**Anna Kaardal**
**120918**
**Obligatory exercise #2**
**15.09.2014**

## Race conditions in an electronic ID-solution for e-voting

### Introduction

In 2011 and 2013, the Norwegian Government conducted Internet voting pilots, where 168.000 (2011) and 250.000 (2013) voters had the choice of casting their ballot over the Internet [1]. The goal was to increase accassability for people who would normally have problems voting[1], like disabled people, or people living outside Norway. Electronic voting might also speed up the counting of ballots. The public trust in e-voting is high (94%[1]), however there are conserns about the security mechanisms, and of course, it is extremely important ensuring consistensy for the votes.

In this exercise, I am going to take a look at a fictivie electronic ID solution for e-voting in Norway, explain some possible race conditions for the solution, and then suggest some possible actions to prevent these race conditions from happening.

### What is a race condition?

A race condition is when the outcome of an operation depends on the recources running in a specific order [2]. If a thread is interrupted by another thread during access or modifcation of shared memory, the outcome might not be what it was intended to be. For instance:
(Some inspiration from [3].)

Sum = 4

<u>Thread1</u>
value1 = 8
sum = sum + value1

<u>Thread2</u>
value2 = 3;
sum = sum - value2

1) Thread1 adds 5 to «sum».
2) Thread2 interrupts Thread1 before finishing writing to the variable, and substracts 3 from «sum», which is still 4, and writes it to the variable. «sum» is now 1.
3) Thread1 gets back the control, and will now finish it`s operation. The integer 12 is written to the variable, and overwrites the value 1.

What was supposed t happen, was first Thread1 to add 8 to the sum (12), then for Thread2 to substract 3 from the sum (12-3 = 9). But the result became 12, because Thread2 interrupted Thread1, and then Thread1 overwrote Thread2.

### Possible race conditions:

There are some possible race conditions which might occur in the voting system if there are no mechanisms ensuring that different users can not interrupt eachother. Two of them are as follows:

A possible race condition might be that two voters votes for the same option at the same time. The addition of the variable is interrupted by another voter, and only one of the votes are counted, for instance:

voter1
vote_count ++

voter2
vote_count ++

Voter1 votes, but voter2 interrupts before voter1 was able to write to the count-variable. The vote from Voter2 is counted, then voter1 gets back the control, and the writing to the count-variable is completed. Voter1 overwrites voter2, and the only vote registrerd is actually the one from voter1.

An other possible race condition might be when a user, user1 logs in to the voting system. The username and password are checked and accepted. Before the system returns the accept, an other user, user2 wants to log on to the system. The user sends the username and password, which are also accepted, the system switches back and returns user1`s interface to user2. User2 will be able to log on as user1. An attacker might try to exploit a vulnerability like this by constantly send logon-request to the database, hoping that somebody is logging on at the same time. This form of exploiting a vulnerability is known as TOCTU (time of check to time of use) [4].

< reads username and password> -> Time of check
<Query the database >
<If username and password correct>
<return accepted to user> -> Time of use

The time between the system checks the requested username and password and return a true/false to the user is the time between the check and the use. Here an attacker can take advantage of the vulnerability, and get the accept for another user. Users who are not attackers might also experience this, but less likely.

**Avoiding race conditions**
To avoid users from interrupting other user`s operations, the critical section has to be locked by some kind of lock mechanism. The critical section from the cases above, are as follows: until the new number is written to the variable from the first race condition I mentioned, and from the username and password are checked, until the accepted / not accepted is returned to the userer (from second exaple). There are different ways to lock the critical section, but one way is to add a mutex, lock where the critical section starts, and unlock the mutex when the the critical section has ended. Using semaphores or event objects might also be an option.

**Conclusion**
Norway has been testing internet voting pilots the the last years. It is extremely important to ensure that all the votes are counted in the right manner, and that attackers are avoided from messing up the votes by exploiting TOCTOU. There are different ways to avoid race conditions.
By using a lock mechanism (mutex, semaphore...) , the operation is performed in an atomic manner, the votes are counted in the right way, without overwriting eachother, and users will not get access to other user`s votings by exploiting TOCTU.

**References:**

[1] BBC feilinformerer om det norske forsøket med internettstemming, **http://www.regjeringen.no/nb/dep/kmd/prosjekter/e-valg-2011-prosjektet/nyttomevalg/nytt-om-e-valg/2013/BBC-informerer-feil-om-det-norske-forsoket-med-internettstemming.html?id=764809**, 16.09.14

[2] Mark Dowd, John McDonald, Justin Schuh, The art of software security assessement, page 759.

[3] Microsoft, Description of Race Conditions and deadlocks http://support.microsoft.com/kb/317723 16.09.14

[4] Mark Dowd, John McDonald, Justin Schuh, The art of software security assessement, page 527.