Cài đặt system call

Mô hình bảo mật của hệ điều hành Linux có dạng vòng tròn (rings model), trong đó các vòng ngoài yêu cầu quyền truy cập ở mức thấp hoặc không yêu cầu quyền, nơi các ứng dụng bên thứ ba hoạt động. Khi càng vào những vòng trong, càng gần cùng kernel thì yêu cầu quyền truy cập càng cao. Có thể thấy, để đảm bảo cơ chế bảo mật, các chương trình ứng dụng ở tầng Userspace không thể truy cập trực tiếp đến các phần cứng, hoặc truy cập vào vùng nhớ của các ứng dụng khác.

Tuy nhiên trên thực tế có nhiều ứng dụng cần kết nối với những thành phần này để có thể thực hiện được công việc của nó. Khi đó, ***system call*** ra đời, là cách để chương trình ở userspace yêu cầu một dịch vụ (service) từ tầng nhân của hệ điều hành, bao gồm các dịch vụ liên quan đến phần cứng, tạo và thực thi 1 tiến trình cũng như giao tiếp với các dịch vụ tích hợp trong phần nhân của hệ điều hành. System call hệ điều hành kiểm soát được những gì chương trình ứng dụng có thể truy cập, đảm bảo tính bảo mật cho hệ thống.

Dưới đây là các bước cài đặt một system call, cách biên dịch và test system call vừa được tạo.

Các syscall được implement trên hệ điều hành Ubuntu 18.04, Kernel được build là phiên bản 4.17.4, các câu lệnh sau được thực hiện trong Kernel với quyền quản trị cao nhất (sudo)

a. *Cài mã nguồn và khai báo syscall*

Tải source kernel 4.17.4 về máy:

```
wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.17.4.tar.xz
```

Lệnh `wget` là lệnh dùng để tải file trên web

Giải nén tập tin vừa tải về vào thư mục /usr/src/

```
  tar -xvf linux-3.16.36.tar.xz -C /usr/src/
```

`tar` command: để giải nén hoặc nén file

`-x`: giải nén file từ một tài liệu (-c là tạo file nén)

`-v`: bật tùy chọn `-verbose`, in ra các lỗi nếu có trong quá trình giải nén

`-f`: viết tắt của `format`, sử dụng định dạng mặc định (.tar)

`-c`: giải nén đến một thư mục cụ thể (trong trường hợp này là /usr/src)

Sau đó, thay đổi thư mục làm việc hiện hành đến nơi ta vừa giải nén file

```
  cd /usr/src/linux-3.16.36/
```

Tạo thư mục mới để chứa mã nguồn system call của mình, đặt tên là mysyscall, sau đó chuyển thư mục làm việc đến thư mục mysyscall này:

```
mkdir mysyscall
cd mysyscall
```

Tạo file mysycall.c để viết mã nguồn của syscall, ở đây dùng trình soạn thảo nano:

```
nano mysyscall.c
```

```
#include <linux/syscalls.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/string.h>
#include "process_name_id.h"

asmlinkage int pnametoid(char* pname){
    // tasklist struct to use
    struct task_struct *task;

    char* s;
    copy_from_user(s,pname,strlen(pname));

    // Iterate through every running process
    for_each_process(task){
        // compares the current process name (defined in task->comm)
to the passed in name
        if(strcmp(task->comm,s) == 0){
            return task_pid_nr(task);
        }
    }
    return -1;
}

asmlinkage int pidtoname(int pid, char* buf, int len) {
    // tasklist struct to use
    struct task_struct *task;

    // Iterate through every running process
    for_each_process(task){
        if(task_pid_nr(task) == pid){
            if (strlen(task->comm) <= len){
                sprintf(buf,"%s",task->comm);
                return 1;
            }
```

```
            else {
                return strlen(task->comm);
            }
        }
    }
    return -1;
}
```

Hàm `int pnametoid(char* pname)`: nhận vào tham số là tên của process, trả về PID của process đó bằng các sử dụng hàm `task_pid_nr(task)` để lấy PID của process task. Nếu không tìm thấy process tương ứng, hàm trả về -1

Hàm `int pidtoname(int pid, char* buf, int len)`: nhận vào PID của process, con trỏ chuỗi kết quả trả về và chiều dài tối đa của tên process. Nếu tìm được process có PID trùng với PID truyền vào, hàm này trả về kết quả là 1 nếu process tìm được có tên dài không vượt quá len, trong trường hợp này tên của process được copy vào chuỗi buf, ngược lại trả về chiều dài của tên process nếu tên của process có độ dài vượt quá giá trị len. Nếu không tìm thấy process tương ứng, hàm trả về -1.

Sau khi tạo xong file mã nguồn, ta tạo thêm file header bằng câu lệnh sau:

```
nano mysycall.h
```

Nội dung file header là khai báo 2 hàm này:

```
asmlinkage int pnametoid(char* pname);
asmlinkage int pidtoname(int pid, char* buf, int len);
```

Tạo file Makefile để biên dịch 2 syscall này. Tại thư mục hiện hành, ta gõ lệnh

```
nano Makfile
```

Sau đó soạn thảo nội dung Makefile là:

```
obj := mysyscall.o
```

Dùng lệnh `cd ..` để thoát khỏi thư mục mysysall

```
cd ..
```

Chỉnh sửa file Makefile của kernel:

```
nano Makefile
```

Thêm thư mục mysyscall vào sau dòng `core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/`. Được `core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ mysyscall/`. Thao tác này giúp cho hệ điều hành khi biên dịch chương trình sẽ biết mã nguồn chương trình ở đâu để tiền hành biên dịch

Mỗi system call được quản lý thông qua tên và một con số gọi là PID, con số này được gán cho system call trong bảng system call của kernel. Dùng lệnh sau mở bảng syscall, để thêm tên system call của mình vào:

```
nano arch/x86/entry/syscalls/syscall_64.tbl
```

Đây là bảng các system call của kernel, mỗi dòng là 1 system call. Danh sách được chia thành 2 phần, các syscall dành cho hệ thống 64-bit (khoảng 320 dòng đầu tiên) và các syscall của hệ thống 32-bit (phần còn lại). Tùy hệ thống mà ta thêm syscall của mình vào phần nào.

```
333     common     mysyscall1 pnametoid
334     common     mysyscall2 pidtoname
```

Sau khi khai báo tên của system call, ta cần khai báo mẫu hàm của system call trong file header của kernel:

```
nano include/linux/syscalls.h
```

Thêm 2 dòng này vào cuối file, ngay trước `#endif`

```
asmlinkage int pnametoid(char* pname);
asmlinkage int pidtoname(int pid, char* buf, int len);
```

b. Compile and install the kernel mới

Sau khi đã viết mã nguồn cho syscall của mình, và khai báo tên cũng như tên hàm cho syscall trong kernel, ta tiến hành biên dịch kernel để có thể cài đặt. Trước khi compile kernel, ta cần chọn những module nào sẽ được load cùng với kernel và những module nào thì không. Dùng lệnh:

```
make menuconfig
```

Sau khi chọn những module cần thiết, ta chọn lần lượt: Save, Ok, Exit, Exit để lưu và thoát.

Để tiến hành compile kernel, ta dùng lệnh:

```
make -j $(nproc)
```

`nproc` để lấy được số core của hệ thống. Dùng lệnh trên để sử dụng tối đa số cores để quá trình compile được diễn ra nhanh hơn. Nếu chỉ muốn dùng 1 core cho việc compile và install kernel thì chỉ cần dùng lệnh make.

Sau đó tiến hành cài đặt các modules cần thiết:

```
make modules_install install -j $(nproc)
```

Cuối cùng là cài đặt kernel

```
make install -j $(nproc)
```

Sau khi các lệnh chạy xong và không gặp lỗi gì, ta khởi động lại hệ điều hành để sử dụng kernel mới

```
reboot
```

c. *Cách test syscall mà ta đã thêm*

Bằng các viết một chương trình ở userspace và gọi syscall với PID là của syscall mình đã tạo, ta sẽ biết được syscall của mình có hoạt động hay không.

Để tạo chương trình test, ta dùng lệnh:

```
nano testMysyscall.c
```

Soạn thảo nội dung file testMysyscall.c như sau

```c
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <string.h>
int main(){
    char name[32];
    puts("Enter process to find");
    scanf("%s",name);
    strtok(name, "\n");
    long int status = syscall(333, name);
    printf("System call pnametoid returned %ld\n", status);

    char name2[32]
    int pid;
    puts("Enter PID to find");
    scanf("%s",&pid);
    status = syscall(334, name2, n, 32);
    printf("Process %s\n", name2);
    printf("System call pidtoname returned %ld\n", status);
```

```
    return 0;
}
```

Dùng gcc để biên dịch chương trình:

```
gcc testMysyscall
```

Chạy chương trình bằng

```
./a.out
```

Nhập vào process name testMysyscall, nếu không gặp lỗi gì thì syscall sẽ trả về PID của process testMysyscall

```
Enter process to find
testMysyscall
System call pnametoid returned 1287
```

Nhập vào PID 1287, nếu không gặp lỗi gì thì syscall sẽ trả về process name testMysyscall

```
Enter PID to find
1287
Process testMysyscall
System call pidtoname returned 1
```

2. Hook vào system call

Hook là kỹ thuật thay đổi hoặc bổ sung một vài tính năng của một hàm (của ứng dụng), hoặc một system call (của hệ điều hành), bằng cách sử dụng một đoạn code chặn lại lời gọi hàm đó và thực thi các tính năng mới cần thiết. Do đó, mỗi khi có một lời gọi hàm đó được thực thi, thì tính năng mới được hook vào sẽ được thực hiện.

Hook đối với system call được thực hiện bằng cách tạo một Loadable Kernel Module (để có thể bật/tắt việc hook dễ dàng bằng cách insert/remove kernel module đó). Kernel module này sẽ thực hiện:

- Tạo một hàm thay thế cho hàm system call cần hook (nếu hàm này chỉ thêm tính năng cho system call thì trong hàm cần có lời gọi system call đó).
- Tìm vị trí con trỏ hàm trỏ đến system call cần hook trong bảng sys_call_table (một bảng của hệ điều hành chứa danh sách các con trỏ hàm đến các system call của hệ thống).
- Thay thế giá trị con trỏ hàm đó trỏ đến hàm thay thế mới.

- Khi remove kernel module, sửa giá trị con trỏ hàm này trỏ lại system call như cũ.

Dưới đây là chi tiết các bước thực hiện việc hook vào 2 system call Open và Write. Đối với syscall Open, ta sẽ thêm tính năng in ra dmesg tên tiến trình mở file và tên file được mở. Đối với `syscall Write`, ta sẽ in ra tên tiến trình ghi, tên file bị ghi và số byte được ghi.

a. Tìm địa chỉ của sys_call_table

Thông tin địa chỉ của bảng sys_call_table được lưu trong file /boot/System.map-4.17.4. Lấy địa chỉ của sys_call_table bằng cách chạy lệnh

```
cat /boot/System.map-4.17.4 | grep sys_call_table
```

```
hoanganh@ubuntu:~$ cat /boot/System.map-4.17.4 | grep sys_call_table
ffffffff81e00180 R sys_call_table
ffffffff81e01560 R ia32_sys_call_table
```

Dựa vào kết quả ta có địa chỉ của sys_call_table là `0xffffffff81e00180`. Khi đó, con trỏ hàm của syscall open sẽ là `sys_call_table[__NR_open]`, của syscall write là `sys_call_table[__NR_write]`.

b. Code hàm hook và các hàm cho kernel module

Ta tạo 2 kernel module, hook-open cho syscall Open và hook-write cho syscall Write. Source code cho 2 kernel module này có trong các file source code đính kèm. Dưới đây trình bày một số hàm quan trọng.

i. `entry_point()` *và* `exit_point()`

```
static int __init entry_point(void){
    //system call table address
    system_call_table_addr = (void*)0xffffffff81e00180;

    // Assign custom_syscall to system call OPEN
    custom_syscall = system_call_table_addr[__NR_open];

    //Disable page protection
    make_rw((unsigned long)system_call_table_addr);

    // Replace system call OPEN by our Hook function
    system_call_table_addr[__NR_open] = hook_function;

    return 0;
}
```

```
static int __exit exit_point(void){
    // Restore system call OPEN
    system_call_table_addr[__NR_open] = custom_syscall;

    //Renable page protection
    make_ro((unsigned long)system_call_table_addr);

    return 0;
}
```

Hàm `entry_point()` thực thi khi ta insert kernel module vào. Đầu tiên ta lưu địa chỉ `sys_call_table` vừa tìm được vào `sys_call_table_addr`. Biến `custom_syscall` sẽ lưu lại địa chỉ của `syscall open` (vì ta cần gọi `syscall open` sau này, lúc con trỏ hàm đến địa chỉ này sẽ bị thay đổi). Hàm `make_rw()` tắt chế độ bảo vệ của `sys_call_table`, để có thể thay đổi giá trị trong bảng. Sau đó, ta gán giá trị của `sys_call_table_addr[__NR_open]` (con trỏ hàm đến `syscall open`) trỏ đến hàm `hook_function()` của chúng ta.

ii. Hàm hook của hook-open

```
// Hook function, replace the system call OPEN
// Take the same parameter as the system call OPEN
asmlinkage int hook_function(const char *pathname, int flags)
{
    // Print calling process name
    printk(KERN_INFO "Calling process:%s\n",current->comm);

    // Print openning file
    printk(KERN_INFO "Openning file:%s\n",pathname);

    return custom_syscall(const char *pathname, int flags);
}
```

Hàm hook thay thế cho syscall open nên nó phải có danh sách tham số và kiểu trả về giống với hàm open. Đầu tiên, tên tiến trình mở file bằng cách sửa dụng macro `current->comm`. Còn tên file được mở cần in ra chính là giá trị xâu pathname truyền vào. Cuối cùng, ta thực hiện hàm open bằng cách gọi `custom_syscall(const char *pathname, int flags);` (vì custom_syscall đang trỏ đến hàm open).

iii. Hàm hook của hook-write

```
// Hook function, replace the system call WRITE
// Take the same parameter as the system call WRITE
ssize_t hook_function(int fd, const void *buf, size_t count)
{
    // Print calling process name
    printk(KERN_INFO "Calling process:%s\n",current->comm);

    char pathname[255];
    fd_to_pathname(fd,pathname);
    printk(KERN_INFO "Written file: %s\n",pathname);

    int written_bytes = custom_syscall(fd, buf, count);

    printk(KERN_INFO "Number of written bytes:%d\n",
written_bytes);

    return written_bytes;
}
```

Hàm hook của hook-write cũng cần cùng danh sách tham số và kiểu trả về với hàm write:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Việc in tên tiến trình gọi syscall write thực hiện tương tự như của hook-open. Tham số của hàm write chỉ có file descriptor (fd) chứ không có tên file (pathname), do đó ta cần viết hàm `fd_to_pathname()` để lấy được tên file. Cuối cùng, thực hiện hàm `write()` (bằng cách gọi con trỏ hàm `custom_syscall` đang trỏ đến `write()`) sẽ trả về số byte đã được viết. Ta có giá trị `written_bytes` và in ra dmesg.

c. *Build và chạy kernel module*

Tạo Makefile như sau:

```
obj-m += hook-open.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

obj-m += hook-write.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
```

Sau đó build kernel module bằng lệnh: `make`

Nếu quá trình build không xảy ra lỗi, kết thúc quá trình sẽ có file hook-open.ko và hook-write.ko được tạo ra. Insert mỗi kernel module bằng lệnh:

```
sudo insmod hook-open.ko
```

Như vậy quá trình hook vào syscall open đã hoàn thành. Test kết quả bằng cách chạy chương trình (test-hook.c) có sử dụng syscall open và write, sau đó vào dmesg (gõ lệnh `dmesg`) sẽ thấy những thông tin tên process và tên file được in ra.

Để dừng việc hook, trở lại syscall open như ban đầu ta chỉ cần remove kernel module:

```
sudo rmmod hook-open
```

3. Tài liệu tham khảo

- https://medium.com/anubhav-shrimal/adding-a-hello-world-system-call-to-linux-kernel-dad32875872?fbclid=IwAR2v28BeWsZt7XATEfNNP7CqGKLclVqxtgnE0CFIpwa5OPYXOzhtjezgaPs
- https://medium.freecodecamp.org/building-and-installing-the-latest-linux-kernel-from-source-6d8df5345980
- https://uwnthesis.wordpress.com/2016/12/26/basics-of-making-a-rootkit-from-syscall-to-hook/
- https://www.tecmint.com/18-tar-command-examples-in-linux/
- https://medium.freecodecamp.org/building-and-installing-the-latest-linux-kernel-from-source-6d8df5345980
-