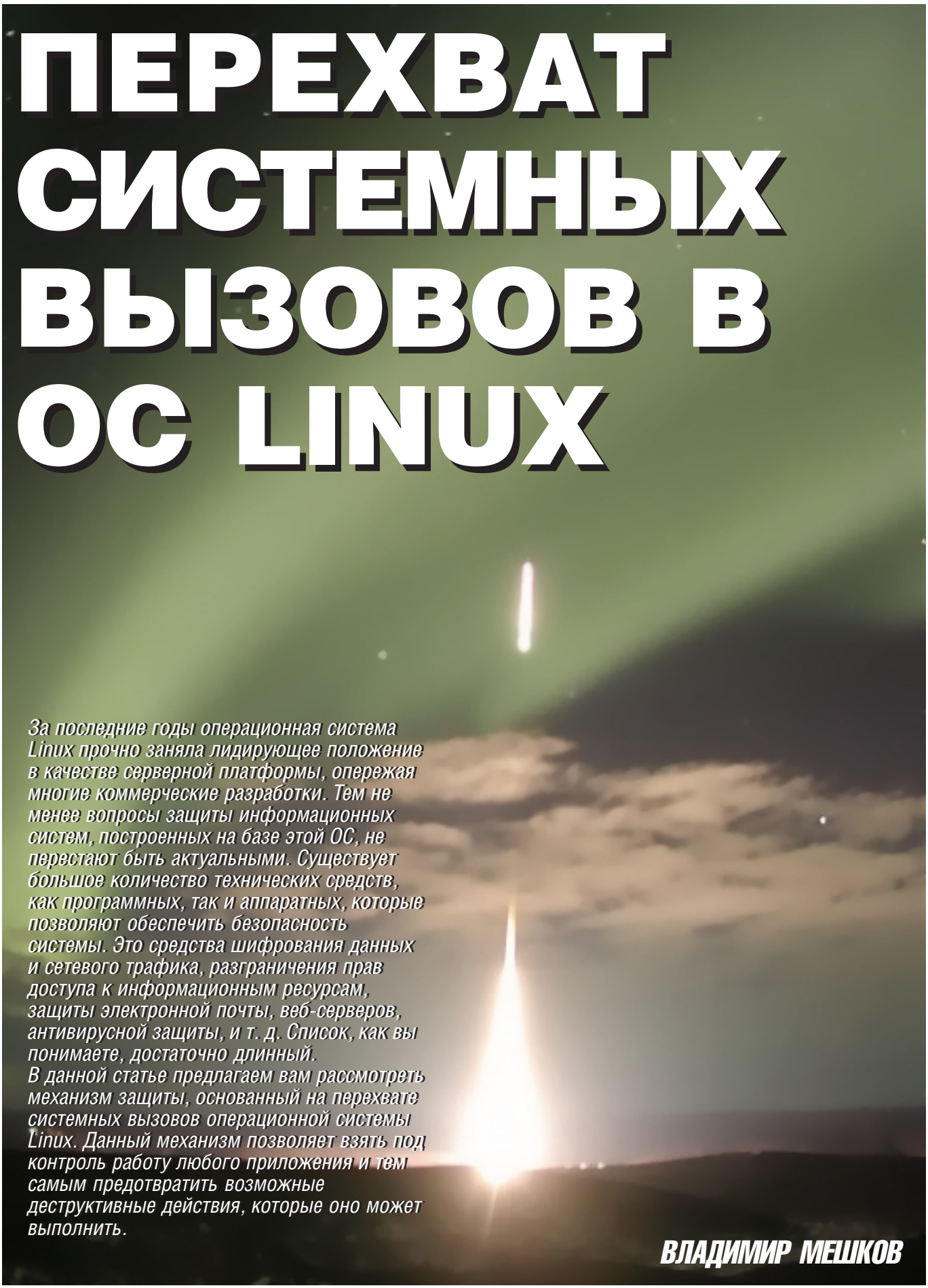
ПЕРЕХВАТ СИСТЕМНЫХ ВЫЗОВОВ В ОС LINUX



За последние годы операционная система Linux прочно заняла лидирующее положение в качестве серверной платформы, опережая многие коммерческие разработки. Тем не менее вопросы защиты информационных систем, построенных на базе этой ОС, не перестают быть актуальными. Существует большое количество технических средств, как программных, так и аппаратных, которые позволяют обеспечить безопасность системы. Это средства шифрования данных и сетевого трафика, разграничения прав доступа к информационным ресурсам, защиты электронной почты, веб-серверов, антивирусной защиты, и т. д. Список, как вы понимаете, достаточно длинный.
В данной статье предлагаем вам рассмотреть механизм защиты, основанный на перехвате системных вызовов операционной системы Linux. Данный механизм позволяет взять под контроль работу любого приложения и тем самым предотвратить возможные деструктивные действия, которые оно может выполнить.

ВЛАДИМИР МЕШКОВ

Системные вызовы

Начнем с определения. Системные вызовы – это набор функций, реализованных в ядре ОС. Любой запрос приложения пользователя в конечном итоге трансформируется в системный вызов, который выполняет запрашиваемое действие. Полный перечень системных вызовов ОС Linux находится в файле /usr/include/asm/unistd.h. Давайте рассмотрим общий механизм выполнения системных вызовов на примере. Пусть в исходном тексте приложения вызывается функция creat() для создания нового файла. Компилятор, встретив вызов данной функции, преобразует его в ассемблерный код, обеспечивая загрузку номера системного вызова, соответствующего данной функции, и ее параметров в регистры процессора и последующий вызов прерывания 0x80. В регистры процессора загружаются следующие значения:

- в регистр EAX – номер системного вызова. Так, для нашего случая номер системного вызова будет равен 8 (см. __NR_creat);
- в регистр EBX – первый параметр функции (для creat это указатель на строку, содержащую имя создаваемого файла);
- в регистр ECX – второй параметр (права доступа к файлу).

В регистр EDX загружается третий параметр, в данном случае он у нас отсутствует. Для выполнения системного вызова в ОС Linux используется функция system_call, которая определена в файле /usr/src/liux/arch/i386/kernel/entry.S. Эта функция – точка входа для всех системных вызовов. Ядро реагирует на прерывание 0x80 обращением к функции system_call, которая, по сути, представляет собой обработчик прерывания 0x80.

Чтобы убедиться, что мы на правильном пути, напишем небольшой тестовый фрагмент на ассемблере. В нем увидим, во что превращается функция creat() после компиляции. Файл назовем test.S. Вот его содержание:

```
.globl _start
.text
_start:
```

В регистр EAX загружаем номер системного вызова:

```
movl $8, %eax
```

В регистр EBX – первый параметр, указатель на строку с именем файла:

```
movl $filename, %ebx
```

В регистр ECX – второй параметр, права доступа:

```
movl $0, %ecx
```

Вызываем прерывание:

```
int $0x80
```

Выходим из программы. Для этого вызовем функцию exit(0):

```
movl $1, %eax movl $0, %ebx int $0x80
```

В сегменте данных укажем имя создаваемого файла:

```
    .data
filename:  .string  "file.txt"
```

Компилируем:

```
gcc -c test.S
ld -s -o test test.o
```

В текущем каталоге появится исполняемый файл test. Запустив его, мы создадим новый файл с именем file.txt.

А теперь давайте вернемся к рассмотрению механизма системных вызовов. Итак, ядро вызывает обработчик прерывания 0x80 – функцию system_call. System_call помещает копии регистров, содержащих параметры вызова, в стек при помощи макроса SAVE_ALL и командой call вызывает нужную системную функцию. Таблица указателей на функции ядра, которые реализуют системные вызовы, расположена в массиве sys_call_table (см. файл arch/i386/kernel/entry.S). Номер системного вызова, который находится в регистре EAX, является индексом в этом массиве. Таким образом, если в EAX находится значение 8, будет вызвана функция ядра sys_creat(). Зачем нужен макрос SAVE_ALL? Объяснение тут очень простое. Так как практически все системные функции ядра написаны на C, то свои параметры они ищут в стеке. А параметры помещаются в стек при помощи макроса SAVE_ALL! Возвращаемое системным вызовом значение сохраняется в регистр EAX.

Теперь давайте выясним, как перехватить системный вызов. Поможет нам в этом механизм загружаемых модулей ядра. Хотя ранее мы уже рассматривали вопросы разработки и применения модулей ядра, в интересах последовательности изложения материала рассмотрим кратко, что такое модуль ядра, из чего он состоит и как взаимодействует с системой.

Загружаемый модуль ядра

Загружаемый модуль ядра (обозначим его LKM – Loadable Kernel Module) – это программный код, выполняемый в пространстве ядра. Главной особенностью LKM является возможность динамической загрузки и выгрузки без необходимости перезагрузки всей системы или перекомпиляции ядра.

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) { ... }
```

Приведем пример простейшего модуля:

```
#define MODULE
#include <linux/module.h>

int init_module(void)
{
printk("Hello World\n");
return 0;
}

void cleanup_module(void)
{
printk("Bye\n");
}
```

Компилируем и загружаем модуль. Загрузку модуля в память осуществляет команда insmod:

```
gcc -c -O3 helloworld.c
insmod helloworld.o
```

Информация обо всех загруженных в данный момент в систему модулях находится в файле /proc/modules. Чтобы убедиться, что модуль загружен, введите команду cat /proc/modules либо lsmod. Выгружает модуль команда rmmod:

```
rmmod helloworld
```

Алгоритм перехвата системного вызова

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.

Выяснить, какие системные вызовы задействуются при работе приложения пользователя, позволяет трассировка. Осуществив трассировку, можно определить, какой именно системный вызов следует перехватить, чтобы взять под контроль работу приложения. Пример использования программы трассировки будет рассмотрен ниже.

Теперь у нас достаточно информации, чтобы приступить к изучению примеров реализации модулей, осуществляющих перехват системных вызовов.

Примеры перехвата системных вызовов

Запрет создания каталогов

При создании каталога вызывается функция ядра sys_mkdir. В качестве параметра задается строка, в которой содержится имя создаваемого каталога. Рассмотрим код, осуществляющий перехват соответствующего системного вызова.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
```

Экспортируем таблицу системных вызовов:

```
extern void *sys_call_table[];
```

Определим указатель для сохранения оригинального системного вызова:

```
int (*orig_mkdir)(const char *path);
```

Создадим собственный системный вызов. Наш вызов ничего не делает, просто возвращает нулевое значение:

```
int own_mkdir(const char *path)
{
return 0;
}
```

Во время инициализации модуля сохраняем указатель на оригинальный вызов и производим замену системного вызова:

```
int init_module()
{
orig_mkdir=sys_call_table[SYS_mkdir];
sys_call_table[SYS_mkdir]=own_mkdir; return 0;
}
```

При выгрузке восстанавливаем оригинальный вызов:

```
void cleanup_module()
{
 sys_call_table[SYS_mkdir]=orig_mkdir;
}
```

Код сохраним в файле sys_mkdir_call.c. Для получения объектного модуля создадим Makefile следующего содержания:

```
CC = gcc
CFLAGS = -O3 -Wall -fomit-frame-pointer
MODFLAGS = -D__KERNEL__ -DMODULE -I/usr/src/linux/include

sys_mkdir_call.o: sys_mkdir_call.c
$(CC) -c $(CFLAGS) $(MODFLAGS) sys_mkdir_call.c
```

Командой make создадим модуль ядра. Загрузив его, попытаемся создать каталог командой mkdir. Как вы можете убедиться, ничего при этом не происходит. Команда не работает. Для восстановления ее работоспособности достаточно выгрузить модуль.

Запрет чтения файла

Для того чтобы прочитать файл, его необходимо вначале открыть при помощи функции open. Легко догадаться, что этой функции соответствует системный вызов sys_open. Перехватив его, мы можем защитить файл от прочтения. Рассмотрим реализацию модуля-перехватчика.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <linux/types.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <asm/uaccess.h>

extern void *sys_call_table[];
```

Указатель для сохранения оригинального системного вызова:

```
int (*orig_open)(const char *pathname, int flag, int mode);
```

Первым параметром функции open является имя открываемого файла. Новый системный вызов должен сравнить этот параметр с именем файла, который мы хотим защитить. Если имена совпадут, будет сымитирована ошибка открытия файла. Наш новый системный вызов имеет вид:

```
int own_open(const char *pathname, int flag, int mode)
{
```

Сюда поместим имя открываемого файла:

```
char *kernel_path;
```

Имя файла, который мы хотим защитить:

```
char hide[]="test.txt"
```

Выделим память и скопируем туда имя открываемого файла:

```
kernel_path=(char *)kmalloc(255,GFP_KERNEL);
copy_from_user(kernel_path, pathname, 255);
```

Сравниваем:

```
if(strstr(kernel_path,(char *)&hide) != NULL) {
```

Освобождаем память и возвращаем код ошибки при совпадении имен:

```
kfree(kernel_path);
return -ENOENT;
}

else {
```

Если имена не совпали, вызываем оригинальный системный вызов для выполнения стандартной процедуры открытия файла:

```
kfree(kernel_path);
return orig_open(pathname, flag, mode);
}
}
```

Далее смотрите комментарии к предыдущему примеру.

```
int init_module()
{
orig_open=sys_call_table[SYS_open];
sys_call_table[SYS_open]=own_open;
return 0;
}

void cleanup_module()
{
sys_call_table[SYS_open]=orig_open;
}
```

Сохраним код в файле sys_open_call.c и создадим Makefile для получения объектного модуля:

```
CC = gcc
CFLAGS = -O2 -Wall -fomit-frame-pointer
MODFLAGS = -D__KERNEL__ -DMODULE -I/usr/src/linux/include

sys_open_call.o: sys_open_call.c
$(CC) -c $(CFLAGS) $(MODFLAGS) sys_open_call.c
```

В текущем каталоге создадим файл с именем test.txt, загрузим модуль и введем команду cat test.txt. Система сообщит об отсутствии файла с таким именем.

Честно говоря, такую защиту легко обойти. Достаточно командой mv переименовать файл, а затем прочесть его содержимое.

Сокрытие записи о файле в каталоге

Определим, какой системный вызов отвечает за чтение содержимого каталога. Для этого напишем еще один тестовый фрагмент, который занимается чтением текущей директории:

```
/* Файл dir.c*/
#include <stdio.h>
#include <dirent.h>

int main ()
{
DIR *d;
struct dirent *dp;

d = opendir(«.»);
dp = readdir(d);

            return 0;

    }
```

Получим исполняемый модуль:

```
gcc -o dir dir.c
```

и выполним его трассировку:

```
strace ./dir
```

Обратим внимание на предпоследнюю строку:

```
getdents (6, /* 4 entries*/, 3933) = 72;
```

Содержимое каталога считывает функция getdents. Результат сохраняется в виде списка структур типа struct dirent. Второй параметр этой функции является указателем на этот список. Функция возвращает длину всех записей в каталоге. В нашем примере функция getdents определила наличие в текущем каталоге четырех записей – «.», «..» и два наших файла, исполняемый модуль и исходный текст. Длина всех записей в каталоге составляет 72 байта. Информация о каждой записи сохраняется, как

мы уже сказали, в структуре struct dirent. Для нас интерес представляют два поля данной структуры:

- d_reclen – размер записи;
- d_name – имя файла.

Для того чтобы спрятать запись о файле (другими словами, сделать его невидимым), необходимо перехватить системный вызов sys_getdents, найти в списке полученных структур соответствующую запись и удалить ее. Рассмотрим код, выполняющий эту операцию (автор оригинального кода – Michal Zalewski):

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <sys/syscall.h>
#include <asm/uaccess.h>

extern void *sys_call_table[];
int (*orig_getdents)(u_int, struct dirent *, u_int);
```

Определим свой системный вызов.

```
int own_getdents(u_int fd, struct dirent *dirp, u_int
count)
  {
  unsigned int tmp, n;
  int t;
```

Назначение переменных будет показано ниже. Дополнительно нам понадобятся структуры:

```
struct dirent *dirp2, *dirp3;
```

Имя файла, который мы хотим спрятать:

```
char hide[]=»our.file»;
```

Определим длину записей в каталоге:

```
tmp=(*orig_getdents)(fd,dirp,count);
if(tmp>0){
```

Выделим память для структуры в пространстве ядра и скопируем в нее содержимое каталога:

```
dirp2=(struct dirent *)kmalloc(tmp,GFP_KERNEL);
copy_from_user(dirp2,dirp,tmp);
```

Задействуем вторую структуру и сохраним значение длины записей в каталоге:

```
dirp3=dirp2;
t=tmp;
```

Начнем искать наш файл:

```
while(t>0) {
```

Считываем длину первой записи и определяем оставшуюся длину записей в каталоге:

```
n=dirp3->d_reclen;
t-=n;
```

Проверяем, не совпало ли имя файла из текущей записи с искомым:

```
if(strstr((char *)&(dirp3->d_name),(char *)&hide) != NULL) {
```

Если это так, затираем запись и вычисляем новое значение длины записей в каталоге:

```
memcpy(dirp3,(char *)dirp3+dirp3->d_reclen,t);
tmp-=n;
}
```

Позиционируем указатель на следующую запись и продолжаем поиск:

```
dirp3=(struct dirent *)((char *)dirp3+dirp3->d_reclen);
}
```

Возвращаем результат и освобождаем память:

```
copy_to_user(dirp,dirp2,tmp);
kfree(dirp2);
}
```

Возвращаем значение длины записей в каталоге:

```
return tmp;
}
```

Функции инициализации и выгрузки модуля имеют стандартный вид:

```
int init_module(void)
{
orig_getdents=sys_call_table[SYS_getdents];
sys_call_table[SYS_getdents]=own_getdents;
return 0;
}

void cleanup_module()
{
sys_call_table[SYS_getdents]=orig_getdents;
}
```

Сохраним исходный текст в файле sys_call_getd.c и создадим Makefile следующего содержания:

```
CC = gcc
module = sys_call_getd.o
CFLAGS = -O3 -Wall
LINUX = /usr/src/linux
MODFLAGS = -D__KERNEL__ -DMODULE -I$(LINUX)/include

sys_call_getd.o: sys_call_getd.c $(CC) -c
$(CFLAGS) $(MODFLAGS) sys_call_getd.c
```

В текущем каталоге создадим файл our.file и загрузим модуль. Файл исчезает, что и требовалось доказать.

Как вы понимаете, рассмотреть в рамках одной статьи пример перехвата каждого системного вызова не представляется возможным. Поэтому тем, кто заинтересовался данным вопросом, рекомендую посетить сайты:

- www.phrack.com
- www.atstake.com
- www.thehackerschoice.com.

Там вы сможете найти более сложные и интересные примеры перехвата системных вызовов. Обо всех замечаниях и предложениях пишите на форум журнала.

При подготовке статьи были использованы материалы сайта http://www.thehackerschoice.com/.