TDA602 Lab1 Report

Group 31: Denis Furian[*] and Elin Björnsson[†]

19th April 2019

1 Part 1: Exploit your program

By running (at least) two parallel processes it is possible to fool the program into giving us more than we can afford. The program reads and writes from the text files `wallet.txt` and `pocket.txt` but, since there is no restriction on file access, one process can subtract money form the `wallet` after the other one has selected an item to buy but before it can complete the purchase.

1.1 What is the shared resource? Who is sharing it?

The shared resources are the files **wallet.txt** and **pocket.txt**. Any instances of the *ShoppingCart* application can read from and write to these files without restrictions.

1.2 What is the root of the problem? Explain in detail how you can attack this system.

Since there are no controls on file accesses, it is never guaranteed that the amount of money in the `pocket` is the same as the last time it was checked. A situation can happen like the one described in **Figure 1**, where *Process 1* alters the content of `wallet.txt` after *Process 2* has confirmed the amount of money.

1.3 Provide the program output and result, explaining the interleaving to achieve them.

As per **Figure 2**, if we launch *ShoppingCart* from two terminals and then complete a purchase on one of them, the other terminal will not keep track of the balance and will think we still have enough money.

If we then try and purchase a car on the other terminal, the program allows the transaction (see **Figure 3**) as it thinks we still have all of our balance.

[*] `furian@student.chalmers.se`
[†] `karelq@student.chalmers.se`

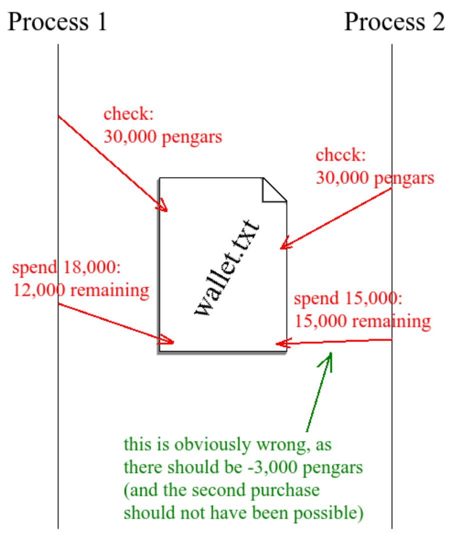
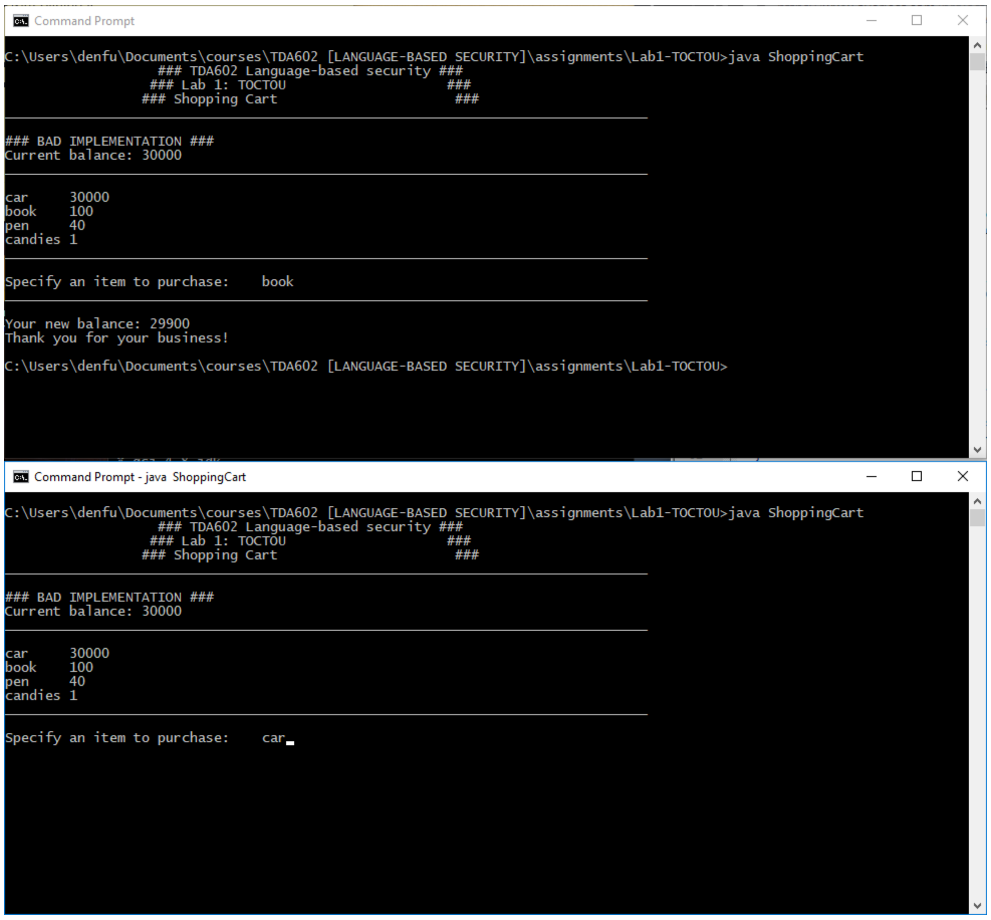


Figure 1: Example of how successive purchases can be carried out when it shouldn't be possible.

Note that this is an exaggeration: while we are forcing this execution, it may happen naturally as an OS interruption may take place at any time and result in the scheduler allowing one process to take action right between the "check balance" and "update balance" commands in the other process. What we actually need is some sort of mutual exclusion to ensure that, no matter what, only one process can update the content of `wallet.txt` at a time.

Mutual exclusion can be achieved in multiple ways but, for the purpose of this assignment, we have used locks. Locks work by "locking away" a resource while it is being used by a process: they usually come with two functions called `lock` and `release` or similarly. The `lock` function can have one of two behaviours: if the shared resource is available, it will grant access to it; if it isn't available because some other process is accessing it, the function will wait until it is freed and, at that point, grant access to the requesting process. Once a process is done working on the shared resource, it will call the `release` function whose job is to reopen the resource for access by the other processes.

We will describe the type of locks we've used in section 2.



```
Command Prompt

C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>java ShoppingCart
                   ### TDA602 Language-based security ###
                  ### Lab 1: TOCTOU                     ###
                 ### Shopping Cart                        ###

### BAD IMPLEMENTATION ###
Current balance: 30000

car     30000
book    100
pen     40
candies 1

Specify an item to purchase:    book

Your new balance: 29900
Thank you for your business!

C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>
```

```
Command Prompt - java ShoppingCart

C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>java ShoppingCart
                   ### TDA602 Language-based security ###
                  ### Lab 1: TOCTOU                     ###
                 ### Shopping Cart                        ###

### BAD IMPLEMENTATION ###
Current balance: 30000

car     30000
book    100
pen     40
candies 1

Specify an item to purchase:    car
```

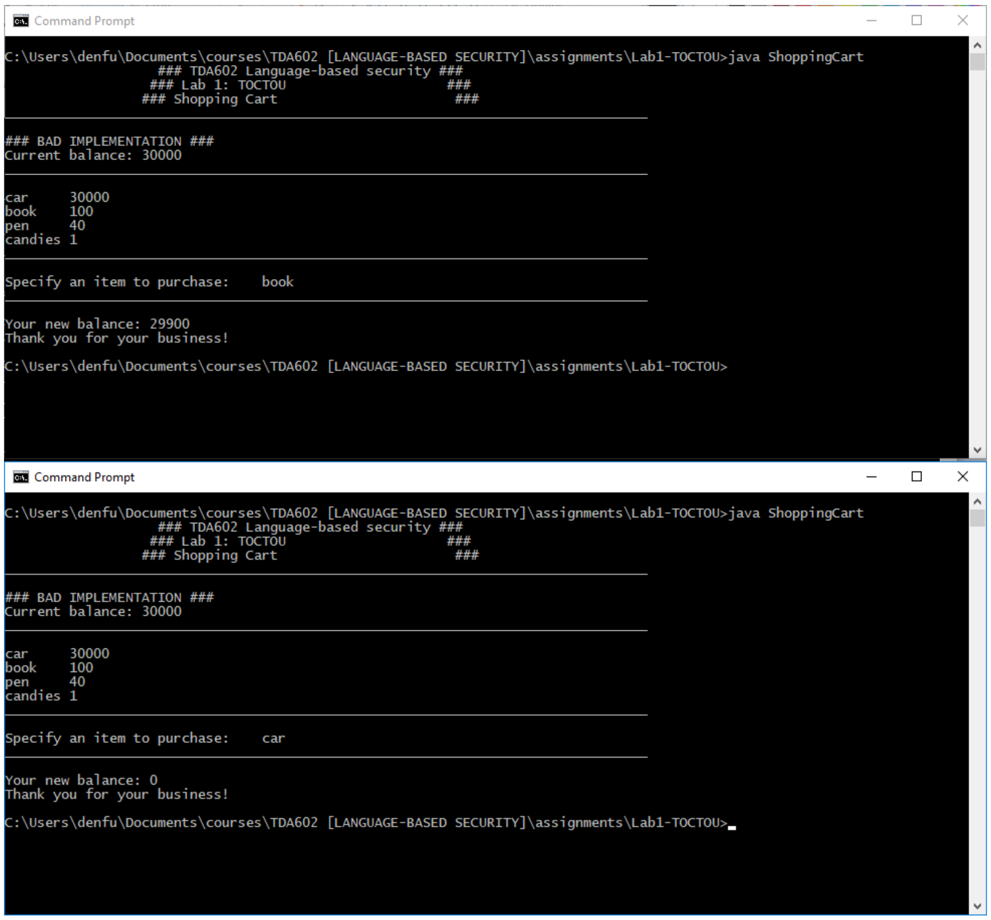
Figure 2: Purchasing a book should result in us not having enough money for a car. (Screenshot from an older version, but the bad behaviour was not changed in the most recent one)

2 Part 2: Fix the API

Once the user chose an item to purchase, the bad implementation would only care about subtracting the price from the user's balance.

The good implementation, however, fixes this by *locking the* `wallet.txt` *file* before carrying out the transaction.

```
public synchronized void safeWithdraw(int valueToWithdraw) throws Exception {
    FileLock lock = file.getChannel().lock();
    try {
        // read balance from file
        int balance = getBalance();
        // verify the amount
```



```
C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>java ShoppingCart
                   ### TDA602 Language-based security ###
                  ### Lab 1: TOCTOU                   ###
                 ### Shopping Cart                      ###
________________________________________________________________________________

### BAD IMPLEMENTATION ###
Current balance: 30000
________________________________________________________________________________

car     30000
book    100
pen     40
candies 1
________________________________________________________________________________

Specify an item to purchase:    book
________________________________________________________________________________

Your new balance: 29900
Thank you for your business!

C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>
```

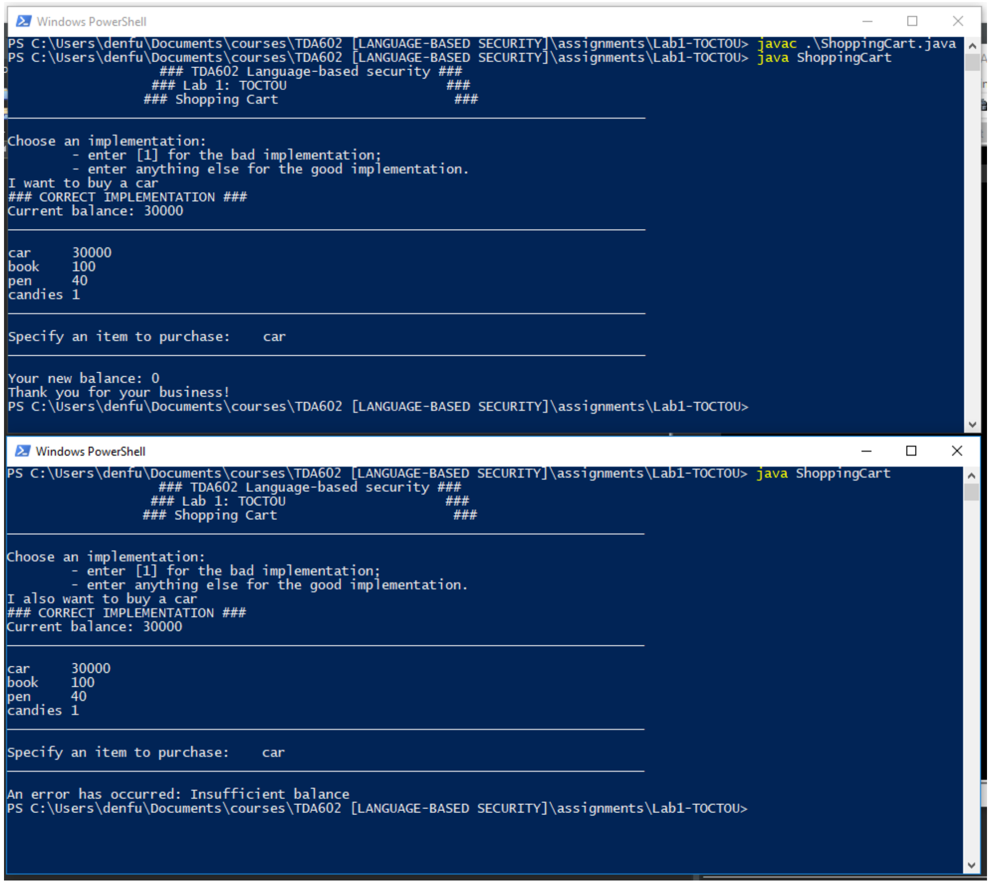
```
C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>java ShoppingCart
                   ### TDA602 Language-based security ###
                  ### Lab 1: TOCTOU                   ###
                 ### Shopping Cart                      ###
________________________________________________________________________________

### BAD IMPLEMENTATION ###
Current balance: 30000
________________________________________________________________________________

car     30000
book    100
pen     40
candies 1
________________________________________________________________________________

Specify an item to purchase:    car
________________________________________________________________________________

Your new balance: 0
Thank you for your business!

C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>
```

Figure 3: Purchasing a book and a car is possible even if we only have money for one of them. (Screenshot from an older version, but the bad behaviour was not changed in the most recent one)

```
            if (balance < valueToWithdraw) {
                // exception if balance is low
                throw new Exception("Insufficient balance");
            } else {
                // update balance otherwise
                setBalance(balance - valueToWithdraw);
            }
        } finally {
            lock.close();
        }
    }
```

(Note that some comments or debugging statements have been removed from this snippet)

Once a process has acquired the lock to `wallet.txt`, no other process can access the file and, on calling the method `safeWithdrawal`, will be kept waiting until the lock is released.



```
Windows PowerShell
PS C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU> javac .\ShoppingCart.java
PS C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU> java ShoppingCart
                   ### TDA602 Language-based security ###
                  ### Lab 1: TOCTOU                    ###
                 ### Shopping Cart                      ###
________________________________________________________________________________

Choose an implementation:
        - enter [1] for the bad implementation;
        - enter anything else for the good implementation.
I want to buy a car
### CORRECT IMPLEMENTATION ###
Current balance: 30000
________________________________________________________________________________

car     30000
book    100
pen     40
candies 1
________________________________________________________________________________

Specify an item to purchase:    car
________________________________________________________________________________

Your new balance: 0
Thank you for your business!
PS C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>
```

```
Windows PowerShell
PS C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU> java ShoppingCart
                   ### TDA602 Language-based security ###
                  ### Lab 1: TOCTOU                    ###
                 ### Shopping Cart                      ###
________________________________________________________________________________

Choose an implementation:
        - enter [1] for the bad implementation;
        - enter anything else for the good implementation.
I also want to buy a car
### CORRECT IMPLEMENTATION ###
Current balance: 30000
________________________________________________________________________________

car     30000
book    100
pen     40
candies 1
________________________________________________________________________________

Specify an item to purchase:    car
________________________________________________________________________________

An error has occurred: Insufficient balance
PS C:\Users\denfu\Documents\courses\TDA602 [LANGUAGE-BASED SECURITY]\assignments\Lab1-TOCTOU>
```

Figure 4: Purchasing two cars with just 30,000 pengars is now no longer possible.

2.1 Were there other APIs suffering from possible races? If so, please explain them and update the APIs to eliminate any race problems.

The other possible race condition is in the **pocket.txt** file, since that file is also read and written by the program instances. We saw fit to update its `addProduct` method to include locks in the same way that `safeWithdraw` does:

```
public void addProduct(String product) throws Exception {
    FileLock fileLock = file.getChannel().lock();
    // now that we are inside the pocket, we can add the product
    this.file.seek(this.file.length());
    this.file.writeBytes('\n' + product);
    // we're done writing: we can release the lock
    fileLock.close();
}
```

2.2 Why are these protections enough?

The locks ensure mutual exclusion in writing access for both shared resources `wallet.txt` and `pocket.txt`, thus making sure that no two processes can access the critical sections at the same time.

This implementation does have some limits, however. First and foremost, the `java.nio.channels.FileLock` class doesn't provide a "hard lock" for files but, rather, tells processes when they should access the shared resource. For this reason, if one process is running the bad implementation there is no way the `FileLock` can prevent it from changing the money inside `wallet.txt` or the items inside `pocket.txt`.

Another issue with the class is that it reportedly only acts on processes[1]: two Java threads within the same process will not be prevented from accessing the files at will.

Lastly, we have discovered an issue with the *Ubuntu* app for Windows 10 in that it seems to ignore locks altogether: two terminals running *ShoppingCart* at the same time will never be blocked. We suspect this may have to do with the implementation of the *Ubuntu* app, since the mutual exclusion is otherwise fully working on both Windows 10 and Ubuntu.

[1] From the documentation: "File locks are held on behalf of the entire Java virtual machine. They are not suitable for controlling access to a file by multiple threads within the same virtual machine."