Obligatory exercise 2

Alexander Gausdal (120927)

Date: 2014-09-17

What is a race condition? (example)

The operating system runs a lot of processes at the same time. Some of these processes shares common storages that they can read and write to. In these cases it's important that the processes works together in a proper manner, or else we might experience problems with race conditions. This example addresses a print spooler, which is used when a process wants to print a file (see figure 1). The process enters the file name of the file it wants to print into the spooler directory. A printer daemon checks for files in the spooler directory and ensures that they are printed out and removed from the directory after the print job is completed. The spooler directory and its daemon uses two shared variables to organize the printing jobs - *out* and *in*. *Out* points to the next file to be printed, and *in* points to the next free slot in the directory. We can see that process A and process B wants to print a file at the same time, which can cause problems like this [1]:

1. Process A stores the value of *in* (7) in a local variable.
2. Suddenly the scheduler switches to process B, because process A has run long enough.
3. Process B will do the same thing as process A - read the value of *in* (7).
4. Process B continues to run and stores the filename of the file it wants to print into slot 7, and increments *in* to be an 8.
5. Now that process B is done, process A will start running again and finish what it started.
6. Process A still has its local variable saved as the value 7, which means it will write its filename into slot 7, overwriting the filename of process B, then incrementing its local variable value and saving it in the *in* variable (7+1=8).

Process B will never get its file printed.

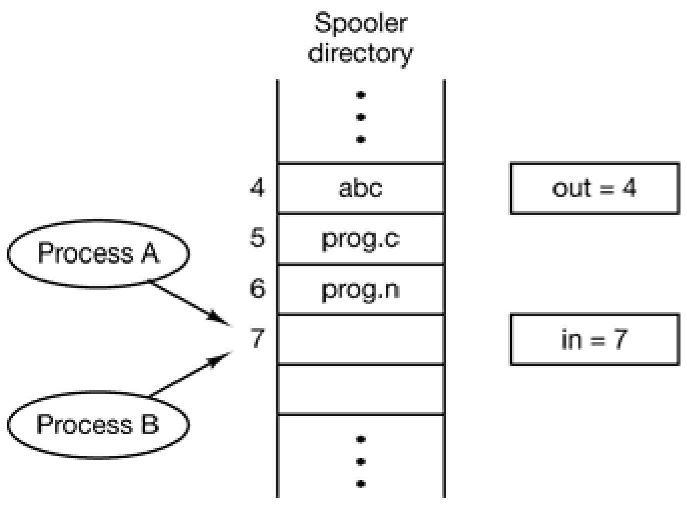


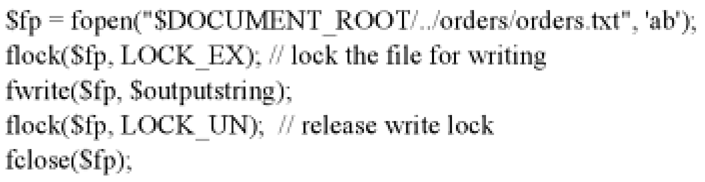
Figure 1 - Spooler directory - *Modern Operating Systems* by *Andrew S. Tanenbaum*

Intro

Let's say I run an online retail business. I have a website where customers can log in with their electronic IDs and start shopping. I've looked at two(+) race condition problems my IT-system could run into.

Ordering a product

The store used to have a flat file database, which stored all data in a single plain text file. The records in the file had no structural relationship between each other. When the store started to become popular it experienced a race condition problem. The problem occurred when two customers tried to order a product at the same time. Both customers will then call the fopen() function (PHP) and start writing to the file at the same time. What will the file look like when they're both done writing? Hard to say. To solve this problem the flock() function can be inserted to the PHP code, which locks the respective file when it's being written to [2]. Example code:



```
$fp = fopen("$DOCUMENT_ROOT/../orders/orders.txt", 'ab');
flock($fp, LOCK_EX); // lock the file for writing
fwrite($fp, $outputstring);
flock($fp, LOCK_UN);  // release write lock
fclose($fp);
```

Now we run into a new race condition problem, when two scripts are trying to acquire a lock at the same time.

The final solution to the problem will be upgrading to a database management system (DBMS). In MySQL there is a *transaction* statement that can be used to avoid race condition problems. It allows you to group your database queries into sets. If one of the queries in the set fails, the database is restored with the *rollback* statement. If no error occurred, the whole set of queries will be committed to the database [3].

Time-of-Check-to-Time-of-Use (TOCTTOU)

A TOCTTOU-attack exploits that a process can change the state of the system between the time an operation is authorized and the time that operation is performed. In other words: A process will invoke a security check which authorizes it to run the programs code, and then suddenly after the authorization another process intercepts and takes over the program [4]. This is a race condition problem in UNIX. The store's UNIX-based application server needs to be protected against this kind of attack, which can lead to an adversary getting root access to the server.

Example: Set-UID program

```
if (!access("/tmp/X", W_OK)) {
  /* the real user ID has access right */
  f = open("/tmp/X", O_WRITE);
  write_to_file(f);
}
else {
  /* the real user ID does not have access right */
  fprintf(stderr, "Permission denied\n");
}
```

First access() checks the real user ID/group ID, and returns 0 if permission to access the file is granted. Access() is a system call typically used by the set-UID root program before accessing file. It will check the real user ID, and not the effective user ID. Open() will check the effective user ID, which is root, because the program is running with the root privilege [5]. For our race condition attack we will try to make /tmp/X represent two different files for the access() and open() calls. After the access() call we will try to change /tmp/X to a symbolic link to /etc/passwd. To make this happen we exploit TOCTTOU, which is the time window between our access and open calls. This short time window will be available if the scheduler decides on a context switch just after the access() call and then grants our process the CPU. Since there's no way we can modify the vulnerable program, we have to run our program in parallel with the target program, and it's really hard to get a perfect timing where our attacking program actually changes the path, because of the very small time window. Therefore it will be necessary to create many attacking processes to complete the attack. Job control signals will assure that the Set-UID program starts and stops in a constant loop. The attack might be more likely to succeed if the system is running slowly, this can be achieved by starting a lot of CPU-intensive programs or flooding the system with network data [6].

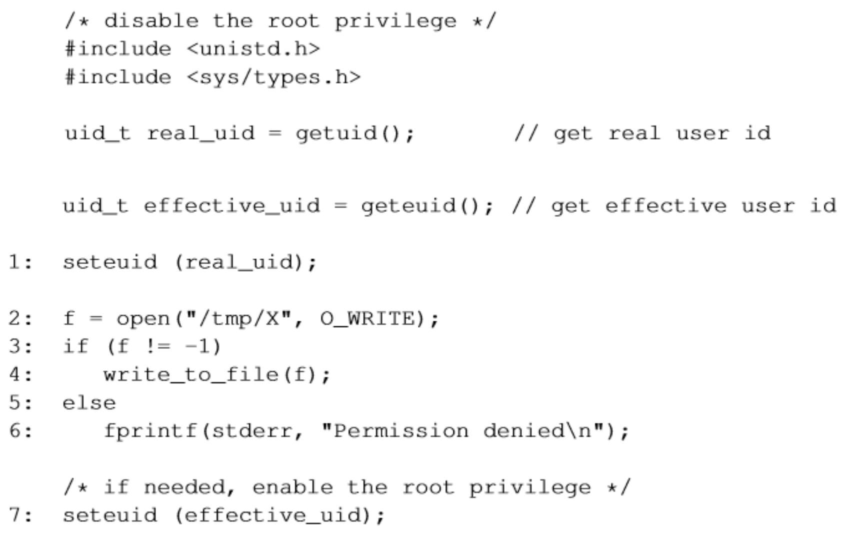
Countermeasures:

- If we can have both access() and open() in the same system call, we would avoid the race condition because context switches can't be done inside a system call. There are some ways of trying to manipulate this, but nothing that can be found in the POSIX standard.
- We could make it really hard for the attacker by making him/her go through a lot of access()/open() iterations. Here the attacker actually needs to win five race conditions to be successful (between the lines: 1-2, 2-3, 3-4, 4-5 and 5-6).

```
1: if (access("tmp/X", O_RDWR))    goto error handling
2: else f1 = open("/tmp/X", O_RDWR);
3: if (access("tmp/X", O_RDWR))    goto error handling
4: else f2 = open("/tmp/X", O_RDWR);
5: if (access("tmp/X", O_RDWR))    goto error handling
6: else f3 = open("/tmp/X", O_RDWR);

Check whether f1, f2, and f3 has the same i-node (using fstat)
```

- If we think about the principle of least privilege, we realize that the open() system call might have too much power. The access() call is supposed to limit open()'s power, but as we've seen this isn't always the case. So if we just reduce the power of the open() system call we could potentially avoid our race condition problem. If we temporarily don't need our program to be as powerful as it is by default, or just don't need it to be ever, we should take away its power. We use the *seteuid()* and *setuid()* system calls to do this. In the code underneath you can see that we set the effective user ID equal to the real user ID. Open() always checks the effective user ID (or group ID) only, for file permission.



```
/* disable the root privilege */
#include <unistd.h>
#include <sys/types.h>

uid_t real_uid = getuid();       // get real user id


uid_t effective_uid = geteuid(); // get effective user id

seteuid (real_uid);

f = open("/tmp/X", O_WRITE);
if (f != -1)
   write_to_file(f);
else
   fprintf(stderr, "Permission denied\n");

/* if needed, enable the root privilege */
seteuid (effective_uid);
```

Extra:

Attackers might also try to exploit the TOCTTOU vulnerability by using it to log into the website as another user. This could be achieved by spamming logon requests to the authentication database and hope that another user tries to log on at the exact same time. The database might struggle with a race condition here and mix up the two login attempts, making user1 log in as user2. Databases usually have solid integrated methods that prevents this form of TOCTTOU-attack from being successful.

Bibliography

[1] A. S. Tanenbaum, Modern Operating Systems.

[2] L. Welling and L. Thomson, "PHP and MySQL Web Development," [Online]. Available: http://books.google.no/books?id=WkqmA1wiqCwC&pg=PT148&dq=race+condition+mysql&hl=no&sa=X&ei=T4sYVO_9BsrMyAOcv4KQDA&ved=0CC8Q6AEwAQ#v=onepage&q=race%20condition&f=false.

[3] "MySQL Transaction," [Online]. Available: http://www.mysqltutorial.org/mysql-transaction.aspx.

[4] Wikipedia, "Time of check to time of use," [Online]. Available: http://en.wikipedia.org/wiki/Time_of_check_to_time_of_use.

[5] T. Jaeger, "Operating System Security," [Online]. Available: http://books.google.no/books?id=P4PYPSv8nBMC&pg=PA49&dq=TOCTTOU&hl=no&sa=X&ei=EyAYVOaAB-K7ygOt94GoDg&ved=0CDUQ6AEwAw#v=onepage&q=TOCTTOU&f=false.

[6] S. University, "Race Condition Vulnerability," [Online]. Available: http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Race_Condition.pdf.

[7] "fopen," [Online]. Available: http://php.net/manual/en/function.fopen.php.