Assignment 2

Race conditions with an electronic id solution

IMT 3501 Software Security

Christian Neßlinger (140153)

Race Conditions[1][2]

Race conditions are situations in which two or more different threads try to operate on the same resource and on the same time. The resource could nearly anything on the computer e. g.:
- a file in the file system or even in the memory
- an entry in a database
- registry flag
- a shared variable when you programming multithreading / asynchronous applications
- ...

Voting systems are in the past famous for vulnerabilities with race conditions e. g. the voting computers in America,[3][4] but also in Norway too.[5]

The following code snippets illustrate the working of race conditions with a shared variable in an asynchronous environment.

```
T0:Thread A get CPU time
T1: X = 112
T2: -- thread is paused because OS
T3:   scheduler gave the CPU time an
T4:   other thread  --
T5: VotesAngelaM = X
T6: VotesAngelaM = VotesAngelaM + 1
```

```
T0: Thread B get CPU time
T1: X = 112
T2: VotesAngelaM = X
T3: VotesAngelaM = VotesAngelaM + 1
```

In this case booth threads starting at the same time (T0) e. g. one thread got the voting signal over over a keyboard and the other one over a new RFID voting device, that is connected with the voting machine. What will happen in this example?
T0: Booth threads are initialized on the same time
T1: Booth threads got the actual number of votes for “Angela Merkel”
T2: Thread “A” is paused because the scheduler from the operations system gave the CPU time to another thread, meanwhile Thread “B” continue the execution
T3: Thread “A” is paused. Thread “B” alters the number of Votes for “Angela Merkel” to 113
T4: Thread “A” is paused. Thread “B” is already finished.
T5: Thread “A” get CPU time and continue the execution.
T6: Thread “A” alters the number of votes for “Angela Merkel” to 113

In this case “Angela Merkel” lose one vote because thread “A” doesn't set the correct value which is 114 (in general is that no problem for me that she will lose a vote, but from the software construction side is that of course a big problem).

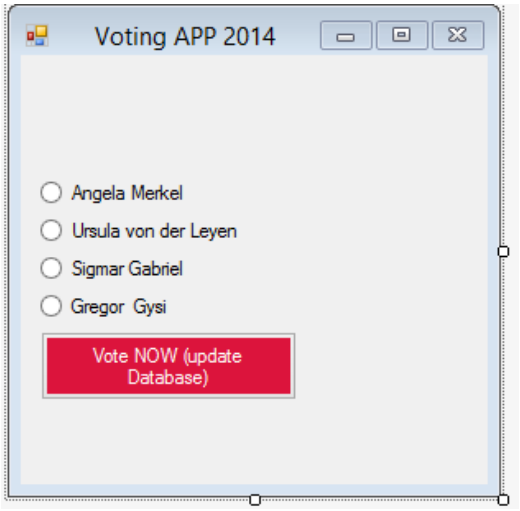
Possible race condition in an “e-voting” system #1

Typical the values from applications or web-platforms are stored in databases. Especially when the applications are bigger. Imagine a Voting app for a PC. You start the app two times (or more often) on the same machine and the app loads the same values from the voting database server. If you now make changes in application one and two to the same time (maybe not you, it could be an attacker) and the application try to store the values in

---

1 Dr. Clay Breshears, Intel Introduction to Parallel Programming video lecture series , http://www.youtube.com/watch?v=6ue-nw-bd9A
2 Flake, Halvar, The art of Software security assessment, ientifying and Preventing Software Vulnerabilities (book from the lecture)
3 http://www.outsidethebeltway.com/diebold_admits_that_its_e-voting_machines_drop_votes/
4 http://www.golem.de/0810/62877.html
5 http://www.forschung.ti.bfh.ch/de/news_details/article/luecke-beim-norwegischen-e-voting.html

the database you may don't know which application has finally executed the last writing operation in the database. The App could maybe look like this small example (from the principal working conditions)



Solution #1 for this problem (single application start):

The programmer could program the application in a way that only single start of the application is allowed.[6] Like the following small example, that checks if an application already running. (I tried this with Visual Studio 2013 an it worked well)

Form1.cs | Form1.cs [Entwurf]

VotingApp | VotingApp.Form1 | isOtherInstanceRunn

```
1-Verweis
public static bool isOtherInstanceRunning( System.Diagnostics.Process otherProcess)
{
    Process[] processes = Process.GetProcessesByName(Process.GetCurrentProcess().ProcessName);
    Process currentProcess = Process.GetCurrentProcess();
    for (int i = 0; i < processes.Length; i++)
    {
        if (processes[i].Id != currentProcess.Id)
        {
            if (processes[i].MainModule.FileName == currentProcess.MainModule.FileName)
            {
                otherProcess = processes[i];
                return true;
            }
        }
    }
    otherProcess = null;
    return false;
}

1-Verweis
private void Form1_Load(object sender, EventArgs e)
{
    Process pr = Process.GetCurrentProcess();
    if (isOtherInstanceRunning(pr) == true)
    {
        Application.Exit();
    }
    else
    {
        MessageBox.Show("Welcome to oure new voting app!");
    }
```

6 http://books.google.no/books?id=QKjrXE550vgC&pg=PA286&lpg=PA286&dq=c%23+mehrfache+ausf%C3%BChrung+verhindern&source=bl&ots=VOMO21zAie&sig=0Q10GfKXfkDmiygUwN-ZtB5E2wg&hl=de&sa=X&ei=ck0ZVIGSN6H-ygPcgYLYBQ&ved=0CB0Q6AEwAA#v=onepage&q=c%23%20mehrfache%20ausf%C3%BChrung%20verhindern&f=false

Solution #2 for this problem (using locks with Transactions in SQL)[7]

If the voting results are stored in a database (e. g. Microsoft SQL Server) a common way to preventing race conditions is to use transactions. With transactions you have different types of locking. If you want be sure you can make a transaction with an "Exclusive" lock. That means, that no other transaction can write and even read the resource that you using. All other transactions must wait until you are finished. Lets make an example how this can help you:

```
T0: Process A start the transaction,
      read the values from the database
      and lock the used resources
      exclusively for this transaction
      (e. g. possible persons to vote
      like Angela Merke, Ursula ...)
T1: Keep connection alive during the
      person thinks about which
      politician he should elect.
T2: Person press the "Vote NOW" button
T3: Update values with the same
      transaction in the database
T4: After a successful update, commit
      the   transaction, now the used
      resources are free for use
```

```
T0: Process B start the transaction, but
      the transaction from process A has
      already locked the resources
T1: Transaction is waiting in the
      scheduler from the SQL-Server
T2: Transaction is waiting in the
      scheduler from the SQL-Server
T3: Transaction is waiting in the
      scheduler from the SQL-Server
T4: Transaction is waiting in the
      scheduler from the SQL-Server
T5: resources are now free and the
      transaction from process B
      continues the execution,      read
      the values from the database  and
      lock the used resources
      exclusively for this transaction
      (e. g. possible persons to vote
      like Angela Merke, Ursula ...)
T6: Keep connection alive during the
      person thinks about which
      politician he should elect.
T7: Person press the "Vote NOW" button
T8: Update values with the same
      transaction in the database
T9: After a successful update, commit
      the   transaction, now the used
      resources are free for use
```

The big advantage of this system is, that the SQL Server manages nearly all important things automatic e. g. the scheduler which decide which transaction was faster and got the rights to lock the resources. But a disadvantage of the system is, if you using exclusively locks you could may cause a deadlock e. g. if an application don't close a transaction. An other disadvantage with this solution is, that in this special case only a serial vote is possible (which means one person is allowed to vote and after this one finished an other one is allowed to vote). That make things very slow. But of course there exist ways to parallelize this.

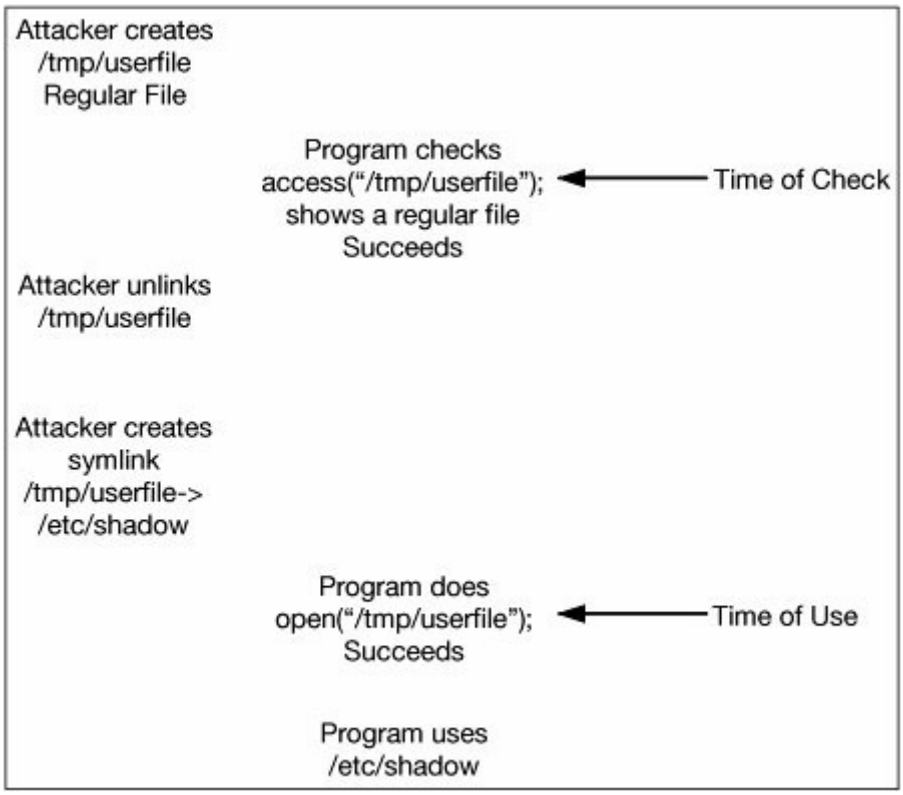
Solution #3

Combine solution #1 and #2 to have the advantages of booth systems.

Possible race condition in an "e-voting" system #2

You also could have problems with time-delay in the voting application, this problem is well known as TOCTOU (time of check to time of use). In the meanwhile between the "time of check" from a resource and the "time of use" an attacker could manipulate the resource without that the program will notice that. The following example should show the problem with that.

7 http://technet.microsoft.com/de-de/library/ms175519%28v=sql.105%29.aspx



From the book "The art of software security assessment", page 527

But how could the voting application from the first example have such a problem?
Lots of programs store values in files before they send the data over the network to the server, because you can't guarantee that a network connection is always stable and typical a network connection is a lot slower than work on the local file system. The advantage is, that the user could make lot of changes in the application and must only transfer the changes one time. You can say, that the application use the file as cache. But you can see in the picture above that a disadvantage is, that an attacker can try to change the file between the "check" and before the "use". Of course that's not easy for the attacker, but in general not impossible because he can try so slow down the system e. g. making some powerful calculations or flooding the network connection with lots of data (all things that have a big amount of I/O operations).

Solution for this problem:
After the "time of check", the software should lock the file e. g. with a write lock or even an encryption. In addition the application could make a file validation not only one time. The application should regular check that the "used file" is the "valid file".[8]

8 Flake, Halvar, The art of Software security assessment, ientifying and Preventing Software Vulnerabilities (book from the lecture)