Concurrency and Race Conditions

141627

Gjøvik University College

Faculty of Computer Science and Media Technology

IMT 3501 Software Security, Hanno Langweg

Assignment 02

Autumn semester 2014

Introduction

As to the Encyclopædia Britannica, the definition of Concurrency is: “Execution of more than one procedure at the same time (perhaps with the access of shared data), either truly simultaneously (as on a multiprocessor) or in an unpredictably interleaved manner.” [1]
This introduces a software vulnerability called Race Condition.
On a multitasking and multithreading operating system, processes are executed sequentially. The scheduler manages the time a process may spend on the processor. As soon as the time is over, a so called Context Switch will happen where the scheduler switches the processor to another waiting process. [2]
These Context Switches can happen at nearly anytime and so be exploited by an adversary.

Overwriting Data

To take this in an example, consider a online shop software with a currency for ordering articles with a function written like this:

```
1 void orderSomething(Article articel) {
2    currentBalance = checkAccountBalance();
3    articlePrice = article.getPrice()
4    if (currentBalance > articlePrice) {
5         orderArticle();
6         reduceBalanceBy(articlePrice);
7    } else { throwError(); }
8 }
```

The function checkAccountBalance() would raise an error is the balance is negative.
If somebody wants to abuse this system, he could exploit the TOCTTOU Race Condition. This is short for Time Of Check To Time Of Use. Since there is a small window from the resource to be checked until to be used, the adversary could run several threads to order the same article even if it crosses his balance.
To do so, he will start a second thread and hopes for a Context Switch in just the moment when thread one finished line 3. Thread two will take the processor and start the same function. At first it will store the values in the two variables currentBalance and articlePrice. Then thread one

and two have the same value stored in the currentBalance variable and can safely process to the condition and order the article. The account balance will get negative afterwards which surely was not intended by the developer nor the store owner.

One solution for this case would be a critical code section which can't be accessed by two threads simultaneously. This can be realized for example with semaphores where every time a resource is accessed, a semaphore value gets decremented. If another process wants to access the same resource, he will have to check the semaphore value first and recognize it is in use and so unavailable. As soon as the other process frees the resource, the second one can access it. This concept is called monitor. [2]

Snoop for volatile data

Another example for a race condition abuse would be code with these two functions:

```
1 website popWebsiteStack() {
2    return websiteStack.pop()
3 }
```

```
1 website getConfidentialWebpage(User user) {
2    webpage = buildWebPageFromConfidentialData();
3    websiteStack.push(webpage);
3    if (checkForAuthentication(user))
4         return websiteStack.pop();
5    else
6         websiteStack.eraseLastElement();
7         throwError();
8 }
```

There are two functions, one for returning a website with confidential data and another one just for getting the top website from the stack. If again an adversary would start two threads, one to simply access the first website on the stack and besides calling the function to view a confidential website, the possibility exists to access the confidential one without having to proof his authenticity.

The solution here is quite simpler. A small change in the codestructure is enough to prevent a race condition here. The website should not be build and stored before the authentication check of the user. By cutting and pasting the lines 2 and 3 in the if clause after the authentication check, the stack could not accidentally pop a confidential website to a non authenticated user.

[1] Encyclopædia Britannica. 15th edition. Encyclopædia Britannica Inc.; 2010.

[2] Juergen Nehmer, Peter Sturm. Systemsoftware: Grundlagen moderner Betriebssysteme. 2nd Edition. dpunkt-Verlag; 2001. Chapter 5 “Threads”.

[3] Juergen Nehmer, Peter Sturm. Systemsoftware: Grundlagen moderner Betriebssysteme. 2nd edition. dpunkt-Verlag; 2001. Chapter 6 “Speicherbasierte Prozessinteraktion”.