Санкт-Петербургский политехнический университет

Институт Компьютерных Наук и Технологий

Кафедра компьютерных систем и программных технологий

Лабораторная работа по дисциплине «Проектирование ОС и компонентов» на тему:

Системные вызовы

Выполнил: студент группы № 53501/3

Дедков С.В.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Душутина Е.В.

Санкт-Петербург

2015

# Оглавление

[Оглавление 2](#_Toc438502877)

[Цель работы 3](#_Toc438502878)

[Принцип работы системных вызовов 4](#_Toc438502879)

[Назначение и функциональность вызовов sys\_stat, sys\_rename, sys\_mkdir, sys\_rmdir, sys\_fstat 7](#_Toc438502880)

[Введение 7](#_Toc438502881)

[Системные вызовы sys\_stat, sys\_fstat 8](#_Toc438502882)

[Сигнатура stat 8](#_Toc438502883)

[Описание stat 8](#_Toc438502884)

[Пример stat 8](#_Toc438502885)

[Пример fstat 10](#_Toc438502886)

[Системные вызовы sys\_mkdir, sys\_rmdir 11](#_Toc438502887)

[Сигнатура mkdir 11](#_Toc438502888)

[Описание mkdir 11](#_Toc438502889)

[Пример mkdir 11](#_Toc438502890)

[Сигнатура rmdir 11](#_Toc438502891)

[Описание rmdir 12](#_Toc438502892)

[Пример rmdir 12](#_Toc438502893)

[Системный вызов sys\_rename 13](#_Toc438502894)

[Сигнатура rename 13](#_Toc438502895)

[Описание rename 13](#_Toc438502896)

[Перехват системных вызовов 14](#_Toc438502897)

[Введение 14](#_Toc438502898)

[Перехват системного вызова mkdir 15](#_Toc438502899)

[Вывод 20](#_Toc438502900)

[Список источников 21](#_Toc438502901)

# Цель работы

* Изучить принцип работы системных вызовов в целом.
* Рассмотреть назначение и функциональность вызовов
  + sys\_stat
  + sys\_rename
  + sys\_mkdir
  + sys\_rmdir
  + sys\_fstat
* Осуществить перехват системного вызова, написать свою реакцию.

# Принцип работы системных вызовов

**Системные вызовы** - обращение из *прикладных* программ к *ядру* ОС для выполнения какой-либо *операции*.

Чтобы обеспечить безопасную и надежную работу программ с системными ресурсами (жесткие диски, видеокарта, сетевые устройства и т. д.) и безопасное межпроцессное взаимодействие *архитектура современных процессоров*, *предусматривает* использование защищенного режима с несколькими уровнями привилегий.

Для *взаимодействия* с системными ресурсами приложения используют *системные вызовы*, которые дают возможность операционной системе обеспечить безопасный доступ к ним. Системные вызовы передают управление *ядру* операционной системы.

Ядро ОС обеспечивает приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации, сервисы файловой системы и сетевых протоколов.

Ядро определяет предоставлять ли приложению запрашиваемые *ресурсы*. Если ресурсы доступны, то ядро выполняет запрошенное действие, затем возвращает управление приложению.

Прикладная программа может обращаться к ядру, например, для:

* управления процессами(создание/удаление процессов и т.п.),
* работы с файлами(чтение/запись и т.п. ),
* управления устройствами(получение/установка атрибутов устройств и т.п.),
* работы с информацией(получить системную дату и т.п.),
* связи и коммуникации(отправка/прием сообщений и т.п.);

Детали этого механизма в *разных архитектурах* немного *различаются*. Например, в процес­сорах *i386* пользовательское приложение выполняет инструкцию программного прерывания *int* со значением *0x80*. Эта инструкция осуществляет переключение на работу с пространством ядра - защищенной областью, - где ядром выполняется обработчик программного прерывания. Обработчик прерывания 0x80 - обработчик системного вызова. Приложение сообщает ядру, какой системный вызов требуется выполнить и с какими параметрами. Это делается посредством *аппаратных регистров*. Системные вызовы обозначаются по *номерам*, начиная с 0. В архитектуре i386, чтобы запросить системный вызов 5 (обычно это вызов open()), пользовательское приложение за­писывает 5 в регистр eax, после чего выдает инструкцию int. Передача параметров обрабатывается схожим образом. Так, в архитектуре i386 регистр применяется для всех возможных параметров — например, регистры ebx, ecx, edx, esi и edi в таком же порядке содержат первые пять параметров.

Рассмотрим простой пример системного вызова, для ввода на экран сообщения.

Листинг – scalls/asm/main.s :

*Примечание*. Здесь и далее исходные коды программ можно найти в репозитории: [*https://github.com/dsvgit/os\_lab\_syscalls*](https://github.com/dsvgit/os_lab_syscalls)

.text # section declaration

# we must export the entry point to the ELF linker or

.global \_start # loader. They conventionally recognize \_start as their

# entry point. Use ld -e foo to override the default.

\_start:

# write our string to stdout

movl $len,%edx # third argument: message length

movl $msg,%ecx # second argument: pointer to message to write

movl $1,%ebx # first argument: file handle (stdout)

movl $4,%eax # system call number (sys\_write)

int $0x80 # call kernel

# and exit

movl $0,%ebx # first argument: exit code

movl $1,%eax # system call number (sys\_exit)

int $0x80 # call kernel

.data # section declaration

msg:

.ascii "Hello, world!\n" # our dear string

len = . - msg # length of our dear string

Makefile:

all: main.s

gcc -c main.s

ld main.o

clean:

-rm \*.out

-rm \*.o

-rm \*~

Результат выполнения:

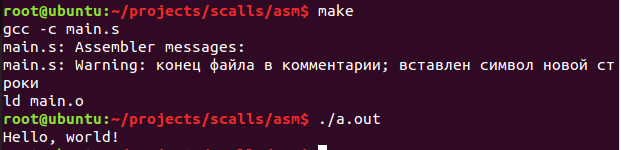


Рисунок 1. Сборка и выполнение примера системного прерывания на языке ассемблер

Лог:

root@ubuntu:~/projects/scalls/asm$ make

gcc -c main.s

main.s: Assembler messages:

main.s: Warning: конец файла в комментарии; вставлен символ новой строки

ld main.o

root@ubuntu:~/projects/scalls/asm$ ./a.out

Hello, world!

root@ubuntu:~/projects/scalls/asm$

Здесь можно увидеть вызов прерывания 0x80 и установку параметров, как это было описано выше:

movl $len,%edx # third argument: message length

movl $msg,%ecx # second argument: pointer to message to write

movl $1,%ebx # first argument: file handle (stdout)

movl $4,%eax # system call number (sys\_write)

int $0x80 # call kernel

В данном случае номер системного вызова - 4 (sys\_write). Таблица системных вызовов Linux - <http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html>, здесь же можно найти путь к исходным кодам.

Далее в работе будут рассмотрены вызовы:

* sys\_stat - 18 - fs/stat.c,
* sys\_rename - 38 - fs/namei.c,
* sys\_mkdir - 39 - fs/namei.c,
* sys\_rmdir - 40 - fs/namei.c,
* sys\_fstat - 28 - fs/stat.c;

# Назначение и функциональность вызовов sys\_stat, sys\_rename, sys\_mkdir, sys\_rmdir, sys\_fstat

## Введение

С точки зрения программиста, системный вызов обычно выглядит как вызов подпрограммы или функции из системной библиотеки. Однако системный вызов, как частный случай вызова такой функции или подпрограммы, следует отличать от более общего обращения к системной библиотеке, поскольку последнее может и не требовать выполнения привилегированных операций.

При программировании на C и C++ для системных вызовов подключаются соответствующие заголовочные файлы.

* sys/stat.h - sys\_stat, sys\_mkdir, sys\_fstat
* stdio.h - sys\_rename
* unistd.h - sys\_rmdir

Далее рассмотрим каждый системный вызов по отдельности.

## Системные вызовы sys\_stat, sys\_fstat

### Сигнатура stat

int stat(const char \*pathname, struct stat \*buf);

### Описание stat

Системный вызов, предназначен для получения информации о файле.

Функция stat() вносит в структуру, на которую указывает buf, информацию, содержащуюся в файле, связанном с указателем pathname - путь до файла.

При успешном заполнении структуры stat возвращается 0. В случае неудачи возвращается —1.

Структура stat включает следующие поля:

struct stat {  
 dev\_t st\_dev; /\* ID of device containing file \*/  
 ino\_t st\_ino; /\* inode number \*/  
 mode\_t st\_mode; /\* protection \*/  
 nlink\_t st\_nlink; /\* number of hard links \*/  
 uid\_t st\_uid; /\* user ID of owner \*/  
 gid\_t st\_gid; /\* group ID of owner \*/  
 dev\_t st\_rdev; /\* device ID (if special file) \*/  
 off\_t st\_size; /\* total size, in bytes \*/  
 blksize\_t st\_blksize; /\* blocksize for filesystem I/O \*/  
 blkcnt\_t st\_blocks; /\* number of 512B blocks allocated \*/  
  
 /\* Since Linux 2.6, the kernel supports nanosecond  
 precision for the following timestamp fields.  
 For the details before Linux 2.6, see NOTES. \*/  
  
 struct timespec st\_atim; /\* time of last access \*/  
 struct timespec st\_mtim; /\* time of last modification \*/  
 struct timespec st\_ctim; /\* time of last status change \*/  
  
 #define st\_atime st\_atim.tv\_sec /\* Backward compatibility \*/  
 #define st\_mtime st\_mtim.tv\_sec  
 #define st\_ctime st\_ctim.tv\_sec  
 };

### Пример stat

Рассмотрим на примере, информацию, которою можно получить воспользовавшись данной функцией. Запустим следующую программу.

Листинг – scalls/stat/main.cpp :

#include <sys/stat.h>

#include <stdlib.h>

#include <iostream>

#include <fstream>

int main() {

std::string fpath = "test.txt";

std::ifstream f ( fpath.c\_str() );

struct stat finfo;

if( !f.good() ) {

std::cout << "Cannot open file.\n";

f.close();

exit(1);

}

f.close();

stat( fpath.c\_str(), &finfo );

std::cout << "ID of device containing file\t\t" << finfo.st\_dev << "\n";

std::cout << "inode number\t\t\t\t" << finfo.st\_ino << "\n";

std::cout << "protection\t\t\t\t" << finfo.st\_mode << "\n";

std::cout << "number of hard links\t\t\t" << finfo.st\_nlink << "\n";

std::cout << "user ID of owner\t\t\t" << finfo.st\_uid << "\n";

std::cout << "group ID of owner\t\t\t" << finfo.st\_gid << "\n";

std::cout << "device ID (if special file)\t\t" << finfo.st\_rdev << "\n";

std::cout << "total size, in bytes\t\t\t" << finfo.st\_size << "\n";

std::cout << "blocksize for filesystem I/O\t\t" << finfo.st\_blksize << "\n";

std::cout << "number of 512B blocks allocated\t\t" << finfo.st\_blocks << "\n";

exit(0);

}

Содержание файла test.txt:

This is a sample file.

Результат выполнения:

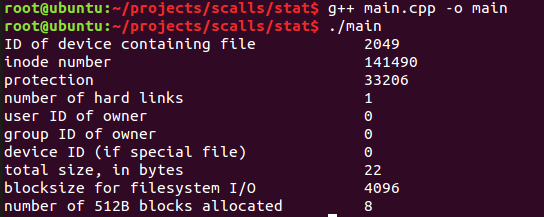


Рисунок 2. Сборка и выполнение примера программы, syscall - stat

Лог:

root@ubuntu:~/projects/scalls/stat$ ./main

ID of device containing file 2049

inode number 141490

protection 33206

number of hard links 1

user ID of owner 0

group ID of owner 0

device ID (if special file) 0

total size, in bytes 22

blocksize for filesystem I/O 4096

number of 512B blocks allocated 8

### Пример fstat

Аналогично работает функция fstat, единственное отличие - она принимает первым параметром не путь до файла, а дескриптор. Пример аналогичной программы с использованием fstat.

Листинг – scalls/fstat/main.cpp :

#include <sys/stat.h>

#include <stdlib.h>

#include <iostream>

#include <fstream>

int main() {

std::string fpath = "test.txt";

struct stat finfo;

FILE \* pf = fopen( fpath.c\_str(), "r" );

if( pf == NULL ) {

std::cout << "Cannot open file.\n";

fclose( pf );

exit( 1 );

}

fstat( fileno(pf), &finfo );

fclose( pf );

std::cout << "ID of device containing file\t\t" << finfo.st\_dev << "\n";

std::cout << "inode number\t\t\t\t" << finfo.st\_ino << "\n";

std::cout << "protection\t\t\t\t" << finfo.st\_mode << "\n";

std::cout << "number of hard links\t\t\t" << finfo.st\_nlink << "\n";

std::cout << "user ID of owner\t\t\t" << finfo.st\_uid << "\n";

std::cout << "group ID of owner\t\t\t" << finfo.st\_gid << "\n";

std::cout << "device ID (if special file)\t\t" << finfo.st\_rdev << "\n";

std::cout << "total size, in bytes\t\t\t" << finfo.st\_size << "\n";

std::cout << "blocksize for filesystem I/O\t\t" << finfo.st\_blksize << "\n";

std::cout << "number of 512B blocks allocated\t\t" << finfo.st\_blocks << "\n";

exit( 0 );

}

Результат выполнения аналогичен предыдущему.

## Системные вызовы sys\_mkdir, sys\_rmdir

### Сигнатура mkdir

int mkdir(const char \*path, mode\_t mode);

### Описание mkdir

Функция mkdir() создает каталог, используя путь, на который указывает параметр path. В случае успеха функция mkdir() возвращает 0, в противном случае возвращается -1. Для задания прав используется второй параметр. Описание типов - <http://pubs.opengroup.org/onlinepubs/007908799/xsh/sysstat.h.html>.

### Пример mkdir

В следующем примере рассматривается использование данной функции.

Листинг – scalls/mkdir/main.cpp :

#include <sys/stat.h>

int main() {

mkdir("test\_dir", S\_IRWXU | S\_IRWXG | S\_IROTH | S\_IXOTH);

return 0;

}

Результат выполнения:

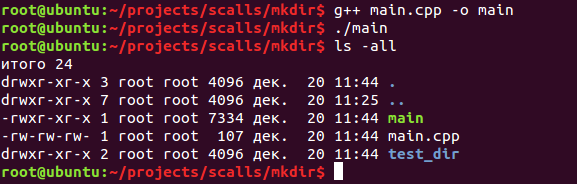


Рисунок 3. Сборка и выполнение примера программы, syscall - mkdir

Лог:

root@ubuntu:~/projects/scalls/mkdir$ g++ main.cpp -o main

root@ubuntu:~/projects/scalls/mkdir$ ./main

root@ubuntu:~/projects/scalls/mkdir$ ls -all

итого 24

drwxr-xr-x 3 root root 4096 дек. 20 11:44 .

drwxr-xr-x 7 root root 4096 дек. 20 11:25 ..

-rwxr-xr-x 1 root root 7334 дек. 20 11:44 main

-rw-rw-rw- 1 root root 107 дек. 20 11:44 main.cpp

drwxr-xr-x 2 root root 4096 дек. 20 11:44 test\_dir

### Сигнатура rmdir

int rmdir(const char \*path);

### Описание rmdir

Функция rmdir() удаляет каталог, путь к которому определен параметром path. Для того, что­бы каталог можно было удалить, он должен быть пустым, но не должен быть текущим или корне­вым каталогом.

В случае успеха функция rmdir() возвращает значение 0. В противном случае возвращается - 1.

### Пример rmdir

Листинг – scalls/rmdir/main.cpp :

#include <iostream>

#include <unistd.h>

int main() {

if ( !rmdir( "test\_dir" ) ) {

std::cout << "test\_dir removed\n";

}

return 0;

}

Результат выполнения:

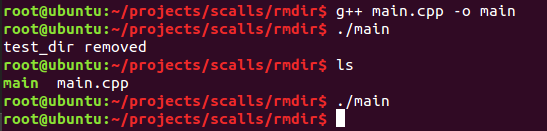


Рисунок 4. Сборка и выполнение примера программы, syscall - rmdir

Лог:

root@ubuntu:~/projects/scalls/rmdir$ g++ main.cpp -o main

root@ubuntu:~/projects/scalls/rmdir$ ./main

test\_dir removed

root@ubuntu:~/projects/scalls/rmdir$ ls

main main.cpp

root@ubuntu:~/projects/scalls/rmdir$ ./main

root@ubuntu:~/projects/scalls/rmdir$

## Системный вызов sys\_rename

### Сигнатура rename

int rename(const char \*oldfname, const char \*newfname)

### Описание rename

Функция rename() изменяет название файла со старого oldfname на новое newfname. Новое имя newfname не должно совпадать с именами, имеющимися в каталоге. Функция rename() возвращает 0 в случае успеха и ненулевую величину - в случае ошибки.

Пример программы.

Листинг – scalls/rename/main.cpp :

#include <stdio.h>

int main () {

int result;

char oldname[] = "oldname.txt";

char newname[] = "newname.txt";

result = rename( oldname , newname );

if ( result == 0 )

puts ( "File successfully renamed" );

else

perror( "Error renaming file" );

return 0;

}

Результат выполнения:

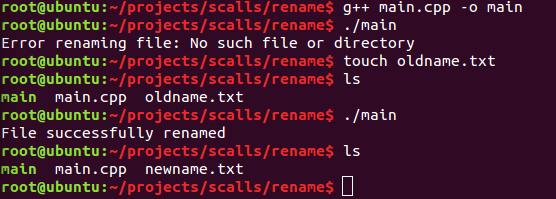


Рисунок 5. Сборка и выполнение примера программы, syscall - rename

Лог:

root@ubuntu:~/projects/scalls/rename$ g++ main.cpp -o main

root@ubuntu:~/projects/scalls/rename$ ./main

Error renaming file: No such file or directory

root@ubuntu:~/projects/scalls/rename$ touch oldname.txt

root@ubuntu:~/projects/scalls/rename$ ls

main main.cpp oldname.txt

root@ubuntu:~/projects/scalls/rename$ ./main

File successfully renamed

root@ubuntu:~/projects/scalls/rename$ ls

main main.cpp newname.txt

# Перехват системных вызовов

## Введение

Перехват системных вызовов позволяет осуществлять контроль ключевого компонента ядра ОС — интерфейса системных вызовов, что, в свою очередь, даёт возможность инспектировать запросы прикладного ПО к сервисам ядра.

Перехват будем осуществлять одним из наиболее распространенных способов - на основе LKM (Loadable Kernel Module).

Загружаемый модуль ядра (общепринятое сокращение LKM - Loadable Kernel Module) - программный код, выполняемый в пространстве ядра. Главной особенностью LKM является возможность динамической загрузки и выгрузки.

Каждый LKM состоит из двух основных функций (минимум):

* функция инициализации модуля. Вызывается при загрузке LKM в память:

int init\_module(void) { ... }

* функция выгрузки модуля:

void cleanup\_module(void) { ... }

Для реализации модуля, перехватывающего системный вызов, необходимо определить алгоритм перехвата. Алгоритм следующий:

* сохранить указатель на оригинальный (исходный) вызов для возможности его восстановления
* создать функцию, реализующую новый системный вызов
* в таблице системных вызовов sys\_call\_table произвести замену вызовов, т.е настроить соответствующий указатель на новый системный вызов
* по окончании работы (при выгрузке модуля) восстановить оригинальный системный вызов, используя ранее сохраненный указатель

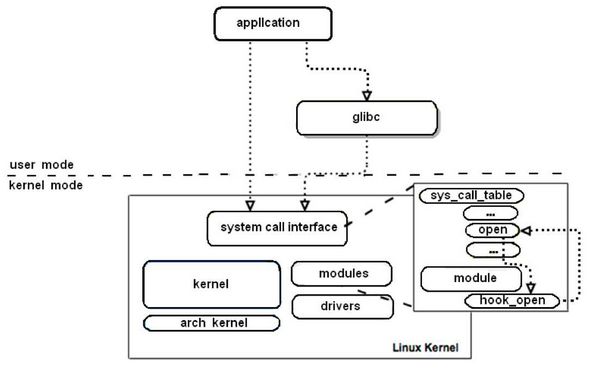


Рисунок 6. Перехват системного вызова с помощью модуля ядра Linux

На рисунке приведена графическая схема перехвата системного вызова, подменой

## Перехват системного вызова mkdir

Файл модуля intercept\_mkdir.c.

Листинг – scalls/lsm/intercept\_mkdir.c :

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/syscalls.h>

#include <linux/delay.h>

#include <asm/paravirt.h>

unsigned long \*\*sys\_call\_table;

unsigned long original\_cr0;

asmlinkage int (\* ref\_sys\_mkdir) (struct inode \*dir, struct dentry \*dentry, int mode);

asmlinkage int new\_sys\_mkdir (struct inode \*dir, struct dentry \*dentry, int mode) {

printk ("mkdir hijacked!\n");

return ref\_sys\_mkdir(dir, dentry, mode);

}

// find sys call table

static unsigned long \*\*aquire\_sys\_call\_table(void)

{

unsigned long int offset = PAGE\_OFFSET;

unsigned long \*\*sct;

while (offset < ULLONG\_MAX) {

sct = (unsigned long \*\*)offset;

if (sct[\_\_NR\_close] == (unsigned long \*) sys\_close)

return sct;

offset += sizeof(void \*);

}

return NULL;

}

// change sys call

static int \_\_init interceptor\_start(void)

{

if(!(sys\_call\_table = aquire\_sys\_call\_table()))

return -1;

original\_cr0 = read\_cr0();

write\_cr0(original\_cr0 & ~0x00010000);

ref\_sys\_mkdir = (void \*)sys\_call\_table[\_\_NR\_mkdir];

sys\_call\_table[\_\_NR\_mkdir] = (unsigned long \*)new\_sys\_mkdir;

write\_cr0(original\_cr0);

return 0;

}

// revert changes

static void \_\_exit interceptor\_end(void)

{

if(!sys\_call\_table) {

return;

}

write\_cr0(original\_cr0 & ~0x00010000);

sys\_call\_table[\_\_NR\_mkdir] = (unsigned long \*)ref\_sys\_mkdir;

write\_cr0(original\_cr0);

msleep(2000);

}

module\_init(interceptor\_start);

module\_exit(interceptor\_end);

MODULE\_LICENSE("GPL");

Makefile.

Листинг – scalls/lsm/Makefile :

obj-m += intercept\_mkdir.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

Сборка модуля:

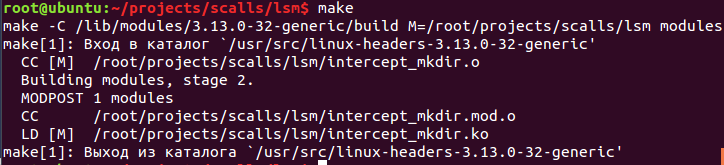


Рисунок 7. Сборка модуля с перехватом системного вызова mkdir

Лог:

root@ubuntu:~/projects/scalls/lsm$ make

make -C /lib/modules/3.13.0-32-generic/build M=/root/projects/scalls/lsm modules

make[1]: Вход в каталог `/usr/src/linux-headers-3.13.0-32-generic'

CC [M] /root/projects/scalls/lsm/intercept\_mkdir.o

Building modules, stage 2.

MODPOST 1 modules

CC /root/projects/scalls/lsm/intercept\_mkdir.mod.o

LD [M] /root/projects/scalls/lsm/intercept\_mkdir.ko

make[1]: Выход из каталога `/usr/src/linux-headers-3.13.0-32-generic'

Установка модуля:



Рисунок 8. Установка модуля

Лог:

root@ubuntu:~/projects/scalls/lsm$ insmod intercept\_mkdir.ko

Проверка работы:

Для проверки создадим папку и проверим вывод при помощи утилиты dmesg.



Рисунок 9. Создание папки после установки модуля с перехватом

Лог:

root@ubuntu:~/projects/scalls/lsm$ mkdir newdir

Частичный вывод результата dmesg:

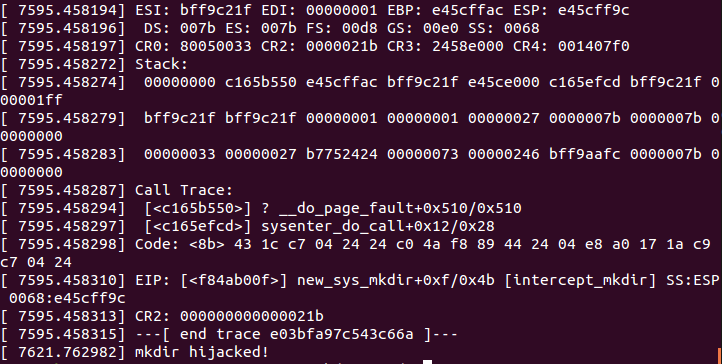


Рисунок 10. Вывод dmesg после создания папки

Лог:

[ 7595.458279] bff9c21f bff9c21f 00000001 00000001 00000027 0000007b 0000007b 00000000

[ 7595.458283] 00000033 00000027 b7752424 00000073 00000246 bff9aafc 0000007b 00000000

[ 7595.458287] Call Trace:

[ 7595.458294] [<c165b550>] ? \_\_do\_page\_fault+0x510/0x510

[ 7595.458297] [<c165efcd>] sysenter\_do\_call+0x12/0x28

[ 7595.458298] Code: <8b> 43 1c c7 04 24 24 c0 4a f8 89 44 24 04 e8 a0 17 1a c9 c7 04 24

[ 7595.458310] EIP: [<f84ab00f>] new\_sys\_mkdir+0xf/0x4b [intercept\_mkdir] SS:ESP 0068:e45cff9c

[ 7595.458313] CR2: 000000000000021b

[ 7595.458315] ---[ end trace e03bfa97c543c66a ]---

[ 7621.762982] mkdir hijacked!

Как видим последнее сообщение - результат перехвата системного вызова mkdir.

Для восстановления стандартного системного вызова воспользуемся командой rmmod:

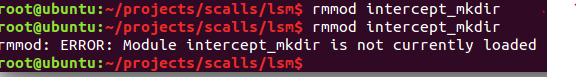


Рисунок 11. Удаления модуля

Лог:

root@ubuntu:~/projects/scalls/lsm$ rmmod intercept\_mkdir

root@ubuntu:~/projects/scalls/lsm$ rmmod intercept\_mkdir

rmmod: ERROR: Module intercept\_mkdir is not currently loaded

Рассмотрим код модуля подробнее:

Объявление нового и старого системных вызовов mkdir:

asmlinkage int (\* ref\_sys\_mkdir) (struct inode \*dir, struct dentry \*dentry, int mode);

asmlinkage int new\_sys\_mkdir (struct inode \*dir, struct dentry \*dentry, int mode) {

printk ("mkdir hijacked!\n");

return ref\_sys\_mkdir(dir, dentry, mode);

}

Сигнатуры этих функций можно найти в файле fs/namei.c:

int vfs\_mkdir(struct inode \*dir, struct dentry \*dentry, umode\_t mode)

Заметим, что новая функция вызывает старую, таким образом сохраняя функциональность.

Поиск таблицы осуществляется в начале функции инициализации модуля:

sys\_call\_table = aquire\_sys\_call\_table()

Функция:

static unsigned long \*\*aquire\_sys\_call\_table(void)

{

unsigned long int offset = PAGE\_OFFSET;

unsigned long \*\*sct;

while (offset < ULLONG\_MAX) {

sct = (unsigned long \*\*)offset;

if (sct[\_\_NR\_close] == (unsigned long \*) sys\_close)

return sct;

offset += sizeof(void \*);

}

return NULL;

}

Возвращает ссылку на таблицу системных вызовов или NULL.

Регистрация функций инициализации и удаления модуля:

module\_init(interceptor\_start);

module\_exit(interceptor\_end);

Стоит отметить, что есть несколько вариантов перехвата системных вызовов:

* найти адрес sys\_call\_table в файле /boot/System.map, содержащем все используемые ядром символы и их адреса в памяти ядра (такой файл создается при каждой компиляции ядра), например так - grep sys\_call\_table/ /boot/System.map | awk ’{print \$$1}’\* и передать в модуль ядра через Makefile. Таким образом, данный метод фактически не позволяет распространять модуль ядра в собранном виде (в ви­де бинарного файла с расширением .ко) - его необ­ходимо будет компилировать на каждой новой ОС;
* найти адрес sys\_call\_table в файле /boot/System.map с помощью функции модуля ядра — в данном случае пересобирать модуль под конкретную систему не обязательно;
* найти адрес sys\_call\_table в памяти ядра ОС с помощью разбора адресов соседних структур (например «loops\_per\_jiffy» и «cpu\_boot\_data»). В данном случае необходимо учитывать, что не во всех версиях ядра ОС таблица системных вызовов будет располагаться именно между указанными структурами;
* найти адрес sys\_call\_table в памяти ядра ОС, для примера это можно сделать обычным перебором со сравнением определенного элемента таблицы системных вызовов с тем значением, которое должно быть в качестве этого элемента

# Вывод

В данной работе был рассмотрен принцип работы системных вызовов в первой части и приведен пример на языке ассемблер. Из которого видно, как на нижнем уровне осуществляются системные вызовы.

Затем рассмотрена функциональность системных вызовов - sys\_stat, sys\_rename, sys\_mkdir, sys\_rmdir, sys\_fstat. Приведены примеры принципов работы с нимим на языке c++.

В завершающей части рассмотрен механизм перехвата системных вызовов путем загрузки модуля ядра.

Это может потребоваться, например, для внедрения собственных ме­ханизмов безопасности в ОС Linux. Фактически требуется написать собственные реализации основ­ных системных вызовов, которые в зависимости от успешности/неуспешности определенных прове­рок будут вызывать/не вызывать выполнение эта­лонных системных вызовов. Таким образом стано­вится возможным реализовать собственные механизмы разграниче­ния доступа, которые будут работать непосредственно до отработки всех штат­ных подсистем разграничения доступа ОС. Сами функции, «переопределяющие» работу системных вызовов логично описать в загружаемом модуле яд­ра, а при инициализации этого модуля необходимо заменять адреса системных вызовов в таблице сис­темных вызовов на адреса переопределяющих их функций

# Список источников

1. Особенности доступа к системным функциям ядра ОС GNU/Linux - <http://www.okbsapr.ru/kanner_2012_5.html>
2. Intercepting System Calls - http://books.gigatux.nl/mirror/networksecuritytools/0596007949/networkst-CHP-7-SECT-2.html
3. linux 3.14 sys\_call\_table interception module <https://bbs.archlinux.org/viewtopic.php?id=139406>
4. Перехват системных вызовов в операционной системе Linux http://samag.ru/archive/article/128
5. Перехват системных вызовов в OS Linux (linux kernel module) - <https://www.opennet.ru/base/dev/intercept_lnx.txt.html>
6. Встраивание в ядро Linux: перехват системных вызовов - <http://habrahabr.ru/company/securitycode/blog/245539/>
7. Лав Р. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014. — 448 с.