David Vierheilig

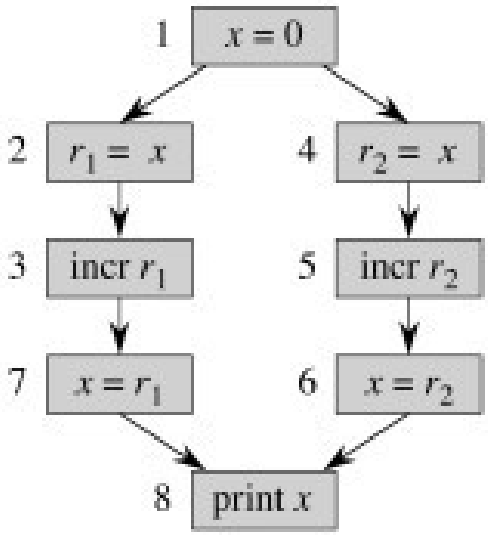
IMT3501
Software Security

Assignment 2 -
electronic id solution

What is a race condition?

A race condition is a line-up in which the result of one operation depends on the chronological behavior of single operations. If 2 or more threads wants to operate with the same source at the same time, we have a race condition, and probably we will receive a false result. In programming it is tried to prevent race conditions. The outcome of race conditions could be bugs. Race conditions are the result of badly programmed software. To prevent race conditions, semaphore can be used. This means that - if a single operation writes to a variable - it is locked. As long as the variable is locked, it cannot be accessed. Once the variable has been released, another single operation can write to it.**[1]**

I want to show it on an example



step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

In this example, both threads read from the variable x. Step 1 shows the variable x with the value "0". Step 2 shows that r1 gets the value "0". In step 3, r1 is incremented and so r1 has the value "1". Now r2 wants to use the variable x. r2 gets the value "0", this is step 4. It gets the value "0", because r1 hasn't written into variable x. Step 5 shows that r2 wants to increment, in source code probably something like that, "r2=r2+1;", r2 has now the value "1". In step 6, the value of r2 is written to x (x has the value "1"), and in step 7 the value of r1 is also written to x (x has the value "1"). Last step shows that x is displayed or printed. The fault in this example is that the right value at the end had to be "2" and not "1". To prevent this, you have to lock the variable, if one operation uses it. After the operation had used the variable x, it can be unlocked and another operation can use the variable.

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
LOAD Count	10			10
		LOAD Count	10	10
		SUB #1	9	10
		STORE Count	9	9
ADD #1	11			10
STORE Count	11			11

In this example we see that Thread A loads the content of the register with the value "10". Then Thread B loads the content of the register with value "10" and subtract it. After that, the content

of the register has the value "9". This value will be stored. Thread A works with its old content (value is still "10") of the register and summate it. So the stored value is "11". The right result had to be the value "10" and not "11".

Race Condition 1

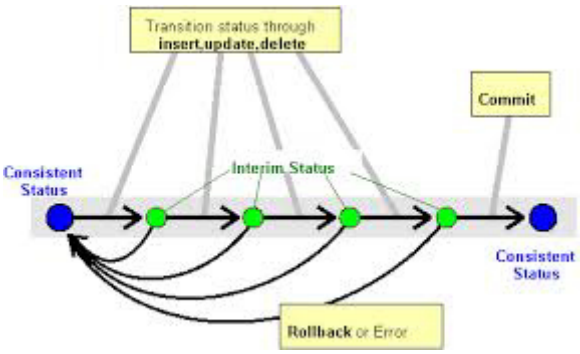
A user is logged into a website. The user starts the website twice, that means he has two open windows of the same website. Now he makes changes on both windows. If the user makes changes at the same time (both windows) and one window needs longer than the other, then the session data will be overwritten by means of the last change.

The built-in PHP session handler locks the session file for writing. That's important to prevent race conditions. The source code looks like this:

```
open("/var/lib/php/session/sess_XXXXXXXXXXXXXXXXXXXXXXXXXX", O_RDWR|O_CREAT,
0600) = 18
flock(18, LOCK_EX)                      = 0
fcntl64(18, F_SETFD, FD_CLOEXEC)        = 0
fstat64(18, {st_mode=S_IFREG|0600, st_size=11, ...}) = 0
pread64(18, "count|i:17;", 11, 0)       = 11
. . . .
pwrite64(18, "count|i:18;", 11, 0)      = 11
close(18)                               = 0
```

Flock function is the important line. For instance to write into the session file, gets access or is denied through the flock function. It makes sure that the file can be written at only and exclusively from 1 instance at the same time.**[2]** Unfortunately this case works only with a single web server. If you have more than one webserver, you have to handle this differently.

You will need a database. With the order "Rollback" the data manipulations were performed using INSERT, UPDATE, DELETE commands are undone. Then the data base is in the same condition before the database was at the beginning of the transaction. This is used if something went wrong during the transaction and that the database is consistent state. Consistent means that the data base is in a correct condition.



If an error occurs, the database can "reset" to the moment it was before the transaction. The changes are assumed if transaction was successful.

Race Condition 2:

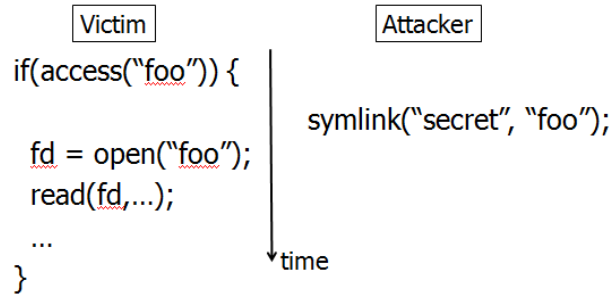
Time-of-check time-of-use (toctou) describes a fault in the program. It is a form of a race condition. The toctou describes a period between the check of a system status (time-of-check, for example write access for a file) and the use of the test result (time-of-use, for example change the write access for a file). The consequence is - that in the period between time-of-check and time-of-use, a file could be plant (for example by an attacker) -to change the result in the further program sequence.**[3]**

Examples:

We are on a typical website. A user is allowed to change particular sites, but the administrator of the application has the possibility to lock sites and so nobody can do changes on this site. The user is logged in with the help of an entry mask. The system checks if the user is allowed to change the site, the system checks his permission. In our case the user is allowed to do changes on this site (time-of-check). If the administrator locks the site and so bans changes, after the user hadn' t saved his changes yet the user will be able to save his changes. 1 minute later the user saves his changes and it will work, because at the time-of-check he had the permission to do changes on this site. At the use of time (time-of-use) the write access will be ignored.**[4]**

The same method can be used by attackers to change a value or content of a file. Following example:

Victim checks file -> it its good -> opens it(time-of-check). Attacker changes interpretation of file name. Victim reads secret file(time-of-use).



At the time-of-check the file was fine. Between the time-of-check and time-of-use the attacker writes some bad things in the code. At the time-of-use the system emenates that the file is fine and uses it. The system doesn't make a check of the file at the time-of-check and uses it, because it has already checked and the system thinks that the file is ok.

An attacker could manipulate the file between the time-of-check and time-of-use. The user or program doesn't recognize that the file was changed.

The impacts could be:

The attacker can get access to unauthorized resources.

The attacker can get write or read access to resources through the race condition.

Possible countermeasures can be used to prevent race conditions. You can ensure that there are locking mechanisms to protect the resource. To make sure that the resource, which is checked, is the same that is in use. You have to ensure that the resource is locked before it is checked (e. g. the used file has to be the same as the checked file).

Resources:

[1] http://www.cs.mtu.edu/~shene/NSF-3/e-Book/RACE/overview.html

[2] http://thwartedefforts.org/2006/11/11/race-conditions-with-ajax-and-php-sessions/

[3] http://deadliestwebattacks.com/tag/toctou/

[4] http://www.guidanceshare.com/wiki/Race_Condition_in_Time_of_Check,_Time_of_Use