Lecture 7: Exception handling and persistence for security

**Lecture topics**
- Exception handling for security
- Persistence format considerations and security
- Questions about the design of bookauction

**Exceptions in Java**
- An exception is an object that signifies that normal execution of the program has been interrupted in some way
  - Something forbidden happened
  - E.g. reference to a member of a null object
  - Some programmer-specified conditions violated
  - E.g., a parameter to a method is out of bounds
- Exceptions are thrown
  - Explicitly, by throw statements
  - Implicitly, by the run-time system
- Exceptions are caught
  - The catch portion of try-catch-finally blocks
  - If there is a chance that an exception can be thrown in a method, either
    - The method must declare that this exception type (or its supertype) can be thrown or
    - The place where the exception may be thrown must appear in a try block

**Why should I use exceptions? Couldn’t I just return status flag from methods**
- It is common in C programming to return a flag value (1 for success, 0 for failure of the operation)
- Exceptions have a number of advantages over the return flag method:
  - Error handling code is logically separated from the regular code
    - This often results in a better formed code, without a bunch of tangled if statements
  - Exceptions are propagated up the call stack automatically
    - To achieve the same functionality manually, a lot of extra work has to be done
  - Specialized error types can be introduced and grouped by inheritance

**How JVM handles exceptions**
- After an exception is thrown, it is propagated down the call stack until a method is found that can handle it

**What if no method on the call stack can handle an exception?**
- There are two distinct cases, based on two types of exceptions Java has
  - **Checked exceptions**
    - Subclasses of class Exception
      - Must be caught or declared as thrown (even by the main method)
      - Usually represent recoverable errors
        - If a file cannot be found, the application must not crash
  - **Unchecked exceptions**
    - Subclasses of Error and RuntimeException
      - Must not be caught or declared
      - Usually represent irrecoverable errors
      - If a null pointer dereference occurs, in general recovery is impossible

**Information carried by an exception**
- The type of an exception
  - Good design practice to use different exception classes for different kinds of exceptional situations
- Stack trace --- the state of the call stack at the moment the exception was thrown
  - Chain of methods and their "active" instructions
  - The printStackTrace method of Exception
- Any additional information
  - E.g. a verbal explanatory message
  - The getLocalizedMessage method of Exception
- Security implication: can an attacker use exception type and stack trace information to gain insight in the part of the application code not accessible to the attacker directly?
  - E.g. code in an EJB
Lecture 7: Exception handling and persistence for security

Exceptions in distributed programs
- Exceptions are used to represent all problems that may happen during remote calls
  - Each method in EJB interfaces declares
    - java.rmi.RemoteException
      - Checked exception
    - This exception is thrown to indicate problems in the network
      - E.g., connection times out
    - Also wraps any exceptions that EJB methods throw
      - If an EJB method throws NullPointerException, the client sees RuntimeException that contains the NullPointerException

Rules for secure class design: validity checks
- Many methods have restrictions on validity of parameters
  - E.g., object references often must not be null
- Many intermediate results can be checked for validity (sanity checks)
- If no validity checks are present, bad things can happen
  - Execution of a method may fail unexpectedly
  - A method may terminate without failure, but cause failure at some later point in the execution
- Use exceptions to implement validity checks
  - Often IllegalArgumentException, IndexOutOfBoundsException, and NullPointerException

Example
```java
interface Homework {
    Iterator parts();
    Student getStudent();
}

public void submitHomework(Homework homework) {
    Grade grade = new Grade(100);
    this.grades.put(homework.getStudent(), grade);
    for (Iterator iter = homework.parts(); iter.hasNext();)
        grade.subtract(pointsOff = this.check(part);
    }
}

```

The finally part of the try-catch-finally block
- The finally part of the try-catch-finally block is guaranteed to be executed, whether or not an exception was thrown inside the try clause
  - Executed after code in the try and catch clauses, but before any exception is thrown that would cause the catch clause to terminate
  - See Finally.java
- What happens if an exception is thrown from the finally clause?
  - If an exception is supposed to be thrown from the catch clause, it is ignored
  - Usually a source of errors
  - See FinallyMissing.java
- Avoid throwing exception from finally or at least avoid this when an exception can be thrown from catch

```
class GradingSystem extends UnicastRemoteObject {
    private Map grades;
    public void submitHomework(Homework homework) {
        Grade grade = new Grade(100);
        this.grades.put(homework.getStudent(), grade);
        try {
            for (Iterator iter = homework.parts(); iter.hasNext();)
                grade.subtract(pointsOff = this.check(part);
        } catch (Exception e) {
            this.grades.remove(homework.getStudent());
        }
    }
}
```
Lecture 7: Exception handling and persistence for security

Persistence in Java: serialization

- Serialization is an easy way to save object state persistently
  - Saves values of all fields of the object
  - For reference fields, performs serialization recursively
- ObjectOutputStream and ObjectInputStream are stream classes responsible for handling objects
  - E.g.
    ```java
    ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("c:\output.txt"));
    Student student = new Student("Joe", 234);
    out.writeObject(student);
    out.close();
    ```

What do I need to do to serialize objects of a specific class?

- Make sure the class implements the Serializable interface
  - It’s a mixin interface; does not have any methods, just marks the class as being allowed to use the serialization mechanism
  - Side effect: all subclasses of a serializable class are automatically serializable
- Make sure that all fields of the class are either
  - Primitive types
  - Serializable reference types
    - Lots of standard classes are serializable (String, Date, …)
  - Marked as transient
  - Their values are not written during serialization and not restored during de-serialization
- Very important for security reasons

Customizing the serialization process

- Create private writeObject and readObject methods in the class whose objects are serialized
  - These methods describe how the object is serialized/de-serialized
    - It’s a good idea to call defaultWriteObject of ObjectOutputStream from writeObject and defaultReadObject of ObjectInputStream from readObject
    - Serialize all non-transient fields
    - See Student.java
- Wait, but if writeObject and readObject are private, how can they be called outside of the class where they are defined?
  - Implementation of object streams is native (in C or C++), so it is not subject to visibility rules

Using transient to prevent serialization of fields

- Fields marked with modifier transient are not serialized
  - Always mark as transient fields that contain
    - Handles to system resources
    - Information relative to address space
  - Make sure to initialize transient fields when de-serializing
  - Otherwise, have default values
    - Serialization does not create objects by calling constructors
  - See TransientFields.java

Externalization

- A type of serialization where
  - The programmer has more control over format
  - The programmer needs to do more work
  - Generally, amount of serialized data is smaller than for serialization
    - Does not save information about superclasses, types of fields, etc.
- The programmer has to implement writeExternal and readExternal methods

What is the most dangerous attack of an application using serialization?

- Tampering with serialized form of an object
  - The idea is to fool the program into
    - Accepting modified values of the object fields
    - Not noticing that values of the object fields changed
    - Working with an object in an illegal state
- Solution?
  - Use encrypted communication channels
  - Always implement readObject, even if default serialization mechanism is used
  - Check validity of fields and relationships among them as the object is re-constructed in readObject
    - E.g. check that all bids on an auction are ordered from lowest to highest
Example attack

Your code:
```java
public class YourClass implements Serializable {
    private byte [] internalArray;
    ...
    private synchronized void writeObject(ObjectOutputStream stream) {
        stream.write(internalArray);
    }
    ...
}
```

Attacker code:
```java
public class AttackerObjectOutputStream extends ObjectOutputStream {
    public void write(byte [] b) {
        Modify b
    }
    ...
}
```

YourClass yc = new YourClass();
```java
AttackerObjectOutputStream aoos = new AttackerObjectOutputStream();
aoos.writeObject(yc);
```

An interesting special case of object mutability problem

Is there a way to make classes non-serializable?

- Why would we want to?
  - A class may contain security-sensitive fields
    - Want to save these fields locally, so don’t want to make them transient
  - Not making a class serializable forces the code that uses objects of this class to deal with serialization
- So, we can just not implement Serializable!
- What if a superclass implements Serializable
- Also, may want to prevent any future subclasses from being serializable
- OK then, just override methods writeObject and readObject, in the following way:
  ```java
  private void writeObject(ObjectOutputStream out) throws IOException {
      throw new NotSerializableException("Objects of this class are not serializable!");
  }
  ...
  ```

Native code

- Java classes can contain native methods
  - Methods that are implemented in C or C++
- None of the Java security checking applies to these
  - No bytecode verification
  - No run-time bounds checking
- If possible, native methods of untrusted code should be examined and analyze for:
  - What they return
  - What they take as parameters
  - Whether they bypass security checks
  - Whether they are public, private, ...
  - Whether they contain method calls which bypass package-boundaries, thus bypassing package protection