TDA602 / DIT101 Language-based security
TOCTOU Attack Experiment Report

Group47: Yicheng Li, Wenjun Tian

April 16, 2025

1 Part 0

1.1 ShoppingCart.java Implementation

The shopping cart implementation has been submitted as an attachment (ShoppingCart.java).

1.2 Compilation and Execution Instructions

1. Navigate to the directory containing ShoppingCart.java
2. Compile the program by executing: `make all`
3. Run the program with: `java ShoppingCart`

2 Part 1

To reliably reproduce the TOCTOU attack, we artificially introduced a breakpoint between `wallet.getBalance()` and `wallet.setBalance()` in the main function, illustrated in 1.

2.1 Attack Procedure

1. Launch two terminal sessions
2. In Terminal 1: Run the program and enter "car" to initiate purchase
3. In Terminal 2: Run another instance and enter "car" to attempt purchase
4. In both terminals: Enter "enter" to continue execution, output logs shown in 2

2.2 Results and Analysis

- Examination of `pocket.txt` reveals two "car" entries, confirming both purchases succeeded
- **Shared Resources**:
  - `backEnd/wallet.txt` (accessed via Wallet class)
  - `backEnd/pocket.txt` (accessed via Pocket class)

  The sharers are multiple instances of the ShoppingCart program run by the same user. Since the file system is shared, multiple processes can access these files simultaneously.
- **Root Cause**: The balance check and deduction operations are non-atomic. When Terminal 1 passes the balance check but hasn't completed the deduction, Terminal 2 can also pass the check, allowing both transactions to succeed.

```java
31
32          try {
33              // fetch the price of product
34              int price = Store.getProductPrice(product);
35
36              // check the balance
37              int currentBalance = wallet.getBalance();
38              if(wallet.getBalance() < price) {
39                  System.out.println("Not enough credits to buy " + product + "!");
40                  break;
41              }
42
43              System.out.println(x:"\n[TOCTOU window] press Enter to continue...");
44              System.in.read();  // system pause
45
46              //withdraw
47              wallet.setBalance(currentBalance - price);
48
49              // add product to pocket
50              pocket.addProduct(product);
51
52              System.out.println("Successfully purchased " + product + " for " + price + " credits");
53
54              // print updated info
55              print(wallet, pocket);
56
57          } catch (Exception e) {
58              System.out.println("Error: " + e.getMessage());
59              // break;
60          }
61
62          product = scan(scanner);
```

Figure 1: Code for Part 1

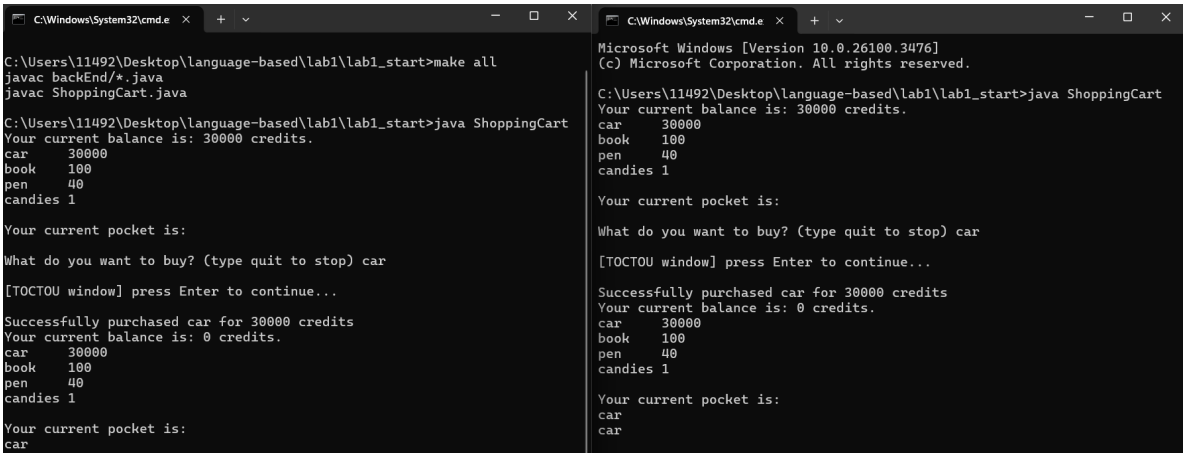

Figure 2: Attack in Part 1

```java
    /**
 * Safely withdraws money from the wallet (atomic check-and-withdraw)
 *
 * @param  valueToWithdraw     amount to withdraw
 * @return                     true if withdraw was successful, false if insufficient funds
 */
public boolean safeWithdraw(int valueToWithdraw) throws Exception {
    // Get exclusive lock on the wallet file
    try (FileLock lock = channel.lock()) {
        int currentBalance = getBalance();
        if (currentBalance < valueToWithdraw) {
            return false;
        }

        System.out.println(x:"\n[TOCTOU window] press Enter to continue...");
        System.in.read();  // system pause

        setBalance(currentBalance - valueToWithdraw);
        return true;
    }
}
```

Figure 3: safeWithdraw()

3 Part 2

3.1 Security Patch Implementation

We modified `Wallet.java` to include file locking using `FileChannel` and `FileLock`, implementing a `safeWithdraw` method that ensures atomic check-and-deduction operations, illustrated in 3.

3.2 Validation Test

1. Repeat the attack procedure from Part 1
2. Terminal 2 cannot enter the vulnerable window - it waits for Terminal 1's lock release
3. After Terminal 1 completes, Terminal 2 fails the balance check
4. `pocket.txt` contains only one "car" entry, output logs shown in 4

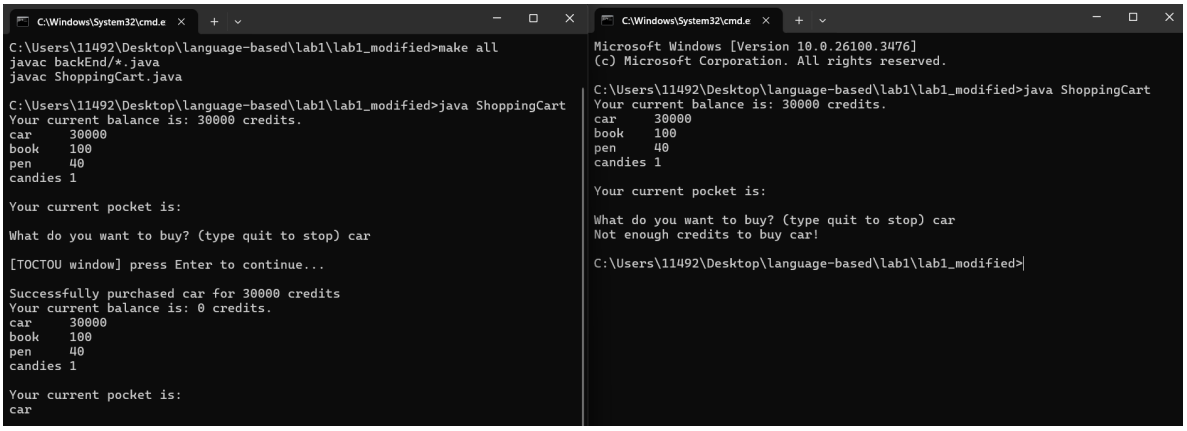

Figure 4: Attack in Part 2

```java
/**
 * Adds a product to the pocket.
 *
 * @param  product           product name to add to the pocket (e.g. "car")
 */
public void addProduct(String product) throws Exception {
    try (FileLock lock = channel.lock()) {//add filelock
        this.file.seek(this.file.length());
        this.file.writeBytes(product+'\n');
    }
}
```

Figure 5: improved Pocket.addProduct()

3.3 Additional Race Condition Considerations

`Pocket.addProduct` needs to address data integrity issues during concurrent writes. A file lock should also be added to prevent product records from becoming interleaved or lost due to overwrites, illustrated in 5. Additionally, although the experiment currently only involves the `add` operation in the `Pocket` class, integrity protection will also be necessary if delete or modify operations are added in the future.

3.4 Design Justification

3.4.1 Adequate Protection

- File locks ensure atomic check-and-deduction
- Minimal lock scope (only critical sections)
- Immediate release via try-with-resources

3.4.2 Preventing Over-Engineering

- Synchronization only where needed
- No application-wide or long-duration locks
- Fine-grained locking (per-file)

3.4.3 Performance Considerations

- OS-level file locks are efficient
- Minimal lock duration
- Maintains concurrency for unrelated operations

3.4.4 Why FileLock instead of ReentrantLock/Synchronized?

The `Wallet` class operates on a file (`Wallet.txt`), which may be accessed by multiple processes (not just threads).

- `FileLock` provides process-level protection, ensuring atomicity even if other processes try to modify the file. Also, `FileLock` supports shared and exclusive locks, which aligns with file I/O semantics.
- `ReentrantLock` and `Synchronized` only work within a single JVM, making them ineffective against cross-process TOCTOU attacks.

```java
/**
 * Gets the wallet balance.
 *
 * @return                  The content of the wallet file as an integer
 */
public int getBalance() throws IOException {
    try (FileLock lock = channel.lock(position:0, Long.MAX_VALUE, shared:true)) { //shared lock
        this.file.seek(pos:0);
        return Integer.parseInt(this.file.readLine());
    }
}
```

Figure 6: Improved `getBalance()`

3.5 More Safety Considerations

3.5.1 Access Control of `setBalance()`

The original implementation exposed `setBalance` as `public`, allowing any external code to modify the wallet balance directly. This is dangerous because it bypasses the `FileLock` protection and breaks encapsulation.
Fix: Changed `setBalance()` to `private`, forcing all balance modifications to go through `safeWithdraw()`, which enforces proper locking.

3.5.2 Implemention of Shared Lock

The modified getBalance() now acquires a shared (read) lock before reading, as is shown in 6. This ensures:

- Other processes cannot write to the file during reading.
- Multiple `getBalance()` calls can read simultaneously.
- Prevents crashes when another process holds an exclusive lock.

The lock is also automatically released via try-with-resources, guaranteeing thread-safety.
*The implementation logic of the shared lock in `Pocket.getPocket()` is completely consistent with `Wallet.getBalance()`, ensuring cross-process/thread safety.

Attachments

The following modified files have been submitted:

- `ShoppingCart.java`
- `ShoppingCart_1.java` (remove _1 suffix for testing)
- `Wallet_1.java` (remove _1 suffix for testing)
- `Pocket_1.java` (remove _1 suffix for testing)