

*Каждый, кому приходилось обеспечивать безопасность сети, прекрасно знает, что без надежного брандмауэра обойтись просто невозможно, и что его грамотная настройка требует от администратора глубоких знаний, опыта, терпения и времени. Однако многие ли интересовались вопросами внутреннего устройства и принципами функционирования брандмауэра? Предлагаем вам заглянуть «внутрь» простого брандмауэра и совершить путешествие по его исходному коду.*

# БРАНДМАУЭР

**ВЛАДИМИР МЕШКОВ**

**Часть 1**

## Общий алгоритм функционирования брандмауэра

В состав брандмауэра входят следующие элементы:

- загружаемый модуль ядра;
- процесс-демон;
- программа инициализации и запуска процесса-демона.

Брандмауэр приводится в действие путем загрузки модуля ядра, инициализации и запуска процесса-демона, который передает модулю данные, определяющие правила фильтрации пакетов. Эти правила демон получает при инициализации.

Задача модуля – взаимодействие с приемной функцией протокола IP с целью анализа сетевого пакета и формирование данных для ведения log-файла.

Поступивший из сети пакет передается для дальнейшей обработки драйверу сетевого адаптера. Драйвер, выполнив возложенные на него обязанности, отдает пакет приемной функции протокола сетевого уровня (в нашем случае это IP). Приемная функция, прежде чем продолжить обработку пакета, передает его для анализа модулю ядра. На основании полученных от процесса-демона правил фильтрации модуль определяет, может ли пакет проследовать далее по сетевому стеку или он должен быть блокирован (отброшен). О своем решении модуль

уведомляет приемную функцию и процесс-демон, который на данном этапе выполняет роль секретаря (ведет log-файл).

Для упрощения задачи брандмауэр фильтрует только входящий трафик на предмет недопущения в систему пакетов, отправителем которых является хост с определенным IP-адресом.

Для того чтобы вся эта схема заработала, в ядро системы необходимо внести некоторые изменения.

## ЯДРО

### Приемная функция протокола IP

Первое, что мы должны сделать, – это модифицировать приемную функцию протокола IP. Данная функция расположена в файле `$(KERNEL_SRC)/net/ipv4/ip_input.c`, где `$(KERNEL_SRC)` – каталог с исходными текстами ядра.

В список заголовочных файлов добавляем еще один header-файл:

```
# ifdef CONFIG_SF_FIREWALL
# include <linux/sf_kernel.h>
# endif
```

Далее в данном файле находим функцию `ip_rcv`. Согласно комментариям она выполняет основную работу по

приему пакетов. В состав функции введем для своих нужд переменную:

```
# ifdef CONFIG_SF_FIREWALL
int err;
# endif
```

Перед операцией проверки контрольной суммы пакета (ip\_fast\_csum) добавляем код, осуществляющий передачу пакета модулю:

```
# ifdef CONFIG_SF_FIREWALL
if ((err=sf_fw_chk(iph,dev,SF_STATE_RECEIVE))!=1)
{
kfree_skb(skb);
return 0;
}
# endif
```

Передача пакета модулю происходит при помощи функции sf\_fw\_chk. Подробно эту функцию мы рассмотрим ниже. Если возвращаемое значение не равно 1, структура skb, содержащая полную информацию о принятом пакете (см. <linux/skbuff.h>), обнуляется, приемная функция отбрасывает этот пакет и ждет очередной.

### Файл sf\_kernel.h

В данном файле определены некоторые константы и содержится список экспортируемых функций. Файл имеет следующее содержание:

```
#define SF_RC_ACCEPT 1- разрешить прием пакета
#define SF_RC_BLOCK 0- заблокировать прохождение пакета
#define SF_STATE_RECEIVE 0- идентификатор приемной функции
                                протокола IP
extern int sf_fw_chk_pass(struct iphdr *, struct net_device
*, int);
extern int sf_fw_chk_block(struct iphdr *, struct net_device
*, int);
extern int (*sf_fw_chk)(struct iphdr *, struct net_device
*, int);
```

Функция sf\_fw\_chk\_pass разрешает прохождение пакета, а функция sf\_fw\_chk\_block – запрещает. Указатель (\*sf\_fw\_chk) настраивается на одну из этих функций в зависимости от ситуаций, которые мы рассмотрим ниже.

Все эти функции определим в файле sf\_stub.c.

Сформированный файл sf\_kernel.h необходимо разместить в каталоге \${KERNEL\_SRC}/include/linux.

### Файл sf\_stub.c

Здесь, как уже было сказано, находятся определения функций sf\_fw\_chk\_pass, sf\_fw\_chk\_block и указателя (\*sf\_fw\_chk). В данный файл необходимо включить следующие заголовочные файлы:

```
#include <linux/config.h>
#include <linux/kernel.h>
#include <linux/netdevice.h>
#include <linux/ip.h>
#include <linux/sf_kernel.h>
```

По умолчанию, пока модуль не загружен, указатель (\*sf\_fw\_chk) настраивается на функцию sf\_fw\_chk\_pass, и все принятые пакеты обрабатываются стандартным способом:

```
#ifdef CONFIG_SF_FIREWALL
int (*sf_fw_chk)(struct iphdr *ip, struct net_device
*rif, int opt) = sf_fw_chk_pass;
```

Функция sf\_fw\_chk\_pass разрешает прием всех поступивших пакетов:

```
int sf_fw_chk_pass(struct iphdr *ip, struct net_device *rif,
int opt)
{
return SF_RC_ACCEPT;
}
```

Функция sf\_fw\_chk\_block блокирует прием пакетов:

```
int sf_fw_chk_block(struct iphdr *ip, struct net_device *rif,
int opt)
{
return SF_RC_BLOCK;
}
#endif /* CONFIG_SF_FIREWALL */
```

Обе функции принимают три параметра:

- заголовок IP-пакета (struct iphdr, определена в файле <linux/ip.h>);
- структуру с информацией о сетевом интерфейсе (struct net\_device, определена в заголовочном файле <linux/netdevice.h>);
- идентификатор приемной функции протокола IP (int opt).

Сформированный файл sf\_stub.c необходимо разместить в каталоге \${KERNEL\_SRC}/net/ipv4. В Makefile, находящийся в этом же каталоге, в секцию obj-у добавляется запись sf\_stub.o для формирования соответствующего объектного модуля и последующего включения его в общий модуль ipv4.o.

### Файл ksyms.c

Заключительным этапом внесения изменений в ядро является добавление экспортируемых функций в таблицу символов ядра, которая расположена в файле \${KERNEL\_SRC}/kernel/ksyms.c. В перечне заголовочных файлов укажем дополнительно файл sf\_kernel.h:

```
#ifdef CONFIG_SF_FIREWALL
#include <linux/sf_kernel.h>
#endif
```

и дополним таблицу символов:

```
#ifdef CONFIG_SF_FIREWALL
EXPORT_SYMBOL(sf_fw_chk_pass);
EXPORT_SYMBOL(sf_fw_chk_block);
EXPORT_SYMBOL(sf_fw_chk);
#endif
```

### Файл config.in

Для включения в ядро всех сделанных изменений добавим в файл \${KERNEL\_SRC}/arch/i386/config.in в секцию "GENERAL SETUP" следующую запись:

```
bool 'SF_FIREWALL_SUPPORT' CONFIG_SF_FIREWALL
```

После этого необходимо перекомпилировать ядро с включенной поддержкой нашего брандмауэра.

## МОДУЛЬ ЯДРА

Тем из вас, кто не знаком с вопросом разработки модулей ядра для операционной системы GNU/Linux, рекомендую для изучения статью «Написание драйверов в Linux: первые шаги» ([http://www.programme.ru/archive/2001/8/082001\\_1.phtml](http://www.programme.ru/archive/2001/8/082001_1.phtml)).

Наш модуль представляет из себя символьное (байт-ориентированное) устройство. Для него необходимо создать файл устройства следующей командой:

```
mknod /dev/firewall c 44 0
```

## Заголовочные файлы и переменные

Нам понадобятся следующие заголовочные файлы:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/netdevice.h>
#include <linux/types.h>
#include <linux/ip.h>
#include <linux/sf_kernel.h>
#include <asm/uaccess.h>
```

Определим старший номер устройства:

```
#define FIREWALL_MAJOR 44 - старший номер устройства.
```

Структура с данными для заполнения log-файла:

```
struct data_log
{
    u32 addr;
    int action;
    int ready;
}
*sf_entry_log;
```

Назначение полей структуры следующее:

- addr – IP-адрес отправителя пакета;
- action – выполненное действие (1 – пакет принят, 0 – пакет отброшен);
- ready – флаг готовности данных для считывания.

## Регистрация устройства в системе

Регистрацию устройства в системе выполняет функция `init_module`:

```
int init_module(void)
{
    if
    (register_chrdev (FIREWALL_MAJOR,"firewall",&firewall_fops))
    {
        printk("unable to get major %d for firewall
device\n", FIREWALL_MAJOR);
        return -EIO;
    }
    return 0;
}
```

Если при регистрации произошла ошибка, измените значение номера `FIREWALL_MAJOR`.

## Функции модуля

Третьим параметром функции регистрации `init_module`

является адрес структуры `struct file_operations firewall_fops` (см. `<linux/fs.h>`). Для нашего модуля мы определим функции открытия устройства, записи/чтения и закрытия. Следовательно, структура `firewall_fops` примет следующий вид:

```
struct file_operations firewall_fops = {
    read:      read_firewall,
    write:     write_firewall,
    open:      open_firewall,
    release:   close_firewall,
};
```

## Функция открытия устройства

Перед началом работы с устройством необходимо открыть его при помощи системного вызова `open()`. Функция имеет следующий вид:

```
static int open_firewall(struct inode *inode, struct file
*file)
{
```

Если устройство уже открыто, сообщить об этом:

```
if(MOD_IN_USE) return -EBUSY;
```

Первое, что мы сделаем при открытии, проверим значение младшего номера:

```
if (MINOR(inode->i_rdev) != 0) return -ENODEV;
```

Младший номер должен быть равен 0. Если это не так, возвращается сообщение об отсутствии данного устройства в системе.

Если все в порядке, проверяем режим работы устройства. Устройство должно быть открыто в режиме записи/чтения:

```
if ((file->f_mode & 1) != 1) return -EBUSY;
```

Следующий шаг – выделение памяти для структур:

```
sf_entry_log=(struct data_log *)kmalloc(sizeof(struct
data_log),GFP_ATOMIC);
iph=(struct iphdr *)kmalloc(sizeof(struct
iphdr),GFP_ATOMIC);
```

Данная операция осуществляет выделение памяти в пространстве ядра. Наличие спецификатора `GFP_ATOMIC` (`GFP` – Get Free Page) указывает на необходимость выделения памяти немедленно (в отличие от `GFP_KERNEL`).

Теперь необходимо снять заглушку, которую мы ранее установили. Напомню, что указатель (`*sf_fw_chk`) был настроен на функцию `sf_fw_chk_pass`, и все принятые пакеты беспроблемно проходили в систему. После открытия модуль снимает заглушку путем настройки указателя (`*sf_fw_chk`) на функцию `sf_check_packet`, которая осуществляет непосредственный анализ принятого пакета на соответствия условиям фильтрации:

```
sf_fw_chk = sf_check_packet;
```

Сама функция `sf_check_packet` будет приведена ниже. Устанавливаем флаги доступности устройства и готовности данных:

```
sf_fw_enabled++;
sf_entry_log->ready=1;
```

Увеличиваем счетчик использования модуля и выходим из функции:

```
MOD_INC_USE_COUNT;
return 0;
}
```

### Функция записи в устройство

В процессе записи модулю передается IP-адрес хоста, чьи пакеты необходимо заблокировать. Функция имеет следующий вид:

```
static ssize_t write_firewall(struct file *file, const char
*buf, size_t count, loff_t *ppos)
{
```

Второй аргумент функции – блок данных, который передается модулю, третий аргумент – размер блока в байтах. Последний аргумент определяет текущую позицию в файле устройства, мы его в данном примере не используем.

В модуль мы будем передавать структуру, содержащую IP-заголовок. В этой структуре поле, соответствующее адресу источника, будет содержать значение, определяющее правило фильтрации. Размер блока данных должен быть равен размеру этой структуры. Проверим это:

```
if(count!=sizeof(struct iphdr)) return -EINVAL;
```

На время записи в устройство данных заблокируем прохождение пакетов. Для этого определим вспомогательный указатель на функцию:

```
int (*sf_fw_chk_save)(struct iphdr *, struct net_device *,
int);
```

Заблокируем прохождение пакетов:

```
sf_fw_chk_save = sf_fw_chk;
sf_fw_chk = sf_fw_chk_block;
```

Считываем блок данных в модуль:

```
copy_from_user(ip,buf,sizeof(struct iphdr));
```

Функция `copy_from_user()` осуществляет копирование блока данных из пространства пользователя в пространство ядра. Данная функция определена в файле `<asm/uaccess.h>`.

Снимаем блокировку прохождения пакетов:

```
sf_fw_chk = sf_fw_chk_save;
```

Фиксируем новую позицию в файле устройства:

```
file->f_pos += count;
```

Возвращаем в вызывающую функцию число записанных байт:

```
count = sizeof(struct iphdr);
return count;
}
```

### Функция чтения из устройства

В процессе чтения из устройства считывается информация для заполнения `log`-файла.

Функция чтения имеет следующий вид:

```
static ssize_t read_firewall(struct file *file, char *buf,
size_t count, loff_t *ppos)
{
```

Второй аргумент данной функции является указателем на структуру `struct data_log`. Проверяем размер запрашиваемых данных:

```
if (count!=sizeof(struct data_log)) return -EINVAL;
```

Проверяем доступность устройства:

```
if (sf_fw_enabled<=0) return -ENODEV;
```

Передаем вызывающей функции запрашиваемые данные:

```
copy_to_user(buf,sf_entry_log,sizeof(struct data_log));
```

Структуру `struct data_log` заполняет функция `sf_check_packet()`.

Фиксируем новую позицию в файле устройства:

```
file->f_pos += count;
```

Сбрасываем флаг готовности данных для считывания. Установку этого флага выполнит функция `sf_check_packet()` после получения очередного пакета:

```
sf_entry_log->ready = 0;
```

Возвращаем в вызывающую функцию число считанных байт:

```
count = sizeof(struct data_log);
return count;
}
```

### Функция `sf_check_packet`

Как уже говорилось, данная функция выполняет проверку принятого пакета на соответствие условиям фильтрации и заполняет информационную структуру `struct data_log` для ведения `log`-файла.

Функция имеет следующий вид:

```
int sf_check_packet(struct iphdr *ip, struct net_device *rif,
int opt)
{
```

Аргументы функции были перечислены выше. Заполнить поле `addr` структуры `sf_entry_log` IP-адресом источника пакета:

```
sf_entry_log->addr = ip->saddr;
```

Проверить адрес принятого пакета:

```
if (ip->saddr == iph->saddr) {
    sf_entry_log->action = SF_RC_BLOCK;
    sf_entry_log->ready = 1;
    return SF_RC_BLOCK;
}
```

Если IP-адрес полученного пакета совпадет с тем, который необходимо заблокировать, приемная функция протокола IP получит команду на сброс пакета, и будет установлен флаг готовности данных для чтения из устройства.

Если это условие не выполняется, то пакет будет допущен для дальнейшей обработки:

```
sf_entry_log->ready = 1;
sf_entry_log->action = SF_RC_ACCEPT;
return SF_RC_ACCEPT;
}
```

## Функция закрытия устройства

Функция имеет следующий вид:

```
static int close_firewall(struct inode *inode, struct file *file)
{
```

Декремент флага доступности устройства:

```
sf_fw_enabled--;
```

Блокируем все поступающие пакеты:

```
sf_fw_chk = sf_fw_chk_block;
```

Освобождаем память:

```
kfree(sf_entry_log);
kfree(iph);
```

Уменьшаем счетчик использования модуля:

```
MOD_DEC_USE_COUNT;
return 0;
}
```

## Снятие регистрации устройства

Снятие регистрации выполняет функция `cleanup_module`:

```
void cleanup_module(void)
{
```

Проверяем, использует кто-нибудь наш модуль или нет:

```
if (MOD_IN_USE)
{
    printk("firewall: busy - remove delayed\n");
    return;
}
```

Устанавливаем заглушку – разрешаем прохождение пакетов в систему:

```
sf_fw_chk = sf_fw_chk_pass;
```

Снимаем регистрацию:

```
unregister_chrdev(FIREWALL_MAJOR, "firewall");
return;
}
```

## Makefile

Для получения загружаемого модуля создадим Makefile следующего содержания:

```
include make.options

# Компилятор
CC = gcc

# Имя модуля
module = sf_device.o

# флаги компиляции

CFLAGS = -O2 -Wall -fomit-frame-pointer
MODFLAGS = -D_KERNEL -DMODULE -I$(LINUX)/include
sf_device.o: sf_device.c
$(CC) -c $(CFLAGS) $(MODFLAGS) sf_device.c
```

Назначение флагов:

- O2 - включить оптимизацию;
- Wall – выводить на экран все предупреждения о возможных ошибках (`warning all`);
- `fomit-frame-pointer` – не сохранять указатель на кадр (`frame pointer`) в регистре для функций, которые не нуждаются в этом. Это позволяет избежать инструкций на сохранение, определение и восстановление указателя на кадр (`frame pointer`), в то же время освобождая регистры для других функций.

Содержание файла `make.options`:

```
# Каталог с исходниками ядра
LINUX = /usr/src/linux
```

## Компиляция и загрузка модуля

Для компиляции достаточно ввести команду `make`. В текущем каталоге появится файл `sf_device.o`. Загрузка модуля осуществляется командой `insmod`:

```
insmod sf_device.o
```

Удаление модуля из памяти выполняет команда `rmmod`:

```
rmmod sf_device
```

Если у вас появятся вопросы по данной части статьи – задавайте их на форуме журнала.

Приведенный код был разработан и протестирован GNU/Linux, дистрибутив Slackware 7.1, ядро 2.4.17, компилятор `gcc-2.95.2`.