Dirty COW Privilege Escalation Practice Report

Nathan White

Jaewon Heo

Orchlon Chinbat

Central Washington University

# Contents

Contents

[Contents 2](#_Toc199895808)

[Prerequisites 4](#_Toc199895809)

[Overview 6](#_Toc199895810)

[Code Examples 7](#_Toc199895811)

[Code 1 8](#_Toc199895812)

[Code 2 14](#_Toc199895813)

[Code 3 18](#_Toc199895814)

[Conclusion 21](#_Toc199895815)

[References 22](#_Toc199895816)

[Figure 1 Network environment overview 4](#_Toc199895817)

[Figure 2 Access example 1 - local access 5](#_Toc199895818)

[Figure 3 Access example 2 - remote access 5](#_Toc199895819)

[Figure 4 Figure 4 Dirty COW poster https://en.wikipedia.org/wiki/Dirty\_COW 6](#_Toc199895820)

[Figure 5 Common libraries used 7](#_Toc199895821)

[Figure 6 Passwd file example 8](#_Toc199895822)

[Figure 7 Passwd file ownership 9](#_Toc199895823)

[Figure 8 Mmap() system call 9](#_Toc199895824)

[Figure 9 User information structure 10](#_Toc199895825)

[Figure 10 Constructing user info 10](#_Toc199895826)

[Figure 11 Generate function 11](#_Toc199895827)

[Figure 12 Madvise system call 11](#_Toc199895828)

[Figure 13 Child process behavior 12](#_Toc199895829)

[Figure 14 Parent process behavior 13](#_Toc199895830)

[Figure 15 Undesired output 1 14](#_Toc199895831)

[Figure 16 Undesired output 2 15](#_Toc199895832)

[Figure 17 Improvement 1 16](#_Toc199895833)

[Figure 18 Improved line construction 17](#_Toc199895834)

[Figure 19 Improved injection 17](#_Toc199895835)

[Figure 20 Get offset function 18](#_Toc199895836)

[Figure 21 Offset aware writing 19](#_Toc199895837)

[Figure 22 Target specific injection 20](#_Toc199895838)

# Prerequisites

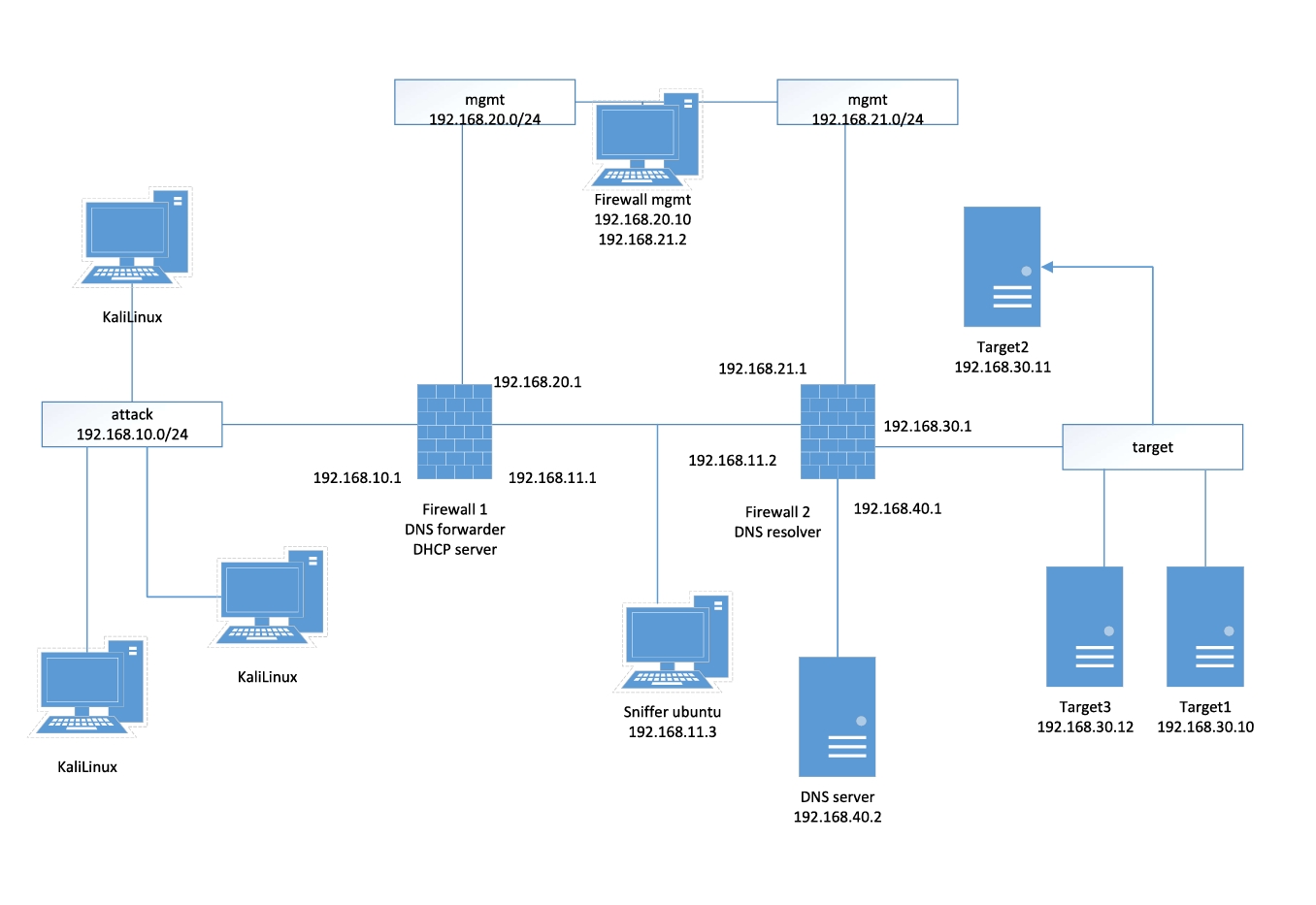


Figure 1 Network environment overview

This exercise was practiced in Target 3 virtual host connected in the target subnet range with IP address 192.168.30.12. The target host is the Metasploitable 2, a Linux distribution which is made intentionally vulnerable to practice various vulnerabilities.

We were logged in locally with user with no root privilege. You can also simulate this exploitation by connecting remotely from one of Kali hosts connected in the attack subnet. It could be done with SSH or NFS mounting.

Lastly, the code used for this exercise was originated by user firefart, and you can access to the original code from: <https://github.com/firefart/dirtycow>

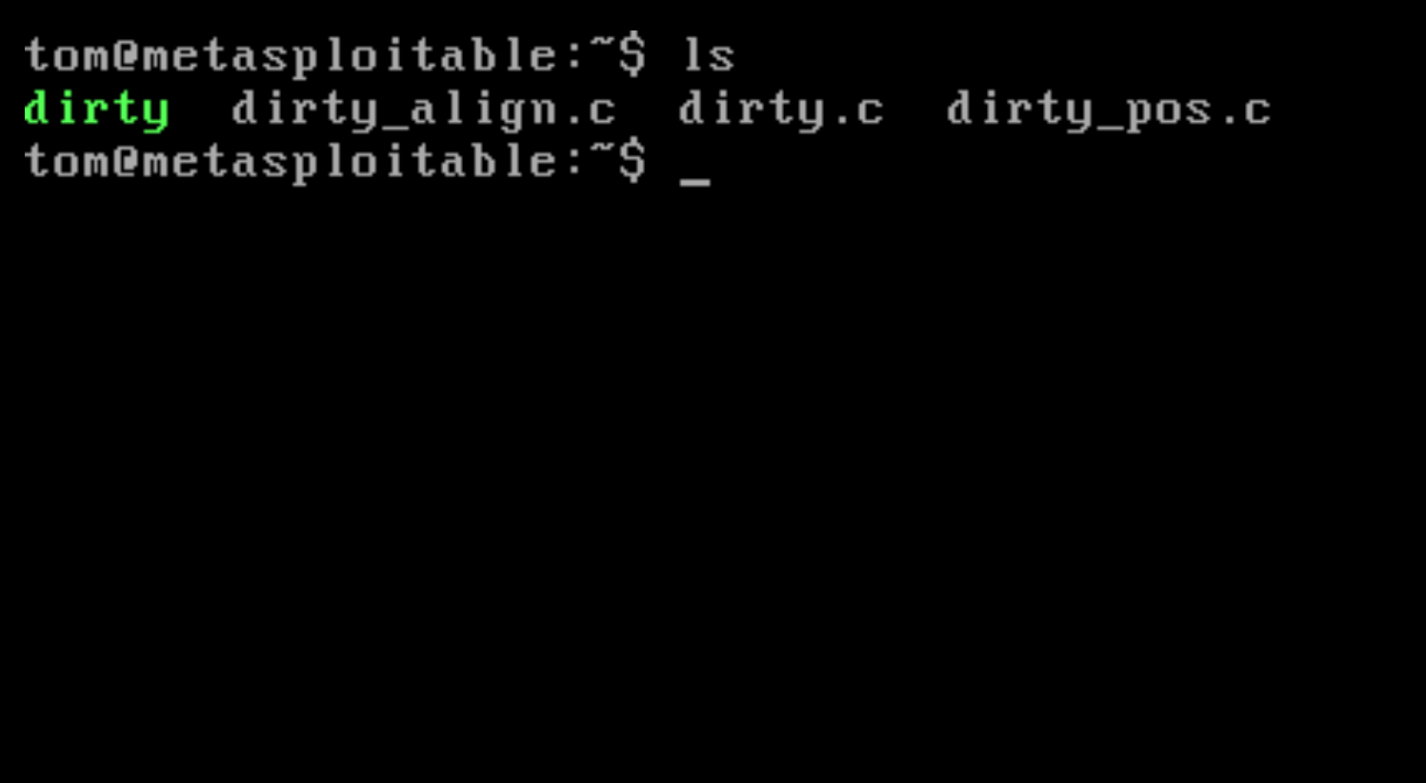


Figure 2 Access example 1 - local access



Figure 3 Access example 2 - remote access

# Overview

 The name of the exploitation method that we are going to use is called Dirty COW (Copy-On-Write). Dirty COW is a privilege escalation vulnerability in the Linux kernel, which discovered in 2016 (Wikipedia, “Dirty COW”). You can find this vulnerability on Linux based operating systems, including Android, using kernel version prior to 4.8.3.

Figure 4 Figure 4 Dirty COW poster https://en.wikipedia.org/wiki/Dirty\_COW

The Dirty COW is a local privilege escalation bug that exploits a race condition in the kernel’s memory management system, specifically in the way the copy-on-write mechanism handles memory mappings marked as read-only and private.

By racing madvise() and ptrace() C system calls, a non-privileged user could gain write access to files that normally cannot modify. In this exercise, we will inject a malicious data into the read-only file /etc/passwd to gain a root privilege.

# Code Examples

For this exercise, we prepared three C programming codes to exploit the kernel. The first one is the original exploit code, sourced from the Dirty COW repository on GitHub (firefart). The other two are modified variants based on the original, created to demonstrate different ways how vulnerability can be leveraged.

Despite the differences, all three versions share common components to exploit. They use Pthread (POSIX thread) for multithreading, mmap() system call to map the target file into the memory, ptrace() system call to attempt unauthorized writes, and lastly madvise() system call to trigger the race condition.

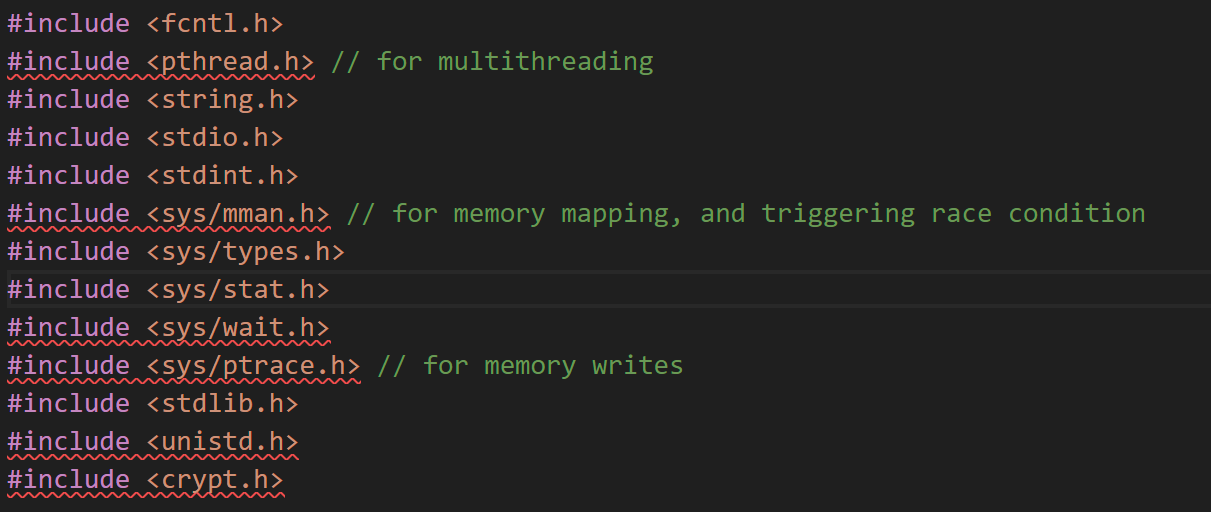


Figure 5 Common libraries used

## Code 1

The first source code used in this exercise is the original Dirty COW exploit written by GitHub user firefart. This code shows how the Dirty COW vulnerability could be exploited to perform local privilege escalation on Linux systems that are prior to kernel version 4.8.3.

This exploit aims to overwrite the /etc/passwd file which stores the user account information, including user ID (UID), group ID (GID), user’s home directory and the login shell. By replacing a normal user’s entry with one that has root’s UID and GID, the attacker can effectively acquire root privileges to the non-privileged account.

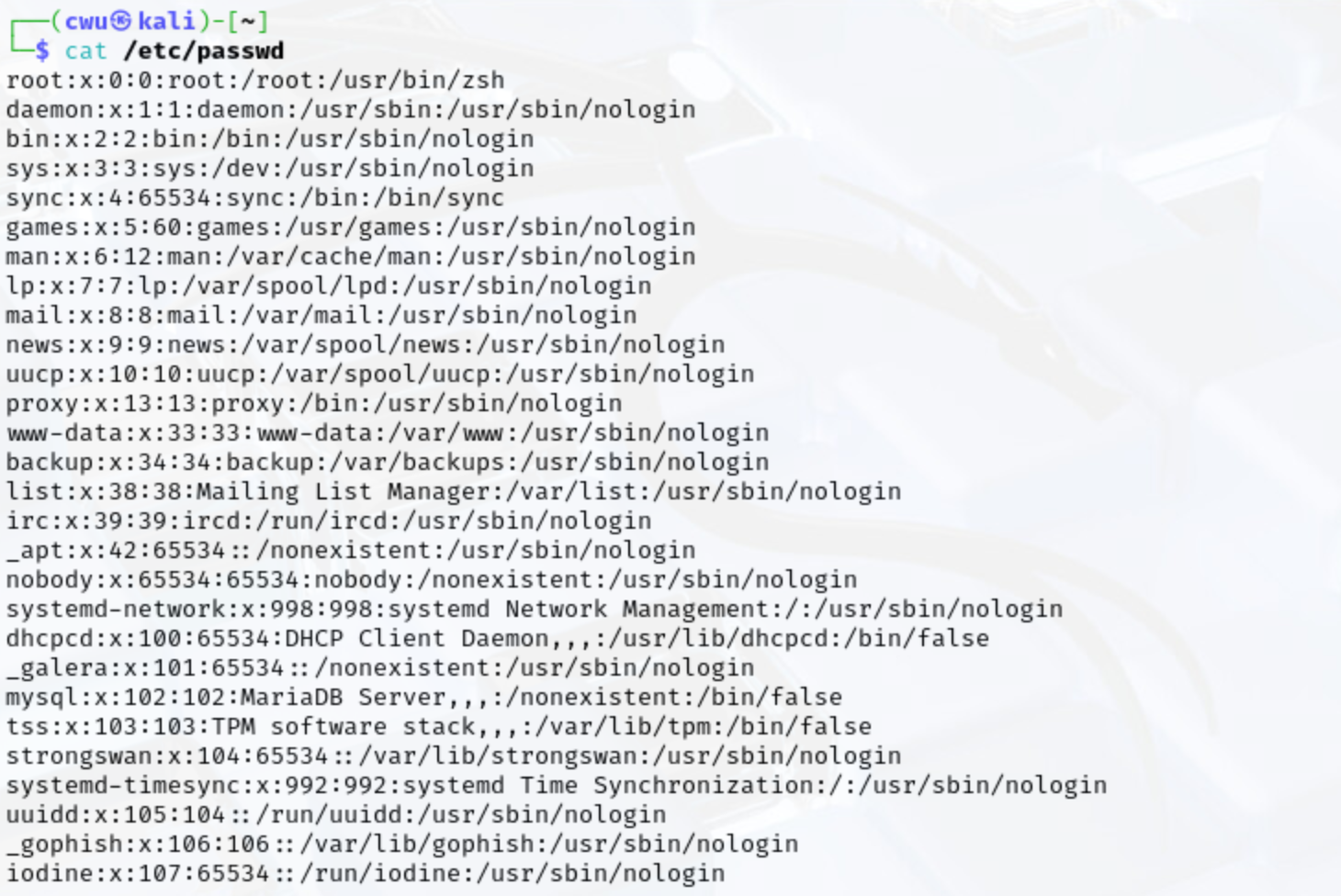


Figure 6 Passwd file example

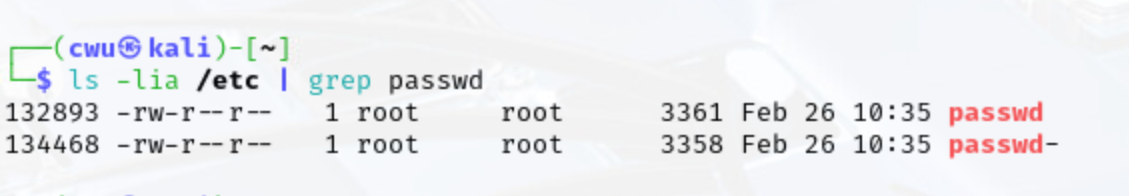
This is typically not possible since the ownership of the passwd file is belonging to the root user. 

Figure 7 Passwd file ownership

As you can see above figure, only root user (owner) can read and write the passwd file and the regular users can only read it. However, the Dirty COW vulnerability allows us to bypass these permission restrictions by exploiting a race condition in the Linux kernel’s memory management system.

The exploit begins by opening and memory-mapping the /etc/passwd file with the mmap() system call.

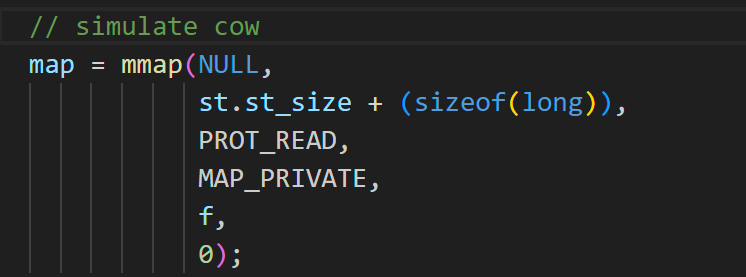


Figure 8 Mmap() system call

This call maps the contents of the file into the process’s address space using the macro, MAP\_PRIVATE, which prevent changes from being written back to the actual file. However, due to the vulnerability under specific conditions, the system may mistakenly allow writings on the file.

This mapped region in process’s memory space allows the exploit to target the section of memory corresponding to a specific user’s line in the /etc/passwd file. The exploit then constructs a replacement line for the user’s entry. This line includes the name of the user, a hashed password, UID, GID, customed home directory and shell location.

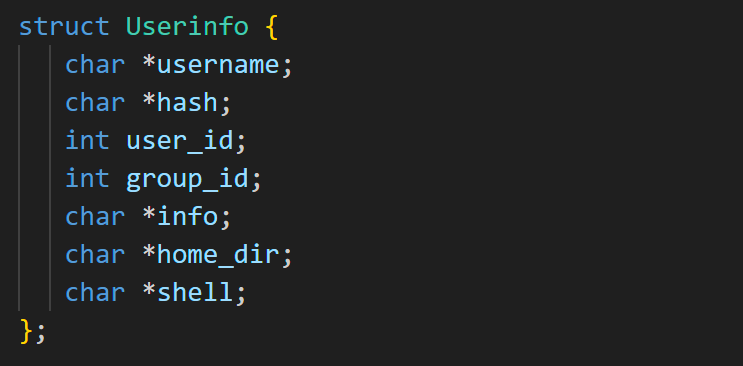


Figure 9 User information structure

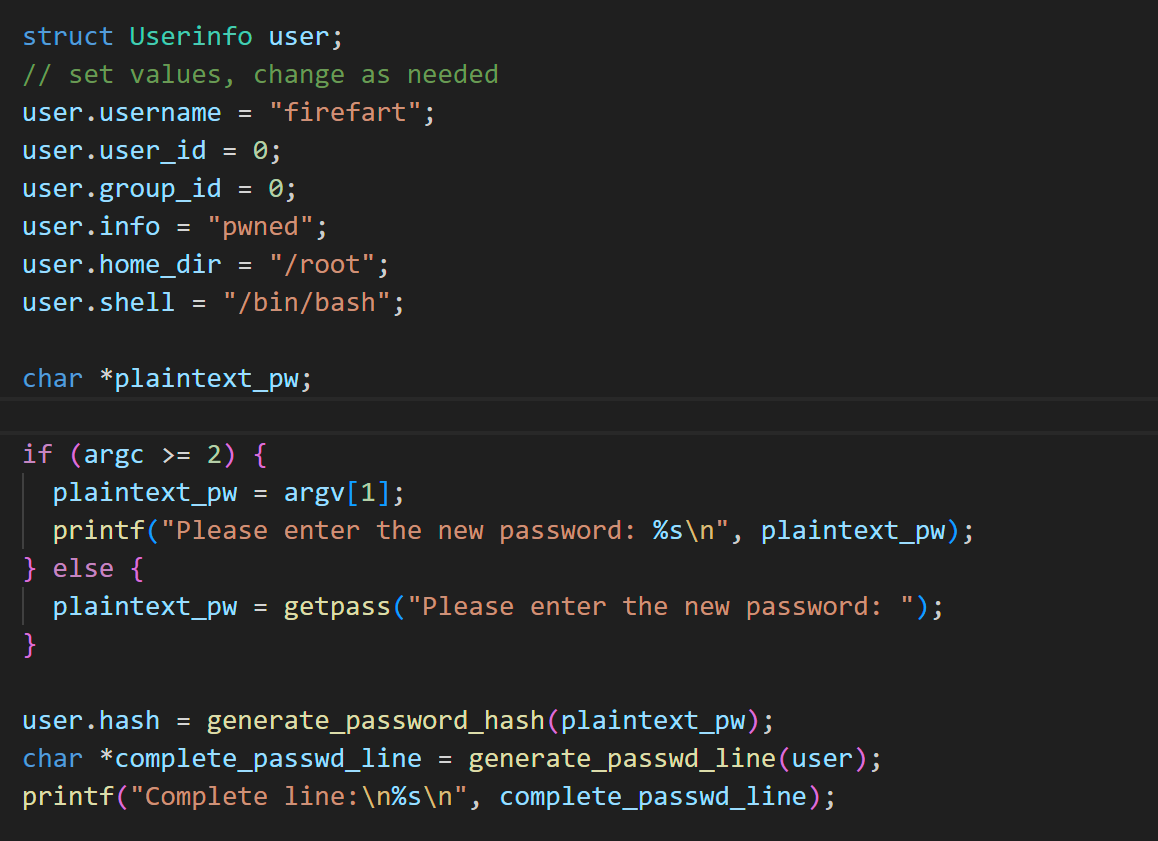


Figure 10 Constructing user info

Above figures show the data structure which stores the user’s line and the process of construction. All information stored in the Userinfo struct then passed to the function generate\_passwd\_line() to gather all information into single line.

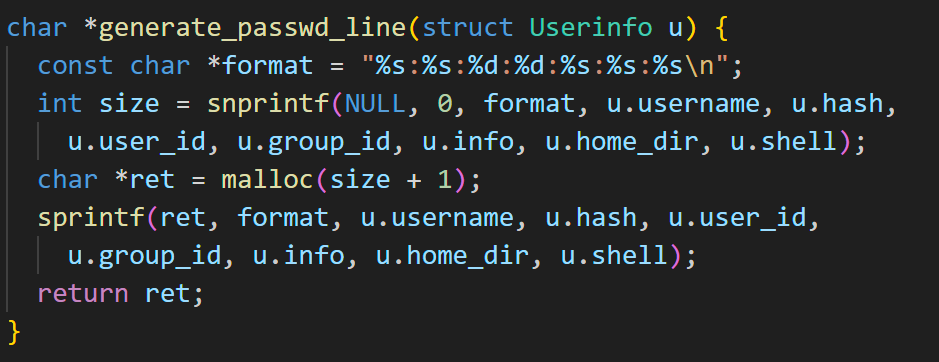


Figure 11 Generate function

Once the target user’s line is constructed, the exploit proceeds to the actual exploitation. It triggers a race condition between memory management and a forced write attempt. This is done using two parallel mechanisms. One, parallelly created user thread repeatedly calls the madvise() system call to advise kernel that this program is no longer in use of the mapped memory. Second, attempts to write into a memory-mapped region using ptrace() system call.

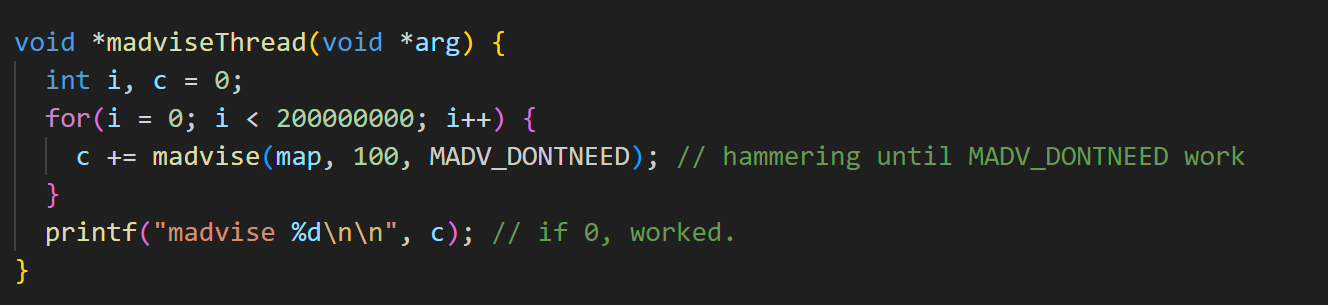


Figure 12 Madvise system call

Above function is used for thread routine which calls madvise system call with the MADV\_DONTNEED macro. The purpose of this system call is to tell the kernel that the process is no longer needs to access to the certain region of memory as mentioned above. The boundary of the region is specified by the second argument of the system call which is the size of the memory.

In response, the kernel discards the memory page and will reload it from the disk when it needs to access it later. This function is hammering the memory-mapped region of the /etc/passwd, attempting to create a race condition. The instability in the memory management system caused by this race will be conjunct with the MAP\_PRIVATE flag which set in the mmap() system call.

Normally, any writes would result in a private copy of the memory and would not modify the file itself. However, due to the flooding calls of madvise(), the kernel may fail to correctly enforce copy-on-write, and lead to modify the actual file on the disk.

Before the flooding, the exploit first creates child process, which will create the thread calling madviseThread() function.

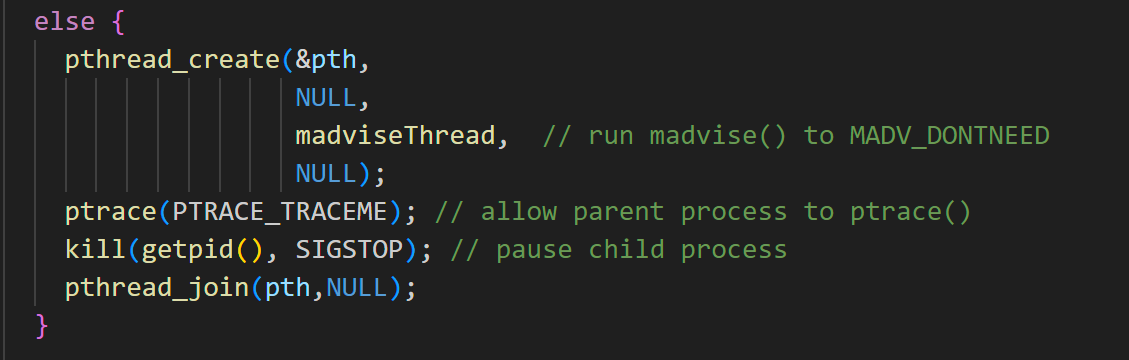


Figure 13 Child process behavior

The child process creates the thread which repeatedly calling madvise(), and then calls ptrace() system call with PTRACE\_TRACEME macro to allow itself to be traced. Then, it suspends itself by calling kill() system call to allow parent process to run again.

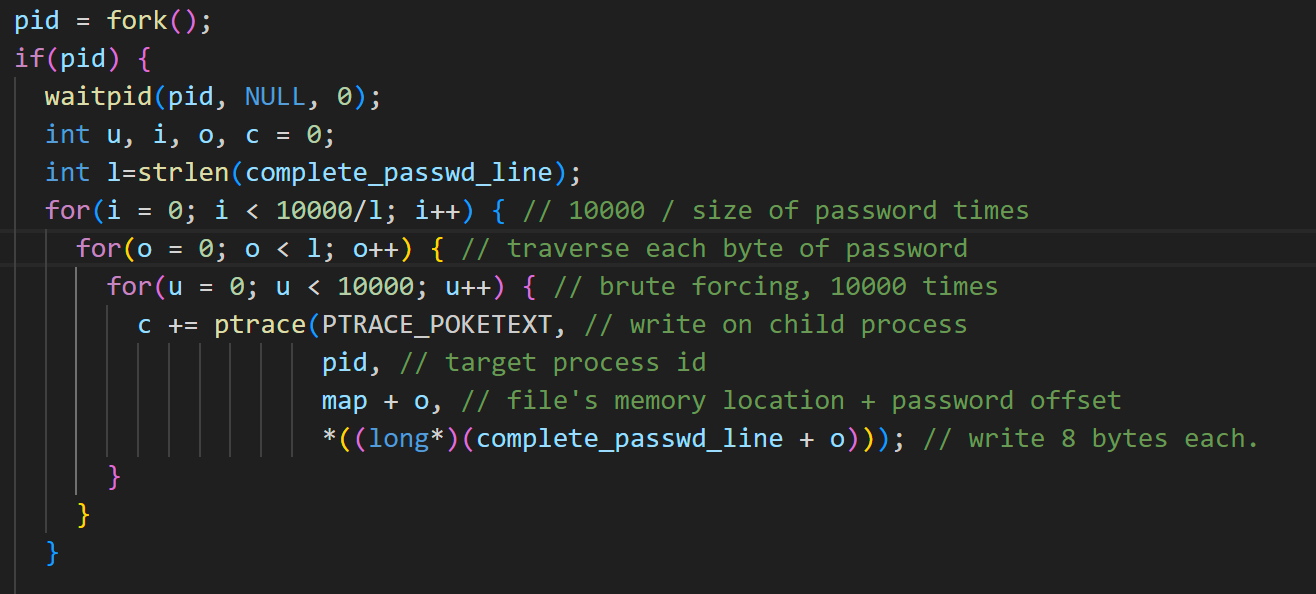


Figure 14 Parent process behavior

Now the parent process, it first waits its child process to finish its task first and then begins attempting to overwrite the mapped memory using ptrace() system call with PTRACE\_POKETEXT macro.

The outer loop repeats the whole writing process to increase change to exploit. The middle loop goes through each byte of the line which is going to be written to /etc/passwd file. The inner loop, brute forces the writing of each byte by 10000 times to increase chance to win the race with the thread.

## Code 2

Before getting into the second code, we need to check the writing part of the first code. It is important to understand the key weakness in the first code to understand the second code. As explained earlier, the first code attempts to overwrite the /etc/passwd file by triggering race condition between the madvise() and ptrace() system calls. Writing each byte to the passwd file could potentially corrupt the data and lead to the undesired output.

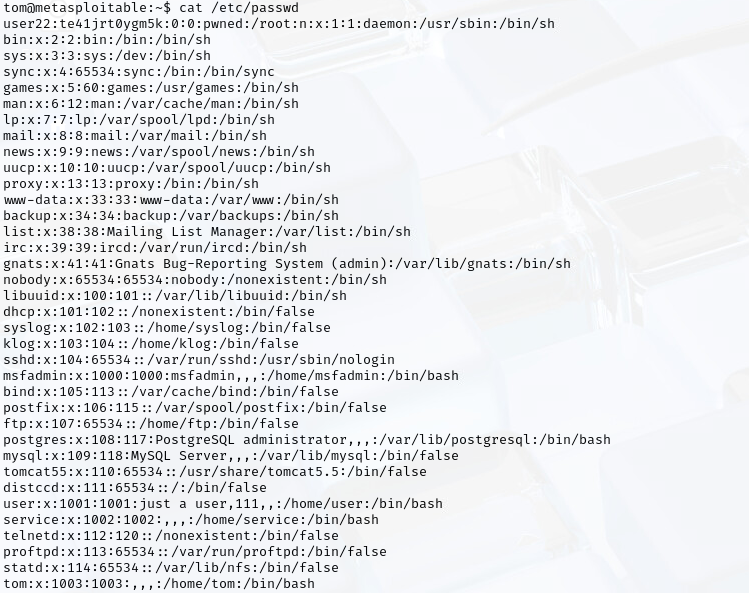


Figure 15 Undesired output 1

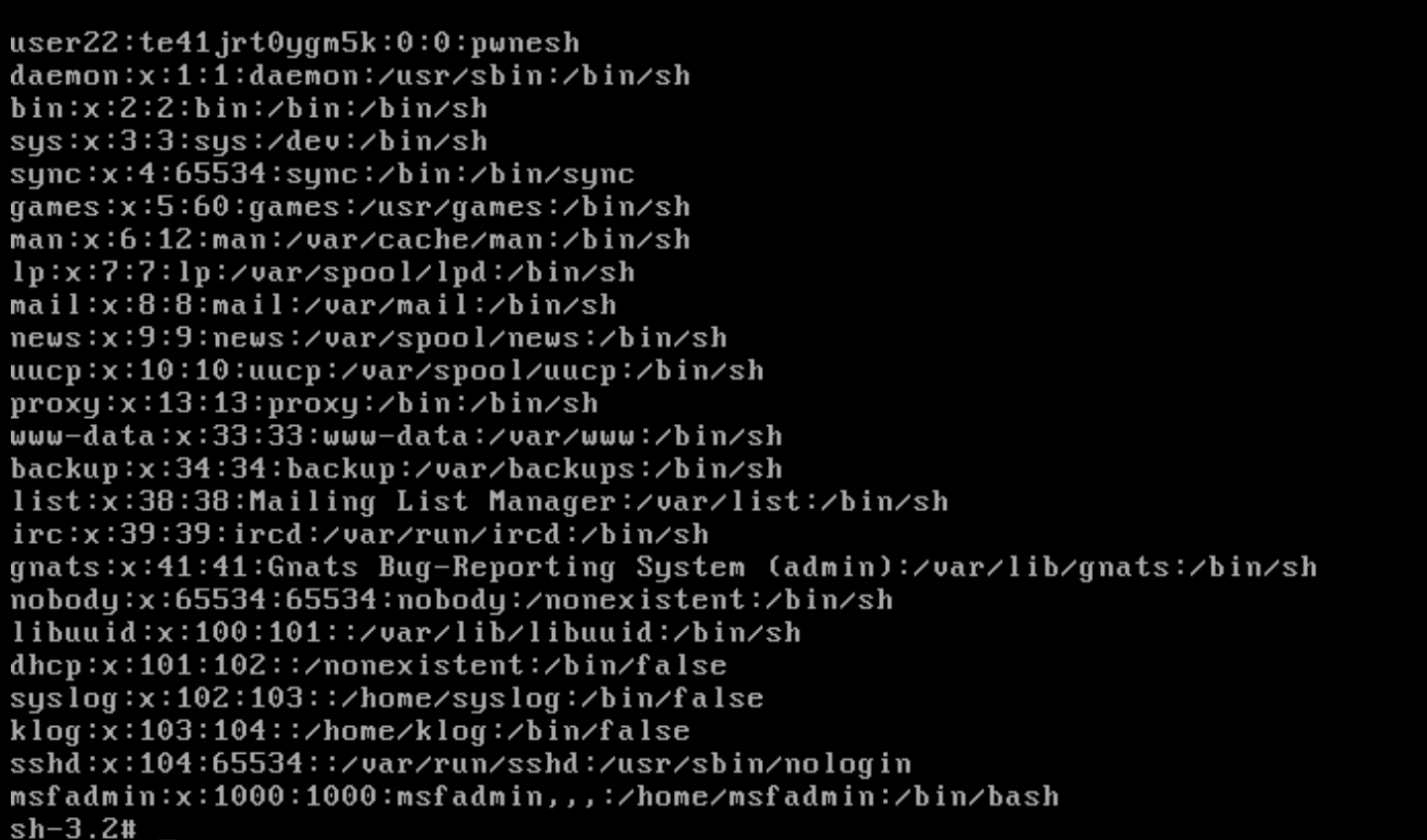


Figure 16 Undesired output 2

Now back to the Figure 14, the middle loop shifts one byte at a time through the entire payload string. At each byte offset, the system call ptrace() write a full word (architecture dependent) due to the PTRACE\_POKETEXT macro. Thus, every ptrace() call writes not only the desired byte, but also unnecessary 7 more bytes which are overlapping with already written bytes.

These unaligned memory writes could cause corruption by partial success. Since we are triggering race condition, some writes may succeed, while some others may fail. This issue may result in garbled lines in /etc/passwd, which incorrect formatting or invalid fields as Figure 15 and Figure 16 above. Due to the parsing failure, attackers may see the issue such as login failures by broken passwd file.

To overcome the issue introduced from the original Dirty COW exploit, the second version of the code introduces two key improvements.

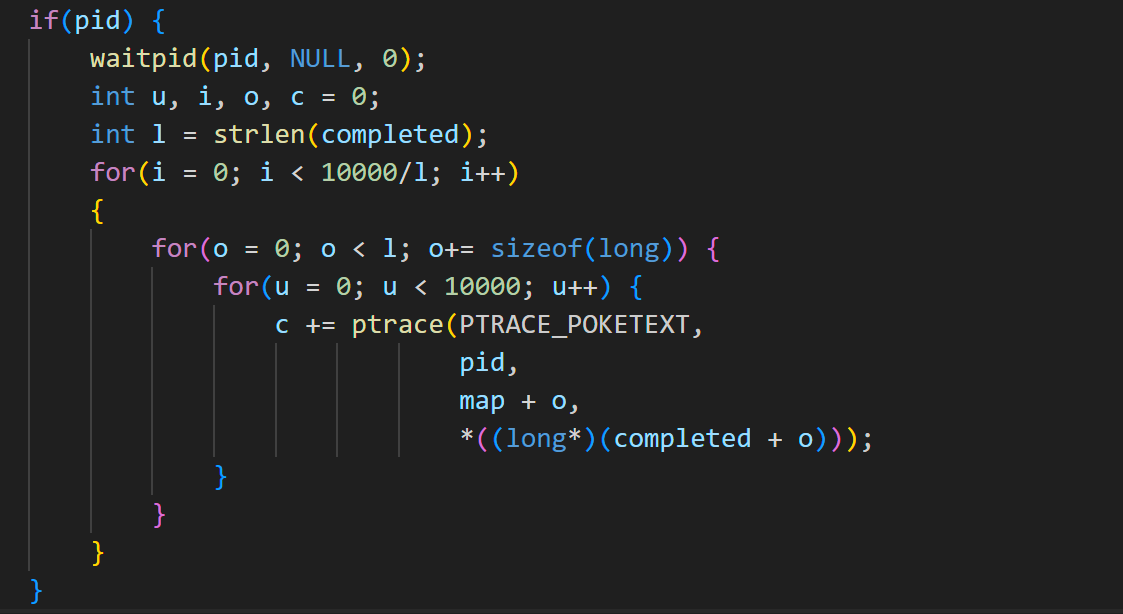


Figure 17 Improvement 1

The first change made is in the write loop. Instead of iterating through each byte, the second version steps through the payload in chunks of sizeof(long). This is because of the ptrace() system call that it writes sizeof(long) bytes for each iteration. By shifting the offset sizeof(long) long, there would be no overlapping, and potentially prevents file corruption.

To do that, we had to have a line which got the size divisible by sizeof(long). To achieve this, we decided to add a null (\0) padding to the end of the line to guarantee the alignment. The improved constructing function at the bottom Figure 18 calculates the size of the line and adds null paddings ensures that every write will target a properly aligned sizeof(long) bytes block and reduces the chance of overlapping.

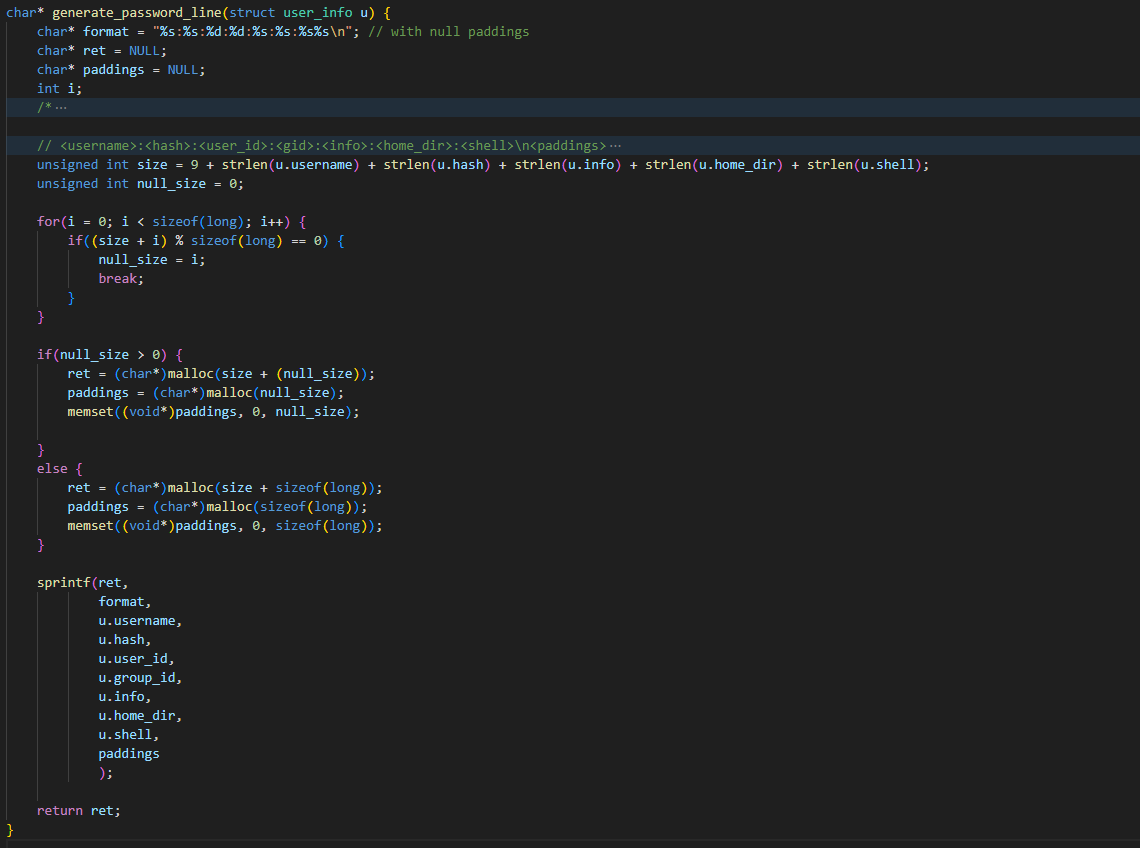


Figure 18 Improved line construction

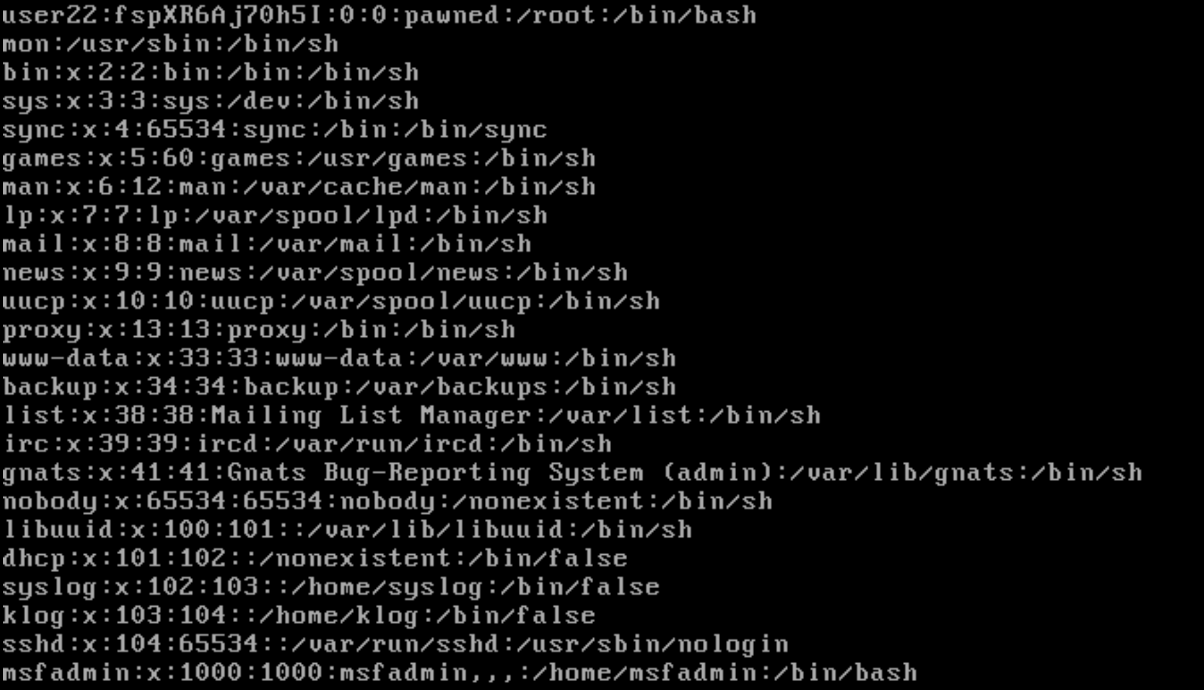


Figure 19 Improved injection

## Code 3

The third variant of the Dirty COW introduces more targeted and precise overwrite strategy. Rather than overwriting from the zero offset (beginning of the file) of the /etc/passwd, this version allows the attacker to specify a target username already in the /etc/passwd file. The program the locates that username and overwrites its line, make that user into a root-level account.

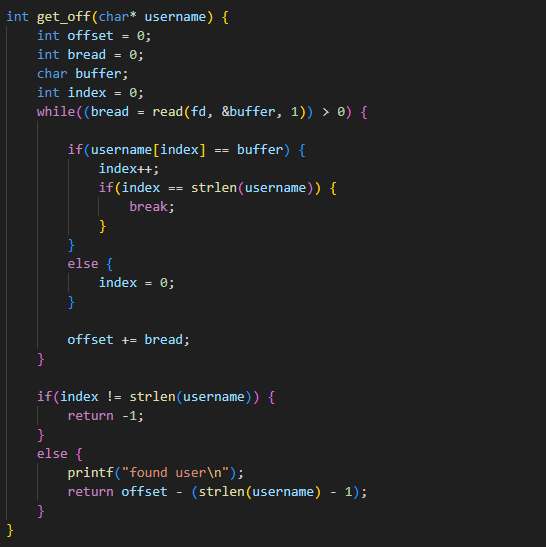


Figure 20 Get offset function

One of the most important features in this version is the get\_off() function. This function reads through the /etc/passwd file, byte by byte, and performs a simple pattern matching to find the starting byte offset of the given username.

The while loop records the offset of the end of the username in the /etc/passwd file, so in order to acquire actual starting offset of the given username, (total length – 1) subtracted from the counted offset.

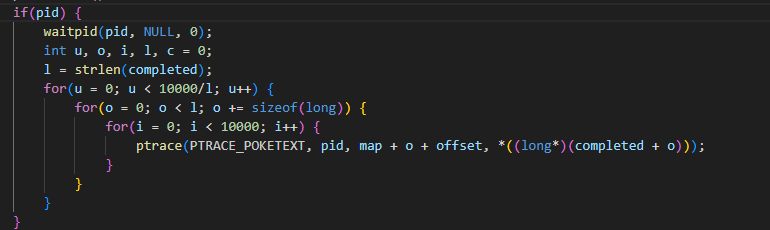


Figure 21 Offset aware writing

Comparing to the second version of the exploit, the second version overwrites from the beginning of the file. This is because variable map points to the beginning of the memory-mapped file. However, in this version, writing code adds the discovered offset to each ptrace() call, to make sure the injected string precisely replaces the targeted user’s line.

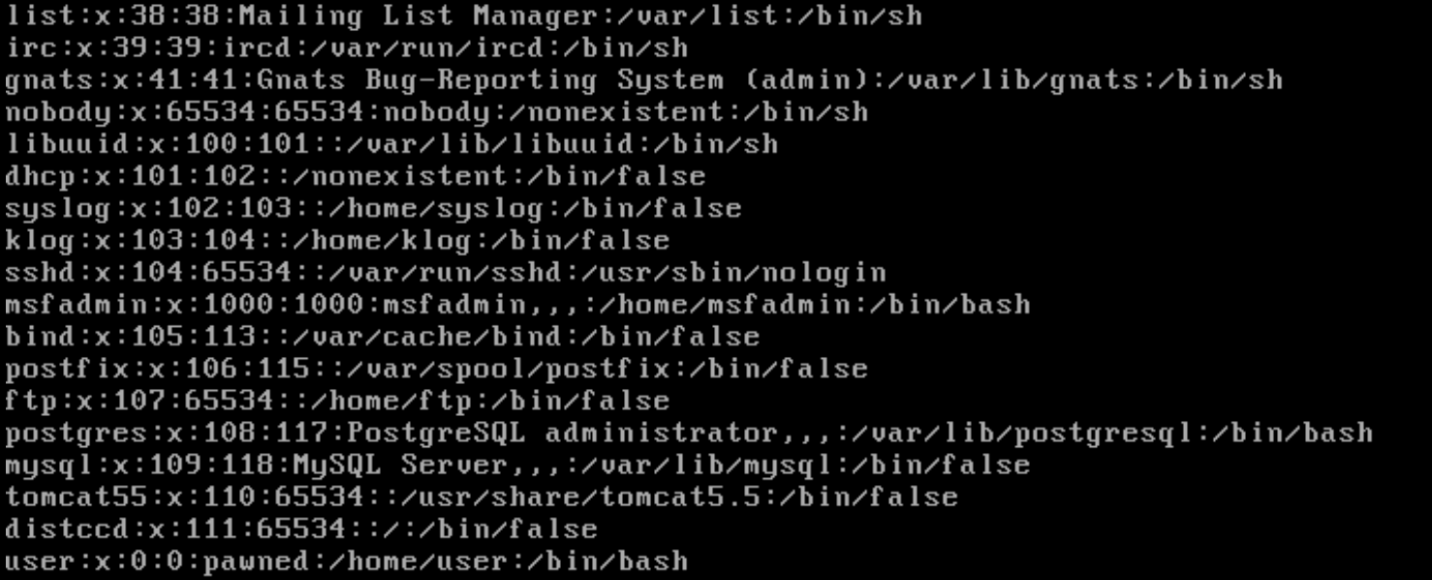


Figure 22 Target specific injection

As above figure shown, username “user”, which is obviously not the first line, got modified. This was done by executing exploit with passing target username as the command line argument.

# Conclusion

This practice shown the exploitation of the Dirty COW vulnerability to achieve privilege escalation on a Linux system. Through three different codes, we explored the mechanics of the vulnerability, which abuses a race condition between the madvise() and ptrace() system calls. This race allowed unauthorized modification of read-only file, in our case /etc/passwd file, which is not allowed to normal users to modify.

The original exploit highlighted the risk of data corruption due to unaligned writes. To prevent that, we created second version of the exploit that ensures integrity by providing write alignment using sizeof(long) chunking and adding null padding. The third version improved little more, by allowing attackers to choose the target.

This practical exploitation reinforced understanding of memory management vulnerabilities and race conditions in the Linux kernel.

# References

firefart. *dirtycow*. GitHub, 20 Oct. 2016, <https://github.com/firefart/dirtycow>.

Wikipedia, “Dirty COW”, <https://en.wikipedia.org/wiki/Dirty_COW>