

Trường đại học Khoa học Tự nhiên - Đại học Quốc gia thành phố Hồ Chí Minh
Khoa Công Nghệ Thông Tin

-----oOo-----

Báo cáo thực hành

Linux kernel module và syscall hooking



Giáo viên hướng dẫn: **Lê Giang Thanh – Nguyễn Thị Thanh Huyền**
Lớp: **Hệ điều hành - CQ2017/21**
Sinh viên thực hiện: **Âu Dương Tấn Sang – 1712145**

Github repository của bài thực hành này: <https://github.com/auduongtansang/SimpleRootkit>

HCM – 11/2019

I. Giới thiệu:

Linux là một hệ điều hành mã nguồn mở, có kernel dạng module. Điều này giúp cho các nhà phát triển hệ thống có thể viết các module và gắn thêm vào kernel, bổ sung cho kernel các tính năng cần thiết theo ý mình. Tuy nhiên các hacker có thể lợi dụng sự tiện ích này để viết các module “đầu độc” hệ điều hành.

Mục tiêu của bài thực hành này là giúp sinh viên hiểu được các khái niệm cơ bản về kernel của hệ điều hành Linux thông qua việc lập trình những module đơn giản bằng ngôn ngữ C. Và từ đó có thể đưa ra các ý tưởng để bảo vệ hệ thống của mình.

II. Tự đánh giá mức độ hoàn thành:

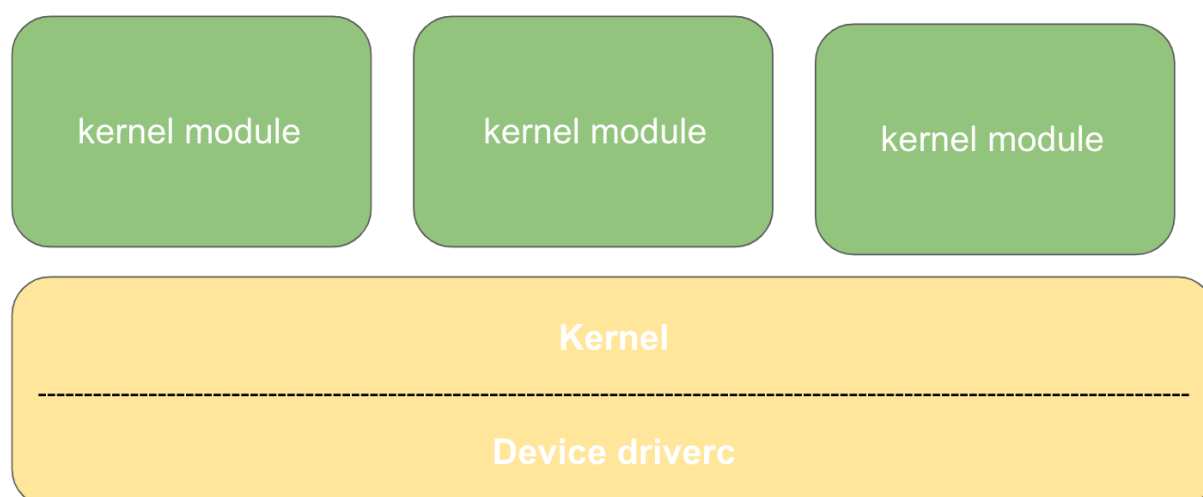
Module tạo số ngẫu nhiên trong kernel.....	100%
Module hook vào syscall open()	100%
Module hook vào syscall write().....	90%

Hạn chế: chưa tìm được tên file bị ghi ở phần hook vào syscall write().

III. Module tạo số ngẫu nhiên:

Kernel module là phần mở rộng có thể gắn thêm vào kernel để thực thi lệnh bằng quyền kernel (hay thực thi lệnh trong kernel-space), điều mà các ứng dụng ở user-space không thể làm được.

Trong hệ điều hành Linux, các device driver cũng là các kernel module, chúng cung cấp cho hệ điều hành khả năng giao tiếp với phần cứng. Trong phần này, ta sẽ viết một device driver để Linux có thể giao tiếp với một phần cứng “tưởng tượng” có thể tạo ra các số ngẫu nhiên.



1. Kernel module và device driver

a. Module đầu tiên:

Một module cơ bản sẽ có cấu trúc mã nguồn như sau (đặt tên file này là *randlkm.c*):

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_AUTHOR("Au Duong Tan Sang");
MODULE_DESCRIPTION("Random number LKM");
MODULE_LICENSE("GPL");

static int __init rand_init(void)
{
    printk("[+] randlkm loaded.\n");
    return 0;
}

static void __exit rand_exit(void)
{
    printk("[-] randlkm unloaded.\n");
    return;
}

module_init(rand_init);
module_exit(rand_exit);
```

Trong đó, *linux/module.h* là file header cần cho mọi module, còn *linux/init.h* chứa định nghĩa cho thủ tục *module_init()* và *module_exit()*. Hai thủ tục này nhằm kích hoạt hai hàm *rand_init()* và *rand_exit()* tương ứng với lúc module được gắn vào và tháo ra khỏi kernel. Các hằng hàm *AUTHOR*, *DESCRIPTION* và *LICENSE* dùng để mô tả thông tin module.

Để biên dịch, tạo một *Makefile* với nội dung:

```
obj-m += randlkm.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Đặt *Makefile* và *randlkm.c* cùng thư mục, mở terminal và chạy *make*. Nếu biên dịch thành công, ta sẽ thấy trong thư mục xuất hiện file *randlkm.ko*, đây chính là file module đã được biên dịch và có thể gắn vào kernel.

Để gắn module vào kernel, chạy lệnh sau bằng quyền root:

```
sudo insmod randlkm.ko
```

Tương tự khi muốn tháo module, chạy lệnh sau cũng bằng quyền root:

```
sudo rmmod randlkm
```

Hàm `rand_init()` và `rand_exit()` có sử dụng lệnh `printk()` để ghi log. Ta có thể dùng nó như một cách đơn giản để debug chương trình trong kernel-space. Để xem log của hệ thống, ta gõ lệnh `dmesg`. Dưới đây là kết quả khi thực hiện toàn bộ các bước trên.

```

adts@adts-VirtualBox: ~/Documents/randlkm
File Edit View Search Terminal Help
adts@adts-VirtualBox:~/Documents/randlkm$ make
make -C /lib/modules/4.15.0-45-generic/build M=/home/adts/Documents/randlkm modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-45-generic'
CC [M] /home/adts/Documents/randlkm/randlkm.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/adts/Documents/randlkm/randlkm.mod.o
LD [M] /home/adts/Documents/randlkm/randlkm.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-45-generic'
adts@adts-VirtualBox:~/Documents/randlkm$ sudo insmod randlkm.ko
[sudo] password for adts:
adts@adts-VirtualBox:~/Documents/randlkm$ gcc -o randtest randtest.c
adts@adts-VirtualBox:~/Documents/randlkm$ sudo ./randtest
Read: 3697610572.
adts@adts-VirtualBox:~/Documents/randlkm$ sudo ./randtest
Read: 1745896688.
adts@adts-VirtualBox:~/Documents/randlkm$ sudo ./randtest
Read: 3105732956.
adts@adts-VirtualBox:~/Documents/randlkm$ sudo ./randtest
Read: 631674815.
adts@adts-VirtualBox:~/Documents/randlkm$ sudo rmmod randlkm
adts@adts-VirtualBox:~/Documents/randlkm$

```

2. Module đầu tiên

Tạm bỏ qua các dòng thực thi chương trình `randtest`, ta sẽ quay lại sau. Cửa sổ bên dưới đang chạy lệnh `dmesg -w` và ta thấy có hai dòng log do lệnh `printk()` ghi vào, có nghĩa rằng module của chúng ta đã hoạt động.

b. Character device file:

Device file là một trung gian giao tiếp giữa chương trình ở user-space và kernel-space. Khi chương trình ở user-space yêu cầu IO, phần cứng sẽ hoạt động và ghi/đọc dữ liệu trên device file, sau đó chương trình ở user-space sẽ đọc/ghi dữ liệu cũng trên device file này.

Có hai loại device file: *block device file* và *character device file*. Sự khác biệt giữa chúng cũng tương tự như sự khác biệt giữa file text và file nhị phân. Để đơn giản, trong phần này ta sẽ dùng character device file để chương trình ở user-land giao tiếp với phần cứng “tưởng tượng” có thể tạo ra các số ngẫu nhiên.

Trong kernel Linux có hỗ trợ sẵn một cấu trúc cho character device file, đó là `miscdevice` trong `linux/miscdevice.h`. Ngoài ra, ta phải định nghĩa các thao tác *open* và *read* trên file này, nghĩa là những công việc cần thực hiện khi chương trình ở user-space gọi `open()` và `read()`. Để làm được điều này, ta dùng cấu trúc `file_operations` trong `linux/fs.h`.

```
static struct file_operations randops =
{
    .read = user_read
};

static struct miscdevice randdev =
{
    .minor = MISC_DYNAMIC_MINOR,
    .name = "randlkm",
    .fops = &randops
};
```

Trong đoạn mã trên, chỉ có thao tác *read* được định nghĩa do ý đồ của tác giả. Để đơn giản, ta không cần quan tâm đến thao tác *open*, khi chương trình ở user-space gọi `read()`, module sẽ tiến hành cả hai việc tạo ra số ngẫu nhiên và gửi số này lên user-space thay vì tách phần tạo số ngẫu nhiên vào thao tác *open*. Lưu ý rằng ta phải định nghĩa hàm `user_read` ở trước `randops` của chúng ta.

```
ssize_t user_read(struct file *fp, char *buf, size_t cnt, loff_t *of)
{
    unsigned int number = 0;
    get_random_bytes(&number, sizeof(number));
    NumToChar(number, data);

    if (copy_to_user(buf, data, strlen(data)) != 0)
    {
        printk("copy_to_user(): fatal error.\n");
        return -1;
    }
    return strlen(data);
}
```

Với hai hàm `get_random_bytes()` và `copy_to_user()` được định nghĩa trong `linux/random.h` và `linux/uaccess.h`, hàm `NumToChar()` dùng để chuyển kiểu số thành kiểu kí tự để gửi lên user-space, ta đã hoàn thành việc chuẩn bị cho device file.

Trong `rand_init()` và `rand_exit()`, thêm hai lệnh tương ứng sau để đăng kí và hủy đăng kí device file này với kernel.

```
misc_register(&randdev);
misc_deregister(&randdev);
```

Việc cuối cùng là ghép tất cả lại với nhau, biên dịch module và gắn vào kernel. Sau khi gắn vào kernel, kiểm tra device file có được tạo và hủy khi gắn và tháo module hay không, ta có thể tìm trong thư mục `/dev`. Nếu thành công, tiếp tục viết một chương trình ở user-space, gọi `open()` và `read()` từ file này.

```
int fd = open("/dev/" "randlkm", O_RDWR);
if (fd < 0)
{
    printf("open() error.\n");
    return -1;
}
if (read(fd, data, len) < 0)
{
    printf("read() error.\n");
    return -1;
}
printf("Read: %s.\n", data);
close(fd);
```

Biên dịch bằng `gcc`, cuối cùng là chạy thử (phải chạy bằng quyền root mới có thể mở các file trong `/dev`). Nếu thành công, ta sẽ nhận được các kết quả như hình bên dưới.

```
adts@adts-VirtualBox: ~/Documents/randlkm
File Edit View Search Terminal Help
adts@adts-VirtualBox:~/Documents/randlkm$ make
make -C /lib/modules/4.15.0-45-generic/build M=/home/adts/Documents/randlkm modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-45-generic'
CC [M] /home/adts/Documents/randlkm/randlkm.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/adts/Documents/randlkm/randlkm.mod.o
LD [M] /home/adts/Documents/randlkm/randlkm.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-45-generic'
adts@adts-VirtualBox:~/Documents/randlkm$ sudo insmod randlkm.ko
[sudo] password for adts:
adts@adts-VirtualBox:~/Documents/randlkm$ gcc -o randtest randtest.c
adts@adts-VirtualBox:~/Documents/randlkm$ sudo ./randtest
Read: 3697610572.
adts@adts-VirtualBox:~/Documents/randlkm$ sudo ./randtest
Read: 1745896688.
adts@adts-VirtualBox:~/Documents/randlkm$ sudo ./randtest
Read: 3105732956.
adts@adts-VirtualBox:~/Documents/randlkm$ sudo ./randtest
Read: 631674815.
adts@adts-VirtualBox:~/Documents/randlkm$ sudo rmmod randlkm
adts@adts-VirtualBox:~/Documents/randlkm$
```

3. Randomnumber LKM

IV. Hooking system call:

Hooking là một dạng chỉnh sửa hay “đầu độc” các system call ở mức kernel-space, mục đích để theo dõi hệ thống, thay đổi hoạt động hoặc phá hoại các chức năng của system call. Kernel module là một cách đơn giản để thực hiện hooking, dĩ nhiên nó cũng rất đơn giản để chống lại.

a. Bảng địa chỉ system call:

Mã nguồn của các system call được lưu ở những vị trí cố định nào đó trong bộ nhớ. Linux dùng một bảng gọi là `sys_call_table` để lưu trữ địa chỉ của các system call. Bản thân `sys_call_table` cũng được lưu trên bộ nhớ nên nó cũng có địa chỉ.

Khó khăn ở các kernel hiện đại, địa chỉ `sys_call_table` không cố định, đây là một trong những cơ chế phòng chống hooking. Tuy nhiên ta hoàn toàn có thể tìm được địa chỉ `sys_call_table` ở thời điểm biên dịch bằng hàm `kallsyms_lookup_name()` trong `linux/kallsyms.h`.

```
void **sys_call_table_addr;

...

hooker_init(void)
{
    sys_call_table_addr = (void*)kallsyms_lookup_name("sys_call_table");
    ...
}
```

Để lấy con trỏ trỏ đến các system call, ta chỉ cần dùng các hằng số numeric reference `__NR` trong `asm/unistd.h`. Ở đây ta dùng `__NR_open` và `__NR_write` để lấy địa chỉ của system call `open()` và `write()`. Ta sẽ backup các địa chỉ này, rồi thay thế bằng địa chỉ của các system call mới do chúng ta viết ra.

Còn một vấn đề nữa với các kernel hiện đại, đó chính là quyền ghi. Các kiến trúc CPU mới thường dùng thanh ghi CR0 để ngăn chặn thay đổi trong kernel, và các kernel mới cũng có cơ chế bảo vệ `sys_call_table` khỏi những chỉnh sửa. Vì vậy, trước khi có thể thay đổi `sys_call_table`, ta phải tắt hết tất cả cơ chế bảo vệ này đi.

```
hooker_init(void)
{
    ...

    write_cr0(read_cr0() & (~0x10000));
    make_page_rw((unsigned long)sys_call_table_addr);
    //<Doing something>
    write_cr0(read_cr0() | 0x10000);
    make_page_ro((unsigned long)sys_call_table_addr);
}
```

Trong đó, các hàm `asm write_cr0()` và `read_cr0()` được hỗ trợ sẵn trên Linux, còn `make_page_rw()` và `make_page_ro()` được định nghĩa như sau:

```
void make_page_rw(unsigned long addr)
{
    unsigned int level;
    pte_t *pte = lookup_address(addr, &level);
    if (pte->pte & ~_PAGE_RW)
        pte->pte |= _PAGE_RW;
}

void make_page_ro(unsigned long addr)
{
    unsigned int level;
    pte_t *pte = lookup_address(addr, &level);
    pte->pte &= ~_PAGE_RW;
}
```

Bây giờ ta hoàn toàn có thể truy cập và chỉnh sửa `sys_call_table`. Đầu tiên, viết một hàm `open()` mới có thêm tác dụng ghi và log tên những tiến trình đã mở file và tên file được mở. Ta sẽ cần đến con trỏ `current` trong `linux/sched.h` và hàm `copy_from_user()` trong `linux/uaccess.h` để thực hiện việc này.

```
asmlinkage int (*original_open)(const char*, int);
asmlinkage int new_open(const char *pathname, int flags)
{
    char path[512];
    copy_from_user(path, pathname, strlen(pathname));
    path[strlen(pathname)] = '\0';

    printk(KERN_INFO "[**] %s has called open() on %s.\n", current->comm,
path);

    return original_open(pathname, flags);
}
```

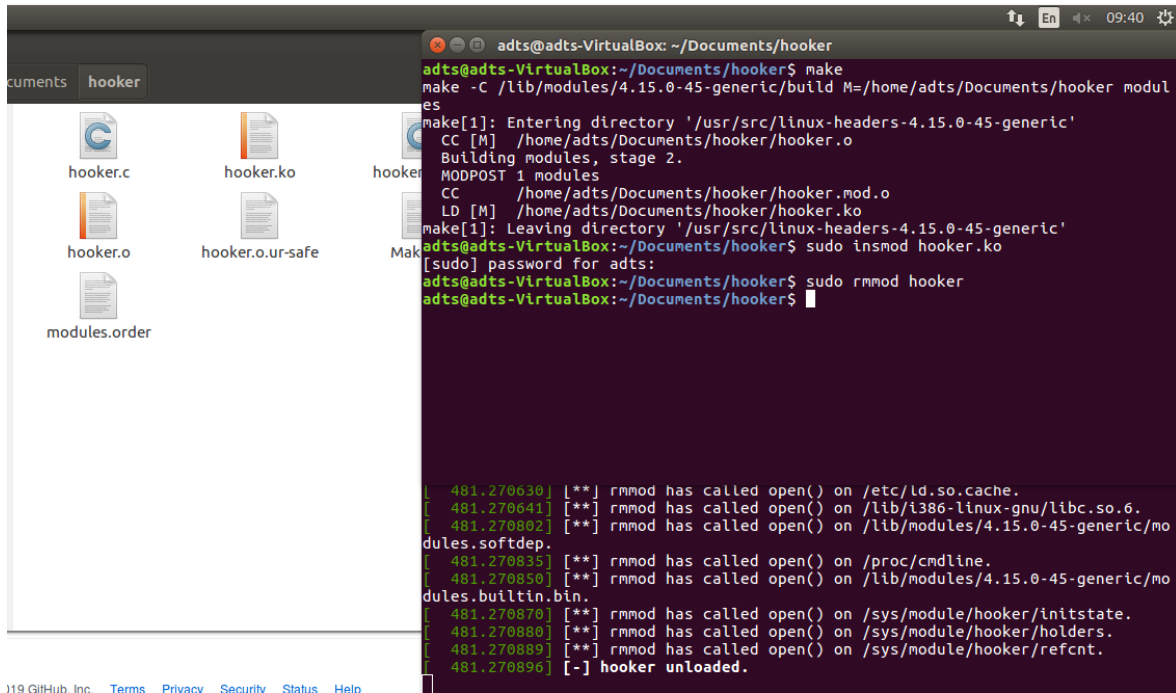
Trong `hooker_init()`, sau khi đã tắt cơ chế bảo vệ, ta dùng các numeric reference `__NR` trong `asm/unistd.h` để truy cập và thay thế syscall `open()` bằng hàm `new_open()` vừa viết.

```
original_open = sys_call_table_addr[__NR_open];
sys_call_table_addr[__NR_open] = new_open;
```


Còn trong `hooker_exit()`, ta phải khôi phục lại syscall `open()` cũ cho kernel.

```
sys_call_table_addr[__NR_open] = original_open;
```

Sau khi hoàn tất, tiến hành biên dịch module và gắn vào hệ thống. Nếu ta thấy có các log của hàm `printk()` trong `new_open()` nghĩa là ta đã hook system call `open()` thành công.

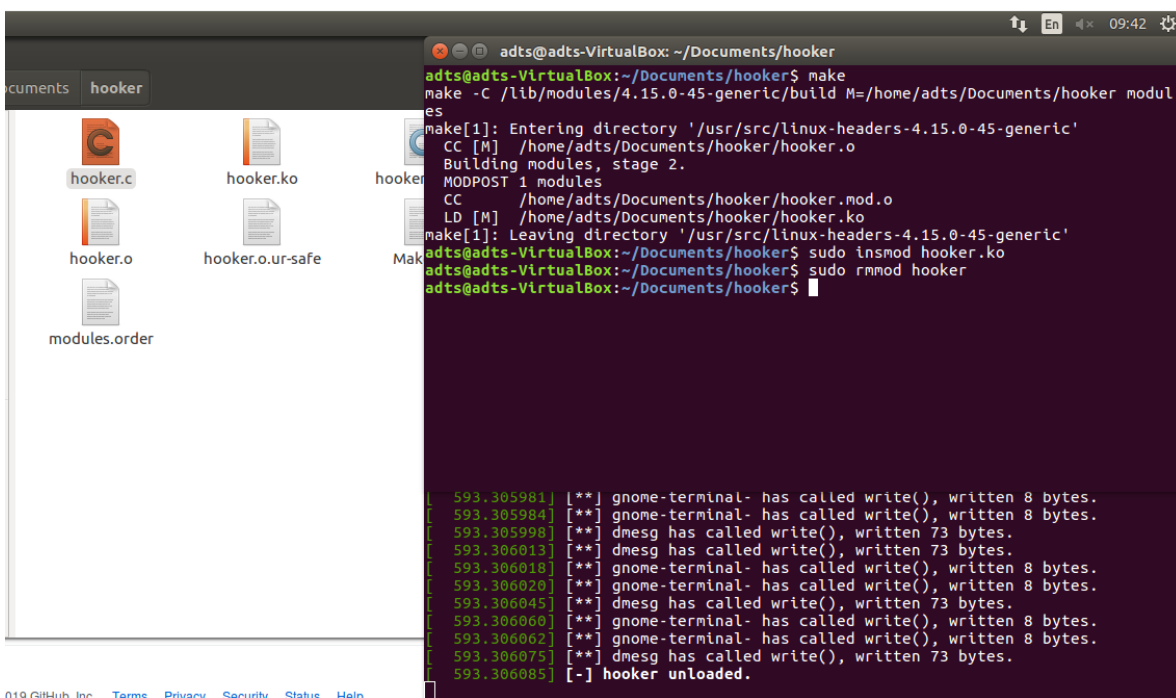


```
adts@adts-VirtualBox: ~/Documents/hooker
adts@adts-VirtualBox:~/Documents/hooker$ make
make -C /lib/modules/4.15.0-45-generic/build M=/home/adts/Documents/hooker modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-45-generic'
CC [M] /home/adts/Documents/hooker/hooker.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/adts/Documents/hooker/hooker.mod.o
LD [M] /home/adts/Documents/hooker/hooker.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-45-generic'
adts@adts-VirtualBox:~/Documents/hooker$ sudo insmod hooker.ko
[sudo] password for adts:
adts@adts-VirtualBox:~/Documents/hooker$ sudo rmmod hooker
adts@adts-VirtualBox:~/Documents/hooker$
```

```
[ 481.270630] [**] rmmod has called open() on /etc/ld.so.cache.
[ 481.270641] [**] rmmod has called open() on /lib/1386-linux-gnu/libc.so.6.
[ 481.270802] [**] rmmod has called open() on /lib/modules/4.15.0-45-generic/modules.softdep.
[ 481.270835] [**] rmmod has called open() on /proc/cmdline.
[ 481.270850] [**] rmmod has called open() on /lib/modules/4.15.0-45-generic/modules.builtin.bin.
[ 481.270870] [**] rmmod has called open() on /sys/module/hooker/initstate.
[ 481.270880] [**] rmmod has called open() on /sys/module/hooker/holders.
[ 481.270889] [**] rmmod has called open() on /sys/module/hooker/refcnt.
[ 481.270896] [-] hooker unloaded.
```

4. Hook open()

Tương tự cho `write()`, ta cũng backup syscall `write()` gốc và thay thế bằng `new_write()`. Lúc này ta ghi log tên tiến trình ghi file và số byte được ghi vào file.

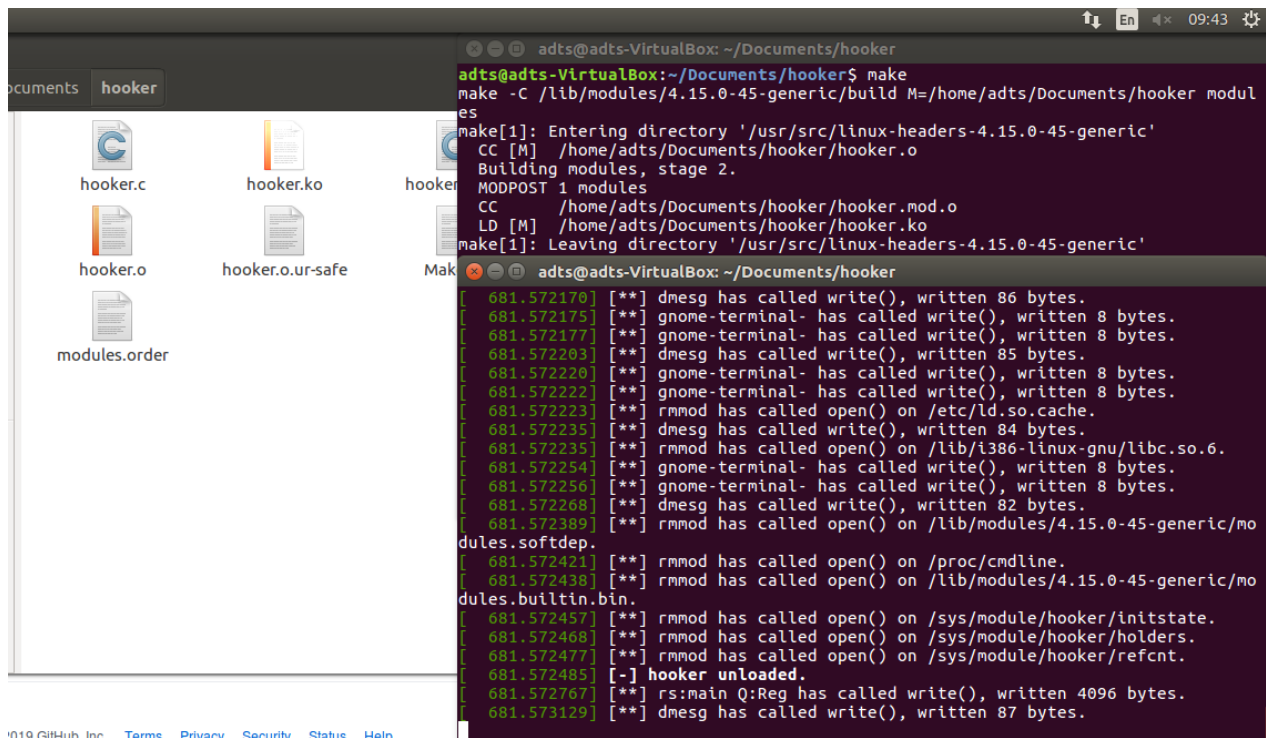


```
adts@adts-VirtualBox: ~/Documents/hooker
adts@adts-VirtualBox:~/Documents/hooker$ make
make -C /lib/modules/4.15.0-45-generic/build M=/home/adts/Documents/hooker modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-45-generic'
CC [M] /home/adts/Documents/hooker/hooker.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/adts/Documents/hooker/hooker.mod.o
LD [M] /home/adts/Documents/hooker/hooker.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-45-generic'
adts@adts-VirtualBox:~/Documents/hooker$ sudo insmod hooker.ko
adts@adts-VirtualBox:~/Documents/hooker$ sudo rmmod hooker
adts@adts-VirtualBox:~/Documents/hooker$
```

```
[ 593.305981] [**] gnome-terminal- has called write(), written 8 bytes.
[ 593.305984] [**] gnome-terminal- has called write(), written 8 bytes.
[ 593.305998] [**] dmesg has called write(), written 73 bytes.
[ 593.306013] [**] dmesg has called write(), written 73 bytes.
[ 593.306018] [**] gnome-terminal- has called write(), written 8 bytes.
[ 593.306020] [**] gnome-terminal- has called write(), written 8 bytes.
[ 593.306045] [**] dmesg has called write(), written 73 bytes.
[ 593.306060] [**] gnome-terminal- has called write(), written 8 bytes.
[ 593.306062] [**] gnome-terminal- has called write(), written 8 bytes.
[ 593.306075] [**] dmesg has called write(), written 73 bytes.
[ 593.306085] [-] hooker unloaded.
```

5. Hook write()

Kết hợp hooking vào cả `new_open()` và `new_write()` để được một module theo dõi hoàn chỉnh.



```

adts@adts-VirtualBox: ~/Documents/hooker
adts@adts-VirtualBox:~/Documents/hooker$ make
make -C /lib/modules/4.15.0-45-generic/build M=/home/adts/Documents/hooker modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-45-generic'
CC [M] /home/adts/Documents/hooker/hooker.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/adts/Documents/hooker/hooker.mod.o
LD [M] /home/adts/Documents/hooker/hooker.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-45-generic'

adts@adts-VirtualBox: ~/Documents/hooker
681.572170 [**] dmesg has called write(), written 86 bytes.
681.572175 [**] gnome-terminal- has called write(), written 8 bytes.
681.572177 [**] gnome-terminal- has called write(), written 8 bytes.
681.572203 [**] dmesg has called write(), written 85 bytes.
681.572220 [**] gnome-terminal- has called write(), written 8 bytes.
681.572222 [**] gnome-terminal- has called write(), written 8 bytes.
681.572223 [**] rmmmod has called open() on /etc/ld.so.cache.
681.572235 [**] dmesg has called write(), written 84 bytes.
681.572235 [**] rmmmod has called open() on /lib/i386-linux-gnu/libc.so.6.
681.572254 [**] gnome-terminal- has called write(), written 8 bytes.
681.572256 [**] gnome-terminal- has called write(), written 8 bytes.
681.572268 [**] dmesg has called write(), written 82 bytes.
681.572389 [**] rmmmod has called open() on /lib/modules/4.15.0-45-generic/modules.softdep.
681.572421 [**] rmmmod has called open() on /proc/cmdline.
681.572438 [**] rmmmod has called open() on /lib/modules/4.15.0-45-generic/modules.builtins.bin.
681.572457 [**] rmmmod has called open() on /sys/module/hooker/initstate.
681.572468 [**] rmmmod has called open() on /sys/module/hooker/holders.
681.572477 [**] rmmmod has called open() on /sys/module/hooker/refcnt.
681.572485 [-] hooker unloaded.
681.572767 [**] rs:main Q:Reg has called write(), written 4096 bytes.
681.573129 [**] dmesg has called write(), written 87 bytes.

```

6. Hook `read()` và `write()`

V. Tài liệu đã tham khảo:

<https://exploit.ph/categories/linux-kernel-hacking/>

<https://memset.wordpress.com/2010/12/03/syscall-hijacking-kernel-2-6-systems/>

*Tài liệu hướng dẫn thực hành của khoa Công nghệ Thông tin – đại học Khoa học Tự nhiên

--- Hết ---