DirtyCOW (CVE-2016-5195)

Aung Khant Ko

DirtyCOW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux kernel. The vulnerability has existed for over a decade, but the patch was pushed only in October 2016. DirtyCOW is caused by a race condition vulnerability, that can be exploited to escalate user privileges, compromising confidentiality and integrity of a system.

Introduction

DirtyCOW (CVE-2016-5195) is a race condition vulnerability reported by Phil Oester in late 2016. The vulnerability was caused by the Linux kernel improperly handling mappings to private copy-on-write pages. This vulnerability can be exploited to make arbitrary writes to privileged files on a system. In this paper, we will explore some preliminary concepts such as race conditions, memory management and copy-on-write before we study the proof-of-concept code for the DirtyCOW exploit.

Race Conditions

Race conditions are bugs in concurrent programs, such as operating systems, where multiple threads or processes read from and write to a shared memory region. The result is non-deterministic and depends on the order of execution (Carr et al. 1).

Race conditions are best demonstrated by examples. Examine the code for a concurrent program in C below.

race.c

```c
#include <stdio.h>
#include <pthread.h>

void *race(void *arg) {
        int *x = (int *) arg;
        for (int i = 0; i < 100000; i++) {
                (*x)++;
        }
}

int main(void) {
        pthread_t t1, t2;
        int sharedInt = 0;

        pthread_create(&t1, NULL, race, (void *) &sharedInt);
        pthread_create(&t2, NULL, race, (void *) &sharedInt);

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        printf(``%d\n'', sharedInt);
        return 0;
}
```

The **main** function creates two threads **t1**, **t2** and initializes an integer **sharedInt** with the value **0**. The two children threads begin execution by invoking the **race** function with the address of

sharedInt as the argument. The **race** function dereferences the pointer to **sharedInt** and increments it 100,000 times. The parent thread waits for threads **t1** and **t2** to finish on lines 18 and 19 and finally prints the value of **sharedInt** on line 20.

Compile the program using the command below.

```
gcc -pthread race.c -o race.out
```

The **-pthread** flag defines the macros required for using the POSIX threads (pthreads) library. After compilation, run the **race.out** executable a few times.

Output of ./race.out

```
$ ./race.out
107264
$ ./race.out
135685
$ ./race.out
105677
$ ./race.out
131199
```

race.out prints seemingly random numbers each time. Logically, each thread executes the **race** function and the value of **x** would be 200,000 when the both threads complete. This non-deterministic behaviour is caused by the increment operation (as well as several other operations) not being atomic.

An atomic operation is an operation that is guaranteed to be executed uninterrupted (Arpaci-Dusseau). Single CPU instruction are always atomic. The increment operation requires three CPU instructions - load data from memory into a register, increment the value in register and write value to memory. Because ordering is not enforced, there is no way of knowing the actual order of execution. One possible ordering can be seen below, where the final value of **x** is **1**, instead of the desired value **2**.

t1	**t2**
load value into register **x = 0, %eax = 0**	
increment value in register **x = 0, %eax = 1**	
	load value into register **x = 0, %eax = 0**
	increment value in register **x = 0, %eax = 1**
	write value from register to memory **x = 1, %eax = 1**
write value from register to memory **x = 1, %eax = 1**	

Simple race conditions, such as the one in the example above, can be easily identified and fixed by introducing mutual exclusion. However, most race condition vulnerabilities are far more subtle. Detecting race conditions is an NP-hard problem, meaning there are no known efficient algorithms for pinpointing them (Carr et al. 12).

The DirtyCOW exploit also follows a similar structure with two racing threads. On their own, both threads execute non-malicious code. However, when the threads are running simultaneously, the kernel can be tricked into performing a malicious action.

Memory Management

Memory management is an important part of operating systems. Modern operating systems accomplish this by virtualising memory to achieve transparency, efficiency and protection (Arpaci-Dusseau). Each process has its own virtual address space where the stack, heap and code regions appear contiguous. However, for reasons stated earlier, the virtual address space is broken down into chunks (known as pages), which are then mapped into the physical address space. The mapped pages may or may not be continuous in physical memory. In fact, unused virtual memory pages won't be mapped at all. The virtual address to physical address translation is done by the Memory Management Unit (MMU), which is part of the CPU.

Copy-On-Write

Copy-On-Write (COW), also known as shadowing, is a technique for efficient sharing of resources while preserving integrity and consistency. When a resource is accessed for read operations, the COW system simply returns a pointer to the data. Only when a process begins writing to the file, the system creates a private copy of the data for that process (Kasampalis 19).

The DirtyCOW exploit involves creating a copy-on-write mapping in the process's virtual address space to a privileged file. As long as the process is only reading from the file, the page in the process's virtual memory will map directly to the file on disk. When a process attempts to write to the file, the kernel will make a private copy of the underlying file and update the mapping to point to the new private copy.

DirtyCOW Proof of Concept

We'll be examining the proof-of-concept code found at `https://github.com/dirtycow/dirtycow.github.io/blob/master/dirtyc0w.c`. The **main** function takes two arguments - the path to the privileged file, and the contents to overwrite the file with. The **mmap** system call creates a mapping to the file in the process's virtual memory. The **PROT_READ** argument specifies that the mapping is read-only and the **MAP_PRIVATE** argument enables copy-on-write mapping.

> **MAP_PRIVATE (man mmap(2)):**
> Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the mmap() call are visible in the mapped region.

creating a file-backed mapping - snippet from dirtyc0w.c

```
/* snippet */
f = open(argv[1], O_RDONLY);
fstat(f, &st);
name = argv[1];
map = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, f, 0);
/* snippet */
```

The first thread invokes the **madviseThread** function. The function loops over the **madvise** system call, which is used to give advice about memory to the kernel. **madvise** takes three arguments - the start of the address range, the length of the range in bytes and the advice. The **madviseThread** function, repeatedly gives the **MADV_DONTNEED** advice to the kernel for the first 100 bytes of the mapped region.

> **MADV_DONTNEED (man madvise(2)):**
> Do not expect access in the near future. (For the time being, the application is finished with the given range, so the kernel can free resources associated with it.)
> After a successful MADV_DONTNEED operation, the semantics of memory access in the specified region are changed: subsequent accesses of pages in the range will succeed, but will result in either repopulating the memory contents from the up-to-date contents of the underlying mapped file

madviseThread - snippet from dirtyc0w.c

```
void *madviseThread(void *arg)
{
  char *str;
  str=(char*)arg;
  int i, c=0;
  for(i=0; i<100000000; i++)
  {
    c+=madvise(map, 100, MADV_DONTNEED);
  }
  printf(``madvise %d\n\n'',c);
}
```

The **MADV_DONTNEED** advice tells the kernel that the process won't be accessing memory in that range anytime soon, allowing the kernel to free up related resources. For example, if the address range is mapped to a private copy-on-write page, the kernel will simply delete that dirty private page. If the process attempts to access the memory again, the kernel will reload the contents from the underlying file (Chao-tic).

The second thread executes the **procselfmemThread** function. This function opens the pseudo-file **/proc/self/mem** and repeatedly writes to the mapped memory region. The function writes to the **/proc/self/mem** file because DirtyCOW relies on the kernel's implementation of virtual memory accesses between processes. Writing directly to the privileged file will result in a segmentation fault. However, attempting to write to a read-only mapping in **/proc/self/main** will trigger the kernel's memory handlers which can be exploited (Chao-tic).

procselfmemThread- snippet from dirtyc0w.c

```
void *procselfmemThread(void *arg)
{
  char *str;
  str=(char*)arg;
  int f=open(``/proc/self/mem'',O_RDWR);
  int i,c=0;
  for(i=0;i<100000000;i++) {
    lseek(f, (uintptr_t) map, SEEK_SET);
    c += write(f,str,strlen(str));
  }
  printf(``procselfmem %d\n\n'', c);
}
```

Everything is a file on Unix based system. Different file types have different implementations for input/output operations. The read/write implementations for **/proc** pseudo-files are defined in **/fs/proc/base.c** of the Linux kernel source code. Whenever a process tries to **write** to a pseudo-file, a call to **mem_write** is made. The function will then execute a series of intermediary functions before reaching vulnerable function, **faultin_page**.

function trace starting at mem_write

```
mem_write (/fs/proc/base.c)
  mem_rw (/fs/proc/base.c)
    access_remote_vm (/mm/memory.c)
      __access_remote_vm (/mm/memory.c)
        get_user_pages_remote (/mm/gup.c)
          __get_user_pages_remote (/mm/gup.c)
            __get_user_pages_locked (/mm/gup.c)
              __get_user_pages (/mm/gup.c)
                faultin_page (/mm/gup.c)
```

The vulnerable section of code can be seen below.

Vulnerable code in /mm/gup.c

```
static long __get_user_pages (...) {
  /* snippet */
  do {
retry:
    cond_resched(); /* please rescheule me!!! */
    page = follow_page_mask(vma, start, foll_flags, &page_mask);
    if (!page) {
      int ret;
      ret = faultin_page(tsk, vma, start, &foll_flags, nonblocking);
      switch (ret) {
        case 0:
          goto retry;
  /* snippet */
}
```

The function **follow_page_mask** attempts to retrieve a page in memory with the given arguments. The **foll_flags** argument contains information about the lookup behaviour - such as how the page is intended to be used.

Since the process is attempting to write to the page, **foll_flags** will have the **FOLL_WRITE** bit set. However, the mapping is marked as read-only and the memory access permissions are being violated. **follow_page_mask** fails to retrieve the page and returns **NULL**.

Instead of throwing a segmentation fault and terminating the process, the function **faultin_page** attempts to resolve the page fault. The handler does this by creating a private copy of the page because the underlying file is read-only and was mapped with copy-on-write enabled. **faultin_page** also unsets the **FOLL_WRITE** bit to prevent an infinite loop in **__get_user_pages** (Chao-tic). The function finally returns **0** and **__get_user_pages** retries.

Under normal circumstances, the process will have its own private page that it can write to. However, the DirtyCOW exploit has a racing thread that's deleting the copy-on-write pages. After **faultin_page** successfully creates a new private page, **__get_user_pages** retries. If **madvise** with **MADV_DONTNEED** is executed right after **faultin_page** completes, the newly copied private page gets deleted and **follow_page_mask** returns **NULL**. The **faultin_page** function executes for the second time, but without the **FOLL_WRITE** bit set. This time, the handler returns the page to the actual underlying file, assuming that the process won't write to it. The privileged file is mapped directly into the process's virtual memory, which the malicious process can write to. The table below shows the order of execution resulting in a write to the privileged file.

procselfmemThread	**madviseThread**
retrieve mapped page **follow_page_mask(...)** returns **NULL** **FOLL_WRITE = 1**	
faultin_page(...) triggered **FOLL_WRITE = 1**	
handler resolves this by creating a private COW page **FOLL_WRITE = 1**	
faultin_page(...) unsets **FOLL_WRITE** bit **FOLL_WRITE = 0**	
faultin_page(...) returns **0** **FOLL_WRITE = 0**	
	COW page is deleted **madvise(.., MADV_DONTNEED)**
retrieve mapped page **follow_page_mask(...)** returns **NULL** **FOLL_WRITE = 0**	
faultin_page(...) triggered **FOLL_WRITE = 0**	
handler resolves this by returning the underlying file **FOLL_WRITE = 0**	

Patch

The patch was released on October 13th, 2016, authored by Linus Trovalds. The patch is relatively short and can be found at `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3a`. Instead of removing the **FOLL_WRITE** bit, a new flag **FOLL_COW** was introduced. The **FOLL_COW** bit indicates whether a copy-on-write has been completed for the page. This preserves the lookup behaviour and also prevents the process from getting stuck in an infinite loop.

According to the commit message, the kernel developers were aware of the issue and Linus attempted to fix it in August 2005. Due to problems with IBM's S390 architecture, the fix was reverted. Rolling back to a vulnerable state sounds horrific, but Linus stated that the race was just theoretical back then. However, computers became faster making it possible to trigger the vulnerability.

Conclusion

The fact that the DirtyCOW vulnerability went unpatched for a decade raises several questions. Why was it left unpatched? Are there more unpatched or undiscovered vulnerabilities in the Linux kernel? Is the Linux kernel even secure?

As Ken Thompson once stated, it is impossible to trust code written by someone else (763). But the majority of the people do not have the time or the ability to write entire operating systems from scratch or manually review and compile source code. These users have no choice but to trust that the people delivering software are not trying to actively exploit their systems.

DirtyCOW is just one of many privilege escalation vulnerabilities in the Linux kernel. Other than having a fancy name, there is nothing special about CVE-2016-5195. Vulnerabilities of all sorts, ranging from denial of service to race conditions, continue to plague software. According to CVEdetails, in 2019, there were 170 vulnerabilities discovered in the Linux kernel, 117 in Mac OS X and 357 in Windows 10. Software is going to remain vulnerable, even those built with the best intentions. Nevertheless, it is important to study vulnerabilities in order to fix and prevent them from occuring again.

Works Cited

Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau “Operating Systems: Three Easy Pieces”, August 2003.

Carr, Steve, et al. “Race Conditions: A Case Study” *The Journal of Computing in Small Colleges 17(1)*, September 2001.

Chao-tic “Dirty COW and why lying is bad even if you are the Linux kernel”, May 2017

Kasampalis, Sakis “Copy On Write Based File Systems Performance Analysis And Implementation”, October 2010

Thompson, Ken “Reflections on Trusting Trust” *ACM Turing Award Lectures*, vol. 27, no. 8, August 1984

Torvalds, Linus “mm: remove gup_flags FOLL_WRITE games from __get_user_pages()”, October 2016, `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3a`