Assignment I – Data Races Prevention & Runtime Policy Enforcement Using Aspect-Oriented Programming

CPS 499-02/592-02

Software/Language Based Security

Fall 2020

Dr. Phu Phung

Evan Krimpenfort

Video Link:

https://udayton.zoom.us/rec/share/gPWa1mPsIr3GApKD3opxcVKfixrE5F4Ik0qyfEnB711FlMv1Z-mtImC-_Ix5XcLb.lgbQECI1wIVT9cvu

Part I: Fixing time-of-check time-of-use (TOCTOU)

Task I:

```
/**
 * New function that allows the user to safely withdraw money from the Wallet.txt file
 */
public int safeWithdraw(int valueToWithdraw) throws Exception
{
    int previousBalance = 0;
    int newBalance = 0;

    while(isLocked) {}  // locking loop

    isLocked = true;
    previousBalance = this.getBalance();
    newBalance = previousBalance - valueToWithdraw;
    this.setBalance(newBalance);
    isLocked = false;

    return previousBalance;
}
```

Figure 1: Snippet of the safeWithdraw function

To make a function that replaces `wallet.setBalance`, we need to do a few things. We need to rewrite the balance and we need to do it in a way that is safe so that there is not a race condition. I first wanted to wrap around my arithmetic inside of a lock. You can see this in figure 1. That way, another user couldn't touch the shared data (access to wallet.txt) while it's being used. Everything inside of the lock was getting the balance, making the deduction, and then setting the balance. Once that was all over, the lock was unlocked for another user to access the shared data. Code is in appendix B.

Task II:

```
/**
 * New function that allows the user to safely deposit money from the Wallet.txt file
 */
public void safeDeposit(int valueToDeposit) throws Exception
{
    int previousBalance = 0;
    int newBalance = 0;

    while(isLocked) {}  // locking loop

    isLocked = true;
    previousBalance = this.getBalance();
    newBalance = previousBalance + valueToDeposit;
    this.setBalance(newBalance);
    isLocked = false;
}
```

Figure 2: Snippet of the safeDeposit function

`Wallet.safeDeposit(..)` is very similar to its partner withdraw. It has a lock over the shared data and unlocks when its done with using the shared data. The only difference seen in figure 2 is instead of deducting from the balance and setting that amount to "wallet.txt," it adds the amount to the retrieved balance and writes that new balance to "wallet.txt."

Task III:

```
out.println("Your balance: " + wallet.getBalance() + " credits");
wallet.safeWithdraw(price); // REPLACED
Pocket pocket = new Pocket();
pocket.addProduct(product);
out.println("You have sucessfully purchased a '" + product + "'");
out.println("Your new balance: " + wallet.getBalance() + " credits");
out.flush();
socket.close();
```

Figure 3: Changes in ShoppingCart.java

Changes made *ShoppingCart.Java* were getting rid of the balance variable that retrieved the balance at the very beginning of the thread. It was also getting rid of any race conditions like the conditional statement between the balance and price. That way, the balance that's been used for that user is correct. Code is in appendix A.

Task IV:

The fix was done by getting rid of the conditional statement between the balance and price variable and making sure that any changing of the shared data was done inside of a lock.

Task V:

```
String product = "";
int price = 0;
while(true)
{
    out.println("What do you want to buy, type e.g., pen?");
    out.flush();
    product = bufferreader.readLine();
    price = Store.getPrice(product);
    if (price > 0) { break; }
    out.println("That Item is not available. Pick another item.");
}
```

Figure 4: Fixed the purchasing

One issue found before was that if a user purchased some item on the list that wasn't there, the item got a default price of zero and still showed up in the "pocket.txt." I fixed that by making sure that the price returned by the Store was always greater than 0. If it wasn't an item on the list surely wasn't purchased and the user was given the chance to purchase something again. You can see my changes in figure 4.

Part II: Enforcing security policies using AOP (AspectJ)

Task I:

```
before(int price): safeWithdraw(price)
{
    try
    {
        Wallet wallet = new Wallet();
        if (wallet.getBalance() - price < 0)
        {
            System.out.println("The amount withdrawn is too much: " + price + " credits"

        }
    }
    catch (Exception e)
    {
        System.out.println("Wallet threw an exception.");
    }
}
```

Figure 5: Before

For Before in figure 5, I used the wallet to check the balance and warn the server about someone buying something with insufficient funds. It's okay to access the shared data here because nothing is being changed. Code is in appendix C.

Task II:

```
after(int price) returning (int withdrawnAmount): safeWithdraw(price)
{
    try
    {
        Wallet wallet = new Wallet();
        if (withdrawnAmount - price < 0)
        {
            wallet.safeDeposit(price);
            System.out.println(withdrawnAmount + " credits was returned to the owner.");
        }
        System.exit(1);
    }
    catch (Exception e)
    {
        System.out.println("Wallet threw an exception.");
    }
```

Figure 6: After

For After in figure 6, `Wallet.safeWithdraw(..)` has been executed. The return value here is what the balance was initially. If that return value subtracted by price gets a value below zero, then the purchase was made with insufficient funds and should return that money back into the client's wallet. The server also ends its program since the client is stealing from the store.

All code can be found in appendix D.

Appendix

Appendix A – ShoppingCart.java | https://github.com/Krimpenfort23/autumn-2020/blob/master/assignments/assignment1/ShoppingCart/ShoppingCart.java

```
import java.util.Scanner;
import java.util.Date;
import java.io.*;
import java.net.*;

public class ShoppingCart
{
    public static void main(String[] args) throws Exception
    {
        try
        {
            int port = 8888;
            ServerSocket serversocket=new ServerSocket(port);
            System.out.println("ShoppingCart program is running on port " + port);
            System.out.println("Waiting for connections from clients");
            while (true)
            {
                Socket socket=serversocket.accept();//establishes connection
                System.out.println("A new customer is connected!");
                OutputStreamWriter outputstreamwriter= new
OutputStreamWriter(socket.getOutputStream());
                BufferedReader bufferreader= new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                Thread thread=new Thread(new ServerThread(socket,outputstreamwriter,bufferreader));
                thread.start();
            }
        }
        catch(IOException ex)
        {
            System.out.println(ex.getMessage());
        }
    }

    public static class ServerThread extends Exception implements Runnable
    {
        final Socket socket;
        final BufferedReader bufferreader;
        final OutputStreamWriter outputstreamwriter;

        public ServerThread(Socket socket, OutputStreamWriter outputstreamwriter, BufferedReader
bufferreader)
        {
            this.socket = socket;
            this.outputstreamwriter=outputstreamwriter;
            this.bufferreader=bufferreader;
        }

        public void run()
        {
            try
            {
                Scanner input = new Scanner(System.in);
                PrintWriter out=new PrintWriter(outputstreamwriter);
```

```
                Wallet wallet = new Wallet();
                out.println("Welcome to Evan Krimpenfort's ShoppingCart. The time now is " + (new
Date()).toString());
                out.println("Please select your product: ");

                for (Object element :Store.asString())
                {
                    out.println(element);
                }

                String product = "";
                int price = 0;
                while(true)
                {
                    out.println("What do you want to buy, type e.g., pen?");
                    out.flush();
                    product = bufferreader.readLine();
                    price = Store.getPrice(product);
                    if (price > 0) { break; }
                    out.println("That Item is not available. Pick another item.");
                }

                out.println("Your balance: " + wallet.getBalance() + " credits");
                wallet.safeWithdraw(price); // REPLACED
                Pocket pocket = new Pocket();
                pocket.addProduct(product);
                out.println("You have sucessfully purchased a '" + product + "'");
                out.println("Your new balance: " + wallet.getBalance() + " credits");
                out.flush();
                socket.close();
            }
            catch(Exception ex)
            {
                System.out.println(ex.getMessage());
            }
        }
    }
}
```

Appendix B – Wallet.java | https://github.com/Krimpenfort23/autumn-2020/blob/master/assignments/assignment1/ShoppingCart/Wallet.java

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

public class Wallet
{
   /**
    * The RandomAccessFile of the wallet file
    */
   private RandomAccessFile file;
   private boolean isLocked;

   /**
    * Creates a Wallet object
    *
```

```
 * A Wallet object interfaces with the wallet RandomAccessFile
 */
 public Wallet () throws Exception
 {
     this.file = new RandomAccessFile(new File("wallet.txt"), "rw");
     isLocked = false;
 }

/**
 * Gets the wallet balance.
 *
 * @return                  The content of the wallet file as an integer
 */
 public int getBalance() throws IOException
 {
     this.file.seek(0);
     return Integer.parseInt(this.file.readLine());
 }

/**
 * Sets a new balance in the wallet
 *
 * @param  newBalance          new balance to write in the wallet
 */
 public void setBalance(int newBalance) throws Exception
 {
     this.file.setLength(0);
     String str = new Integer(newBalance).toString()+'\n';
     this.file.writeBytes(str);
 }

/**
 * Closes the RandomAccessFile in this.file
 */
 public void close() throws Exception
 {
     this.file.close();
 }

/**
 * New function that allows the user to safely withdraw money from the Wallet.txt file
 */
 public int safeWithdraw(int valueToWithdraw) throws Exception
 {
     int previousBalance = 0;
     int newBalance = 0;

     while(isLocked) {}  // locking loop

     isLocked = true;
     previousBalance = this.getBalance();
     newBalance = previousBalance - valueToWithdraw;
     this.setBalance(newBalance);
     isLocked = false;

     return previousBalance;
 }
```

```
   /**
    * New function that allows the user to safely deposit money from the Wallet.txt file
    */
    public void safeDeposit(int valueToDeposit) throws Exception
    {
        int previousBalance = 0;
        int newBalance = 0;

        while(isLocked) {}  // locking loop

        isLocked = true;
        previousBalance = this.getBalance();
        newBalance = previousBalance + valueToDeposit;
        this.setBalance(newBalance);
        isLocked = false;
    }
}
```

Appendix C – ShoppingCartAspectJ.aj | https://github.com/Krimpenfort23/autumn-2020/blob/master/assignments/assignment1/part2/ShoppingCartAspectJ.aj

```
public aspect ShoppingCartAspectJ
{
    pointcut safeWithdraw(int price): call(* Wallet.safeWithdraw(int)) && args(price);
    before(int price): safeWithdraw(price)
    {
        try
        {
            Wallet wallet = new Wallet();
            if (wallet.getBalance() - price < 0)
            {
                System.out.println("The amount withdrawn is too much: " + price + " credits");
            }
        }
        catch (Exception e)
        {
            System.out.println("Wallet threw an exception.");
        }
    }
    after(int price) returning (int withdrawnAmount): safeWithdraw(price)
    {
        try
        {
            Wallet wallet = new Wallet();
            if (withdrawnAmount - price < 0)
            {
                wallet.safeDeposit(price);
                System.out.println(withdrawnAmount + " credits was returned to the owner.");
                System.exit(1);
            }
        }
        catch (Exception e)
        {
            System.out.println("Wallet threw an exception.");
        }
```

```
    }
}
```

Appendix D – All Code
https://github.com/Krimpenfort23/autumn-2020/tree/master/assignments/assignment1