****

**EE6042- Host and Network Security**

**Group Project**

Due Date: Friday the 21st of April, 2017

Course: Information and Network Security MENG

Department: Electronic & Computer Engineering

Student: Hugh Mullins, Student ID: 16106989

Student: Sunday Jimoh, Student ID: 16090462

Student: Conor McDonnel, Student ID: 16120604

Student: Raksha Ramdas, Student ID: 16052439

Student: Samaa Alsafwani, Student ID: 16087941

**Table of Contents**

[1 Dirty Copy-on-Write (CVE-2016-5195) 4](#_Toc480210213)

[1.1 Introduction 4](#_Toc480210214)

[1.2 Impacts of Exploiting the Dirty COW Vulnerability 4](#_Toc480210215)

[1.3 How to execute a Dirty COW Exploit 5](#_Toc480210216)

**Table of Figures**

[Figure 1: Analysis of the impact of The Dirty COW exploit on a system 5](#_Toc480203924)

# Dirty Copy-on-Write (CVE-2016-5195)

## Introduction

Dirty copy-on-write (Dirty COW) is a security vulnerability present in the Linux kernel since version 2.6.22 which was released in September, 2007 and it has been exploited actively since October, 2016. Its CVE designation is CVE-2016-5195. It affects all major vendors of Linux including Ubuntu, Debian, Red Hat, Fedora and even Android, which is also powered by the Linux kernel. It was discovered by a Linux security researcher, Phil Oester.

Copy-on-write (COW), sometimes referred to as implicit sharing or shadowing, manages memory resources and allows for more than one process to share a page until a user writes to the page, this is known in programming as marking a page dirty.

Dirty COW is a local privilege escalation bug that exploits a race condition in mm/gup.c in the Linux kernel 2.x through 4.x before version 4.8.3, taking into consideration the way the Linux kernel’s memory subsystem handles the copy-on-write (COW) breakage of private read-only memory mappings. An unprivileged user or a local attacker could exploit this flaw to gain write access to otherwise read-only memory mappings and thus increase their privileges on the system. This vulnerability could be used by the attacker to modify existing setuid files with instructions to gain administrative privileges. The exploitation of the Dirty COW bug leaves no traces of anything abnormal in the system logs.

## Impacts of Exploiting the Dirty COW Vulnerability

Some of the impacts of exploiting the Dirty COW bug on Linux based systems include:

* The flaw allows attackers with local system accounts to modify on-disk binaries, bypassing the standard permission mechanisms that would prevent modification without an appropriate permission set. This is achieved by the racing the madvise (MaADV\_DONTNEED) system call while having the page of the executable mmapped in memory.
* The Linux kernel contains many binaries which are read-only, and can only be modified or written to by a user of higher permissions, such as the root. When an attacker escalates system privileges using the Dirty COW exploit they can change files, such as /bin/bash, so that it performs additional, unexpected functions such as a key logger to steal passwords and other sensitive information from the legitimate users.
* Although it is a local privilege escalation bug, it can be used by remote attackers in conjunction with other known exploits that allow remote execution of non-privileged code to achieve remote root access on an affected computer. The attack itself does not leave any traces in the system log.
* It can also be used to obtain root permissions in any Android device up to Android version 7.

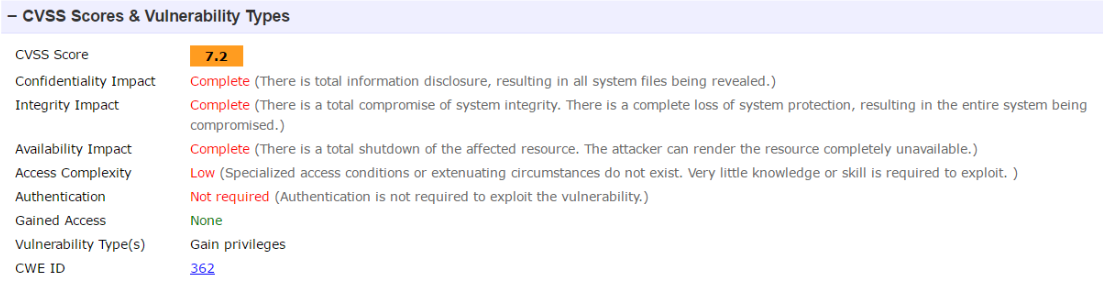


Figure 1: Analysis of the impact of The Dirty COW exploit on a system

## How to execute a Dirty COW Exploit

Some of the known techniques of exploiting the Dirty COW bug include:

* This first Dirty COW exploit seen in the wild takes two arguments a filename and a string to be written to the file.  The file specified is opened as read-only and is then mmap’ed into the current process read-only and with [MAP\_PRIVATE](http://man7.org/linux/man-pages/man2/mmap.2.html) specified to create a private COW mapping.  It then starts two threads. The first thread opens /proc/self/mem read-write and loops millions of times writing the specified data to the command line at the offset of the mmap’ed file. Writing to the /proc/self/mem triggers COWs, but since these pages are marked private, the memory pages written to will not be written back to the mmapped file. The second thread loops multiple times, calling the madvise system on the same offset w/ the flag MADV\_DONTNEED which tells the kernel to discard pages that are mmap’ed. It continues writing to memory and, at the same time, informs the kernel that memory is no longer needed thus exposing a race condition where changes to private pages could be propagated on the underlying file.
* Another exploit uses the pokemon exploit as a base and automatically generates a new passwd line. The original /etc/passwd is then backed up to /tmp/passwd.bak and overwritten with the new line. The user will be prompted for the new password when the binary is run. After running the exploit, you would be able to login with the newly created user.

# The Exploit: Step-by-step

The exploit was performed on a centos virtual machine with a linux kernel version 3.10.0.

The script for CVE-2016-5195 can be downloaded directly from github. The exploit has its own webpage - [https://dirtycow.ninja](https://dirtycow.nija) - which links directly to its github page.

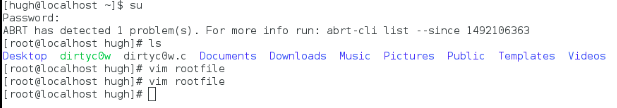
1. To begin, it is necessary to download the dirtyc0w.c script:



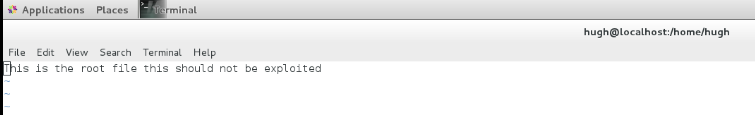
2. Following this, the script must be compiled:



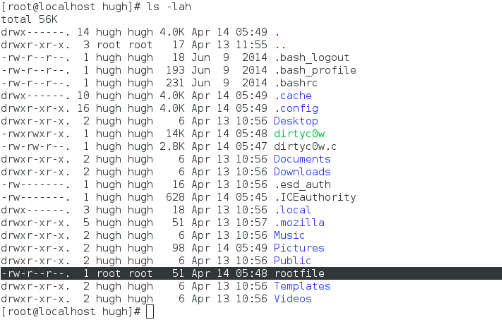
3. In order to prove the exploit, we created a file as the root user. The root user can read and write to this file, but any other user can only read it.



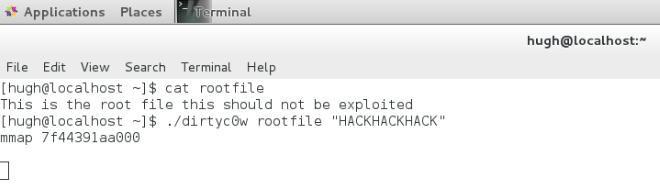
The file reads “This is the root file this should not be exploited”



We can see the permissions for the file here:



5. The exploit is then performed:



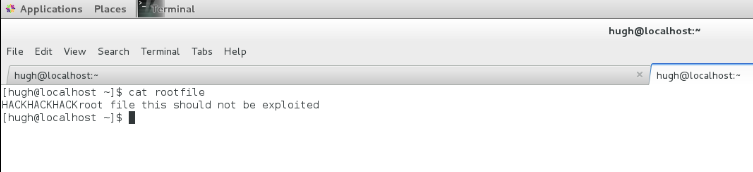
The format for the command is as follows:

[Executable Program] [File to be Exploited] [What to write to file]

Which in this case is:

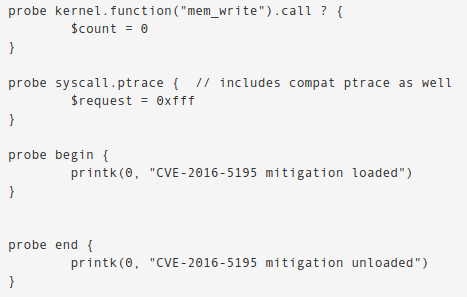
./dirtyc0w rootfile “HACKHACKHACK”.

6. Finally, the file must be examined to ensure the exploit has been successful.

As we can see, “HACKHACKHACK” has been written to the file.

# Solution

Our proposed solution uses a scripting language and tool named SystemTap. SystemTap is used for dynamically instrumenting running production on linux systems. Our solution utilises a very simple SystemTap script released by redhat.



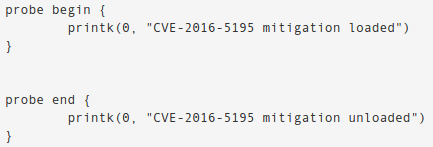
To begin, the script places a probe for when the kernel function “mem-write” is called.



Following this, a probe is set at the ptrace syscalls. The function then disables them.



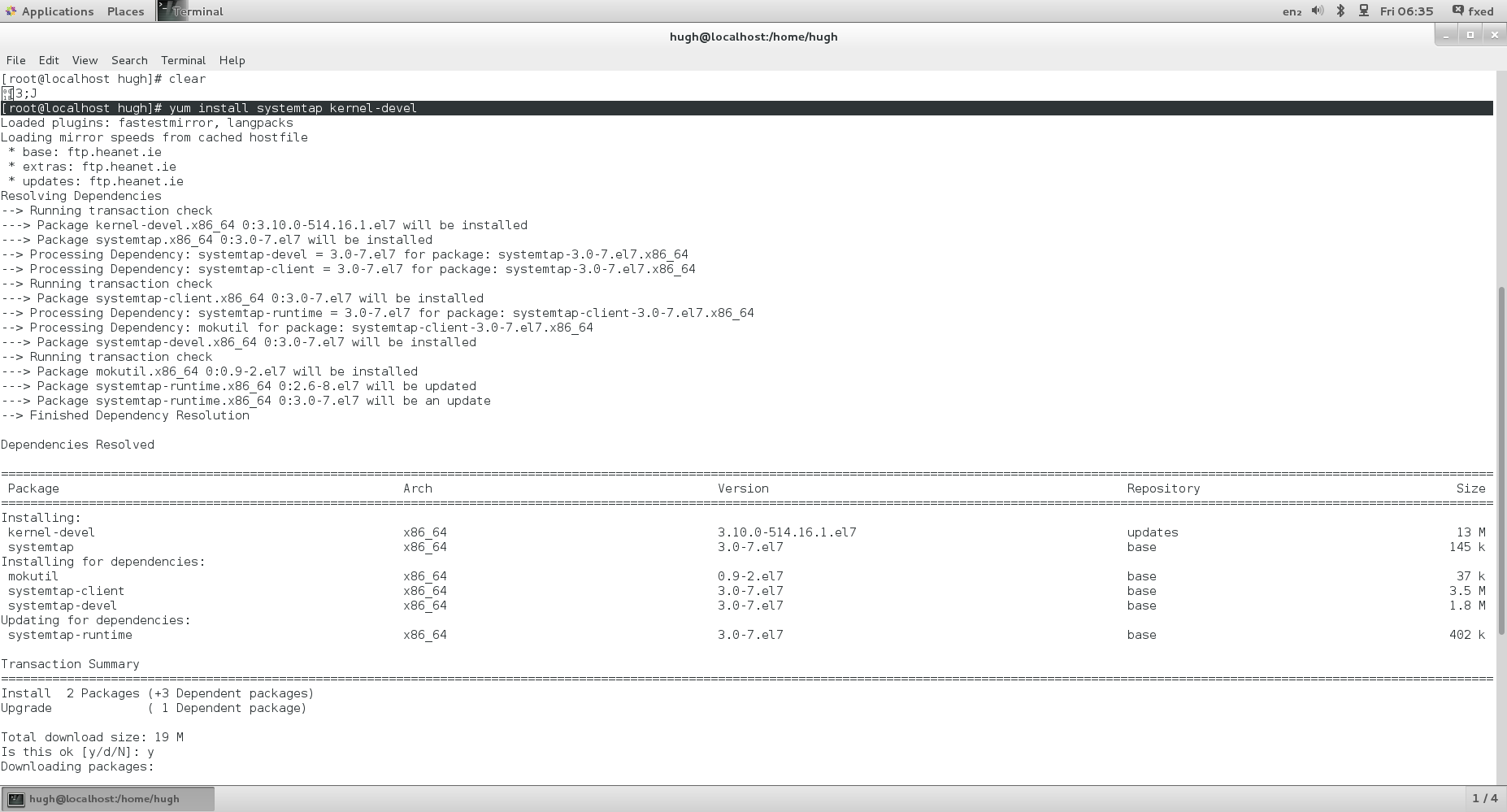
To conclude the script, “probe begin” and “probe end” are included. This bit of code ensures the text included in the script, “CVE-2016-5195 mitigation loaded” and “CVE-2016-5195 mitigation unloaded”, are included in the kernel log buffer. This is done via the printk function. This registers exactly when the mitigation is loaded and unloaded in the system logs, which would allow for a complete audit.



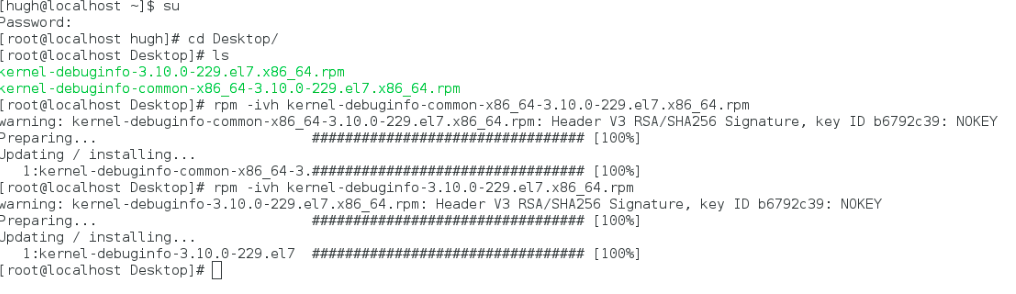
This is important as before otherwise there would be no log of an adversary attempting to perform the exploit.

## Implementing the Solution

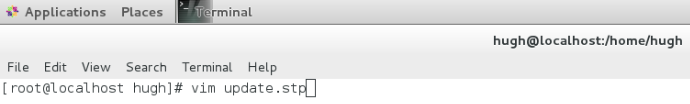
## 1. To begin, SystemTap must be installed on the system:

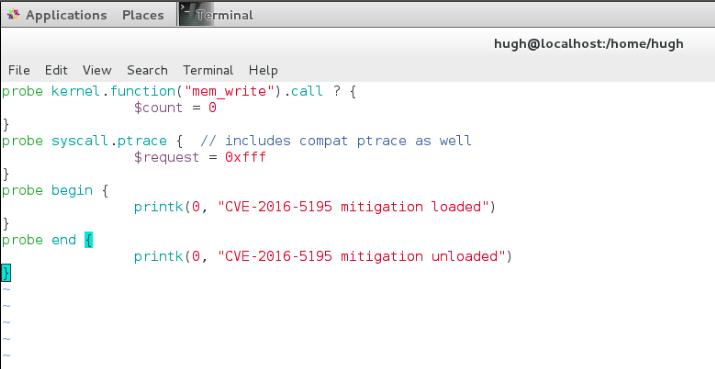


2. It is then necessary to install the debuginfo package for centos. These packages include debugging symbols etc., and are useful later on in the solution.

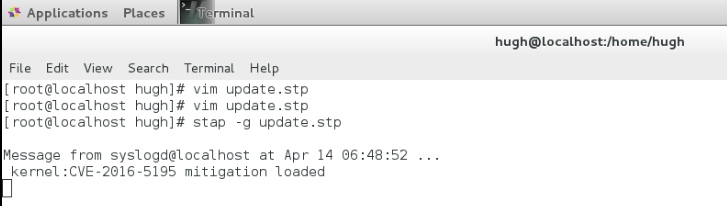


1. Following this, the SystemTap script discussed in the previous section is created:





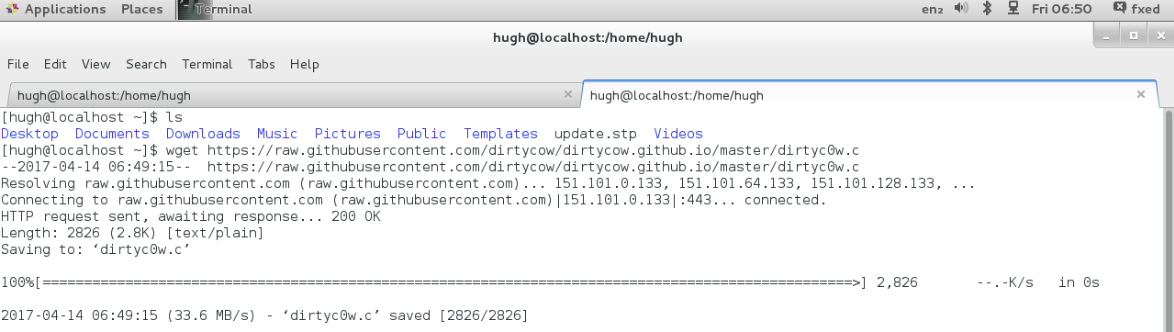
Once it is created, the script must be executed:



## Testing the Solution

In order to test the solution we have implemented, the exploit must once again be performed.

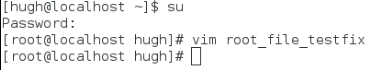
1. The exploit script must be downloaded.

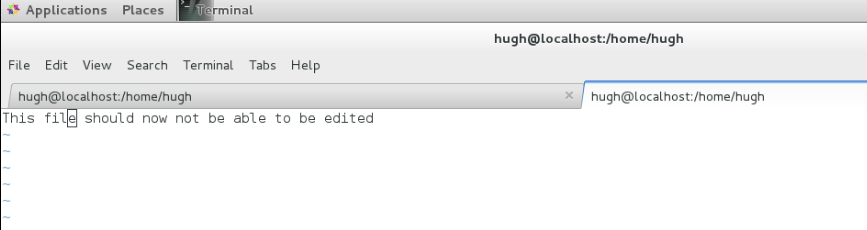


2. The script must then be executed

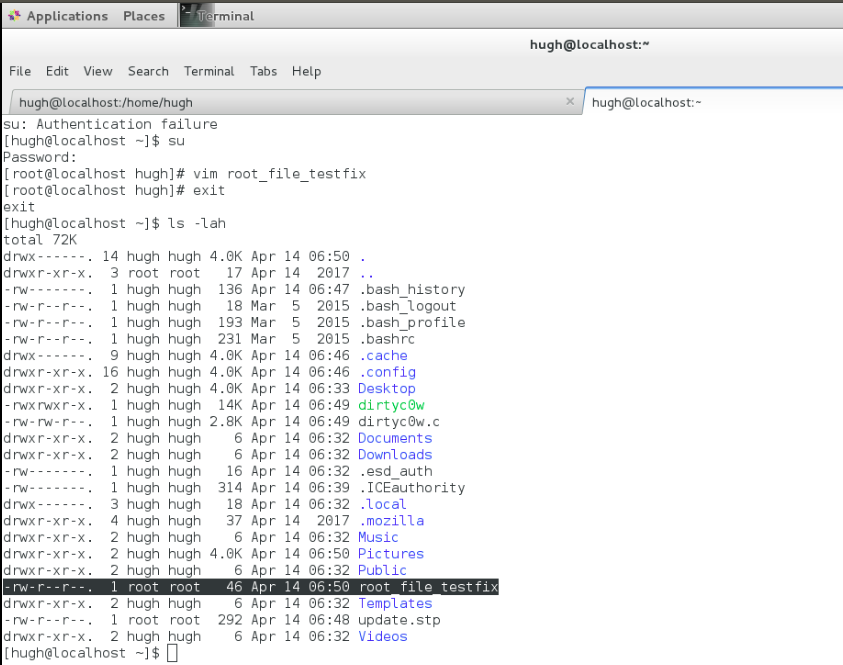


3. A file is then created named “root\_file\_testfix”.



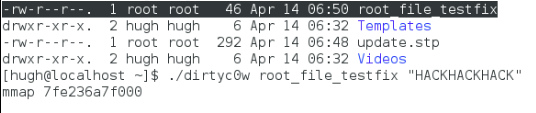


It reads “This file should now not be able to be edited”.

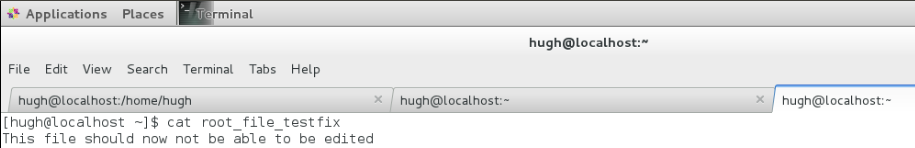


The figure above shows the permissions of “root\_file\_testfix”. This file can only be written to by the root user. Prior to the implementation of the solution, a general user would be able to write to this file after using the exploit.

4. Finally the exploit is executed just as it was in the previous section.



As we can see, the file is not written to:



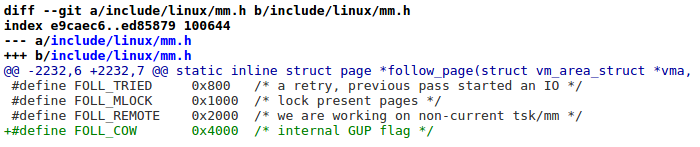
Prior to the solution being implemented, the file would have read “HACKHACKHACK This file should now not be able to be edited”

# Discussion

Throughout this report, the dirtycow exploit was performed on a centos virtual machine with linux kernel version 3.10.0. The solution outlined is only applicable to centos machines, which are using vulnerable linux kernels. Redhat released this solution as a number of web servers running centos could not immediately upgrade to a newer distribution or kernel. Cpanel, a popular web server control panel, is often installed on centos web servers. Upgrading the kernel would require the system to reboot, which would cause downtime to an incredibly large number of websites. Our solution was used to prevent this. It is necessary to conduct the steps for our proposed solution each time the centos server is rebooted i.e. Execute “update.stp” again. We can assume that if the server is going to be rebooted, the kernel will be upgraded therefore the exploit will be prevented.

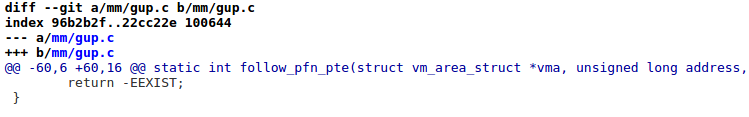
The solution that has been widely accepted within the linux community is to update the operating system with the latest security patch available from the OS repository.

The fix within the linux kernel was completed in October 2016. A new internal flag named “FOLL\_COW” was introduced. This flag informs the system that a COW has already been completed. Another flag, which was already present in the system, is then used to validate that the FOLL\_COW flag is still valid. The files edited are located within the /mm/ directory. This is the linux Memory Manager.

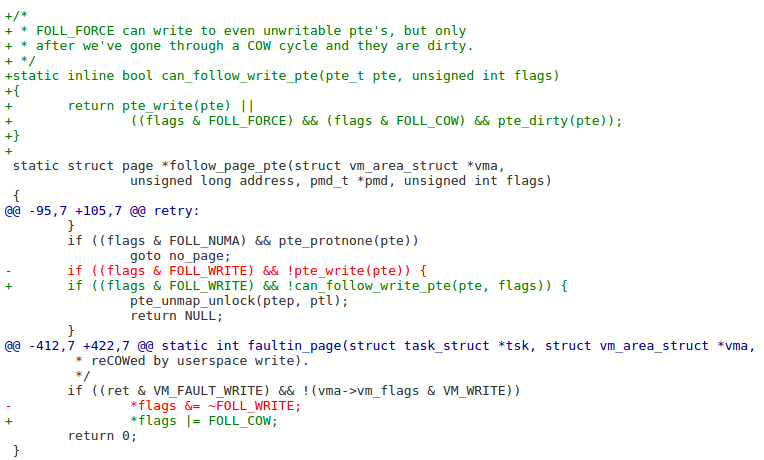


Above we can see the commit where FOLL\_COW is defined. It is defined within the “mm.h” file.

Following this, the GUP script was edited.



GUP stands for Get User Pages. “vm” above stands for Virtual Memory. PTE stands for Page Table Entry.



Above we can see where code has been both added and removed. The first section ensures copy and write is complete, and only then allows writing to it. The OR assignment operator also ensure the flag can only be evaluated once.