Báo cáo đồ án 2:

Linux Kernel Modules & Syscalls Hooking

Môn: Hệ điều hành

Sinh viên thực hiện:

Nguyễn Ngọc Băng Tâm - 1712747

Bùi Thị Cẩm Nhung - 1712645

11/2019

1. Tổng quan

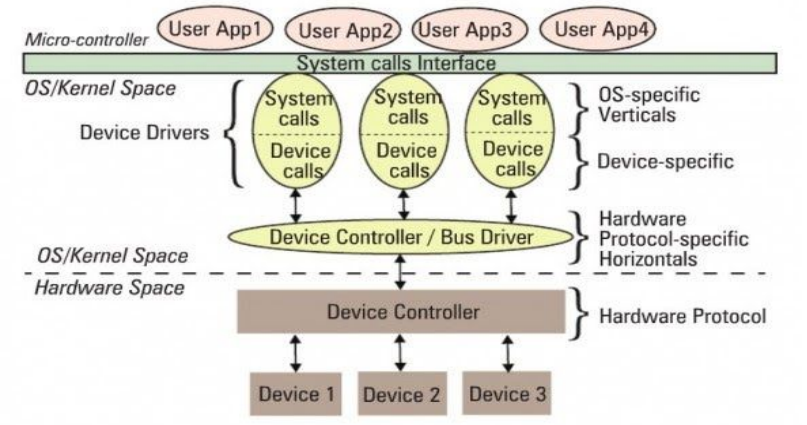
1.1 Mô tả đồ án

Kernel Module là tên gọi chung của các đoạn code có khả năng nạp vào và gỡ ra khỏi kernel khi cần thiết mà không cần phải reboot lại toàn bộ hệ thống. Chính vì sự linh hoạt này mà nó còn có tên gọi khác là Loadable Kernel Module (LKM).

Sự xuất hiện của LKM mang nhiều ý nghĩa lớn: giúp thu gọn kích thước của kernel, đồng thời đem đến sự linh hoạt khi thêm mới hoặc thay đổi driver trong hệ thống, đặc biệt đối các máy server.

LKM được chia làm 3 loại chính: device driver, system call và file system.

Mô tả một cách đơn giản, device driver là một trình điều khiển cho phép quản lý, giám sát việc vận hành một thiết bị phần cứng cụ thể gắn với máy tính (chẳng hạn bàn phím, chuột). Mỗi device driver cung cấp cho người dùng một giao diện gọi các hàm hệ thống system call đến tầng ứng dụng. Đây cũng là ranh giới giữa user space và kernel space.



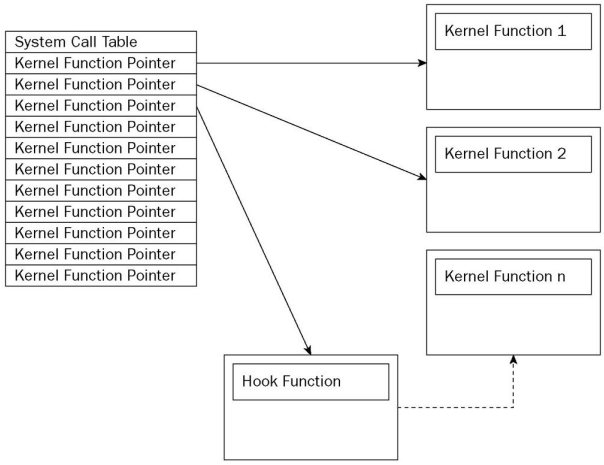
Hình 01. Các thành phần của driver trên Linux

Character Device Driver là device driver mà ở đó việc vận hành các thiết bị (chẳng hạn thiết bị đọc, viết) được thực hiện tuần tự từng byte một (byte-oriented). Một character device driver tiếp nhận một lời gọi system call từ user thông qua file thiết bị gọi là character device file, nói chính xác hơn là thông qua tên file của thiết bị đó.

Phần 1 của đồ án này tập trung tìm hiểu về cách xây dựng một LKM phát sinh số ngẫu nhiên; đồng thời tạo một character device driver cho phép các tiến trình ở user space open và read các số ngẫu nhiên đó.

Hooking là một kĩ thuật giúp can thiệp vào hành vi của một hệ điều hành, một phần mềm hoặc một thành phần ứng dụng nào đó nhằm nắm quyền kiểm soát, kiểm lỗi hoặc mở rộng tính năng của thành phần đó.

Có nhiều loại hooking khác nhau hướng vào các thành phần khác nhau. Trong đó, system call hooking là một phương pháp hook vào kernel hiện hành nhằm thay đổi cách thức hoạt động vốn có của một system call nào đó theo chủ ý của người hook. Cách làm được xem là đơn giản nhất để thực hiện một system call hooking là truy cập và thay đổi trực tiếp trên bảng system call trong kernel space.



Hình 02. Mô tả bảng system call sau khi được hook

Phần 2 của đồ án yêu cầu thực hiện system call hooking với hai syscall open và write; đồng thời ghi vào dmesg tên tiến trình mở / ghi file cùng với tên file được mở / ghi (cùng số byte nếu có).

1.2 Đánh giá mức độ hoàn thành

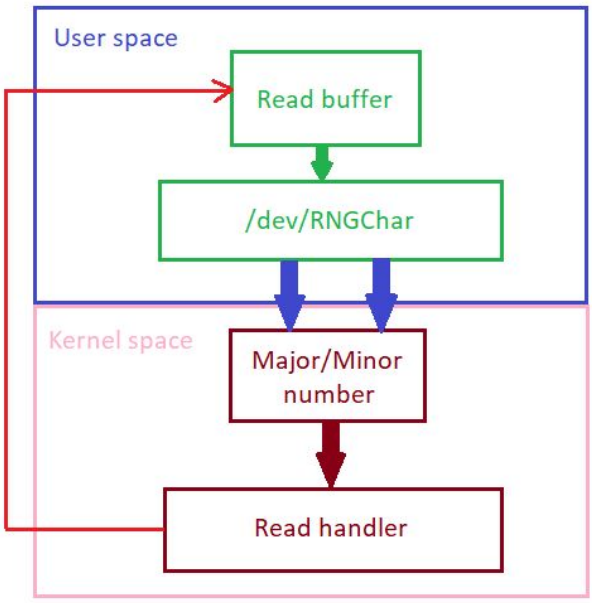
- Đánh giá tổng thể: 100%
- Chi tiết từng yêu cầu:

Yêu cầu	Hoàn thành	Người thực hiện
Module tạo số ngẫu nhiên	x	Tâm
Tạo character device driver cho phép tiến trình ở user space open và read các số ngẫu nhiên được tạo	x	Tâm
Chương trình hook vào system call open	x	Nhung
Chương trình hook vào system call write	x	Nhung

2. Thiết kế

2.1 Random Number Generator

Cấu trúc chương trình sẽ gồm 2 tập tin: 1 kernel module phát sinh số ngẫu nhiên và 1 chương trình gọi từ user space xuống kernel.



Hình 03. Cơ chế hoạt động của character device driver

2.1.1 Tổng quan về tạo kernel module phát sinh số ngẫu nhiên

Để viết kernel module, ta cần biết nó được viết bằng C thuần và không thể sử dụng các thư viện truyền thống trong user space như stdlib.h. Chính vì vậy, Linux kernel sẽ đóng vai trò là thư viện, cung cấp các hàm và macro để ta phát triển kernel module. Các thư viện này sẽ có dạng <linux/tên_thư_viện.h>.

Kernel module không có hàm main, nhưng vẫn tuân theo lập trình hướng đối tượng. Do đó, một kernel module bao gồm ít nhất 2 hàm cơ bản là constructor (có macro tương ứng là __init) và destructor (có macro tương ứng là __exit). Sau đó, cần truyền chúng vào module_init và module_exit để thực hiện khi lắp / tháo module vào / ra khỏi kernel.

Để phát sinh số ngẫu nhiên, ta sử dụng hàm get_random_bytes nằm trong thư viện <linux/random.h>. Ngoài ra, để giới hạn số ngẫu nhiên phát sinh không vượt quá một giá trị MAX, ta thực hiện chia lấy dư số được phát sinh cho MAX.

Đoạn code dưới đây minh họa cho việc phát sinh số ngẫu nhiên khi viết module, trên thực tế khi kết hợp vào character device sẽ được chỉnh sửa cho phù hợp.

```
#define MAX 1000

int randNum;
get_random_bytes(&randNum, sizeof(randNum));
randNum %= MAX;
```

2.1.2 Tạo character device phát sinh số ngẫu nhiên ở kernel

Để tạo character device driver phát sinh số ngẫu nhiên, thực hiện:

1. Đăng ký số hiệu file thiết bị
2. Đăng ký toán tử file sẽ sử dụng
3. Đăng ký lớp thiết bị ảo (virtual device class) và khởi tạo thiết bị tự động với số hiệu đã đăng ký
4. Cài đặt toán tử mở file và đọc file, trong đó toán tử đọc sẽ trả số phát sinh ngẫu nhiên về user

Cấu trúc dữ liệu: có 3 cấu trúc dữ liệu trong thao tác với file: struct file_operations, struct file và struct inode.

- struct file_operations: chứa các hàm toán tử muốn thực hiện đối với file thiết bị (ở đồ án này ta quan tâm tới toán tử open, release và read).

```
struct file_operations {
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
};
```

- struct file: chỉ file cần thao tác, đứng dưới góc nhìn của người dùng.
- struct inode: chỉ file cần thao tác, nhưng đứng từ góc nhìn hệ thống.

Có thể sử dụng thêm struct cdev để đăng ký với hệ thống về sự tồn tại của character device nếu dùng cách cài đặt 2 như trình bày ở phần 2.1.3.

Phương pháp cài đặt: có 2 cách đăng ký số hiệu file và toán tử file:

1. Đăng ký cùng lúc số hiệu và toán tử file bằng hàm register_chrdev

2. Đăng ký số hiệu file trước và toán tử file riêng lẻ
    + Hàm register_chrdev_region (đăng ký tĩnh số hiệu) hoặc alloc_chrdev_region (xin cấp phát động số hiệu).
    + Hàm cdev_add để thêm character device vào hệ thống sau khi khởi tạo struct cdev chứa struct toán tử file cần dùng.

Các hàm sử dụng trong chương trình sẽ có phần khác nhau khi cài đặt theo 2 cách khác nhau. *Trong đồ án này, nhóm lựa chọn cài đặt theo cách 1.* Các bước thực hiện gồm:

1. Khai báo các cấu trúc dữ liệu cần thiết
2. Đăng ký số hiệu file thiết bị và toán tử file. Để xin cấp phát số hiệu major động, truyền tham số major = 0.

```
void register_chrdev (unsigned int  major, const char *deviceName, const struct file_operations *fops)
```

3. Đăng ký lớp thiết bị ảo (virtual device class)

```
struct class* class create (struct module *classOwner, const char *className)
```

4. Khởi tạo device driver theo lớp thiết bị ảo và số hiệu file đã đăng ký. Sau khi thực hiện lệnh này, device driver sẽ được tạo trong folder /dev/<device_name>.

```
struct device* device create (struct class* deviceClass, struct device* parent, dev_t devt, const char* deviceName, ...)
```

5. Cài đặt chi tiết các toán tử open, read và release. Trong toán tử read, thực hiện phát sinh số ngẫu nhiên như phần 2.1.1 và sao chép về vùng nhớ của user space bằng hàm copy_to_user.

```
unsigned long copy_to_user (void user* user_space_buffer, const void* kernel_space_buffer, unsigned long numBytes)
```

6. Cài đặt hàm destructor với các hàm hủy theo thứ tự ngược lại với lúc khởi tạo:
    + Hủy device driver
    + Hủy đăng ký lớp thiết bị ảo
    + Xóa lớp thiết bị ảo
    + Hủy đăng ký và xóa character device

```
void device_destroy (struct class* deviceClass,  dev_t devt)
void class_unregister (struct class* deviceClass)
void class_destroy (struct class* deviceClass)
void unregister_chrdev (unsigned int majorNumber, const char *deviceName)
```

Lưu ý: Bước 2 - 4 được cài đặt trong hàm constructor của module.

2.1.3 Tạo chương trình ở user space gửi yêu cầu đến kernel:

Viết chương trình đơn giản với hàm main thực hiện việc mở file device driver đã tạo trong folder /dev/<device_name>.

Lưu ý: Chế độ mở file trong đồ án là O_RDONLY.

```
int open (const char *filePath, int openMode, ...)
```

Thực hiện đọc dữ liệu từ file device driver:

```
ssize_t read (int file, void *buffer, size_t numBytes)
```

2.2 System call hooking

2.2.1 Tổng quan về system call hooking

Để thực hiện system call hooking, ta sẽ cần xây dựng một LKM có khả năng nạp vào và tháo ra khỏi kernel trong thời gian thực nhằm chiếm được quyền thực thi trong kernel mode.

Như đã đề cập ở phần đầu, cách hook mà ta lựa chọn là truy cập trực tiếp vào bảng system call và thay đổi giá trị của kernel function pointer mong muốn. Quá trình này trải qua 4 bước làm chính sau đây:

1. Lưu địa chỉ gốc, tức kernel function pointer của system call cần hook.
2. Xây dựng hàm hook. Chú ý rằng trước khi hàm kết thúc phải trả về hàm gốc, tức kernel function pointer gốc của system call. Hàm hook phải có chữ ký hàm giống hàm gốc được hook.
3. Thay đổi kernel function pointer của system call cần hook thành địa chỉ của hàm hook khi khởi tạo module. Tuy nhiên để làm được điều này, ta cần tắt memory protection.
4. Phục hồi địa chỉ gốc của system call được hook khi tháo module, bật lại memory protection như ban đầu.

Lưu ý rằng để truy cập được vào bảng system call, trước tiên ta phải biết địa chỉ của nó bằng cách gõ lệnh sau và ghi nhớ mã hex địa chỉ (thay 3.16.36 bằng phiên bản của kernel tương ứng):

```
cat /boot/System.map-3.16.36 | grep sys_call_table
```

Tuy nhiên cách làm này không thực sự linh hoạt vì đòi hỏi phải thực hiện việc tìm kiếm thủ công ở mỗi lần hook và không thể trực tiếp đem module xây dựng được sang một kernel khác.

Đoạn code dưới đây mô tả tổng quan quá trình tự động tìm kiếm địa chỉ của bảng system call và thực hiện hook đối với hai syscal là open và write:

```
void **system_call_table_addr;

asmlinkage long      (*original_open) (const char *, int, mode_t);
asmlinkage ssize_t   (*original_write) (int, const void *, size_t);

asmlinkage long hooked_open (const char *pathname, int flags, mode_t mode)
{
   [...]
```

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

asmlinkage ssize_t hooked_write (int fd, const void *buf, size_t nbytes) {
   [...]
   return original_write(fd, buf, nbytes);
}

/* Make page writable */
int make_rw (unsigned long address) {
   unsigned int level;
   pte_t *pte  = lookup_address(address, &level);
   if (pte->pte &~_PAGE_RW) {
      pte->pte |=_PAGE_RW;
   }
   return 0;
}

/* Make page write protected */
int make_ro (unsigned long address) {
   unsigned int level;
   pte_t *pte  = lookup_address(address, &level);
   pte->pte    = pte->pte &~_PAGE_RW;
   return 0;
}

static int __init entry_point (void) {
   /* Dynamically looking for system call table address */
   system_call_table_addr = (void **)
kallsyms_lookup_name("sys_call_table");

   /* Save current system call */
   original_open  = system_call_table_addr[__NR_open];
   original_write = system_call_table_addr[__NR_write];

   /* Load hooked system call */
   make_rw((unsigned long) system_call_table_addr);
   system_call_table_addr[__NR_open]   = hooked_open;
   system_call_table_addr[__NR_write]  = hooked_write;
   make_ro((unsigned long) system_call_table_addr);
   return 0;
```

```
}

static void __exit exit_point (void) {
   /* Restore original system call */
   make_rw((unsigned long) system_call_table_addr);
   system_call_table_addr[__NR_open]   = original_open;
   system_call_table_addr[__NR_write]  = original_write;
   make_ro((unsigned long) system_call_table_addr);
}

module_init(entry_point);
module_exit(exit_point);
```

2.2.2 Hàm hook syscall open

Mục tiêu: ghi vào dmesg tên tiến trình mở file và tên file được mở.

Để lấy tên tiến trình mở file, ta thực hiện theo 2 bước sau:

1. Truy cập vào biến toàn cục con trỏ kiểu task_struct có tên là current được định nghĩa trong thư viện <linux/sched.h>. Con trỏ current này có nhiệm vụ trỏ đến process hiện hành.
2. Lấy thuộc tính comm (tức command name) của con trỏ current tương ứng với tên của process hiện hành.

Tên file được mở là một tham số được truyền vào khi gọi open. Tuy nhiên con trỏ kiểu char này đang trỏ đến một vùng nhớ ở user space còn syscall đang chạy ở kernel space. Chính vì vậy, ta cần sử dụng copy_from_user để copy vùng nhớ này từ user space sang kernel space.

Để ghi thông tin vào dmesg, ta gọi hàm printk (hàm tương tự của printf nhưng ở kernel space).

```
asmlinkage long hooked_open (const char *pathname, int flags, mode_t mode)
{
   char *kfname   = kmalloc(1024, GFP_KERNEL);
   copy_from_user(kfname, pathname, 1024);
   printk(KERN_INFO "Process %s opens %s", current->comm, kfname);
   kfree(kfname);
   return original_open(pathname, flags, mode);
```

```
}
```

2.2.3 Hàm hook syscall write

Mục tiêu: ghi vào dmesg tên tiến trình ghi file, tên file được ghi và số byte ghi.

Ta thực hiện các bước làm tương tự như trên để lấy tên tiến trình ghi file.

Số byte ghi là một tham số được truyền vào khi gọi write.

Khi gọi write, ta còn truyền một thêm một tham số nữa là file descriptor. File descriptor là một số nguyên dùng để định danh các file trong hệ thống. Dựa vào file descriptor, ta có thể suy ra tên file bằng cách tra vào bảng file descriptor table được định nghĩa trong thư viện <linux/fdtable.h>.

```
asmlinkage ssize_t hooked_write (int fd, const void *buf, size_t nbytes) {
   /* Retrieve pathname from file descriptor using fdtable */

   char *buffer   = kmalloc(1024, GFP_KERNEL);
   char *kfname   = d_path(&fcheck_files(current->files, fd)->f_path,
buffer, 1024);
   printk(KERN_INFO "Process %s writes %zu bytes to %s", current->comm,
nbytes, kfname);
   kfree(buffer);
   return original_write(fd, buf, nbytes);
}
```

3. Tài liệu tham khảo

1. Linux Device Drivers, 2nd Edition
2. Write a Linux Kernel Module: Character Device Driver
3. Linux Kernel Module