM) calls the no-argument constructor of the most derived superclass that *does not* implement `java.io.Serializable` either directly or indirectly. Calling the no-argument constructor allows it to fix the state of this superclass.

In the following code snippet, class `A`'s no-argument constructor is called when `C` is deserialized because `A` does not implement `Serializable`. Subsequently, `Object`'s constructor is invoked. This procedure cannot be carried out programmatically, so the JVM generates the equivalent bytecode at runtime. Typically, when the superclass's constructor is called by a subclass, the subclass remains on the stack. However, in deserialization this does not happen. Only the unvalidated bytecode is present, which allows any security checks within the superclass's constructor to be bypassed because the complete execution chain is not scrutinized.

```
class A { // Has Object as superclass
  A(int x) { }
  A() { }
}

class B extends A implements Serializable {
  B(int x) { super(x); }
}

class C extends B {
  C(int x) { super(x); }
}
```

At this point, there is no subclass code on the stack and the superclass's constructor is executed with no restrictions because `doPrivileged()` allows the immediate caller to exert its full privileges. Because the immediate caller `java.util.Calendar` is trusted, it exhibits full system privileges.

A custom class loader can be used to exploit this vulnerability. Instantiating a class loader object requires special permissions that are made available by the security policy that is enforced by the `SecurityManager`. An unsigned applet cannot carry out this step by default. However, if an unsigned applet can execute a custom class loader's constructor, it can effectively bypass all the security checks (it has the requisite privileges as a direct consequence of the vulnerability). A custom class loader can be designed to extend the system class loader, undermine security, and carry out prohibited actions such as reading or deleting files on the user's file system. Moreover, legitimate security checks in the constructor are meaningless because the code is granted all privileges. The following noncompliant code example illustrates the vulnerability:

```
try {
  ZoneInfo zi = (ZoneInfo) AccessController.doPrivileged(
    new PrivilegedExceptionAction() {
      public Object run() throws Exception {
        return input.readObject();
      }
  });
  if (zi != null) {
    zone = zi;
  }
} catch (Exception e)
{
  // Handle error
}
```

Compliant Solution (CVE-2008-5353: `Zoneinfo`)

This vulnerability was fixed in JDK v1.6 u11 by defining a new `AccessControlContext` `INSTANCE`, with a new `ProtectionDomain`. The `ProtectionDomain` encapsulated a `RuntimePermission` called `accessClassInPackage.sun.util.calendar`. Consequently, the code was granted the minimal set of permissions required to access the `sun.util.calendar` class. This whitelisting approach guaranteed that a security exception would be thrown in all other cases of invalid access. The code also uses the two-argument form of `doPrivileged()`, which strips all permissions other than the ones specified in the `ProtectionDomain`.

```
private static class CalendarAccessControlContext {
  private static final AccessControlContext INSTANCE;
    static {
      RuntimePermission perm =
          new RuntimePermission("accessClassInPackage.sun.util.calendar");
      PermissionCollection perms = perm.newPermissionCollection();
      perms.add(perm);
      INSTANCE = new AccessControlContext(new ProtectionDomain[] {
          new ProtectionDomain(null, perms)
      });
    }
  }

// ...
try {
  zi = AccessController.doPrivileged(
      new PrivilegedExceptionAction<ZoneInfo>() {
        public ZoneInfo run() throws Exception {
          return (ZoneInfo) input.readObject();
        }
      }, CalendarAccessControlContext.INSTANCE);
} catch (PrivilegedActionException pae) { /* ... */ }
if (zi != null) {
  zone = zi;
}
```

Risk Assessment

Deserializing objects from an unrestricted privileged context can result in arbitrary code execution.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
SER08-J	High	Likely	Medium	P18	L1

Related Guidelines

MITRE CWE	CWE-250, Execution with Unnecessary Privileges

Bibliography

[API 2014]	
[CVE 2011]	CVE-2008-5353