Java Secure Coding

Denial of Service (DoS)

Input Validation

Mutability

Variable Scope

Thread Safety

Exception Handling

Role-Based Authentication

What We Will Cover

- Denial of Service (DoS)
- Input Validation
- Mutability
- Variable Scopes
- Thread Safety
- Exception Handling
- Role-Based Authentication

Denial of Service (DoS)

Denial of Service (DoS)

Input Validation

Mutability

Variable Scope

Thread Safety

Exception Handling

Role-Based Authentication

DoS Resources Attacked

- Affects Resources
  - CPU
  - Memory
  - Disk Space
  - etc.

DoS Examples

- Requesting images with large size
- Failure of Sanity checking of sizes by integer overflow errors
- Memory allocation to an object graph much more than usual
- Zip bombs: Huge decompressed file from a tiny zip file
- Billion laughs attack: Growing XML documents dramatically during parsing

DoS Examples, cont'd

- Increasing executing cost for example from `O(n)` to `O(n^2)`
- Exhibiting catastrophic backtracking by regular expression
- Processor time may be consumed by XPath expressions
- Infinite loops
- and many more ...

(DoS): Release Resources

- A good sample pattern to extracting the paired acquire and release operations in Java SE 8:

```
long sum = readFileBuffered(InputStream in -> {
            long current = 0;
            for (;;) {
                int b = in.read();
                if (b == -1) {
                    return current;
                }
                current += b;
            }
        });
```

DoS: Integer Overflow

- The following piece of code avoids overflow:

```
1 private void checkGrowBy(long extra) {
2             if (extra < 0 || current > max - extra) {
3                 throw new IllegalArgumentException();
4             }
5         }
```

Input Validation

Denial of Service (DoS)

Input Validation

Mutability

Variable Scope

Thread Safety

Exception Handling

Role-Based Authentication

Validate All Inputs

- Inpute Sources:
  - Method Arguments
  - External Streams
- If from untrusted sources, validate it

Validate Output From Objects As Input

- Example: `Class` object returned by `ClassLoaders`
- ClassLoader instances which
  - get passed as arguments
  - or are set in `Thread` context can be controled by attacker
- So, don't make many assumptions when calling this method

Define Wrappers Around Native Methods

- Java code unlike native methods is secure, so :
  - Do not make a native method public
  - Instead expose the functionality through a public java-based wrapper method
  - Example: Next slide

Example of Wrapper

```
1  public final class NativeMethWrap {
2
3          // private native method
4          private native void nativeOperation(byte[] data, int offset,
5                                              int len);
6
7          // wrapper method performs checks
8          public void doOperation(byte[] data, int offset, int len) {
9              // copy mutable input
10             data = data.clone();
11
12             // validate input
13             // Note offset+len would be subject to integer overflow.
14             // For instance if offset = 1 and len = Integer.MAX_VALUE,
15             //   then offset+len == Integer.MIN_VALUE which is lower
16             //   than data.length.
17             // Further,
18             //   loops of the form
19             //       for (int i=offset; i<offset+len; ++i) { ... }
20             //   would not throw an exception or cause native code to
21             //   crash.
22
23             if (offset < 0 || len < 0 || offset > data.length - len) {
24                 throw new IllegalArgumentException();
25             }
26
27             nativeOperation(data, offset, len);
28         }
29     }
```

Quiz

- What is DoS?
  - (in the context of security)
- What areas of Java SE can be attacked?
- What are the defences against DoS?

Lab

- Input validation:
  - Overview: A Java code to set Boiler temperature using JNI implementation
  - Requirements: Linux, JDK 64bit, gcc compiler 64-bit
  - Approximate time: 60 minutes
  - Instructions: labs/java-security-validation/labs/input_validation.md
  - https://github.com/elephantscale/secure-coding-labs/blob/main/java_security_validation/labs/input_validation.md

Mutability

Denial of Service (DoS)

Input Validation

Mutability

Variable Scope

Thread Safety

Exception Handling

Role-Based Authentication

Introduction

- Mutable object: it is possible to change the state and fields after creation
- Immutable object: you cannot change anything after creation
- Most classes are created as mutable
- Mutable classes may result in a variety of security issues

How to Create Mutable and Immutable

```
1  class MutableClass{
2    private string value;
3
4    public MutableClass(string value) {
5        this.value = "SomeString";
6    }
7
8    //getter and setter for value
9  }
10
11 class ImmutableClass {
12   private final string value;
13
14   public ImmutableClass(string value) {
15       this.value = "SomeString";
16   }
17
18   //only getter
19 }
```

Immutability In Value Types

- Should not be subclassable
- Hiding constructors helps more flexibility in creation of instances
- Is a protection against mutable inputs and outputs

Copy Mutable Output Values

- Example of copying a trusted mutable object:

```
1 public class OutputCopy {
2             private final java.util.Date date;
3             ...
4             public java.util.Date getDate() {
5                 return (java.util.Date)date.clone();
6             }
7         }
```

Copies of Mutable Classes

- Mutable objects can be changed during and after execution of the method or constructor call
- Types which are subclassed can behave incorrectly
- Following example creates a copy of mutable object, calls a copy constructor

Copy of Mutable as Subclass

```
1  public final class CopyMutableInput {
2              private final Date date;
3
4              // java.util.Date is mutable
5              public CopyMutableInput(Date date) {
6                  // create copy
7                  this.date = new Date(date.getTime());
8              }
9          }
```

Support Copy Functionality

- Let it be possible creating safe copy
- Static creation method, a copy constructor and public copy method for final classes may help in this regard
- Do not use `java.lang.Cloneable` mechanism

Overridable Identity Equality

- Overridable methods sometimes behave strange
- You may get `True` value from two different objects by `Object.equal`
- Example: When using key in a `Map`, an object may be able to pass itself off as a different object that it should not have access to
- Solution: If possible collection implementation that enforces identity equality like `IdentityHashMap`

Collection Implementation Example

```
1  private final Map<Window,Extra> extras = new IdentityHashMap<>();
2
3          public void op(Window window) {
4              // Window.equals may be overridden,
5              // but safe as we are using IdentityHashMap
6              Extra extra = extras.get(window);
7          }
```

Package Private Key

- If such a collection is not available: Package private key helps

```
1  public class Window {
2              /* pp */ class PrivateKey {
3                  // Optionally, refer to real object.
4                  /* pp */ Window getWindow() {
5                      return Window.this;
6                  }
7              }
8              /* pp */ final PrivateKey privateKey = new PrivateKey();
9
10             private final Map<Window.PrivateKey,Extra> extras =
11                                                 new WeakHashMap<>();
12             ...
13         }
14
15         public class WindowOps {
16             public void op(Window window) {
17                 // Window.equals may be overridden,
18                 // but safe as we don't use it.
19                 Extra extra = extras.get(window.privateKey);
20                 ...
21             }
22         }
```

Input to Untrusted Object As Output

- Previous instructions on output objects are applicable when passed to untrusted objects
- Apply proper copying

```
1 private final byte[] data;
2
3         public void writeTo(OutputStream out) throws IOException {
4             // Copy (clone) private mutable data before sending.
5             out.write(data.clone());
6         }
```

Output From Untrusted Objects As Input

- Previous instructions on input objects are applicable when returned from untrusted objects
- Apply proper copying and validation

```
1  private final Date start;
2          private Date end;
3
4          public void endWith(Event event) throws IOException {
5              Date end = new Date(event.getDate().getTime());
6              if (end.before(start)) {
7                  throw new IllegalArgumentException("...");
8              }
9              this.end = end;
10         }
```

Wrapper Methods

- If you need public access to a internal state of calss declaring a private field and enabling access via public wrapper method would help
- If you need access from sublclsses declaring a private field and enabling access via protected wrapper method is a good idea
- Wrapper methods enable us validate input before setting a new value

Example of Wrapped State

```
1  public final class WrappedState {
2              // private immutable object
3              private String state;
4
5              // wrapper method
6              public String getState() {
7                  return state;
8              }
9
10             // wrapper method
11             public void setState(final String newState) {
12                 this.state = requireValidation(newState);
13             }
14
15             private static String requireValidation(final String state) {
16                 if (...) {
17                     throw new IllegalArgumentException("...");
18                 }
19                 return state;
20             }
21         }
```

Making Public Static Fields Final

```
1 public class Files {
2               public static final String separator = "/";
3               public static final String pathSeparator = ":";
4         }
```

Public Static Final Field Values

- Only immutable and unmodifiable values should be stored in public static fields
- Following example shows prevention the list from being modified

```
1 import static java.util.Arrays.asList;
2         import static java.util.Collections.unmodifiableList;
3         ...
4         public static final List<String> names = unmodifiableList(asList(
5             "Fred", "Jim", "Sheila"
6         ));
```

Variable Scope

Denial of Service (DoS)
Input Validation
Mutability
Variable Scope
Thread Safety
Exception Handling
Role-Based Authentication

Class Level Scope

- Also called member variables
- Are declared inside the class but outside any function
- Can be access outside the class with these rules:

Modifier	Package	Subclass	World
public	Yes	Yes	Yes
protected	Yes	Yes	No
Default (no modifier)	Yes	No	No
private	No	No	No

Class Level Declaration

```
1  public class ClassLevelVar
2  {
3      int num;
4      private String address
5      void method1() {
6          }
7      int method2() {
8          }
9      char a;
10 }
```

Method Level Scope

- Also called local variables
- Declared inside the method
- Only accessible inside the method
- When execution of the method finishes, they disapear

Method Level Declaration

```
1  public class MethodLevelVar
2  {
3      void method()
4      {
5          // Local variable
6          int x;
7      }
8  }
```

Block Scope

- Also called Loop variables
- Are declared inside brackets { }
- Are valid only inside the bracket
- Example:

```
1  public class BlockScope
2  {
3      public static void main(String args[])
4      {
5          {
6              int temp = 10;
7              System.out.println(temp);
8          }
9
10
11         System.out.println(temp); // you will get an error
12     }
13 }
```

Method Scopes

- The discussion of scopes for methods follows the same guidelines as for variables

Class Scopes

- Public and package access
- Private class dosn't make sense
- Default modifier is package access

Thread Safety

Denial of Service (DoS)

Input Validation

Mutability

Variable Scope

Thread Safety

Exception Handling

Role-Based Authentication

Introduction

- Many techniques provide secure threading some of them are:
  - No state
  - No shared state (one of the best ways)
  - Message passing
  - Immutable state
  - Synchronized blocks
  - Volatile fields

No State

- an instance or static variable may be used by multiple threads
- Avoid instance or static variables
- Example: (part of `class.lang.Math`)

```
1  public static int subtractExact(int x, int y) {
2
3      int a = b - c;
4
5      if (((b ^ c) & (b ^ a)) < 0) {
6
7          throw new ArithmeticException("integer overflow");
8
9      }
10
11     return a;
12
13 }
```

No Shared State

- You cannot avoid state? At least don't share it
- One way is extending the thread class and adding an instance variable
- Pool and workQueue are local to a single worker thread in the example
- Example:

```
1  package java.util.concurrent;
2
3  public class ForkJoinWorkerThread extends Thread {
4
5      final ForkJoinPool pool;
6
7      final ForkJoinPool.WorkQueue workQueue;
8
9  }
```

Message Passing

- You don't want to share the state? Let the threads communicate
- How? pass messages between them
- An example of sending a message with `Akka` framework:

```
1 target.tell(message, getSelf());
```

- And receive a message:

```
1  @Override
2
3  public Receive createRcv() {
4
5      return rcvBuilder()
6
7          .match(String.class, s -> System.out.println(s.toLowerCase()))
8
9          .build();
10
11 }
```

Immutable State

- If you don't want the message to be changed by another thread make it immutable
- When implementing an immutable class, declare its fields as final
- Example:

```
1  public class aFinalField
2
3  {
4
5      private final int finalField;
6
7      public aFinalField(int value)
8
9      {
10
11         this.finalField = value;
12
13     }
14
15 }
```

Synchronized Blocks

- Put a lock inside a sync block
- To be sure two threads won't execute this section simultaneously

```
1 synchronized(lock)
2 {
3
4     counter++;
5
6 }
```

Volatile Fields

- By declaration of a variable you tell the JVM and the compiler to return the latest written value

```
1  public class aVolatileField
2
3  {
4
5      private volatile int  volatileField;
6
7  }
```

Lab

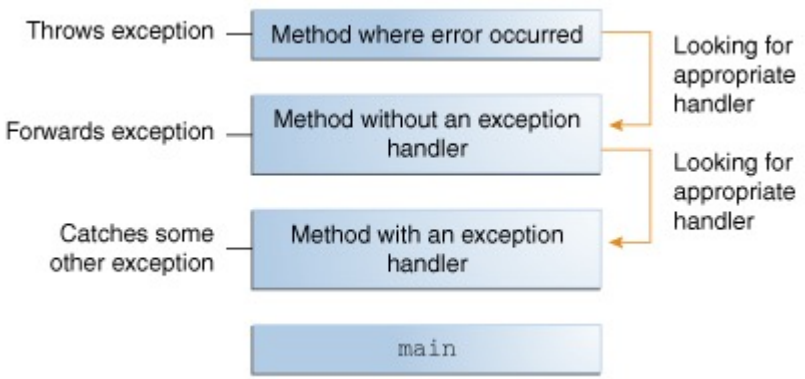
- Thread safety:
  - Overview: A Railway Ticket Booking System for explaining the Thread-safe and Exception Handling.
  - Requirements: Linux, JDK 1.8 and Apache Maven 3.5.4
  - Approximate time: 60 minutes
  - Instructions: labs/thread-safety-labs/labinfo/thread_safe.md & thread_unsafe.md
  - https://github.com/elephantscale/secure-coding-labs/tree/main/thread-safety-labs/labinfo

Exception Handling

Denial of Service (DoS)
Input Validation
Mutability
Variable Scope
Thread Safety
Exception Handling
Role-Based Authentication

What Is An Exception?

- Exception = exceptional event
- A disruption during normal flow of program's instruction
- An object is created containing information about the error



Kinds of Exception?

- Checked exception; subject to exception handling
- Error
- Runtime exception
- An exception must be enclosed by one of these:
- `try` statement
- `throws` clause

How to Throw an Exception?

- It's done by `throw` statement:

```
1  public Object pop() {
2      Object obj;
3
4      if (num == 0) {
5          throw new EmptyStackException();
6      }
7
8      obj = objectAt(num - 1);
9      setObjectAt(num - 1, null);
10     num--;
11     return obj;
12 }
```

The try-with-resources Statement

- Ensures that the resource will be closed at the end
- Example:

```
1  public static void viewTable(Connection conn) throws SQLException {
2
3      String query = "select COFFEE_NAME, SUPPLIER_ID, PRICE, SALES, TOTAL from COFFEES";
4
5      try (Statement stmt = conn.createStatement()) {
6          ResultSet rs = stmt.executeQuery(query);
7
8          while (rs.next()) {
9              String coffeeName = rs.getString("COFFEE_NAME");
10             int supplierID = rs.getInt("SUPPLIER_ID");
11             float price = rs.getFloat("PRICE");
12             int sales = rs.getInt("SALES");
13             int total = rs.getInt("TOTAL");
14
15             System.out.println(coffeeName + ", " + supplierID + ", " +
16                                price + ", " + sales + ", " + total);
17         }
18     } catch (SQLException e) {
19         JDBCTutorialUtilities.printSQLException(e);
20     }
21 }
```

Advantages of Exceptions

- Separating error-handling code from regular code
- Propagating errors up the call stack
- Grouping and differentiating error types

Role-Based Authentication

Denial of Service (DoS)
Input Validation
Mutability
Variable Scope
Thread Safety
Exception Handling
Role-Based Authentication

Introduction

- Allows user to authenticate to a role
- For every instance of authentication specify the following attributes:
- Conflict resolution level
- Autentication configuration
- Login success URL
- Authentication post processing classes

Login URLs

- Can be specified in The User Interface Login URL.
- Calls the role authentication module
- Redirection :
- Upon successful or failed login, `Access Manager` redirects the user to the right page