хачим ядро xBSD

крис касперски ака мыщъх, no email

техника ### тактика внедрения в ядро xBSD и принципы перехвата системных функций мало чем отличаются от LINUX и NT, однако, здесь есть своя специфика, о которой мы и собираемся рассказать, подкрепив слова конкретными листингами, учитывающими архитектурные особенности FreeBSD, NetBSD и OpenBSD

введение

Ядро изолированно от адресного пространства прикладных приложений и для взаимодействия с ним операционная система представляет ряд интерфейсов. FreeeBSD и NetBSD имеют монолитное ядро, поддерживающее загрузку динамических модулей, очень похожих на модули LINUX и чем-то напоминающие NT-драйвера. Загрузка модуля осуществляется "на лету", не требуя перезагрузки операционной системы, что очень хорошо. Естественно, для этого требуется права root'a, которые необходимо каким-то образом заполучить (но это уже тема для отдельного разговора). В GENERIC-ядре OpenBSD модули по умолчанию включены, но многие администраторы собирают монолитное ядро без поддержи модульности, считая, что это увеличивает защищенность, лишая атакующего возможности внедрять в ядро вредоносный код, однако…

Все xBSD-системы поддерживают псевдоустройство /dev/[k]mem (аналогичное тому, что имеется в LINIX) и библиотеку функций **`libkvm`** для работы с ним, прямых аналогов которой в LINUX нет. Поэтому, даже когда модули недоступны, у нас по-прежнему остается возможность модификации ядра, осуществляемая непосредственно с прикладного уровня, при условии, что мы владеем правами root'a.

Рассмотрим все эти способы поподробнее.

Рисунок 1 UNIX-подобным системам приходится конкурировать не только с Windows, но и воевать между собой

LKM-модули NetBSD и OpenBSD

Операционные системы NetBSD и OpenBSD поддерживают **LKM-модули** (***L**oadable **K**ernel **M**odule – Загружаемые Модули Ядра*), "слизанные" с SunOS 4 и совместимые на интерфейсном уровне, реализованном через псевдоустройство **`/dev/lkm`**, с которым прикладные приложения взаимодействуют посредством вызовов `ioctl` (подробнее см. "**`man 4 lkm`**").

В OpenBSD модули могут быть загружены только на нулевом уровне безопасности, если же уровень отличен от нуля, а загрузить модули все-таки необходимо, следует отредактировать файл `/etc/rc.securelevel`, загружая модули _до_ того, как уровень безопасно будет установлен на необходимую величину, но в этом случае о динамической загрузке следует забыть, расстаравшись с одним из наиболее элегантных свойств оси.

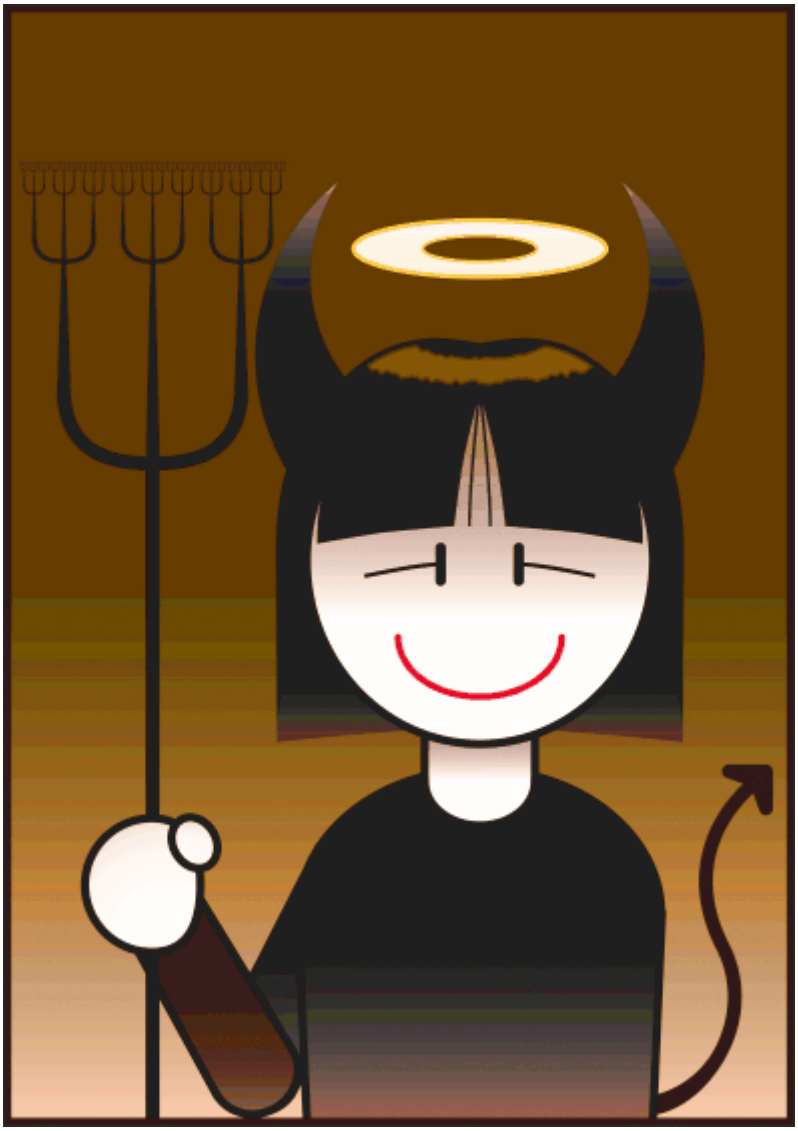

Рисунок 2 OpenBSD – самая защищенная ось в линейке BSD

Модули бывают разных типов: **System Call Modules**, реализующие новые системные вызовы (или замещающие уже существующие); **Virtual File System Modules**, поддерживающие виртуальные файловые системы; **Device Drive Modules**, управляющие существующими или несуществующими шинами и устройствами; **Execution Interpreter Modules**, отвечающие за загрузку различных исполняемых форматов; **Miscellaneous Modules** к которым относятся все модули, не попадающие ни под одну из классификаций (подробнее см. "`man 9 module`").

Каждый тип модуля имеет свои особенности реализации, однако, нам они без разницы. Управлять оборудованием мы не собираемся, устанавливать новую файловую систему — тоже.

Перехват системных вызов может быть осуществлен из любого модуля, а не только MOD_SYSCALL. Это можно сделать непосредственно в процедуре начальной инициализации модуля, что избавит нас от необходимости заполнять все служебные структуры, которые в случае Device Drive модулей довольно громоздки.

Примеры готовых модулей можно найти непосредственно в самой NetBSD/OpenBSD, обратившись к каталогу "`/usr/share/lkm/`" или скачать их напрямую из Сети: http://www.openbsd.org/cgi-bin/cvsweb/src/share/lkm. Забавно, но в OpenBSD эти файлы не модифицировались свыше 6 лет! Примеры из NetBSD посвежее будут — "всего" 5 лет выдержки, однако, по большому счету никакой разницы между ними нет и они практически одни-в-один повторяют друг друга.

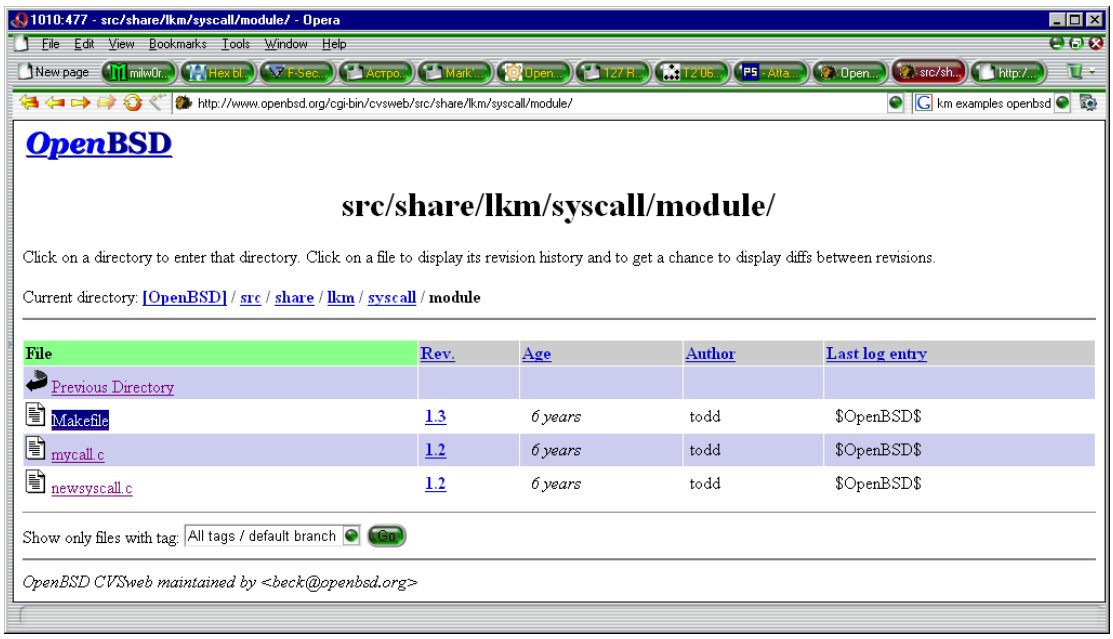

Рисунок 3 демонстрационные примеры LKM-модулей, входящие в состав OpenBSD, последний раз модифицировались 6 лет назад!

Ниже приведен скелет простейшего MOD_SYSCALL-модуля, перехватывающего системный вызов #1 (`mkdir`) и устанавливающий на него свою хакерскую "заглушку", выводящую на экран "rock you" и мерзко пищащую спикером. При желании, из нее можно вызвать оригинальную функцию `mkdir`, передав управление по адресу, сохраненному в переменной, `old_mkdir`, (более подробно о создании LKM-модулей под OpenBSD и NetBSD можно прочитать в статьях "**OpenBSD Loadable Kernel Modules**" и "**Writing a LKM Driver for NetBSD**", адреса которых приведены во врезке):

```
/* модуль, перехватывающий mkdir, и работающий под Net- и OpenBSD */
/* ============================================================== */
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/ioctl.h>
#include <sys/cdefs.h>
#include <sys/conf.h>
#include <sys/mount.h>
#include <sys/exec.h>
#include <sys/lkm.h>
#include <sys/proc.h>
#include <sys/syscallargs.h>
#include <sys/syscall.h>

/* объявляем переменную old_mkdir, в которую позже будет записан */
/* оригинальный адрес системного вызова mkdir                    */
int (*old_mkdir) (struct proc * p, void * v, register_t retval);


/* функция-заглушка, устанавливаемая на место mkdir              */
int hack(struct proc *p, void *v, int *retval)
{
```

```
        printf ("rock you!\x7\n"); return 0;
}

/* процедура начальной загрузки модуля                          */
static int load(struct lkm_table *lkmtp, int cmd)
{
        if (cmd == LKM_E_LOAD) /* загрузка модуля */
        {
                printf ("syshack loadedd\n");

                /* сохраняем адрес оригинального вызова mkdir    */
                (sy_call_t *)old_mkdir = sysent[SYS_mkdir].sy_call;

                /* устанавливаем вместо mkdir свою "заглушку"    */
                sysent[SYS_mkdir].sy_call = (sy_call_t *)hack;
        }

        if (cmd == LKM_E_UNLOAD) /* выгрузка модуля */
        {
                printf ("syshack unloadedd\n");

                /* снимаем свою заглушку, возвращая на место mkdir */
                sysent[SYS_mkdir].sy_call=(sy_call_t*)old_mkdir;
        }
        return(0);
}

/* точка входа в модуль                                         */
int entry(struct lkm_table *lkmtp, int cmd, int ver)
{
        /* сердце модуля - макрос DISPATCH */
        DISPATCH(lkmtp, cmd, ver, load, load, lkm_nofunc);
}
```

Листинг 1 файл syshack.c, реализующий простейший LMK-модуль, демонстрирующий технику перехвата системных вызов под Net- и OpenBSD

Сердцем модуля является макрос **DISPATCH**, передающий управление функции инициализации и деинициализации (в нашем случае она называется load), вызываемой при загрузке и выгрузке модуля. Для перехвата/освобождения системного вызова mkdir используется прямая правка таблицы системных вызовов **sysent**. В принципе, можно было воспользоваться макросом MOD_SYSCALL, но это не хакерство и вообще неинтересно.

Компиляция нашего модуля осуществляется так:

```
# gcc -D_LKM -D_KERNEL -I/sys -c syshack.c
```

Листинг 2 компиляция LKM-модулей компилятором gcc

При желании можно воспользоваться Make-файлом следующего содержания (и тогда для сборки модуля будет достаточно набрать "make"):

```
KSRCS=syshack.c
KOBJS=syshack.o

KMOD=syshack
CFLAGS= -D_LKM -D_KERNEL -I/sys
```

Листинг 3 Make-файл, собирающий LKM-модули

Как видно, Make-файл практически полностью повторяет ключи компиляции gcc, а именно: -D_LKM и -D_KERNEL, впрочем, -D_LKM вроде бы можно и опустить ("вроде бы" — потому что на всех системах мыщъх это не проверял).

За загрузку модуля в память ядра отвечает утилита **modload** (см. "man 8 modload"), вызываемая следующим образом:

```
# modload -o hack -eentry syshack.o
Module loaded as ID 0
```

Листинг 4 загрузка LKM-модуля в память ядра утилитой modload, здесь: hack – имя модуля в памяти, entry – точка входа в модуль, syshack.o – имя скомпилированного объектного файла

Проверить успешность загрузки модуля можно утилитой "**modstat**" (см. "man 8 modstat"):

```
# modstat
Type     Id Off Loadaddr Size Info     Rev Module Name
SYSCALL   0 210 e0b92000 0002 e0b93008   2 hack
```

Листинг 5 утилита modstat показывает наличие модуля hack в памяти, значит, загрузка прошла успешно!

Если модуль действительно загружен, то появится строчка с его именем (в данном случае — "hack") и с этого момента любые попытки создать новый каталог утилитой mkdir будут обречены на провал вплоть до того времени пока мы не выгрузим модуль из памяти утилитой **modunload** (см. "man 8 modunload"):

```
# modunload -n hack
```

Листинг 6 выгрузка модуля из памяти утилитой modunload

Перехват остальных системных вызовов осуществляется аналогичным образом. При желании, наш модуль может скрывать от глаз администратора некоторые процессы или файлы, стелсируясь на уровне ядра, но об этом мы поговорим как ни будь в другой раз, а пока же нам предстоит разобраться с FreeBSD, в которой модули реализованы совсем иначе.

KLD-модули FreeBSD

Ранние версии FreeBSD поддерживали LKM-модули наравне со своими конкурентами, но начиная с FreeBSD 3.0 интерфейс модулей был изменен на **KLD**, что расшифровывает как ***D**ynamic **K**ernel **L**inker – Динамическое Связывание Ядра* и LKM-модули отошли на задний план (в текущих версиях FreeBSD их поддержка прекращена).

Рисунок 4 по сравнению с Net/OpenBSD, FreeBSD развивается намного более активно и связывают (линкуют) пингвина только так!

В практическом плане это в первую очередь означает, что старые исходные тексты необходимо переделывать, а в некоторых случаях — чуть ли не переписывать заново. На этом

фоне, преимущества нового типа модулей полностью девальвируются. Кстати говоря, штатное руководство (см. "`man KLD`") лишь заявляет о преимуществах, но не перечисляет их и за разъяснением приходится обращаться к статье "**Attacking FreeBSD with Kernel Modules**", адрес которой приведен во врезке.

```
          response from the standard input begins with the character 'y' or
          'Y', the file copy is attempted.  (The -i option overrides any pre-
          vious -f options.)

# cp -R /usr/share/examples/kld/* /root/A
# man -k KLD
kld(4) - dynamic kernel linker facility
kldconfig(8)            - display or modify the kernel module search path
kldfind(2)              - returns the fileid of a kld file
kldfirstmod(2)          - return first module id from the kld file specified
kldload(2)              - load KLD files into the kernel
kldload(8)              - load a file into the kernel
kldnext(2)              - return the fileid of the next kld file
kldstat(2)              - get status of kld file
kldstat(8)              - display status of dynamic kernel linker
kldsym(2)               - look up address by symbol name in a KLD
kldunload(2)            - unload kld files
kldunload(8)            - unload a file from the kernel
# ls -l /usr/share/examples/kld
total 10
-rw-r--r--  1 root  wheel  3650 Jan 28  2002 Makefile
drwxr-xr-x  4 root  wheel   512 Sep 14  2004 cdev
drwxr-xr-x  3 root  wheel   512 Sep 14  2004 dyn_sysctl
drwxr-xr-x  4 root  wheel   512 Sep 14  2004 syscall
#
```

Рисунок 5 запрос "man -k KLD" обнаруживает множество документов, относящихся к загружаемым модулям ядра во FreeBSD

Если не углубляться в детали, то LKM-модуль – это ELF-файл, загружаемый в адресное пространство ядра, а KLD – это часть самого ядра, которая, в отличии от LKM, может быть загружена в любое время _без_ поддержки со стороны прикладного уровня. То есть, ядро в процессе старта системы как бы собирает себя из блоков, загружаемых/выгружаемых в любой момент времени.

KLD модули предоставляют намного больше возможности для разработчиков драйверов, но на нас это никак не распространяется. Перехват системных реализуется так же, как и раньше. Меняется только декларация модуля, макросы и некоторые структуры данных .

Примеры готовых модулей можно найти в каталоге "`/usr/share/examples/kld/`" или стянуть их из Сети: http://www.freebsd.org/cgi/cvsweb.cgi/src/share/examples/kld/. "Зрелость" файлов варьируется от нескольких месяцев до 7 (!) лет.

```
Stop in /usr/share/examples/kld/dyn_sysctl.
*** Error code 1

Stop in /usr/share/examples/kld.
# ls
Makefile        cdev            dyn_sysctl      syscall
# cd cdev
# ls
Makefile        README          module          test
# cd module
# ls
@               cdev.h          cdev.o          machine         setdef1.c
Makefile        cdev.kld        cdevmod.c       setdef0.c       setdef1.o
cdev.c          cdev.ko         cdevmod.o       setdef0.o       setdefs.h
# kldload ./cdev.kld
kldload: Unsupported file type
kldload: can't load ./cdev.kld: Exec format error
# kldload ./cdev.ko

Sample Loaded kld character device driver
Copyright (c) 1998
Rajesh Vaidheeswarran
All rights reserved
#
```

Рисунок 6 откомпилированный и загруженный демонстрационный KLD-модуль, представляющий собой драйвер символьного устройства

Программа, чей код приведен ниже, демонстрирует технику перехвата системного вызова под FreeBSD из KLD-модуля:

```
/* модуль, перехватывающий mkdir, и работающий под FreeBSD          */
/* based on:                                                         */
/* syscall.c by Assar Westerlund and hacked_mkdir.c by Joseph Kong  */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/sysproto.h>
#include <sys/systm.h>
#include <sys/syscall.h>

/* функция-заглушка, устанавливаемая на место mkdir              */
static int hack (struct proc *p, void *arg)
{
        printf ("rock you!\x7\n"); return 0;
}

/* элемент структуры sysent, описывающий наш системный вызов     */
static struct sysent hack_sysent = {
        1,                      /* sy_narg */
        hack                    /* sy_call */
};

/* процедура начальной загрузки модуля                           */
static int load (struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd)
        {
        case MOD_LOAD:                        /* загрузка модуля */
                printf ("syshack loadedd\n");

                /* устанавливаем вместо mkdir свою "заглушку"    */
                sysent[SYS_mkdir]=hack_sysent;
                break;

        case MOD_UNLOAD:                      /* выгрузка модуля */
                printf ("syshack unloadedd\n");

                /* снимаем свою заглушку, возвращая на место mkdir */
                sysent[SYS_mkdir].sy_call=(sy_call_t*)mkdir;
                break;

        default:
                error = EINVAL;
                break;
        }
        return error;
}

/* структура, описывающая основные параметры модуля              */
static moduledata_t syscall_mod = {
        "Intercept",
        load,
        NULL
};

/* сердце программы — макрос DECLARE_MODULE, декларирующей модуль */
DECLARE_MODULE(syscall, syscall_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Листинг 7 файл syshack.c, реализующий простейший LMK-модуль, демонстрирующий технику перехвата системных вызов под FreeBSD

Наша базовая процедура `load` практически никак не изменилась (поменялись лишь определения cmd-кодов), а вот в декларации модуля произошли большие перемены. Функция с макросом `DISPATCH` исчезла, а вместе с ней исчезла и необходимость указывать точку входа в

модуль при его загрузке в память ядра. Новый макрос **DECLARE_MODULE** не только задает точку входа в модуль вместе с его типом, но так же определяет порядок загрузки! Вообще-то, этот макрос — не единственный и с не меньшим успехом мы могли бы воспользоваться DEV_MODULE или SYSCALL_MODULE (см. "man 9 DEV_MODULE" и "man 9 SYSCALL_MODULE"), но это уже дело вкуса, споры о которых рискуют превратиться в священные войны, а ведь прежде, чем воевать, модуль еще откомпилировать надо!

В общем случае, сборка осуществляется следующим Make-файлом, причем строки KO и KLMOD не являются обязательными:

```
SRCS    = syshack.c
KMOD    = syshack
KO      = ${KMOD}.ko
KLDMOD  = t

.include <bsd.kmod.mk>
```

Листинг 8 Make-файл, собирающий KLD-модули

Если компиляция прерывается сообщением "can't find kernel source tree" это значит, что у вас не установлены исходные тексты ядра или bsd.kmod.mk-файл не может их найти. Установить недостающие компоненты можно в любой момент, запустив утилиту /stand/sysinstall и отметив пункт "Kernel Developer – Full binaries and doc, kernel source only". Выходит, чтобы откомпилировать KLD-модуль необходимо иметь сырцы ядра! Вот такая она, FreeBSD! Ни NetBSD, ни OpenBSD ничего подобного не требуют! (что ж, вполне логично, LKM-модули, в отличии от KLD, не являются частью ядра).

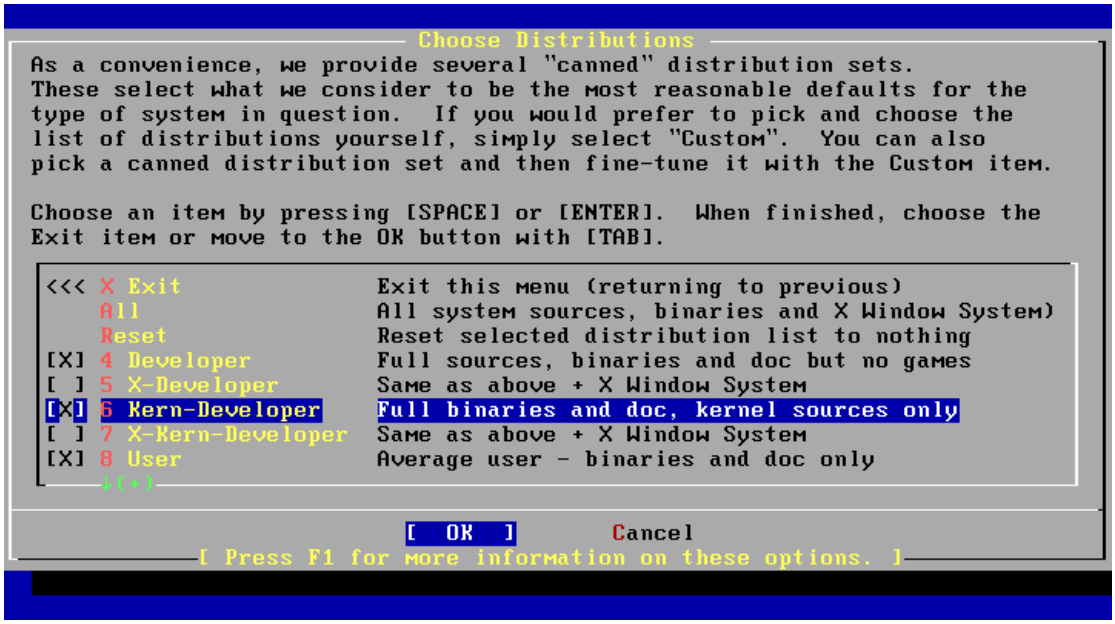

Рисунок 7 компиляция KLD-модулей требует наличия исходных текстов ядра, которые легко установить, запустив утилиту /stand/sysinstall

После компиляции на диске образуется множество "левых" файлов и ссылок на системные каталоги (которые можно тут же удалить), среди которых затерялся файл с расширением .ko – это и есть наш модуль (в данном случае он называется syshack.ko).

Загрузка модуля в память осуществляется утилитой **kldload** (см "man 8 kldload") которой указывается имя модуля (если модуль расположен в текущей директории, необходимо предварить его ./) и, при желании, ключ -v для более жестокой проверки корректности модуля (но наш модуль корректен, так что дополнительные меры контроля совершенно излишни).

```
# ls
Makefile        syshack.c       syshack.ko      syshack.o
# kldload ./syshack
syshack loadedd
# Jun 22 13:47:54  /kernel: syshack loadedd
Jun 22 13:47:54  /kernel: syshack loadedd

# kldstat
Id Refs Address    Size     Name
 1    4 0xc0100000 394090   kernel
 2    1 0xc0cac000 3000     daemon_saver.ko
 3    1 0xc0caf000 14000    linux.ko
 8    1 0xc08e3000 2000     syshack.ko
# mkdir XXX
rock you!
# Jun 22 13:48:08  /kernel: rock you!
Jun 22 13:48:08  /kernel: rock you!

# ls
Makefile        syshack.c       syshack.ko      syshack.o
# kldunload syshack
syshack unloadedd
# Jun 22 13:49:07  /kernel: syshack unloadedd
Jun 22 13:49:07  /kernel: syshack unloadedd

# mkdir XXX
# ls
Makefile        XXX             syshack.c       syshack.ko      syshack.o
#
#
```

Рисунок 8 загрузка и выгрузка KLD-модуля

Убедиться в успешности загрузки поможет утилита **kldstat** (см. "man 8 kldstat"), которая будучи запущенной без аргументов, выводит "свиток" всех имеющихся модулей. Если среди них присутствует syshack.ko, то операция перехвата прошла успешно и теперь всякая попытка создания новой директории будет обречена на провал. Вплоть до выгрузки модуля из памяти, конечно, что можно сделать в любой время утилитой **kldunload** (см. "man 8 kldunload") указав ей имя модуля без расширения.

Полный протокол работы с системой приведен ниже. Там же находятся и комментарии.

```
# kldstat                     ; запускаем kldstat, чтобы просмотреть список модулей
Id Refs Address    Size     Name
 1    3 0xc0100000 394090   kernel           ; ядро
 2    1 0xc0cac000 3000     daemon_saver.ko  ; хранитель экрана
 3    1 0xc0caf000 14000    linux.ko         ; эмулятор LINUX'a
; как видно, syshack модуля среди них нет (было бы удивительно, если бы он был)


# ls                          ; просматриваем текущий каталог утилитой ls
Makefile        syshack.c       syshack.ko      syshack.o
; файл syshack.ko это и есть откомпилированный KLD-модуль

# kldload ./syshack           ; загружаем наш модуль в память
syshack loaded
# Jun 22 13:58:20  /kernel: syshack loadedd
Jun 22 13:58:20  /kernel: syshack loadedd
; модуль рапортует об успешной загрузке
; и система дублирует это сообщение, указывая время его появления

# kldstat                     ; снова просматриваем список загруженных модулей
Id Refs Address    Size     Name
 1    4 0xc0100000 394090   kernel           ; ядро
 2    1 0xc0cac000 3000     daemon_saver.ko  ; хранитель экрана
 3    1 0xc0caf000 14000    linux.ko         ; эмулятор LINUX'a
10    1 0xc08e3000 2000     syshack.ko       ; вот он, наш модуль!
; как видно syshack появился в списке модулей,
; значит, загрузка и перехват системного вызова mkdir прошли успешно

# mkdir TEST-DIR                              ; пытаемся создать каталог TEST-DIR
rock you!                                     ; сообщение нашего модуля
# Jun 22 13:58:57  /kernel: rock you!
Jun 22 13:58:57  /kernel: rock you!
; ...но вместо создания нового каталога,
```

```
; mkdir пищит спикером и посылает нас на хутор за бабочками!

# ls                                        ; просматриваем текущий каталог
Makefile        syshack.c       syshack.ko      syshack.o
; директории TEST-DIR действительно нет,
; вот что значит правильно организованный перехват!

# kldunload syshack                         ; выгружаем модуль из памяти
syshack unloadedd
# Jun 22 14:00:44  /kernel: syshack unloadedd
Jun 22 14:00:44  /kernel: syshack unloadedd
; модуль выгрузил себя из памяти,
; восстановив оригинальный mkdir

# mkdir TEST-DIR                            ; пытаемся создать TEST-DIR еще раз
; теперь на экран не выводится никаких сообщений

# ls                                        ; проверяем успешность создания TEST-DIR
Makefile        TEST-DIR        syshack.c       syshack.ko      syshack.o
; каталог TEST-DIR действительно создан!
; значит, mkdir был восстановлен правильно!

# kldstat                                   ; просматриваем список модулей
Id Refs Address    Size     Name
 1    3 0xc0100000 394090   kernel           ; ядро
 2    1 0xc0cac000 3000     daemon_saver.ko  ; хранитель экрана
 3    1 0xc0caf000 14000    linux.ko         ; эмулятор LINUX'a
; модуля syshack в этом списке нет,
; значит, его выгрузка прошла успешно
```

Листинг 9 полный протокол перехвата и освобождения системного вызова mkdir посредством KLD-модулей

работа с libkmv
или что делать, когда модули недоступны

Все операционные системы семейства BSD поддерживают библиотеку **libkvm** (***K**ernel **V**irtual **M**emory*), предоставляющую унифицированный доступ к памяти ядра и, как KLM-модули впервые появившуюся в Sun OS, но в отличии от них, сохранившую полную обратную совместимость. Другими словами, программа, написанная для FreeBSD, при переносе на OpenBSD или NetBSD, не потребует никаких изменений!

Фактически, библиотека `libkvm` представляет собой высокоуровневую обертку вокруг псевдоустройства `/dev/mem`, изображающего из себя физическую оперативную память (псевдоустройство `/dev/kmem` включает в себя лишь виртуальную память ядра после трансляции адресов). Аналогичное псевдоустройство имеется и в LINUX'e, но соответствующей библиотеки для него нет, что жутко напрягает. Тем не менее, с псевдоустройством `/dev/[k]mem` на всех системах можно работать и напрямую через обычный ввод/вывод, для обеспечения полной переносимости, однако, целесообразность этого решения весьма сомнительна, поэтому в рамках этой ### статьи мы сосредоточимся исключительно на библиотеке `libkvm`, а остальные способы доступа к ядерной памяти оставим за кадром, тем более, что мыщъх уже писал про них в ### статье "перехват библиотечных функций в linux и bsd", так что не будем повторяться.

Прежде, чем работать с виртуальной памятью ядра, ее необходимо открыть, вызвав функцию **kvm_open** (см. "`man kvm_open`") и передав ей в качестве имени файла NULL, тогда (при успешном завершении операции, обеспеченных правами root'a) она вернет дескриптор. Если вместо NULL указать имя файла-образа ядра или кору, то открыты будут они, а не "живое" ядро в памяти, но нам это не нужно.

Передавая полученный дескриптор функциям **kvm_read**/**kvm_write**, мы сможем читать/писать память по заданным виртуальным адресам. Но какие именно адреса мы хотим читать? Вернее, как найти среди множества адресов полезную информацию, например, таблицу системных вызовов? В этом нам поможет функция **kvm_nlist**, разбирающая таблицу символов и возвращающая адрес элемента по его имени. Единственным ее аргументом (не считая дескриптора памяти ядра) является указатель на массив структур `nlist`, описанных в одноименном включаемом файле. В поле **n_name** заносимся имя интересующего нас элемента

и, если этот элемент действительно присутствует в таблице символов, тогда после завершения функции в поле **`n_value`** возвращается его виртуальный адрес.

Приведенная ниже программа (любезно позаимствованная из статьи "**Playing Games With Kernel Memory... FreeBSD Style**", опубликованной в #63 PHACK'e) определяет адрес таблицы системных вызов, адрес "нашего" системного вызова и адрес функции, по которой данный вызов располагается в памяти. Программа требует два аргумента — имя системного вызова (например, `mkdir`) и его номер (в случае `mkdir` равный 1), рекомендуя обратиться к файлу `/usr/src/sys/sys/syscall.h`, если номер вызова нам не известен (вообще-то данный файл располагается в каталоге `/usr/include/sys/`, но это неважно). На самом деле, имя системного вызова используется только для контроля, что такой вызов действительно существует, то есть никак не используется и для вычисления адреса используется номер syscall'a, который преобразуется в индекс таблицы системных вызовов. Это грубая недоработка! Если мы успешно определи адрес syscall'a по имени, то зачем нам его номер?! Если же мы можем (а мы можем) определять адреса syscall'ов по номеру через индекс в таблице системных вызовов, зачем нам нужно имя?!

Тем не менее, мыщъх решил оставить листинг "как есть", чтобы продемонстрировать два различных способа определения адресов системных вызовов.

```
/* программа демонстрирующая технику определения адресов системных вызовов, */
/* работающая на всем зоопарке BSD-подобных систем                           */
/* Based on Stephanie Wehner's checkcall.c,v 1.1.1.1                         */
#include <stdio.h>
#include <fcntl.h>
#include <kvm.h>
#include <nlist.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/sysent.h>
#include <sys/syscall.h>

int main(int argc, char *argv[])
{
        char errbuf[_POSIX2_LINE_MAX];
        kvm_t *kd; u_int32_t addr; int callnum; struct sysent call;
        struct nlist nl[] = { { NULL }, { NULL }, { NULL }, };

        if(argc != 3)
        {
                printf("Usage:\n%s <name of syscall> <syscall number>\n\n", argv[0]);
                printf("See /usr/src/sys/sys/syscall.h for syscall numbers\n");exit(0);
        }

        /* Find the syscall */
        nl[0].n_name = "sysent"; nl[1].n_name = argv[1]; callnum = atoi(argv[2]);

        /* Initialize kernel virtual memory access */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);

        /* Find the addresses */
        kvm_nlist(kd, nl);

        if(!nl[0].n_value)
                return fprintf(stderr,"ERROR: %s not found\n", nl[0].n_name);
        else
                printf("%s is 0x%x at 0x%x\n",nl[0].n_name,nl[0].n_type,nl[0].n_value);

        /* Calculate the address */
        addr = nl[0].n_value + callnum * sizeof(struct sysent);

        /* Print out location */
        if(kvm_read(kd, addr, &call, sizeof(struct sysent)) < 0)
                return fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
        else
                printf("sysent[%d] is at 0x%x and will execute function"
                        " located at 0x%x\n", callnum, addr, call.sy_call);

        kvm_close(kd);
}
```

Листинг 10 findsyscall.c, определяющий виртуальные адреса системных вызовов и работающий на Free-, Net- и OpenBSD

При трансляции листинга компилятору необходимо указать на библиотеку `libkvm` (при этом "`lib`" как всегда опускается), иначе линкер начнет материться на неразрешимые ссылки.

```
#gcc find_syscall.c -o find_syscall -lkvm
```

Листинг 11 компиляция программы find_syscall.c

Откомпилировав программу, попробуем определить адрес системного вызова `mkdir`. На моей машине (FreeBSD 4.5) результат выглядит так:

```
#./find_syscall mkdir 1
Finding syscall 1: mkdir

sysent is 0x4 at 0xc03ca480
sysent[1] is at 0xc03ca488 and will execute function located at 0xc01ba2cc
```

Листинг 12 find_syscall успешно определила адрес таблицы системных вызов и адрес самого mkdir

Воспользовавшись функцией `kvm_write`, мы без труда заменим указатель на `mkdir` в таблице системных вызовов или внедрим jump в начало самой `mkdir` (но последний способ — не очень надежен и даже на однопроцессорных машинах может приводить к сбоям, поскольку существует вероятность, что правка функции совпадет с ее вызовом).

Остается решить последний вопрос — куда перенаправлять перехваченный системный вызов? На пользовательское адресное пространство — нельзя, система таких шуток не понимает. Теоретически, можно найти свободное место в ядре (заполненное, например, NOP'ми), записав в него крошечный "бустер", выделяющий немного памяти через `malloc` для размещения основного кода перехватчика, который затягивается внутрь ядра через **`copyin`**, но никакой гарантии, что свободное место найдется у нас нет, поэтому лучше (и надежнее!) размещать перехватчик поверх какого-нибудь редко используемого системного вызова, например, устаревшего, но до сих пор поддерживаемого `lstat`, проходящего под номером 40 или `SYS_ptrace/SYS_ktrace`, "ослепив" кучу утилит, предназначенных для выявления вредоносных программ, что в конечном счете идет только на благо собственной маскировки.

В остальном же, внедрение кода перехватчика внедряется по сценарию, уже описанному мыщъх'ем в ## статье "хак ядра NT" опубликованной в прошлых номерах "Хакера".

заключение

Перехват системных вызовов — прерогатива не только зловредных программ, тем же самым занимаются и средства защиты, активно использующие интерфейс kvm, который поддерживает даже суперзащищенная OpenBSD. И вообще, следует различать действие и его мораль. А мораль такова, что распространенность BSD-систем создает все предпосылки для локальных и удаленных атак с применением всех доступных средств и интерфейсов. Главное — знать как. Все остальное — дело техники и… фантазии. В модификации ядра есть непередаваемое очарование, притягивающее к себе словно магнитом и заставляющее рыскать в поисках скудной документации по всей сети, перечитывать man и конечно же, экспериментировать!

Проблема в том, что код, работающий на одной системе, может оказаться совершенно неработоспособным на другой, поэтому желательно иметь в своем распоряжении хотя бы по одной версии каждой из BSD-систем. Для этой цели хорошо подходят виртуальные машины типа VM Ware. Дисковое пространство давно перестало быть проблемой, а в нормальной конфигурации (то есть без иксов) BSD-системы свободно умещаются в половину гигабайта — смехотворная по нынешним временам величина!

>>> врезка сравнение различных систем

	FreeBSD	NetBSD	OpenBSD
ядро	монолитное с модулями	монолитное с модулями	монолитное (иногда с модулями)
тип модулей	KLD	LKM	LKM
загрузка модуля	kldload	modload	modload
выгрузка модуля	kldunload	modunload	modunload
статистика по модулям	kldstat	modstat	modstat

исходные тексты ядра	требует	не требует	не требует
интерфейс kvm	поддерживается	поддерживается	поддерживается

Таблица 1 основные способы проникновения в ядро в различных BSD-системах

>>> врезка ссылки по теме

- **Introduction to NetBSD loadable kernel modules**:
  - статья, рассказывающая как реализовать простейший модуль символьного устройства, выводящий числа Фибоначчи, под NetBSD (на английском языке): http://home.unix-ag.org/bmeurer/NetBSD/howto-lkm.html;
- **Writing a LKM Driver for NetBSD**:
  - пишем модуль, реализующий драйвер последовательного порта, и просматриваем дерево драйверов под NetBSD (на английском языке): http://etudiant.epita.fr/~jaquem_o/NetBSD_LKM_Driver;
- **OpenBSD Loadable Kernel Modules**:
  - познавательная статья, подробно рассказывающая о LKM-модулях под OpenBSD с кучей различных примеров (на английском языке): http://undeadly.org/cgi?action=article&sid=20010812210650; а вот ее зеркало: http://ezine.daemonnews.org/200109/openbsd-lkm.html;
- **Dynamic Kernel Linker Facility - KLD**:
  - исходный текст простейшего KLD-модуля под FreeBSD с пояснениями: http://docs.mandragor.org/files/Operating_systems/BSDs/FreeBSD_Developers_Handbook/driverbasics-kld.html;
- **Fun and Games with FreeBSD Kernel Modules**:
  - статья, посвященная внедрению в ядро FreeBSD и сокрытию файлов, модулей, процессов и сетевых соединений от администратора (на английском языке): http://www.r4k.net/mod/fbsdfun.html;
- **Attacking FreeBSD with Kernel Modules**:
  - реализуем KLD-модуль, атакующий ядро FreeBSD и перехватывающий системные вызовы с маскировкой от администратора (на английском языке): http://packetstormsecurity.org/papers/unix/bsdkern.htm;
- **Infecting loadable kernel modules**:
  - заражение модулей ядра на Net-, Open- и FreeBSD (на английском языке): www.phrack.org/phrack/61/p61-0x0a_Infecting_Loadable_Kernel_Modules.txt;
- **Playing Games With Kernel Memory… FreeBSD Style**:
  - несмотря на то, что в заголовок статьи вынесена одна лишь FreeBSD, она неявно охватывает и Net-/OpenBSD, поскольку использует интерфейс kvm, демонстрируя технику перехвата системных вызовов (на английском языке): www.phrack.org/phrack/63/p63-0x07_Games_With_Kernel_Memory_FreeBSD_Style.txt;