TOCTOU Lab Report

Part 1

Shared Resource

The shared resource in this system is `wallet.txt`, which is managed by the `Wallet` class. It stores the user's balance.

Who Is Sharing It?

The main thread of the program (`ShoppingCart.java`) uses this file to read the balance and perform withdrawal operations.
However, when multiple instances of the frontend are running simultaneously, they all interact with the same `wallet.txt` file, effectively sharing access to this resource during the purchasing process.

Root Cause of the Problem

The key issue lies in the separation of two operations:

- **Checking** the balance: `wallet.getBalance()`
- **Modifying** the balance (deduction): `wallet.setBalance(...)`

These two actions are performed separately. The time gap between them creates a **race condition window**.
If another thread or process modifies the wallet during this window, the balance check becomes outdated, potentially allowing unauthorized purchases.

How to Exploit the System (TOCTOU Attack)

This vulnerability can be exploited by simulating a TOCTOU attack using multiple concurrent frontend instances (here we use two instances):

Step 1: Thread T1

Start the program: `java ShoppingCart` . Choose a product, e.g., **Car**, which costs `30000` , equal to the current wallet balance. Pause the execution before the program deducts the balance (using a debugger)

```
16      public static void main(String[] args) throws Exception { args = String[0]@8
17          Wallet wallet = new Wallet(); wallet = Wallet@9
18          Pocket pocket = new Pocket(); pocket = Pocket@10
19          Scanner scanner = new Scanner(System.in); scanner = Scanner@11
20
21          print(wallet, pocket); wallet = Wallet@9, pocket = Pocket@10
22          String product = scan(scanner); product = "car", scanner = Scanner@11
23
24          while(!product.equals(anObject:"quit")) { product = "car"
25              int balance = wallet.getBalance(); balance = 30000, wallet = Wallet@9
26              if(balance < Store.getProductPrice(product)){ //Checks if the balance is smaller than the
27                  return; //Quits if that is the case
28              }
29          else { //Product was not too expensive
30              wallet.setBalance(wallet.getBalance()-Store.getProductPrice(product)); // Removes the proc
31              pocket.addProduct(product); //Adds product to pocket
32              System.out.println("Your new balance is: " + wallet.getBalance() + " credits."); //Prints
33          }
34
```

```
Your current balance is: 30000 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:

What do you want to buy? (type quit to stop) car
```

Step 2: Thread T2

Start another instance of java ShoppingCart. Choose a cheaper product (≤ 30000). Since T1 hasn't updated the wallet yet, the balance is still seen as 30000. T2 successfully purchases the item and modifies the wallet.

```
Your current balance is: 30000 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:

What do you want to buy? (type quit to stop) book
Your new balance is: 29900 credits.
Your current balance is: 29900 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:
book
```

Step 3: Resume T1

T1 continues and completes the withdrawal of 30000. The system has now overdrawn the wallet.

```
Your current balance is: 30000 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:

What do you want to buy? (type quit to stop) car


Your new balance is: -100 credits.
Your current balance is: -100 credits.
car     30000
book    100
pen     40
candies 1

Your current pocket is:
book
car
```

Part 2: Fix the API (Revised)

Addressing the Root Cause

The vulnerability stems from a classic **Time-of-Check to Time-of-Use (TOCTOU)** flaw. In the original system, two separate operations — `wallet.getBalance()` and `wallet.setBalance(...)` — are used to check and update the user's balance. This separation introduces a time window where the state can change due to concurrent access.

Our initial fix used `ReentrantLock`, which offers thread-level synchronization. However, the TOCTOU attack occurs at the **process level**, involving two separate JVM instances accessing the same file. Thus, `ReentrantLock` alone is insufficient.

Why ReentrantLock Is Not Enough

`ReentrantLock` protects shared data from concurrent access by **threads** in the same process. It has no effect across **different processes**, which is where the actual vulnerability is exploited in this lab. Therefore, additional protection is required.

Introducing FileLock

To solve the problem at the process level, we use `FileLock` from `java.nio.channels`. This allows us to place exclusive or shared locks on the file being accessed:

- **Exclusive locks** prevent any other process from reading or writing to the file. Used in `safeWithdraw`.
- **Shared locks** allow concurrent readers but no writers. Used in `GetBalance`.

`ReentrantLock` is still used for thread-level protection within the same process.

Revised `Wallet.java` Implementation

```java
package backEnd;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.channels.FileChannel;
import java.nio.channels.FileLock;
import java.util.concurrent.locks.ReentrantLock;

public class Wallet {
    /**
     * The RandomAccessFile of the wallet file
     */
    private final RandomAccessFile file;
    private final FileChannel channel;
    private final ReentrantLock lock = new ReentrantLock();

    /**
     * Creates a Wallet object
     *
     * A Wallet object interfaces with the wallet RandomAccessFile
     */
    public Wallet () throws Exception {
        this.file = new RandomAccessFile(new File("backEnd/wallet.txt"), "rw");
        this.channel = file.getChannel();
    }

    /**
     * Gets the wallet balance (thread-safe + shared file lock)
     *
     * @return The content of the wallet file as an integer
     */
    public int getBalance() throws IOException {
        lock.lock();
        FileLock fileLock = null;
        try {
            fileLock = channel.lock(0L, Long.MAX_VALUE, true); // Shared lock for readi
            this.file.seek(0L);
            String line = this.file.readLine();
            return (line == null || line.trim().isEmpty()) ? 0 : Integer.parseInt(line.
        } finally {
```

```
        if (fileLock != null) fileLock.release();
        lock.unlock();
    }
}

/**
 * Sets a new balance in the wallet (thread-safe + exclusive file lock)
 *
 * @param newBalance New balance to write in the wallet
 */
public void setBalance(int newBalance) throws Exception {
    lock.lock();
    FileLock fileLock = null;
    try {
        fileLock = channel.lock(); // Exclusive lock
        this.file.seek(0L);
        this.file.setLength(0);
        String str = Integer.toString(newBalance) + '\n';
        this.file.writeBytes(str);
    } finally {
        if (fileLock != null) fileLock.release();
        lock.unlock();
    }
}

/**
 * Atomically withdraws money from the wallet (thread-safe + exclusive file lock)
 *
 * @param valueToWithdraw amount to withdraw
 * @return true if successful, false if insufficient funds
 */
public boolean safeWithdraw(int valueToWithdraw) throws Exception {
    lock.lock();
    FileLock fileLock = null;
    try {
        fileLock = channel.lock(); // Exclusive file lock
        this.file.seek(0L);
        String line = this.file.readLine();
        int currentBalance = (line == null || line.trim().isEmpty()) ? 0 : Integer.
        if (currentBalance < valueToWithdraw) {
            return false;
        } else {
            this.file.setLength(0L);
```

```
                this.file.seek(0L);
                this.file.writeBytes((currentBalance - valueToWithdraw) + "\n");
                return true;
            }
        } finally {
            if (fileLock != null) fileLock.release();
            lock.unlock();
        }
    }

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

Why We Need File Locks for Reading

Reading a shared file without acquiring a lock may lead to **data corruption** or **runtime crashes**, especially in a multi-process environment.

Platform Differences

- **On Windows**: Attempting to read a file that is exclusively locked by another process can result in an exception or blocked access.
- **On Unix/Linux**: Although reads are often allowed during writes, this can lead to **inconsistent or partially written data** being read.

This makes it essential to synchronize reads not just at the thread level, but also at the **process level**.

The Solution: Shared File Locks

To safely read data, we use a **shared file lock** via `java.nio.channels.FileLock`. This type of lock:

- Allows multiple readers to access the file concurrently.
- Prevents writers from modifying the file during a read.

In our `getBalance()` method, we apply a shared lock as follows:

```
fileLock = channel.lock(0L, Long.MAX_VALUE, true); // shared lock
```

The `ShoppingCar.java` is below:

```
import backEnd.*;
import java.util.Scanner;

public class ShoppingCart {
    private static void print(Wallet wallet, Pocket pocket) throws Exception {
        System.out.println("Your current balance is: " + wallet.getBalance() + " credit
        System.out.println(Store.asString());
        System.out.println("Your current pocket is:\n" + pocket.getPocket());
    }

    private static String scan(Scanner scanner) throws Exception {
        System.out.print("What do you want to buy? (type quit to stop) ");
        return scanner.nextLine();
    }

    public static void main(String[] args) throws Exception {
        Wallet wallet = new Wallet();
        Pocket pocket = new Pocket();
        Scanner scanner = new Scanner(System.in);

        print(wallet, pocket);
        String product = scan(scanner);

        while(!product.equals("quit")) {
            int price = Store.getProductPrice(product); //Gets the price of the request
            if (price == -1) {
                System.out.println("Product not found.");
        }
        else if(wallet.safeWithdraw(price)){ //Product was not too expensive
            pocket.addProduct(product); //Adds product to pocket
            System.out.println("You bought " + product ); //Prints success message
        }else{
            System.out.println("Insufficient balance for " + product + "."); //Prints e
        }
            // Just to print everything again...
            print(wallet, pocket);
            product = scan(scanner);
        }
    }
}
```