Reverse Engineering: Java Reflection

Table of contents

1 Introduction

In this lab, we will get our hands dirty with java. Note that this section will NOT involve assembly language. The files provided should be safe files as we wrote them, so completing this on your host machine is fine, however, we recommend that you run it on your VM when using unknown files.

Recall that Java is one of the languages that is partially compiled into byte code. The byte code can be distributed to any user who wants to execute it without needing to make any accommodations for different target systems. Additionally, one of the features of byte code is it provides some kind of security such that users cannot easily figure out what the code is.

2 Java Install

If you already have java on your machine, whether it is your host or virtual machine, you are welcome to skip this section.

To install java on a Windows machine, go to https://www.oracle.com/java/technologies/downloads/#jdk24-windows.

To install java on Linux Mint or Ubuntu, you can run the following commands

```
sudo apt update
sudo apt install default-jdk
```

To install java on a Mac, run the following command:

```
brew install --cask oracle-jdk
```

3 Confirm Java Installation

To confirm you have the JDK installed properly, run the following commands. Recall that the `javac` command is used to compile a java file into the bytecode and the `java` command can execute that bytecode.

```
java --version
```

```
javac --version
```

If both commands run without an error, then you are good to move on.

4 Download and Unzip the Files

For this lab you will need a few files to work with. **Do NOT use an IDE such as VSCode** for this lab as it may decompile the bytecode for you, and that ruins the magic of learning about reverse engineering with java. Instead, use your terminal with nano or vim.

The files you will need for this lab are contained on Canvas inside `java reflection lab files.zip`.

Once downloaded, unzip the files before continuing.

When unzipped, you should see two directories `Countdown` and `Test`.

5 Directory 1 - Test

To complete the following, navigate into the Test directory.

5.1 Initial Investigation

We will start by investigating the files we have. In particular, the `Test.class` file and the `TestTest.class` file.

Recall that when you compile a `.java` file, you will receive as output one or more `.class` files. Let's start by looking at the bytecode of the Test.class file.

```
cat Test.class
```

Looking at the output from the previous command tells us very little about what the file does. Now let's look at TestTest.class

```
cat TestTest.class
```

Again, not much information can be extracted from TestTest.class.

Let's try running the file with the following command.

```
java Test.class
```

What information was gained about this file? Not much, but we now know the `Test` class does not have a `main()` method inside of it.

Let's look run the `TestTest.class` file.

```
java TestTest.class
```

You should get an output of content printing to the terminal. What information was gained from this? Possibly there is some looping happening in either TestTest.class or Test.class. If you go back and look at the contents of Test, you will see that the string that prints from running TestTest is found inside the bytecode of Test. So you know that TestTest is mostlikely invoking Test.

5.2 Inspect Fields

Using Java's reflection library, we can get more information out of the Test.class file and the TestTest.class file.

Create a java file called `InspectFields.java` inside the same directory as Test.class and place the following code inside it.

```
import java.lang.reflect.Field;

class InspectFields{

    public static void main(String[] args){
        Field[] fields = Test.class.getDeclaredFields();

        for (Field f : fields){
            System.out.println(f);
        }
    }

}
```

Every class in java allows us to call on `.class.getDeclaredFields()` to get information about the member variables declared directly in a class. The code for out Inspect class does that to the Test class. You should get the following output (or similar):

```
private java.lang.String Test.s
private int Test.x
```

We now know that Test.class has two private instance variables, one named `s` that is a String and one named `x` that is an int. This aligns perfectly with what we saw when we ran TestTest, a String and an int.

We can view the values for these fields by creating an instance of the class, and accessing the fields on each one. Modify `InspectFields.java` file to contain an instance of the class and the code that will allow us to access the field values indicated in the following code bock.

```
class InspectFields{

    public static void main(String[] args){

        Test testInstance = new Test();  // create an instance of Test
```

```
        Field[] fields = Test.class.getDeclaredFields();

        String s; // a variable to hold the value of s
        int x; // a variable to hold the value of x

        for (Field f : fields){
            System.out.println(f);
        }

        // access the values of each field
        try {
            // since both s and x are private in the Test class, we must use setAccessible()
            fields[0].setAccessible(true);
            fields[1].setAccessible(true);

            // get and print the String field
            s = fields[0].get(testInstance).toString();
            System.out.println(s);

            // get and print the int field
            x = (int) fields[1].get(testInstance);
            System.out.println(x);
        } catch (Exception e) {
            // don't do anything, or print something if you want
        }
    }

}
```

The output should be similar to the following:

```
private java.lang.String Test.s
private int Test.x
some random string
10
```

We can now see that the values of the String and the int were `some random string` and `10` respectively.

5.3 Set Field

Just as we were able to view the field, we can edit it as well. Create a file named `Edit.java` to edit the int field for Test.

```
import java.lang.reflect.Field;

class Edit {

    public static void main(String[] args){
        Test testInstance = new Test();
        Field[] fields = Test.class.getDeclaredFields();

        try {
            // set accessible since it is a private field
            fields[1].setAccessible(true);

            // set the value to 3 instead of 10
            fields[1].set(testInstance, 3);

            // get the value to see if it changed
            int x = (int) fields[1].get(testInstance);
            System.out.println(x);

        } catch (Exception e) {

        }
    }
}
```

Note that if the Test class had a `main()` method we could have called it with the line `testInstance.main(null);`

5.4 Inspect Methods

Java's reflection library comes with the ability to get the methods that are in another class, even if that class is already compiled into bytecode.

Create a file called InspectMethods.java in the same directory as Test and place the following code in it.

```java
import java.lang.reflect.Method;

public class InspectMethods {

    public static void main(String[] args){
        Method[] methods = Test.class.getDeclaredMethods();

        for (Method method : methods){
            System.out.println(method);
        }
    }

}
```

The output should look similar to the following:

```
public void Test.method1()
private void Test.method2()
public void Test.method3(int)
```

We now know that the Test class has three methods named method1, method2, and method3. One of the methods is private, one takes an int, and all three do not return anything.

5.5 Invoking the Methods

Now that we know what each method is, let's invoke them to get a better understanding of what each one does.

Create a file named InvokeMethods.java in the same directory and add the following code to it.

```java
import java.lang.reflect.Method;

public class InvokeMethods{

    public static void main(String[] args) throws Exception {
        // create a reference to test.class for convenience
        Class<Test> tc = Test.class;

        // create an instance of Test for invoking methods
        Test testInstance = new Test();
```

```
        // get the method named "method1" and invoke it
        System.out.println("\nStarting Method 1")
        Method method1 = tc.getDeclaredMethod("method1");
        method1.invoke(testInstance);

        // get "method2" and invoke it. Note that method2 was private.
        System.out.println("\nStarting Method 2")
        Method method2 = tc.getDeclaredMethod("method2");
        method2.setAccessible(true); // to access private methods
        method2.invoke(testInstance);

        // get "method3" and invoke it.
        System.out.println("\nStarting Method 3")
        // note the addition of int.class on the next line
        Method method3 = tc.getDeclaredMethod("method3", int.class);
        // note the passing in of the value 3 on the next line
        // consider 3 as a randomly chosen value
        method1.invoke(testInstance, 3);
    }
}
```

The output should be similar to the following:

```
Starting Method 1
s = some random string
x = 10

Starting Method 2
Should not be able to evoke a private method

Starting Method 3
Multiple times?
s = some random string
x = 10
s = some random string
x = 10
s = some random string
x = 10
```

Now you might be able to make some conclusions about what each method of the Test file does. For method1, you can see that it prints two lines: a string for the variable s, and a value 10 for the variable x.

The second method appears to just print “Should not be able to evoke a private method.”

The third method appears to loop method 1 the number of times indicated by the argument passed into it.

6 Directory 2 - Countdown

Again, for this portion, it is recommended that you do not use an IDE such as VSCode that will automatically decompile the code for you in such a simple case (for the sake of practicing). Instead, consider using vim or nano.

In the Countdown directory, you will find a file named Countdown.class. Run the file to see what it does.

```
java Countdown
```

Notice how it is counting down from 2,000,000 and doing so rather slowly.

Apply the concepts from the Test Directory portion of this document to see if you can get to zero quicker.