Trường đại học Khoa học Tự nhiên - Đại học Quốc gia thành phố Hồ Chí Minh

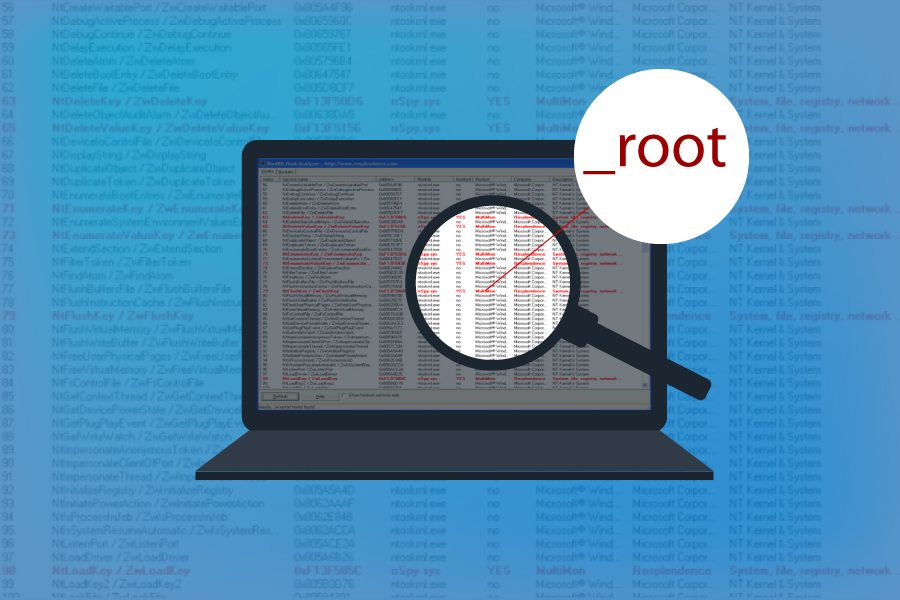
Khoa Công Nghệ Thông Tin

------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%

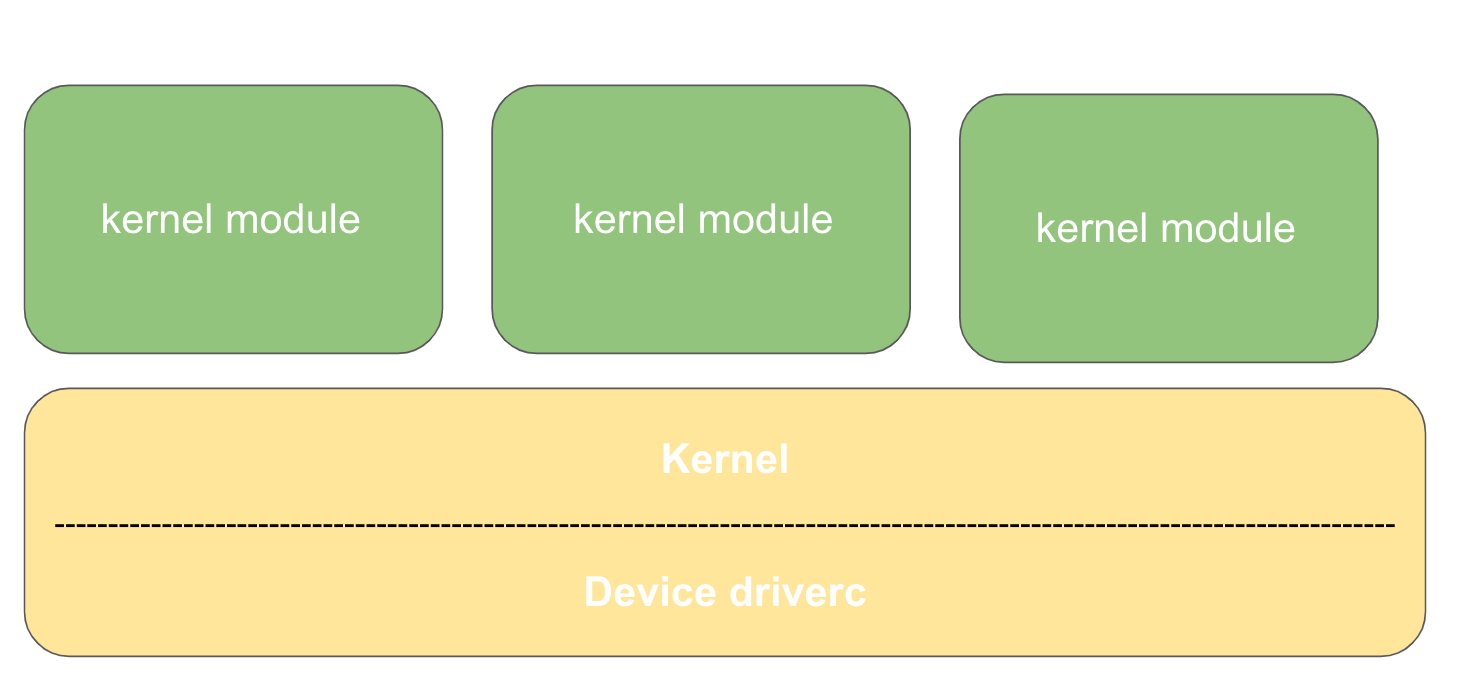
Module hook vào syscall write() 90%

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.

A screenshot of a cell phone

Description automatically generated

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 miscdevive 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.

A screenshot of a cell phone

Description automatically generated

3. Random number 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\_add = (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.

A screenshot of a cell phone

Description automatically generated

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.

A screenshot of a cell phone

Description automatically generated

5. Hook write()

Kết hợp hooking vào cả new\_open() và new\_write() để được một module theo dõi hoàn chỉnh.

A screenshot of a computer

Description automatically generated

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 ---