Race Conditions

Per Christian Kofstad
IMT3662, Mobile Development Theory
Assignemnt 1 - s.nr: 120916

In this article I will look at a race condition example from a bank transaction.

Race condition is two parts working simultaneously on the same data, ending up with a corrupted result after unsecured transaction actions. The testguide made by OWASP [1] describes race condition as follow:

> *A race condition is a flaw that produces an unexpected result when the timing of actions impact other actions. An example may be seen on a multi threaded application where actions are being performed on the same data. Race conditions, by their very nature, are difficult to test for.*

The Microsoft Support [2] team describes race condition in the following way:

> *A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.*

For race conditions to occur you need at least two users that perform a task on a shared variable at the same time. In order to find scenarios where this occurs we need to search for two requirements. Multi-tread systems with variables that more than one thread can write to.

1 Our Example

In order to find systems that fit this criteria, we basically could look everywhere where we have multi-user functionality with user write permissions. And the place where the variable is stored, must be run by a program with multiple working threads.

My example is a online bank example inspired by an article i found on defuce.ca [3].

Imagine two people sending money to a third person. Bob, Alice and Carol all have 100 kroner. Bob and Alice both want to send Carol 20 kroner each. Bob and Alice send them on the exact same time. And the transactions happens simultaneously on the same data but on two different threads, there is a race condition. It does not need to end in corrupted result, but it might. We use some pseudo-code to explain how.

```
1   Alice = 100
3   Bob   = 100
4   Carol = 100
5
6  T #1 - Alice checks if there is 20 kroner to transfer
7  T #2 - Bob checks if there is 20 kroner to transfer
8
9  T #1  - Alice get value of Carol; Carol = 100
10 T #2  - Bob gets value of Carol; Carol = 100
11
12 T #1  - Alice sets new value for own account 80;
13 T #2  - Bob sets new value for own account 80;
14
15 T #1  - Alice overwrites new value of Carol with new value;
           Carol = 120
16 T #2  - Bob overwrites new value of Carol with new value;
           Carol = 120
```

We can see from the example that the problem occurs on line 9 and 10 when both Alice and Bob gets the same value from Carol and does not understand that they are working on two operations on the same variable at once. This results in a corrupted result from line 13 and onwards. The transaction gets corrupt when Bob overwrites Carol's value without knowing that Bob then "erases" the transaction that just have been made between Alice and Carol. This means that both 20 kroner is "lost" in the system without ever being used or transferred anywhere.

Another example is when someone is adding money to an account when money is taken out of the account at the same time.

```
1   Alice = 100
3   Bob   = 100
4   Carol = 100
5
6  T #1 - Alice checks if there 20 kroner to transfer
7  T #2 - Bob checks if there 60 kroner to transfer
8
9  T #1  - Alice get value of Bob; Bob = 100
10 T #2  - Bob gets value of Carol; Carol = 100
11
12 T #1  - Alice sets new value for own account 80;
13 T #2  - Bob sets new value for own account 80;
14
15 T #1  - Alice overwrites new value of Bob with new value;
           Carol = 120
16 T #2  - Bob overwrites new value of Carol with new value;
           Carol = 160
```

In this example you see that Bob transfers money to Carol, but at the same time Alice transfers money to Bob. This result in an overwrite of Bob's account, and it seems like Bob ends up 60 kroner more than he should have on his account. Since the transfer thread over to Bob's account, didn't see that another thread was moving money out of Bob's account at the same time.

2 Mutex

Two threads working on the same variable does not normally know about each other, but there are ways to inform the threads that there are other threads working on specific parts of the system. You can add a mutex [4] to critical shared objects. This must be activated before a thread checks the current value of the variable, and not unlocked before after the all the values is completely changed. This means that only one thread can access both the transfer and the receiver account at the time, and no one is allowed to interfere.

3 Starvation

Mutex [4] is a great method to ensure that operations is happening without interrupting each other. However if there is used mutex on variables that many threads work on during a period of time, you may risk to create a starvation condition. Starvation means that there are threads waiting for a resource, in this case a variable that have to wait a very long time to get the resource because it is constantly locked to other threads. If you want to use mutex, make sure not to use it longer than the system needs to, in order to do "safe calculations".

4 Check - Prepare - Try

A method that might be better is to run the transaction in different phases. You first checks if there is available cash on the account you want to transfer to and from. Then you prepare all the numbers that you will end up with. Then you run a lock before you double check if all values is valid, if the values is valid, you change the values in one operation. If numbers is not valid you get the new numbers, prepares again and try to change again all over again. This will run until, one of two states will occur, you either do not have efficient money on the account you are pulling money from, or the transaction is completed successfully.

5 Conclusion

Race condition is a dangerous scenario, and may corrupt data. You need to implement countermeasures against race conditions. If you implement countermeasures, make sure to implement them right.

References

[1] M. Meucci, “Owasp testing guide - testing for race contesting,” *OWASP.org*, 2008. [Online]. Available: https://www.owasp.org/index.php/Testing_for_Race_Conditions_(OWASP-AT-010)

[2] T. M. S. team, “Description of race conditions and deadlocks,” *support.microsoft.com*, 2014. [Online]. Available: http://support.microsoft.com/kb/317723

[3] T. Hornby, “Practical race condition vulnerabilities in web applications,” *defuce.ca*, 2014. [Online]. Available: https://defuse.ca/race-conditions-in-web-applications.htm

[4] B. Schell and C. Marti, “Mutex,” *Webster’s New World Hacker Dictionary*, 2010. [Online]. Available: http://www.yourdictionary.com/mutex