Secure Coding Guidelines for Java SE

Taken from:

https://www.oracle.com/technetwork/java/seccodeguide-139067.html

8 Serialization and Deserialization

Note: Deserialization of untrusted data is inherently dangerous and should be avoided.

Java Serialization provides an interface to classes that sidesteps the field access control mechanisms of the Java language. As a result, care must be taken when performing serialization and deserialization. Furthermore, deserialization of untrusted data should be avoided whenever possible, and should be performed carefully when it cannot be avoided (see 8-6 for additional information).

Guideline 8-1 / SERIAL-1: Avoid serialization for security-sensitive classes

Security-sensitive classes that are not serializable will not have the problems detailed in this section. Making a class serializable effectively creates a public interface to all fields of that class. Serialization also effectively adds a hidden public constructor to a class, which needs to be considered when trying to restrict object construction.

Similarly, lambdas should be scrutinized before being made serializable. Functional interfaces should not be made serializable without due consideration for what could be exposed.

Guideline 8-2 / SERIAL-2: Guard sensitive data during serialization

Once an object has been serialized the Java language's access controls can no longer be enforced and attackers can access private fields in an object by analyzing its serialized byte stream. Therefore, do not serialize sensitive data in a serializable class.

Approaches for handling sensitive fields in serializable classes are:

- Declare sensitive fields `transient`
- Define the `serialPersistentFields` array field appropriately
- Implement `writeObject` and use `ObjectOutputStream.putField` selectively
- Implement `writeReplace` to replace the instance with a serial proxy
- Implement the `Externalizable` interface

Guideline 8-3 / SERIAL-3: View deserialization the same as object construction

Deserialization creates a new instance of a class without invoking any constructor on that class. Therefore, deserialization should be designed to behave like normal construction.

Default deserialization and `ObjectInputStream.defaultReadObject` can assign arbitrary objects to non-transient fields and does not necessarily return. Use

`ObjectInputStream.readFields` instead to insert copying before assignment to fields. Or, if possible, don't make sensitive classes serializable.

```
        public final class ByteString implements java.io.Serializable {
            private static final long serialVersionUID = 1L;
            private byte[] data;
            public ByteString(byte[] data) {
                this.data = data.clone(); // Make copy before assignment.
            }
            private void readObject(
                java.io.ObjectInputStream in
            ) throws java.io.IOException, ClassNotFoundException {
                java.io.ObjectInputStreadm.GetField fields =
                    in.readFields();
                this.data = ((byte[])fields.get("data")).clone();
            }
            ...
        }
```

Perform the same input validation checks in a `readObject` method implementation as those performed in a constructor. Likewise, assign default values that are consistent with those assigned in a constructor to all fields, including transient fields, which are not explicitly set during deserialization.

In addition create copies of deserialized mutable objects before assigning them to internal fields in a `readObject` implementation. This defends against hostile code deserializing byte streams that are specially crafted to give the attacker references to mutable objects inside the deserialized container object.

```
        public final class Nonnegative implements java.io.Serializable {
            private static final long serialVersionUID = 1L;
            private int value;
            public Nonnegative(int value) {
                // Make check before assignment.
                this.data = nonnegative(value);
            }
            private static int nonnegative(int value) {
                if (value < 0) {
                    throw new IllegalArgumentException(value +
                                                       " is negative");
                }
                return value;
            }
            private void readObject(
                java.io.ObjectInputStream in
            ) throws java.io.IOException, ClassNotFoundException {
                java.io.ObjectInputStreadm.GetField fields =
                    in.readFields();
                this.value = nonnegative(field.get(value, 0));
            }
            ...
        }
```

Attackers can also craft hostile streams in an attempt to exploit partially initialized (deserialized) objects. Ensure a serializable class remains totally unusable until deserialization completes successfully. For example, use an `initialized` flag. Declare the flag as a private transient field and only set it in a `readObject` or `readObjectNoData` method (and in constructors) just prior to returning successfully. All public and protected methods in the class must consult the `initialized` flag before proceeding with their normal logic. As discussed earlier, use of an `initialized` flag can be cumbersome. Simply ensuring that all fields contain a safe value (such as null) until deserialization successfully completes can represent a reasonable alternative.

Security-sensitive serializable classes should ensure that object field types are final classes, or do special validation to ensure exact types when deserializing. Otherwise attacker code may populate the fields with malicious subclasses which behave in unexpected ways. For example, if a class has a field of type `java.util.List`, an attacker may populate the field with an implementation which returns inconsistent data.

Guideline 8-4 / SERIAL-4: Duplicate the SecurityManager checks enforced in a class during serialization and deserialization

Prevent an attacker from using serialization or deserialization to bypass the `SecurityManager` checks enforced in a class. Specifically, if a serializable class enforces a `SecurityManager` check in its constructors, then enforce that same check in a `readObject` or `readObjectNoData` method implementation. Otherwise an instance of the class can be created without any check via deserialization.

```
public final class SensitiveClass implements java.io.Serializable {
     public SensitiveClass() {
        // permission needed to instantiate SensitiveClass
        securityManagerCheck();

        // regular logic follows
    }

    // implement readObject to enforce checks
    //   during deserialization
    private void readObject(java.io.ObjectInputStream in) {
        // duplicate check from constructor
        securityManagerCheck();

        // regular logic follows
    }
}
```

If a serializable class enables internal state to be modified by a caller (via a public method, for example) and the modification is guarded with a `SecurityManager` check, then enforce that same check in a `readObject` method implementation. Otherwise, an attacker can use deserialization to create another instance of an object with modified state without passing the check.

```
public final class SecureName implements java.io.Serializable {

    // private internal state
    private String name;

    private static final String DEFAULT = "DEFAULT";
    public SecureName() {
        // initialize name to default value
        name = DEFAULT;
    }

    // allow callers to modify private internal state
    public void setName(String name) {
        if (name!=null ? name.equals(this.name)
                       : (this.name == null)) {
            // no change - do nothing
            return;
        } else {
            // permission needed to modify name
            securityManagerCheck();

            inputValidation(name);

            this.name = name;
        }
    }

    // implement readObject to enforce checks
    //   during deserialization
    private void readObject(java.io.ObjectInputStream in) {
        java.io.ObjectInputStream.GetField fields =
            in.readFields();
        String name = (String) fields.get("name", DEFAULT);

        // if the deserialized name does not match the default
        //   value normally created at construction time,
        //   duplicate checks


        if (!DEFAULT.equals(name)) {
            securityManagerCheck();
            inputValidation(name);
        }
        this.name = name;
    }

}
```

If a serializable class enables internal state to be retrieved by a caller and the retrieval is guarded with a `SecurityManager` check to prevent disclosure of sensitive data, then enforce that same check in a `writeObject` method implementation. Otherwise, an attacker can serialize an object to bypass the check and access the internal state simply by reading the serialized byte stream.

```
public final class SecureValue implements java.io.Serializable {
    // sensitive internal state
    private String value;

    // public method to allow callers to retrieve internal state

    public String getValue() {
        // permission needed to get value
        securityManagerCheck();
```

```
                return value;
            }


            // implement writeObject to enforce checks
            //  during serialization
            private void writeObject(java.io.ObjectOutputStream out) {
                // duplicate check from getValue()
                securityManagerCheck();
                out.writeObject(value);
            }
        }
```

Guideline 8-5 / SERIAL-5: Understand the security permissions given to serialization and deserialization

Permissions appropriate for deserialization should be carefully checked. Additionally, deserialization of untrusted data should generally be avoided whenever possible.

Serialization with full permissions allows permission checks in `writeObject` methods to be circumvented. For instance, `java.security.GuardedObject` checks the guard before serializing the target object. With full permissions, this guard can be circumvented and the data from the object (although not the object itself) made available to the attacker.

Deserialization is more significant. A number of `readObject` implementations attempt to make security checks, which will pass if full permissions are granted. Further, some non-serializable security-sensitive, subclassable classes have no-argument constructors, for instance `ClassLoader`. Consider a malicious serializable class that subclasses `ClassLoader`. During deserialization the serialization method calls the constructor itself and then runs any `readObject` in the subclass. When the `ClassLoader` constructor is called no unprivileged code is on the stack, hence security checks will pass. Thus, don't deserialize with permissions unsuitable for the data. Instead, data should be deserialized with the least necessary privileges.

Guideline 8-6 / SERIAL-6: Filter untrusted serial data

Serialization Filtering was introduced in JDK 9 to improve both security and robustness when using Object Serialization. Security guidelines consistently require that input from external sources be validated before use; serialization filtering provides a mechanism to validate classes before they are deserialized. Filters can be configured that apply to every use of object deserialization without modifying the application. The filters are configured via system properties or configured using the override mechanism of the security properties. A typical use case is to black-list classes that have been identified as potentially compromising the Java runtime. White-listing known safe classes is also straight-forward (and preferred over a black-list approach for stronger security). The filter mechanism allows object-serialization clients to more easily validate their inputs. For a more fine-grained approach the `ObjectInputFilter` API allows an application to integrate finer control specific to each use of `ObjectInputStream.`

RMI supports the setting of a serialization filters to protect remote invocations of exported objects. The RMI Registry and RMI distributed garbage collector use the filtering mechanisms defensively.

Support for the configurable filters has been included in the CPU releases for JDK 8u121, JDK 7u131, and JDK 6u141.

For more information and details please refer to [17] and [20].

9 Access Control

Although Java is largely an *object-capability* language, a stack-based access control mechanism is used to securely provide more conventional APIs.

Many of the guidelines in this section cover the use of the SecurityManager to perform security checks, and to elevate or restrict permissions for code. Note that the SecurityManager does not and cannot provide protection against issues such as side-channel attacks or lower level problems such as Row hammer, nor can it guarantee complete intra-process isolation. Separate processes (JVMs) should be used to isolate untrusted code from trusted code with sensitive information. Utilizing lower level isolation mechanisms available from operating systems or containers is also recommended.

Guideline 9-1 / ACCESS-1: Understand how permissions are checked

The standard security check ensures that each frame in the call stack has the required permission. That is, the current permissions in force is the *intersection* of the permissions of each frame in the current access control context. If any frame does not have a permission, no matter where it lies in the stack, then the current context does not have that permission.

Consider an application that indirectly uses secure operations through a library.

```
package xx.lib;

public class LibClass {
    private static final String OPTIONS = "xx.lib.options";

    public static String getOptions() {
        // checked by SecurityManager
        return System.getProperty(OPTIONS);
    }
}

package yy.app;

class AppClass {
    public static void main(String[] args) {
        System.out.println(
            xx.lib.LibClass.getOptions()
        );
    }
}
```

When the permission check is performed, the call stack will be as illustrated below.

```
+--------------------------------+
| java.security.AccessController |
|   .checkPermission(Permission) |
+--------------------------------+
| java.lang.SecurityManager      |
|   .checkPermission(Permission) |
+--------------------------------+
| java.lang.SecurityManager      |
|   .checkPropertyAccess(String) |
+--------------------------------+
| java.lang.System               |
|   .getProperty(String)         |
+--------------------------------+
| xx.lib.LibClass                |
|   .getOptions()                |
+--------------------------------+
| yy.app.AppClass                |
|   .main(String[])              |
+--------------------------------+
```

In the above example, if the `AppClass` frame does not have permission to read a file but the `LibClass` frame does, then a security exception is still thrown. It does not matter that the immediate caller of the privileged operation is fully privileged, but that there is unprivileged code on the stack somewhere.

For library code to appear transparent to applications with respect to privileges, libraries should be granted permissions at least as generous as the application code that it is used with. For this reason, almost all the code shipped in the JDK and extensions is fully privileged. It is therefore important that there be at least one frame with the application's permissions on the stack whenever a library executes security checked operations on behalf of application code.

Guideline 9-2 / ACCESS-2: Beware of callback methods

Callback methods are generally invoked from the system with full permissions. It seems reasonable to expect that malicious code needs to be on the stack in order to perform an operation, but that is not the case. Malicious code may set up objects that bridge the callback to a security checked operation. For instance, a file chooser dialog box that can manipulate the filesystem from user actions, may have events posted from malicious code. Alternatively, malicious code can disguise a file chooser as something benign while redirecting user events.

Callbacks are widespread in object-oriented systems. Examples include the following:

- Static initialization is often done with full privileges
- Application main method
- Applet/Midlet/Servlet lifecycle events
- `Runnable.run`

This bridging between callback and security-sensitive operations is particularly tricky because it is not easy to spot the bug or to work out where it is.

When implementing callback types, use the technique described in Guideline 9-6 to transfer context.