SWE 619 Assignment 12

// TODO

Goal: Applying lessons learned.

Write a brief report, and include enough evidence (output, screen shots, etc.) that the GTA can figure out that you actually completed the assignment.

1. For most of the semester, we have focused on design considerations for constructing software that does something we want it to do. For this last assignment, I would like students to appreciate just how vulnerable software is to malicious parties intent on attacking their software. Students who find this assignment amusing might wish to take CS 468: Secure Programming and Systems.

   There are two attacks documented in Bloch's Item 88: *Write readObject() methods defensively.* One is called BogusPeriod, and the other is called MutablePeriod. Implement either (your choice) of these attacks (basically involves typing in code from Bloch) and verify that the attack takes place.

I choose to demonstrate the MutablePeriod attack. For the demonstration, we need to create a few classes first. Hence, I first created a **new gradle application** as follows:

- Open command prompt
- `cd` to the directory where you want to initialise your project
- Type `gradle init`
- Follow the onscreen instructions and Gradle should create a workspace for you

```
> Task :init
Get more help with your project: https://docs.gradle.org/5.6.2/userguide/tutorial_java_projects.html

BUILD SUCCESSFUL in 1m 17s
2 actionable tasks: 2 executed
C:\Users\haram\OneDrive\Desktop\SWE 619 Assignment 5>
```

Now, we have our project ready to roll!

LET'S GET STARTED!!!

1. Coding up the Period class:

Bloch's Item 50 contains an immutable Period class with mutable private Date fields. The class preserves its invariants and immutability by using defensive-copy for Date objects in its constructor and accessors. The Period class is as follows:

Period.java:

```
package Assignment;

import java.io.Serializable;
import java.util.Date;



// Immutable class that uses defensive copying
public final class Period implements Serializable{
    /**
     *
     */
    private static final long serialVersionUID = 1L;
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period
```

```
     * @param  end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end   = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(
                          start + " after " + end);
    }

    public Date start () { return new Date(start.getTime()); }

    public Date end ()   { return new Date(end.getTime()); }

    public String toString() { return start + " - " + end; }

    // Remainder omitted
}
```

2. Coding up the MutablePeriod class:

Serializing the Period class does more harm than good. For starters, it allows attackers to pass in invalid object instances as byte streams. The readObject method is used to create an object from a byte stream passed in as a parameter without any type safety mechanisms implemented. Normally, the byte stream is generated by serializing a normally constructed instance. The problem arises when readObject is passed with a byte stream that is artificially constructed to generate an object that violates the invariants of its class. Such a byte stream can be used to create an impossible object, which could not have been created using a normal constructor.

This however, can be fixed by coding a readObject method which checks for validity of invariants and throws an InvalidObjectException if invariants are not satisfied. But this fix does not clear the so called “immutable” Period class of all threats and attacks.

Despite this effort of fixing the Period class, it still allows the attacker to pass in a perfectly valid object instance as a byte stream and appending rogue object references to private

fields of the Period class at the end. The attacker reads the Period instance from the ObjectInputStream and then reads the “rogue object references” that were appended to the stream. These references give the attacker access to the objects referenced by the private Date fields within the Period object. By mutating these private Date fields, the attacker can mutate the Period instance.

The attack is demonstrated as follows:

MutablePeriod.java

```
package Assignment;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Date;

public class MutablePeriod {
    // A period instance
    public final Period period;

    // period's start field, to which we shouldn't have access
    public final Date start;

    // period's end field, to which we shouldn't have access
    public final Date end;

    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream out = new ObjectOutputStream(bos);

            // Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));

            /*
             * Append rogue "previous object refs" for internal Date
fields in Period. For
```

```
             * details, see "Java Object Serialization
Specification," Section 6.4.
             */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // The start field
            ref[4] = 4; // Ref # 4
            bos.write(ref); // The end field

            // Deserialize Period and "stolen" Date references
            ObjectInputStream in = new ObjectInputStream(new
ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObject();
            start = (Date) in.readObject();
            end = (Date) in.readObject();
        } catch (Exception e) {
            throw new AssertionError(e);
        }
    }

    public static void main(String[] args) {
        MutablePeriod mp = new MutablePeriod();
        Period p = mp.period;
        Date pEnd = mp.end;

        // Let's turn back the clock
        pEnd.setYear(78);
        System.out.println(p);

        // Bring back the 60s!
        pEnd.setYear(69);
        System.out.println(p);
    }

}
```

On running the main method for this one, we get the following output:

```
D:\GMU\SWE 619\Assignment 12\src\main\java
λ java Assignment/MutablePeriod
Wed Nov 20 11:47:19 EST 2019 - Mon Nov 20 11:47:19 EST 1978
Wed Nov 20 11:47:19 EST 2019 - Thu Nov 20 11:47:19 EST 1969
```

Conclusion:

Thus, in this assignment, we perused Bloch's Item 88: "Write readObject methods defensively" and implemented the attack which allowed an attacker to mutate private fields even in an immutable class. We coded up the immutable Period class and the attackers MutablePeriod class and verified that the attack takes place.