Race Condition Vulnerability

1 Race Condition Vulnerability

- The following code snippet belongs to a privileged program (e.g. `Set-UID` program), and it runs with the root privilege.

```
1: if (!access("/tmp/X", W_OK)) {
2:   /* the real user ID has access right */
3:   f = open("/tmp/X", O_WRITE);
4:   write_to_file(f);
5: }
6: else {
7:   /* the real user ID does not have access right */
8:   fprintf(stderr, "Permission denied\n");
9: }
```

  - The `access()` system call checks whether the *real* user ID or group ID has permissions to access a file, and returns 0 if it does. This system call is usually used by set-uid program before accessing a file on behalf of the real user ID (not the effective user ID).
  - The `open()` system call also conducts access control, but it only checks whether the *effective* user ID or group ID has permissions to access a file.
  - The above program wants to write to file `"/tmp/X"`. Before doing that, it ensures that the file is indeed writable by the real user ID. Without such a check, the program can write to this file regardless of whether the real user ID can write to it or not, because the program runs with the root privilege (i.e., the effective user ID checked by `open()` is root).

- Assume that the above program somehow runs very very slowly. It takes one minute to run each line of the statement in this program. Please think about the following:
  - Can you use this program to overwrite another file, such as `/etc/passwd`?
  - You cannot modify the program, but you can take advantage of that one minute between every two statements of the program.
  - The `/tmp` directory has permission `rwxrwxrwx`, which allows any user to create files/links under this directory.
  - Hint: `/tmp/X` does not need to be a real file, it can be a symbolic link.

- Attack Ideas:
  - If we let `/tmp/X` point out `/etc/passwd` before Line 1, the `access()` call will find out that the real user ID does not have the right to modify `/etc/passwd`; hence, the execution will go to the `else` branch. Therefore, before Line 1, `/tmp/X` must be a file that is writable by the real user ID.
  - Obviously, if we do not do anything after Line 1, the `/tmp/X` will be opened, and the attacker cannot gain anything.

  - Let us focus on the time window between Line 1 and Line 3. Since we assume that the program runs very slowly, we have a one-minute time window after Line 1 and before Line 3. Within this time window, we can delete `/tmp/X` and create a symbolic link used the same name, and let it point to `/etc/passwd`.
  - What will happen if we do the above between Line 1 and Line 3.
    - The program will use `open()` to open `/etc/passwd` by following the symbolic link.
    - The `open()` system call only checks whether the effective user (or group) ID can access the file. Since this is a `Set-UID` root program, the effective user ID is `root`, which can of course read and write `/etc/passwd`.
    - Therefore, Line 4 will actually write to the file `/etc/passwd`. If the contents written to this file is also controlled by the user, the attacker can basically modify the password file, and eventually gain the root privilege. If the contents are not controlled by the user, the attacker can still corrupt the password file, and thus prevent other users from logging into the system.

- Back to reality: the program runs very fast, and we do not have that one-minute time window. What can we do?

- Race-Condition Attack:

  - Cause `/tmp/X` to represent two different files for the access and open calls?
  - Before `access(/tmp/X, W_OK))`, the file `/tmp/X` is indeed `/tmp/X`.
  - After `access(/tmp/X, W_OK)`, change */tmp/X* to */etc/passwd*.
  - How is this possible?
    - There is a short time window between `access()` and `open()`.
    - The window between the checking and using: Time-of-Check, Time-of-Use (TOCTOU).
    - CPU might conduct context switch after `access()`, and run another process.
    - If the attack process gets the chance to execute the above attacking steps during this context switch window, the attack may succeed.
    - Since we cannot guarantee that a context switch occurs between the Line 1 and 3 of the target program, even if the attack program gets the chance to run during the context switch window, the attack may not work. However, if running once does not work, we can run the attack and the target program for many times.

- Improving Success Rate: The most critical step of a race-condition attack must occur within TOCTOU window. Since we cannot modify the vulnerable program, the only thing that we can do is to run our attacking program in parallel with the target program, hoping that the change of the link does occur within that critical window. Unfortunately, we cannot achieve the perfect timing. Therefore, the success of attack is probabilistic. The probability of successful attack might be quite low if the window is small. How do we increase the probability?

  - Slow down the computer by running many CPU-intensive programs.
  - Create many attacking processes.

- Another example (a set-uid program)

```
file = "/tmp/X";
fileExist = check_file_existence(file);
if (fileExist == FALSE){
   // The file does not exist, create it.
   f = open(file, O_CREAT);
}
```

  - In `Unix`, to create a file, we use `open()` system call.
  - `open(file, O_CREAT)` creates a file if the file does not exist; if the file already exists, it simply opens the file.

- Why is it vulnerable?
  - Race condition: make the file non-existing during the check, and make it point to `/etc/passwd` after the check.

2 Countermeasures

- Approaches
  - Turn check and use operations into one atomic operation: if we can use one system call to achieve both check and use purpose, we will not have race conditions. In most operating systems, system calls cannot be preempted by another user-space process, therefore, there will not be a context switch within a system call invocation.
  - Ensure that the same file name points to the same file (i.e. the same I-node) during the check and use operations.
  - Make the probability of winning the race condition very very low.
  - Do not use too much privilege if unnecessary.
- **Use atomic operation**
  - If a system call can do both checking to use within the same system call, it is safe, because context switch will not happen within a system call.
  - `open(file, O_CREAT | O_EXCL)` can check file existence and file open in an atomic operation (i.e., within the same system call). it returns error if the file already exists; otherwise, it creates the file. The `mkstemp()` function generates a unique temporary filename from template. This function uses `open()` with `O_EXCL` flag, to prevent the race condition problem.
  - Similarly, we can create another option for `open()` to conduct `access()` and `open()` together. Although such an option does not exist in the POSIX standard, technically, this can be done. Namely, we can define an option called `O_REAL_USER_ID`. When we call `open()` using `open(file, O_WRITE | O_REAL_USER_ID)`, we ask `open()` to check both the effective user ID and the real user ID, and only open the file when both IDs have the permissions to open the file. In practice, pushing the POSIX standard committee to accept this new option might not be easy.
- **Check-use-check-again approach**

- `lstat(file, &result)` can get the status of the file. If the file is a symbolic link, it returns the status of the link (not the file pointed to by the link). We can use `lstat()` to check the file status before the TOCTOU window and then do another check after the window. If the results are different, we can detect the race condition. Let us see the following solution:

```
    struct stat statBefore, statAfter;

1:  lstat("/tmp/X", &statBefore);

2:  if (!access("/tmp/X", O_RDWR)) {
      /* the real UID has access right */
3:    f = open("/tmp/X", O_RDWR);
4:    lstat("/tmp/X", &statAfter);

5:    if (statAfter.st_ino == statBefore.st_ino)
6:    { /* the I-node is still the same */
7:       Write_to_file(f)
8:    }
9:    else perror("Race Condition Attacks!");
10: }
11: else fprintf(stderr, "Permission denied\n");
```

- However, the above solution does not work (a new race condition exists between `open()` and the second `lstat()`). To exploit this vulnerability, the attacker needs to conduct two race condition attacks, one between line 2 and 3, and the other between line 3 and 4. Although the probability of winning both race is much lower than the previous case, it is still possible.
- To fix the problem, we want to use `lstat()` on the file descriptor `f`, rather than on the file name. Although `lstat()` cannot do that, `fstat()` can do it.

```
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>
  #include <stdio.h>

  int main()
  {
      struct stat statBefore, statAfter;

1:    lstat("/tmp/X", &statBefore);
2:    if (!access("/tmp/X", O_RDWR)) {
        /* the real UID has access right */
3:      int f = open("/tmp/X", O_RDWR);
4:      fstat(f, &statAfter);
5:      if (statAfter.st_ino == statBefore.st_ino)
6:      { /* the I-node is still the same */
7:        write_to_file(f);
8:      }
9:      else perror("Race Condition Attacks!");
10:   }
11:   else fprintf(stderr, "Permission denied\n");
12: }
```

- Question: Are there race conditions between `lstat()` and `fstat()`? How about using symbolic link (e.g. to /etc/shadow) at line 1, then quickly switch to /tmp/X before line 2, and then quickly switch back to the symbolic link before line 3?

  Answer: This attack is impossible. The function call `lstat("/tmp/X",...)` returns the status of the link if the `"/tmp/X"` is a symbolic link, instead of the status of the file pointed to by the link. In other words, when `/tmp/X` points to `/etc/shadow`, the i-node returned by `lstat(/tmp/X,...)` will be the i-node of `/tmp/X`, while the i-node returned by `fstat(f,...)` will be the i-node of the actual file (in this case, it will be the i-node of `/etc/shadow`). These two i-nodes are different even if `/tmp/X` points to `/etc/shadow`.
- Note: not all the calls have these two versions, one for the filename, and the other for the file descriptor (Think: if `access()` can also work on file descriptor, the solution will be much easier).

- **Check-use-repeating approach:** Repeat access and open for several iterations. In the following example, the attacker needs to win five race conditions (between 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)
```

- **Based on the Principle of Least Privilege:**
  - In the program that uses `access()` and `open()`, we realize that `open()` is too powerful than what we need (it only checks the effective user id); that is why we have to use `access()` to make sure that we do not misuse the power. The lesson that we learned from the race-condition attack is that such a checking is not always reliable.
  - Another approach to prevent a program from misusing the power is to not give the program the power. This is exactly the essence of the principle of least privilege: if we temporarily do not need the power, we should disable it; if we permanently do not need the power, we should just discard it. Without the power, even if the program makes some mistakes, the damage will be much reduced.
  - In `Unix`, we can use `seteuid()` or `setuid()` system calls. to disable/enable or delete the power.

```
    /* 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);
```