Security Survey

B

Objectives

After completing this lesson, you should be able to:

- Identify potential Denial of Service vulnerabilities
- Secure confidential information
- Support data integrity
- Explain reasons for input validation
- Limit accessibility and extensibility of sensitive objects
- Consider security measures for serialization

About This Lesson

- This lesson offers a sampling to get you thinking about security.
- There's much more to learn before becoming a security expert.
- In general, it's good to be paranoid and think critically of your code.
- Consider all the angles.
- Paranoia keeps you on guard against even the slightest vulnerabilities.
  - Assume bad actors are always out to get you.
  - Assume developers and users will make mistakes.

Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- Developing Secure Objects
- Secure Serialization and Deserialization

Denial of Service (DoS) Attack

Symptoms

- Legitimate users are unable to access resources and services.
- There is excessive resource consumption. No resources remain to do legitimate work.

Causes

- A file or code construct grows too large.
- A service or connection is overwhelmed with bogus requests.

Prevention

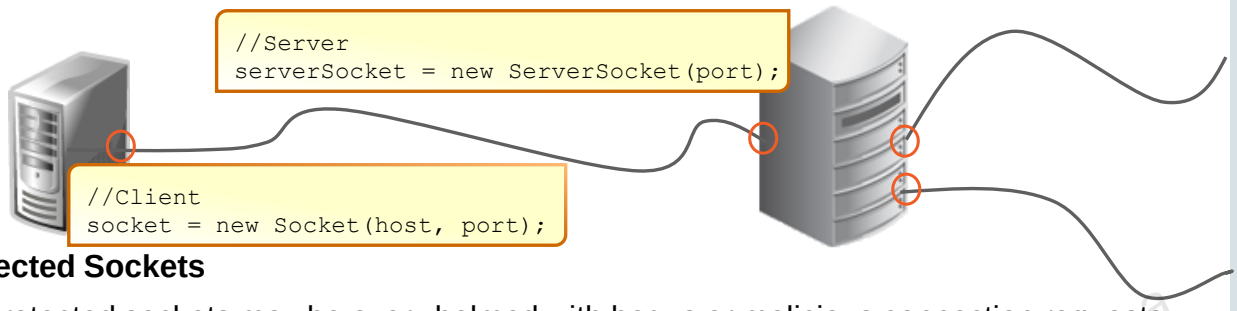
- Use permissions to restrict access to code that consumes vulnerable resources.
- Validate all inputs into a system.
- Release resources in all cases.
- Monitor excessive resource consumption disproportionate to that used to request a service.

Input into a system should be checked so that it will not cause excessive resource consumption disproportionate to that used to request the service. When inputs are not checked, a system is vulnerable to attacks that hog resources and prevent legitimate users from accessing services and resources. Commonly affected resources include CPU cycles, memory, disk space, and file descriptors.

Sockets

About Sockets

- Sockets are the endpoints in a connected network.
- Ensure these endpoints aren't vulnerable to malicious connections.



Unprotected Sockets

- Unprotected sockets may be overwhelmed with bogus or malicious connection requests.
- Restrict connections and privileged access.
- Validate server addresses and check permissions before creating new sockets.

Assume variables `socket` and `serverSocket` are declared elsewhere.

Other DoS Examples

Zip Bomb

- An innocuously sized file requires an outrageous amount of space after decompression.
- Limit the decompressed data size. Don't rely on the file's compressed size or metadata.

Billion Laughs

- XML entity expansion causes an XML document to grow dramatically during parsing.
- Set `XMLConstants.FEATURE_SECURE_PROCESSING` to enforce reasonable limits.

Unsustainable Growth

- Ensure that constructs like collections and buffers can't grow to unmanageable sizes.

Zip bombs are perpetrated by short file that are very highly compressed. For instance, ZIPs, GIFs and gzip encoded http contents. When decompressing files, it's better to limit the decompressed data size rather than relying on the compressed size or metadata.

Billion laughs attacks occur when XML entity expansion causes an XML document to grow dramatically during parsing. This is explained more later in this lesson. To prevent this type of attack, set the XMLConstants.FEATURE_SECURE_PROCESSING feature to enforce reasonable limits.

Collections are sized dynamically. Ensure that a collection cannot grow so big that it unreasonably hogs all available memory.

Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- Developing Secure Objects
- Secure Serialization and Deserialization

Enemies Target Confidential Information

About Confidential Information

- This includes users' personal data, security credentials, or information revealing how your system functions.
- Enemies exploit this information for gain or chaos, harming your users and your system.
- If information is exposed or tampered with, your system is no longer trustable.

Prevention

- Clearly identify and encapsulate confidential information.
- Be as restrictive as possible with permissions.
- Limit scope and access to objects holding confidential information.
- Store trusted information in unmodifiable or immutable objects.
- Do not reveal confidential information to untrusted code, libraries, exceptions, or logs.

Purge Sensitive Information from Exceptions

- Sanitize exceptions so confidential information is not reported.
- This example output dangerously clues enemies to the location of your application's configuration file.

```
Cannot find main configuration file:
  C:\config\badfile.properties (The system cannot find the path specified)
  java.io.FileNotFoundException: C:\config\badfile.properties
    (The system cannot find the path specified)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:138)
    at java.io.FileInputStream.<init>(FileInputStream.java:97)
    at com.example.sec.FileException.readProps(FileException.java:20)
    at com.example.sec.FileException.main(FileException.java:12)

Java Result: -1
```

Internal exceptions should be caught and sanitized before propagating them to upstream callers. The type of an exception may reveal sensitive information, even if the message has been removed. For instance, exceptions related to file access like `FileNotFoundException` may reveal whether a file exists. An attacker can learn about your directory system by providing various file names as input and analyzing the resulting exceptions.

Do Not Log Highly Sensitive Information

- This example log exposes a user's ID, address, and credit card number.
- Your enemies may use this information to commit fraud and identity theft.

```
Feb 08, 2020 10:55:14 AM com.example.sec.DetailedLogger logMessages
SEVERE: ID 123-45-6778 for User John Adams does not match.
Feb 08, 2020 10:55:14 AM com.example.sec.DetailedLogger logMessages
SEVERE: User John Adams located at 123 Drury Lane, Quincy, MA
Feb 08, 2020 10:55:14 AM com.example.sec.DetailedLogger logMessages
SEVERE: Cannot find exp date for credit card 1111-1111-1111-1111
```

This example log reveals a user's ID, address, and credit card number, which are highly sensitive. This type of information should not be kept for longer than necessary nor where it may be seen, even by administrators. It should not be sent to log files, and its presence should not be detectable through searches.

Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- Developing Secure Objects
- Secure Serialization and Deserialization

Validate Inputs

- Expect all kinds of crazy inputs:
  - Casual typos
  - Misunderstandings in formatting
  - Special characters, which could be disguised commands
  - Devious inputs that reveal information about the system
  - Malicious inputs to hack and exploit security holes
- Programs must anticipate this unpredictability and elegantly recover from any issues.
  - Never trust inputs from untrusted sources.
  - Use well-tested, specialized libraries and Java APIs.
  - Avoid ad hoc code for validation.

Numbers That Make Programs Go Awry

- Be careful checking resource limits and dealing with numeric inputs.
- There are APIs to guard against overflowed number space:

```
final int MAX = Integer.MAX_VALUE;
System.out.println(MAX);                          // 2147483647
System.out.println(MAX +1);                       //-2147483648
System.out.println(Math.addExact(MAX, 1));        // java.lang.ArithmeticException
                                                  // Also try multiplyExact
                                                  // and decrementExact
```

- There are APIs to guard against bad floating point values:

```
if (Double.isInfinite(untrusted_double)){
    // guards against 1/Double.MIN_VALUE
    // and 1/-Double.MIN_VALUE
}
```

```
if (Double.isNaN(untrusted_double)){
    // guards against division by 0.0
    // and infinities minus infinities
}
```

The space reserved to store an integer value is -2^15 to 2^15 -1. If you exceed the lower or upper bound, you'll wrap around the value space. This is known as a silent overflow. There are methods that guard against this by producing exceptions. In this example, `addExact` produces an `ArithmeticException` when attempting to add 1 to the highest possible integer value.

The result of a floating point operation may be too high or low to be represented by the memory space that backs a primitive floating point value. These values may be considered positive or negative infinities. Similarly, you may have issues dealing with operations that divide by zero, which are not a number (NaN). There are methods to check for these scenarios. These two examples show `boolean` methods for checking that a `floating point` is infinite or NaN.

Directory Traversal Attacks with `../`

- `../` requests the parent directory
- Your enemies may include `../` in paths.
  - They'll gradually learn your directory structure.
  - They want a path to sensitive information.
- Absolute paths may contain `../`.
- Canonicalize your path names before validation.
- Canonical paths are absolute, unique, and stripped of `../`
- Consider these methods:
  - `String getCanonicalPath()` `//java.io.File`
  - `File getCanonicalFile()` `//java.io.File`
  - `Path toRealPath(LinkOption... options)` `//java.nio.file`
  - `Path normalize()` `//java.nio.file`

```
File f = new File("../project");
String s = f.getCanonicalPath();
// "C:\\Windows\\project"
```

Both `getCanonicalPath` and `getCanonicalFile` are part of the original Java io API. `getCanonicalPath` returns a canonical path as a `String`. `getCanonicalFile` is similar but returns a `File` object instead of a `String`. These methods are called on File objects. The example calling `getCanonicalPath` shows how a file path containing `../` can be saved as a `String`. The resulting canonical path is absolute with `../` stripped out.

Both `toRealPath` and `normalize` are part of Java's NIO2 file API. `toRealPath` returns a `Path` for a file that must exist. An array of `LinkOptions` may be supplied that indicates how to handle symbolic links like `"../"`. `normalize` also returns a `Path`, but the file doesn't necessarily have to exist. These methods are called on instances that implement the `Path` interface. The actual behaviors of these methods may vary depending on how the interface is implemented.

SQL Injection Through Dynamic SQL

- Like Java `String`s, SQL statements can be assembled dynamically by concatenating a mix of hard-coded text, variables, and user input values.
- Matching single quotes (`'`) mark the end points of text.

```
String query = "SELECT * FROM Employees WHERE LastName ='" +userInput +"'";
```

- Using another `'`, an input field is exploited to inject malicious code into the statement.
- Assume the user enters this in an input dialog for the `userInput` String:

```
'; DROP DATABASE criticalDataBase; //
```

- The resulting `String query` is set to run two queries.
  - A meaningless query to find an employee with a blank last name.
  - A malicious query that deletes a critical database!

The intended behavior is for the user to input a value for a person's last name, which is used in the query. The query is stored as a Java String. It's formatted to include a pair of single quotes which are intended to format the user input into text within the query. An enemy may circumvent this setup by starting their input with a single quote. They're free to continue writing malicious queries which are eventually executed.

In this example, the user's input begins with a single quote and semicolon. This effectively creates a meaningless query to find an employee with a blank last name. This is followed by a malicious query that deletes a critical database. The user's input concludes with two slashes, which effectively comments out the remaining syntax from the original `String query`.

Safer SQL

- `PreparedStatement`s and substituted parameters shown in the JDBC lesson are a better approach.
- They ensure that parameter values are never interpreted as SQL.

```
String query = "SELECT * FROM Employee WHERE LastName = ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setString(1, userInput);
```

- In cases where `PreparedStatement`s do not work, use validation methods provided by JDBC. For example, `Statement.enquoteLiteral`.

```
StubStatement stmt = new StubStatement;
String lastName = stmt.enquoteLiteral("O'Kelly");
// lastName is: 'O''Kelly'
```

In the first example, the `query` is still stored as a `String`. Each `?` in `query` indicates a parameter to substitute. A `PreparedStatement pStmt` is created based on the query. The `Connection con` is defined elsewhere. `PreparedStatement setXXX` methods set values for each `?` and help validate input. The first argument indicates which `?` to set. The second argument indicates the replacement value.

The second example creates an instance of `StubStatement`, which is an example class that implements the `Statement` interface. The default method `enquoteLiteral` is part of the `Statement` interface. It returns a String enclosed in single quotes. Any occurrence of a single quote within the `String` is replaced by two single quotes. In this case, calling the method on the `String` argument `"O'Kelly"` returns `'O''Kelly'`.

XML Inclusion

- Many types of attacks exploit XML Document Type Definitions (DTDs), resulting in various degrees of harm:
  - Denial of Service
  - Exposed private files and data
- What's the purpose of a DTD?
  - Outlining how XML elements should be structure
  - Defining substitutable values known as entities

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE employee[
    <!ELEMENT employee (name, company)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT company (#PCDATA)>
    <!ENTITY sun "Sun Microsystems">
]>

<employee>
    <name>Kenny O'Kelly</name>
    <company>&sun;</company>
</employee>
```

```
Name: Kenny O'Kelly
Company: Sun Microsystems
```

At the top of this XML example file, `DOCTYPE` is the DTD that outlines the XML for representing an employee. Elements are used to outline an employee's `name` and `company`. An entity is used to define a substitute value of "Sun Microsystems". When the actual employee is created later in the file, `&sun;` is used to substitute in the value of the `sun` entity. When this is read into Java code, the resulting name is Kenny O'Kelly and company is Sun Microsystems.

This code is available in the `xml` package of the `SecurityTesting` project.

The Problem with XML Entities

- Entities may also be defined externally in other files and URLs.
- This leaves XML susceptible to XML External Entity (XXE) attacks.

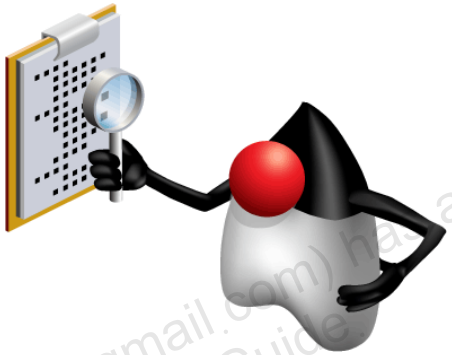
```
…
    <!ENTITY sun SYSTEM "http://[redacted].com/malicious.dtd">
…
```

  - A Billion Laughs DoS occurs when `malicious.dtd` recursively require more and more entities.
  - Private data may be exposed through any element referencing `&sun;` if `malicious.dtd` points to a sensitive file.
- If you must use XML, guard against attacks by:
  - Setting resource limits
  - Validating inputs
  - Reducing privileges by using the most restrictive XML parser configurations
  - Disabling external entities and DTDs all together

XXE attacks are similar to SQL injection in the sense that malicious code from the outside world is introduced into your application. The previous example has been changed. The value for `sun` is no longer defined internally. Instead, it's defined externally in a dtd file. This could potentially be from an untrusted or malicious source.

Failure to Verify Bytecode

- Verify bytecode against tampering and dangerous behavior.
- Many sources may generate or modify bytecode you need. Don't trust these sources.
- Beware of the command line arguments `-Xverify:none` or `-noverify`, which disable bytecode verification.

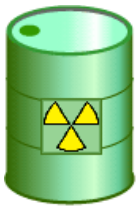


In a sufficiently complex system, Java bytecode is generated and modified all of the time by many different tools, products, and frameworks. Do you know how your Java EE application server compiles JSP files? Do any of the frameworks in your application dynamically generate new classes during runtime? How about profiling / instrumentation tools like Introscope or even the method profiling built into JRockit Mission Control? By disabling bytecode verification, you are trusting all these components in your entire stack are completely safe and bug-free. This is almost always a very bad idea. For more information, see: https://blogs.oracle.com/buck/never-disable-bytecode-verification-in-a-production-system

Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- Developing Secure Objects
- Secure Serialization and Deserialization

Isolate Unrelated Code



- Isolate and contain less-trustworthy code.
- Don't allow interference between unrelated code.
- Encapsulate with the most restrictive permissions possible.
- `public static` fields should be `final`.
  - Mutable statics are a vulnerability for code that directly or indirectly depends on their values.

```
public class ClassA {
    public static int year = 1974;
}
```


Encapsulate with the most restrictive permissions possible. Traditionally, classes and interfaces may be `public` if they're part of a published API. Otherwise, declare them package-private. Class members of an API may similarly be `public` or `protected` as appropriate. Otherwise, declare them `private` or package-private.

Stronger Encapsulation with Modules

- Restrict access further through modules and packages.
- In addition to `private`, package, and `protected`, there are 3 types of `public`.
- The `modules-info` files determine who may instantiate this example `public` class:

```
package pkgA;

public class ClassA {…}
```

1. `public` to everyone.

```
module A{
  exports pkgA;
}
```
```
module B{
  requires A
}
```
✓
```
module C{
  requires A
}
```
✓

2. `public` to specific modules.

```
module A{
  exports pkgA to B;
}
```
```
module B{
  requires A
}
```
✓
```
module C{
  requires A
}
```
✗

3. `public` only in its own module.

```
module A{
  //exports nothing
}
```
```
module B{
  requires A
}
```
✗
```
module C{
  requires A
}
```
✗

Use modules and packages as another means of restricting access as much as possible.

In this example, `ClassA` exists in `pkgA` package within module `A`. The class is public, but it's not necessarily accessible to other classes and modules. The same thing is true about the class's public fields and methods. It all depends on declarations made in `module-info` files. In scenario 1, the class is public to everyone because its package is exported from module `A`. Modules `B` and `C` require module `A`, meaning classes inside these modules may instantiate `ClassA`. In scenario 2, the class is public only to specific modules. In this case module `B`. Classes within Module `B` may instantiate `ClassA`, but classes within Module `C` may not. Scenario 3 has the most restrictive form of public access. `ClassA` is public only within Module `A`. Classes within Modules `B` and `C` may not instantiate `ClassA`.

This code is available in the `ModulesTest` project.

Stronger Encapsulation Against Reflection

With modules, private data is no longer accessible via reflection.

```
module A{
  exports pkgA;
}
```

```
module B{
  requires A
}
```

```
package pkgA;  //in Module A
public class ClassA {
    public static final int YEAR = 1974;
    private String secretCrush = "Kenny O'Kelly";
}
```

```
package pkgB;  //in Module B
public class NewMain {

    public static void main(String[] args) {
        ClassA test = new ClassA();
        try {
            Field secret =  Class.forName("pkgA.ClassA").getDeclaredField("secretCrush");
            secret.setAccessible(true);
            System.out.println(secret.get(test));
        }…
```

```
java.lang.reflect.InaccessibleObjectException: Unable to make field private java.lang.String
pkgA.ClassA.secretCrush accessible: module A does not "opens pkgA" to module B
```

`Private` data isn't accessible between modules. It used to be possible to access `private` data via reflection. Now with modules, `private` data is no longer accessible between modules or JARs via reflection. This example shows what happens when an attempt is made. `ClassA` in package `pkgA` exists within module `A`. It contains two fields, one meant to be `public` and one meant to be `private`. Although module `A` exports `pkgA` and module `B` requires module `A`, only `public` data is accessible from module `A`. The main class in `pkgB` of module `B` attempts to learn a `private secretCrush` from a `ClassA` instance via reflection. An approach like this may have worked in the past, but now produces an error. Keep this in mind as you produce and consume JARs for libraries.

Limit Extensibility

Non-`final` and non-`private` classes and methods can be maliciously overridden by an attacker.

- Hide methods as necessary.
- Design classes and methods for inheritance, or declare them `final` or `private`.
- Instantiate with a constructor after the potential object is verified safe.
- Don't call overridable methods from constructors.

```
// Unsubclassable class
public final class SensitiveClass {
    private final Behavior behavior;

    // Hidden constructor
    private SensitiveClass(Behavior behavior) {
        this.behavior = behavior;
    }

    // Guarded construction
    public static SensitiveClass newSensitiveClass
            (Behavior behavior) {
        // ... validate any arguments ...
        // ... perform security checks ...
        return new SensitiveClass(behavior);
    }
}
```

This code shows an example of a secure, unsubclassable class. The class is `final`, and therefore unsubclassable. Its fields and constructor are `private`. The only way to create an instance of this object type is to first call the static factory method `newSensitiveClass`. This method first validates any arguments and performs security checks before calling the constructor to actually instantiate the object. A safe and secure instance of a `SensitiveClass` is returned. Overridable methods shouldn't appear within this method or the constructor. Such methods may be maliciously overridden by an attacker to circumvent security checks, expose sensitive data, or perform other compromising actions.

This approach also guards against partially initialized objects, which are vulnerable to attack. When a constructor in a non-`final` class throws an exception, attackers can attempt to gain access to partially initialized instances. You should ensure a non-`final` class remains totally unusable until its constructor completes successfully. By guarding the constructor like in this example, we ensure only safe and secure instances are sent for construction.

Beware of Superclass Changes

- Superclass changes affect subclass behavior by:
  - Changing the implementation of an inherited method that's not overridden
  - Introducing new methods
- Changes may break subclass assumptions and lead to security vulnerabilities.

Class Hierarchy **Inherited Methods**

```
java.util.Hashtable                 put(key, val)
        ^                           remove(key)
        | extends                   entrySet()      // Added in JDK 1.2
        |                                           // Supports removal
java.util.Properties                                // Lacks security checks
        ^
        | extends
        |
java.security.Provider              put(key, val) // SecurityManager checks
                                    remove(key)   // SecurityManager checks
```

In this hierarchy, the `Provider` subclass inherits certain methods from `Hashtable`, including put and remove. `Provider.put` maps a cryptographic algorithm `name`, like RSA, to a class that implements that algorithm. To prevent malicious code from affecting its internal mappings, `Provider` overrides `put` and `remove` to enforce `SecurityManager` checks.

The `Hashtable` class was enhanced in JDK 1.2 to include a new method, `entrySet`, which supports the removal of entries from the `Hashtable`. The `Provider` class was not updated to override this new method. This oversight allowed an attacker to bypass the `SecurityManager` check enforced in `Provider.remove`, and to delete `Provider` mappings by simply invoking the `Hashtable.entrySet` method.

Problem: Vulnerable Object Fields

- Encapsulation may not be enough to protect fields from malicious manipulation.
- Keys to exploitation are:
  - Objects provided to a constructor
  - Objects returned from a getter method

```
public class Example {
   private final Date date;
   public Example(Date date) {
      this.date = date;
   }
   public Date getDate() {
      return date;
   }
}
```

```
...
Date badDate = new Date();
Example obj = new Example(badDate);

badDate.setYear(666);
obj.getDate().setMonth(6);
System.out.println(obj.getDate());
```

```
Tue Jul 01 18:48:08 EDT 2566
```

`Date` is an object field in the `Example` class. The field is set with a constructor. Although the field is private and has a traditional getter method, this isn't enough to prevent unwanted manipulation, which effectively negates the security benefits of encapsulation.

The code on the right creates a `Date` object `badDate` and supplies it to the constructor. Both this reference, and the `Example` class's `date` field point to the same location in memory. This means the `badDate` reference has access to freely manipulate a private field in another class! The getter method provides the same dangerous level of access because it returns the `date` object. You can see from the output how both means are effective.

This code is available in the `date` package of the `SecurityTesting` project.

Solution: Create Copies of Mutable or Subclassable Input Values

- Most methods like constructors and setters should first copy any objects they receive.
- Getters should return copies. There are a few ways to do this, including:
  - Calling a constructor
  - Calling the `clone` method, if the object type is trustable or `final`

```
public class Example {
   private final Date date;
   public Example(Date date) {
      this.date = new Date(date.getTime());
   }
   public Date getDate() {
      return (Date)date.clone();
   }
}
```

```
...
Date badDate = new Date();
Example obj = new Example(badDate);

badDate.setYear(666);
obj.getDate().setMonth(6);
System.out.println(obj.getDate());
```

```
Mon Apr 01 18:51:08 EDT 2019
```

The `Example` class now implements several protective measures. The constructor copies the object it receives before setting the value of the `date` field. The object supplied by the outside world will not be the same as the field value. The getter also supplies a copy. The code shows two ways to return copies. One is to create a new object by calling a constructor. The other is to clone the original object. This method returns an object that must be cast.

The code on the right remains is unchanged. Attempts to manipulate the private `date` field are now unsuccessful.

Using the `clone` method may not be wise if you're copying a non-`final` object type. A non-`final` class may be extended by a malicious class that also implements `java.lang.Cloneable`. Consequentially, the base class object reference may point to an instance of a mischievous subclass clone.

File Security Bug Reports

- Reports let others know what to beware of and plan accordingly until a fix is ready.
- Customers may not appreciate you keeping secrets when they're at risk.
- It's dangerous to mess with code you don't understand.
- Reports help the right people fix issues and anticipate/prevent similar issues in the future.

Topics

- Denial of Service
- Confidential Information
- Integrity of Inputs
- Developing Secure Objects
- Secure Serialization and Deserialization

Serialization and Deserialization

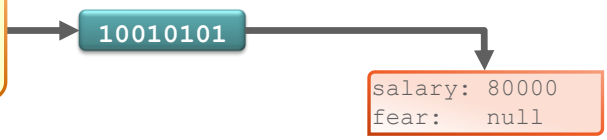
- Serialization converts an object's state into a byte stream.
  - Instance fields
  - Not static fields
  - Not methods
- Deserialization converts the byte stream into a copy of the object.
- Serialization is useful for storing state in a database or transferring state over a network.
- The class or its superclass must implement the `Serializable` interface.

Problem: Fields Are Accessible After Serialization

Solution 1: Hide sensitive fields from serialization by declaring them `transient`.

```
public class Employee implements Serializable {
   private int salary;
   private transient String fear;
…
```

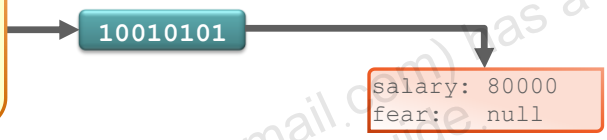
```
salary: 80000
fear:   Spiders
```



Solution 2: Declare a special array field `serialPersistentFields`, which specifies each field to serialize.

```
public class Employee implements Serializable {
   private int salary;
   private String fear;
   private static final ObjectStreamField[]
         serialPersistentFields = {
         new ObjectStreamField("salary", int.class)};
…
```

```
salary: 80000
fear:   Spiders
```



You should avoid serializing security-sensitive classes, and carefully guard sensitive information within classes you do serialize. Java's access controls cannot be enforced after serialization. In effect, serialization provides a public interface to fields. An attacker may analyze the byte stream to learn sensitive data.

Guard against this by declaring sensitive fields `transient`. A `transient` field is not serialized. This is illustrated by the first example. An employee's deepest fear is considered sensitive information, and is marked transient to prevent serialization. When the example object is serialized and then deserialized, the resulting copy knows nothing of the original object's fears. The `fear` field is `null`.

Alternatively, you may declare a special array field `serialPersistentFields`, which specifies each field you want to serialize. Solution 2 produces the same results. This solution overrides any `transient` declarations. Even if the second example were edited to declare the `salary` field `transient`, the resulting object after deserialization still has the same salary as the original object.

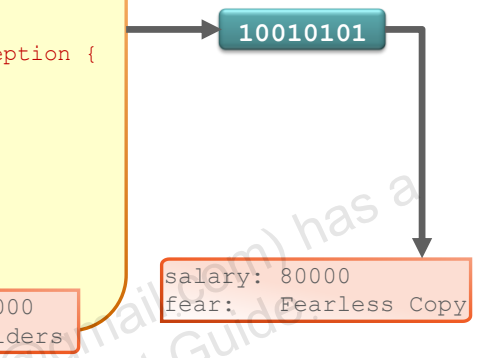
These solutions are available in the `serialize` package of the `SecurityTesting` project.

Solution 3a: Implement `writeObject` and `readObject` Methods

- `writeObject` specifies data serialized to the byte stream.
- `readObject` deserializes from the byte stream and specifies how copies are created.
- The order of statements within these methods must align.

```
public class Employee implements Serializable {
   private int salary;
   private String fear;
…
   private void writeObject(ObjectOutputStream out) throws IOException {
        out.writeObject(this.salary);
        out.writeObject("Fearless Copy");
    }

   private void readObject(ObjectInputStream in) throws
                IOException,ClassNotFoundException {
        this.salary = (int)in.readObject();
        this.fear = (String)in.readObject()
    }}
```



The `writeObject` method specifies what data to serialize to the byte stream. You're not limited to serializing just class fields. You can serialize whatever data you want from this method. In this example, the `Employee`'s `fear` still must be kept secret. Instead of adding the actual vulnerable value to the byte stream, a replacement `String` is serialized.

The `readObject` method deserializes the byte stream and specifies how copies are made. The state of the copies are derived from the byte stream in this example. But this isn't necessarily required. For example, instead of serializing `"Fearless Copy"` to the byte stream, we could achieve the same effect by letting the `readObject` method set `this.fear` to `"Fearless Copy"`.

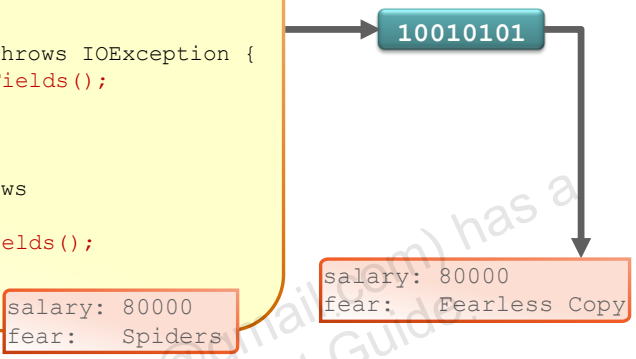
The order that data is serialized and deserialized matters greatly. If the order of statements in readObject were reversed in this example, a ClassCastException is thrown due to the mismatch in data types between `int` and `String`.

This solution also circumvents any `transient` declarations. Even if the `salary` field were declared `transient`, the resulting object after deserialization still has the same salary as the original object.

Solution 3b: Implement `writeObject` with `PutField`, and `readObject` with `GetField`

- Use `ObjectOutputStream.PutField` to selectively serialize data.
- The order of statements is less important.
- This solution is brittle. Refactored fields may mismatch `String`s in these methods.

```
public class Employee implements Serializable {
   private int salary;
   private String fear;
…
   private void writeObject(ObjectOutputStream out) throws IOException {
        ObjectOutputStream.PutField fields = out.putFields();
        fields.put("salary", this.salary);
        fields.put("fear", "Fearless Copy");
        out.writeFields();     }
   private void readObject(ObjectInputStream in) throws
                IOException,ClassNotFoundException {
        ObjectInputStream.GetField fields = in.readFields();
        this.fear = (String)fields.get("fear", null);
        this.salary = fields.get("salary", 0);     }}
```



The `writeObject` method still specifies what data to serialize to the byte stream. Now individual fields are specified by name, along with their intended values. An instance of `ObjectOutputStream.PutField` is created. Data is serialized by specifying key-value pairs with the instance's `put` method. Finally, `writeFields` is called on the output stream.

The `readObject` deserializes the byte stream to set the value of a copy's fields. An instance of `ObjectInputStream.GetField` is first created. The order of statements afterwards is less important, because data is retrieved based on key-value pairs from the `get` method. The `get` method also specifies a default value. This method returns primitives and `Objects`, which means an `Object` may need to be cast before it's set as a copy's field value.

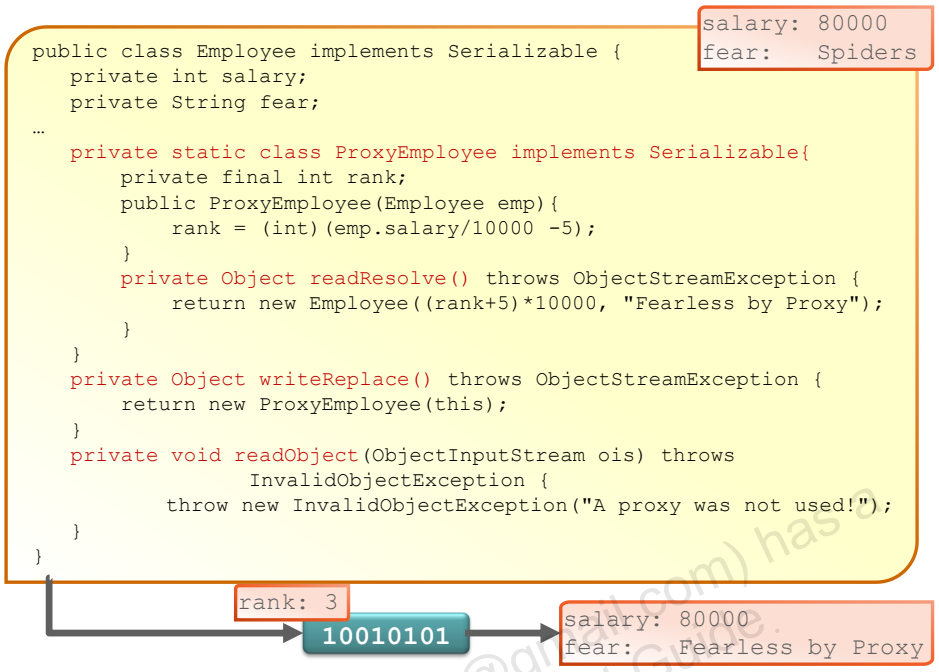
Although the order that data is serialized and deserialized is less important when fields are handled by name, the solution is more-brittle. `String` names used in the `put` and `get` methods must match the class's field names. If a field is refactored, the corresponding `String` name might not be updated. This results in an `IllegalArgumentException`.

This solution does not circumvent `transient` declarations. Fields aren't recognized if they're declared `transient`. For example, if the `fear` field were declared `transient`, the code fails to compile.

Solution 4: Implement a Serialization Proxy Pattern

- Don't expose your object to serialization.
- Instead, serialize a proxy object type with `writeReplace`.
- The proxy is written as an inner class.
- Deserialize the proxy back to a copy of the original object type with `readResolve`.
- Conclude with another `readResolve` for safety.

```
public class Employee implements Serializable {
   private int salary;
   private String fear;
…
   private static class ProxyEmployee implements Serializable{
       private final int rank;
       public ProxyEmployee(Employee emp){
           rank = (int)(emp.salary/10000 -5);
       }
       private Object readResolve() throws ObjectStreamException {
           return new Employee((rank+5)*10000, "Fearless by Proxy");
       }
   }
   private Object writeReplace() throws ObjectStreamException {
       return new ProxyEmployee(this);
   }
   private void readObject(ObjectInputStream ois) throws
                 InvalidObjectException {
          throw new InvalidObjectException("A proxy was not used!");
   }
}
```



The `writeReplace` method is written in the `Employee` class. It returns a proxy as an Object. The proxy object is a substitute for the real object type. It's written as a `private` inner class that should also implement `Serializable`. Its fields are `private final`. Its constructor requires an instance of the outer object as a parameter to setup a proxy instance.

If the byte stream is analyzed by an enemy, they can only see rank. They won't have any clues into the company's salaries, how rank equates to salary, or know sensitive information like employee fears are kept on file.

The inner class's `readReslolve` method translates the proxy back into the original object type. It creates a copy instance. The `void readObject` method in the outer class is a safeguard. It ensures a proxy was actually used. Otherwise, the `Employee` fields may be vulnerable to serialization.

Deserialize Cautiously

- Deserialization is a form of object construction.
- It effectively creates a public constructor that may sidestep security checks.
- Deserialization of untrusted data is dangerous.
- Like other means of setting state, you should first validate values.

Summary

In this lesson, you should have learned how to:

- Identify potential Denial of Service vulnerabilities
- Secure confidential information
- Support data integrity
- Explain reasons for input validation
- Limit accessibility and extensibility of sensitive objects
- Consider security measures for serialization

For more information on the Java secure coding guideline, see:
https://www.oracle.com/technetwork/java/seccodeguide-139067.html