

<p>一、实验目的及实验内容 (本次实验所涉及并要求掌握的知识; 实验内容; 必要的原理分析)</p> <p>实验目的: 了解 NetFilter 框架, 掌握 Linux 内核模块编程的方法, 熟悉 Netfilter 框架进行数据拦截的方法, 能够利用 Netfilter 框架实现网络数据包的控制。</p> <p>实验要求: 1. 阅读课本第 16-17 章。 2. 掌握 Linux 内核模块的编程方法; 3. 掌握 Netfilter 框架钩子函数设计与实现方法; 4. 能够利用 Netfilter 框架和钩子函数实现以下功能: 禁止 Ping 发送; 禁止某个 IP 地址数据包的接收; 禁止某个端口的数据响应;</p>	<p>小题分:</p>
<p>二、实验环境及实验步骤 (本次实验所使用的器件、仪器设备等的情况; 具体的实验步骤)</p>	<p>小题分:</p>
<p>实验环境:</p> <p>系统版本: Ubuntu18.04, 64 位 内核版本: 4.15.0-106-generic</p> <p>实验步骤:</p> <p>下面将从以下三个大方面进行详细说明;</p> <ol style="list-style-type: none"> 1. 进行 Linux 内核模块的简单编程 (hello world)。 2. 利用 Netfilter 框架和钩子函数完成题目要求的几个功能。 3. 写测试程序测试 2 中写的模块。 <p>1. 进行 Linux 内核模块的简单编程 (hello world)。</p> <p>netfilter 框架程序的编写是在内核层进行的, 在内核层编写程序和应用层编写程序有很大的区别。典型的应用程序有一个 main 程序, 而内核模块则需要一个初始化函数和清理函数, 在向内核中插入模块时调用初始化函数, 卸载内核模块时调用清理函数。Linux 的内核编程通常采用可加载模块的方式, 与直接编进内核相比较, 可加载内核模块有很大的方便性:</p> <ul style="list-style-type: none"> • 不用重新编译内核。 	

- 可以动态加载和卸载，调试使用方便。

下面通过一个“Hello World”例子对内核模块程序设计进行介绍。

1) 编写 c 程序:

```
#include <linux/module.h>
#include <linux/init.h>
/*版权声明*/
MODULE_LICENSE("Dual BSD/GPL");
/*初始化模块*/
static int __init helloworld_init(void)
{
    printk(KERN_ALERT "Hello Kernel!\n");    /*打印信息*/

    return 0;
}
/*清理模块*/
static void __exit helloworld_exit(void)
{
    printk(KERN_ALERT "Goodbye,Kernel!\n");    /*打印信息*/
}
module_init(helloworld_init);    /*模块初始化*/
module_exit(helloworld_exit);    /*模块退出*/
/*作者、软件描述、版本等声明信息*/
MODULE_AUTHOR("Zhang Bowei");    /*作者声明*/
MODULE_DESCRIPTION("Hello World DEMO");    /*描述声明*/
MODULE_VERSION("0.0.1");    /*版本*/
MODULE_ALIAS("Chapter 17, Example 1");    /*模块别名*/
```

这个“Hello World”模块只包含内核模块的加载、卸载函数和简单的授权、作者、描述等信息声明。

2) 编写 makefile:

可以如下编写:

```

ifneq ($(KERNELRELEASE),)
obj-m :=hello.o
else
KDIR:=/lib/modules/$(shell uname -r)/build
all:
    make -C $(KDIR) M=$(shell pwd) modules
clean:
    rm -f *.ko *.o *.mod.o *.mod.c *.symvers
endif

```

- 3) 编译模块，输入命令 `sudo make` 进行编译，并查看编译结果：

```
root@ubuntu:/project# make
make -C /lib/modules/5.5.13/build M=/project modules
make[1]: Entering directory '/usr/linux-5.5.13'
  CC [M]  /project/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /project/hello.o
see include/linux/module.h for more information
  CC [M]  /project/hello.mod.o
  LD [M]  /project/hello.ko
make[1]: Leaving directory '/usr/linux-5.5.13'
root@ubuntu:/project#
```

- 4) 使用命令 `insmod hello.ko` 将安装模块到内核里，并使用指令 `lsmod` 显示当前被载入的模块，结果如下所示。可以看到我们自己所创建的模块已经被成功加载。

```
root@ubuntu:/project# insmod hello.ko
root@ubuntu:/project# lsmod
Module                Size  Used by
hello                 16384  0
vmw_vsock_vmci_transport 28672  1
```

- 5) 使用命令 `rmmmod hello` 将我们创建的 `hello` 模块卸载。此时 使用 `lsmod` 显示当前被加载的模块，可以看到已经没有 `hello` 模块。

```

root@ubuntu:/project# rmmod hello
root@ubuntu:/project# lsmod
Module                  Size  Used by
vmw_vsock_vmci_transport 28672  1
vsock                   45056  3 vmw_vsock_vmci_transport
crc10dif_pclmul         16384  1
crc32_pclmul            16384  0
vmw_balloon             24576  0
ghash_clmulni_intel     16384  0
snd_ens1371             28672  6
snd_ac97_codec          139264  1 snd_ens1371
gameport                16384  1 snd_ens1371
aesni_intel             372736  0
crypto_simd             16384  1 aesni_intel
ac97_bus                 16384  1 snd_ac97_codec
snd_pcm                 114688  3 snd_ac97_codec,snd_ens1371

```

6) 输入命令 `tail /var/log/syslog` 查看系统的日志信息：

```

Apr  1 02:44:05 ubuntu kernel: [ 3610.885447] Hello Kernel!
Apr  1 02:44:10 ubuntu kernel: [ 3615.835225] Goodbye Kernel!

```

可以看到内核输出了信息“Hello Kernel”和“Goodbye, Kernel”，这正是我们在 hello 模块中所定义的。实验结果正确。

2. 利用 Netfilter 框架和钩子函数题目要求的几个功能。

netfilter 中共有 5 个钩子，分别是 PREROUTING、POSTROUTING、INPUT、FORWARD 和 OUTPUT。

netfilter 在设计的时候，考虑到了应用中的各种情况。在 IPv4 的协议栈中，netfilter 在 IP 数据包的路线上仔细选取了 5 个挂接点（HOOK）。这 5 个点中，在合适的位置对 NF_HOOK 宏函数进行了调用，如下图所示，这 5 个点的含义如下所述。

- **NF_IP_PRE_ROUTING**: 刚刚进入网络层而没有进行路由之前的网络数据会通过此点（进行完版本号、校验和等检测）。
- **NF_IP_FORWARD**: 在接收到的网络数据向另一个网卡进行转发之前通过此点。
- **NF_IP_POSTROUTING**: 任何马上要通过网络设备出去的包通过此检测点，这是 netfilter 的最后一个设置检测的点，内置的目的地址转换功能（包括地址伪装）在此点进行。
- **NF_IP_LOCAL_IN**: 在接收到的报文做路由，确定是本机接收的报文之后。
- **NF_IP_LOCAL_OUT**: 在本地报文做发送路由之前。

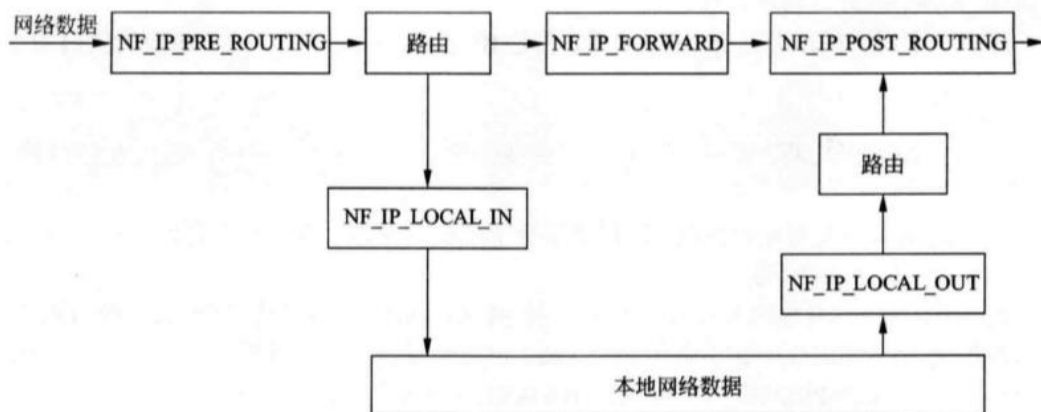


图 17.7 netfilter 的 5 个钩子

当物理网络上有网络数据到来时，iprcvO 函数会接收到。此函数在最后会调用 NFHOOK 将控制权交给在 PREROUTING 点的处理规则处理。如果此处没有挂接钩子函数，则由函数 ip_rcv_finish 来查询路由表，判断此数据是发给本地还是转发给另一个网络。

如果此网络数据是发给本地的，就会调用 ip_local_deliverO 函数。该函数在最后调用宏 NF_HOOK，由 netfilter 的 INPUT 处理规则处理。INPUT 处理完后交给传输层，传给应用层中的用户进程。

如果此数据是转发，会调用 iprev_finish 函数，查询路由表。调用 ip_route_input 函数后将控制权交给 ipforward（）函数。ipforward 函数在最后会调用 NF_HOOK 函数宏，由 netfilter 的 FORWARD 处理规则处理。处理完毕后调用 ipforward finish 函数，由其中的 ip_sendO 函数将数据发送。在发出此数据之前会通过 NF_HOOK 函数宏，由 netfilter 的 POSTROUTING 处理规则处理，处理完毕后，将网络数据转给网络驱动程序，通过网络设备发送到物理网络上。

当本地机器要发送网络数据时，netfilter 会在将数据交给规则 POSTROUTING 处理之前，由处理规则 OUTPUT 先进行处理。

下面详细说明代码部分：

1) 首先引入头文件并定义一些结构体和变量：

```

#include <linux/netfilter_ipv4.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>                /*IP头部结构*/
#include <net/tcp.h>                 /*TCP头部结构*/
#include <linux/if_ether.h>
#include <linux/if_packet.h>
#ifndef __NF_SOCKETPTE_H__
#define __NF_SOCKETPTE_H__
/* cmd命令定义:
SOE_BANDIP: IP地址接收禁止命令
SOE_BANDPORT: 端口禁止命令
SOE_BANDPING: ping禁止
*/
#define SOE_BANDIP      0x6001
#define SOE_BANDPORT    0x6002
#define SOE_BANDPING    0x6003
/* 禁止端口结构*/
typedef struct nf_bandport
{
    /* band protocol, TCP?UDP */
    unsigned short protocol;

    /* band port */
    unsigned short port;
} nf_bandport;
/* 与用户交互的数据结构 */
typedef struct band_status
{
    /* IP发送禁止, IP地址, 当为0时, 未设置 */
    unsigned int band_ip;

    /* 端口禁止, 当协议和端口均为0时, 未设置 */
    nf_bandport band_port;

    /* 是否允许ping回显响应, 为0时响应, 为1时禁止 */
    unsigned char band_ping;
} band_status;
#endif /* __NF_SOCKETPTE_H__ */

```

这里定义了三个命令 SOE_BANDIP, SOE_BANDPORT, SOE_BANDPING。

这三个变量可以在用户层被调用, 从而传入内核层实现特定的功能。

之后的一个结构体 struct band_status 这个结构体变量就是用户层传入内核层的数据。

然后定义一些宏:

```

/* NF初始化状态宏 */
#define NF_SUCCESS 0
#define NF_FAILURE 1
/* 初始化绑定状态 */
band_status b_status;
/*快速绑定操作宏*/
/* 判断是否禁止TCP的端口*/
#define IS_BANDPORT_TCP(status)( status.band_port.port != 0 && status.band_port.protocol == IPPROTO_TCP)
/*判断是否禁止UDP端口 */
#define IS_BANDPORT_UDP(status)( status.band_port.port != 0 && status.band_port.protocol == IPPROTO_UDP)
/* 判断端是否禁止 PING */
#define IS_BANDPING(status)( status.band_ping )
/* 判断是否禁止IP协议 */
#define IS_BANDIP(status)( status.band_ip )

```

同时，定义了一个全局变量 bstatus，用于控制整个程序的禁止状态。

2) 既然是一个模块，那么就需要模块的基本代码信息：

尽管没有强制要求必须声明许可证，但是在进行模块编写的时候最好指定。内核可以识别如下 4 种许可方式：GPL、Dual BSD/GPL、Dual MPL/GPL、Proprietary。没有采用以上许可证方式的声明则假定为私有的，内核加载这种模块会被“污染”。

```

/* 版权声明*/
MODULE_LICENSE("Dual BSD/GPL");

```

然后模块的初始化和退出：

```

/* 初始化模块 */
static int __init init(void)
{
    nf_register_hook(&nfin);           /*注册LOCAL_IN的钩子*/
    nf_register_hook(&nfout);          /*注册LOCAL_OUT的钩子*/
    nf_register_sockopt(&nfsockopt);   /*注册扩展套接字选项*/

    printk(KERN_ALERT "2017301510052-zbw netfilter init successfully\n");
    /*打印信息*/

    return NF_SUCCESS;
}

/* 清理模块 */
static void __exit exit(void)
{
    nf_unregister_hook(&nfin);          /*注销LOCAL_IN的钩子*/
    nf_unregister_hook(&nfout);         /*注销LOCAL_OUT的钩子*/
    nf_unregister_sockopt(&nfsockopt);  /*注销扩展套接字选项*/
    printk(KERN_ALERT "2017301510052-zbw netfilter clean successfully\n");
}

module_init(init);                    /*初始化模块*/
module_exit(exit);                    /*模块退出*/

```

最后对模块的作者、描述、版本声明、别名声明等信息进行设置。

```

/* 作者、描述、版本、别名*/
MODULE_AUTHOR("Zhang Bowei");        /*作者声明*/
MODULE_DESCRIPTION("netfilter DEMO"); /*模块描述信息声明*/
MODULE_VERSION("0.0.1");              /*模块版本声明*/
MODULE_ALIAS("ex17.2");               /*模块别名声明*/

```


3) 定义（初始化）钩子和套接字选项

上面初始化和清理模块用到了 `nf_register_hook` 和 `nf_unregister_hook` 这两个注册和注销钩子函数，以及 `nf_register_sockopt` 和 `nf_unregister_sockopt` 这两个注册和注销扩展套接字选项的函数。

在使用这些函数之前，需要先定义 `nfin` 钩子，`nfout` 钩子，`nf` 套接字选项。先看第一个 `nfin` 钩子：

```
/* 初始化nfin钩子，在钩子LOCAL_IN上 */
static struct nf_hook_ops nfin =
{
    .hook = nf_hook_in,
    .hooknum = NF_INET_LOCAL_IN,
    .pf = PF_INET,
    .priority = NF_IP_PRI_FILTER
};
```

这里用到了 `nf_hook_ops` 结构体，原型和参数如下：

```
struct nf_hook_ops
{
    struct list_head list;           /*钩子链表*/
    nf_hookfn *hook;                 /*钩子处理函数*/
    struct module *owner;            /*模块所有者*/
    int pf;                           /*钩子的协议族*/
    int hooknum;                     /*钩子的位置值*/

    int priority;                     /*钩子的优先级,默认情况下为继承优先级*/
};
```

- ❑ `list`: 结构 `nf_hook_ops` 构成一个链表，`list` 是此链表的表头，把各个处理函数组织成一个表。初始值为 `{NULL, NULL}`。
- ❑ `hook`: 用户自定义的钩子函数指针，它的返回值必须为 `NF_DROP`、`NF_ACCEPT`、`NF_STOLEN`、`NF_QUEUE`、`NF_REPEAT`、`NF_STOP` 之一。
- ❑ `pf`: 协议族，表示这个 `HOOK` 属于哪个协议族；例如对 `ipv4` 而言，设定为 `PF_INET`。
- ❑ `hooknum`: 用户想注册的钩子位置，取值为 5 个钩子（`NF_INET_PRE_ROUTING`、`NF_INET_LOCAL_IN`、`NF_INET_FORWARD`、`NF_INET_LOCAL_OUT`、`NF_INET_POST_ROUTING`、`NF_INET_NUMHOOKS`）之一。一个挂接点可以挂接多个钩子函数，谁先被调用要看优先级。
- ❑ `priority`: 优先级，目前 `netfilter` 在 `IPv4` 中定义了多个优先级，取值越小优先级越高。

调用的是 `nf_hook_in` 函数，下面会讲。

同样的，定义 `nfout` 钩子：


```

/*初始化nfout钩子, 在钩子LOCAL_OUT上*/
static struct nf_hook_ops nfout=
{
    .hook = nf_hook_out,
    .hooknum = NF_INET_LOCAL_OUT,
    .pf = PF_INET,
    .priority = NF_IP_PRI_FILTER
};

```

这个和 nfin 钩子类似，只不过调用的是 nf_hook_out 函数。

最后定义 nf 套接字选项：

```

/* 初始化nf套接字选项 */
static struct nf_sockopt_ops nfsockopt = {
    .pf = PF_INET,
    .set_optmin = SOE_BANDIP,
    .set_optmax = SOE_BANDIP+3,
    .set = nf_sockopt_set,
    .get_optmin = SOE_BANDIP,
    .get_optmax = SOE_BANDIP+3,
    .get = nf_sockopt_get,
};

```

这里使用了 nf_sockopt_ops 结构体，原型和参数如下：

```

struct nf_sockopt_ops
{
    struct list_head list;
    int pf;
    int set_optmin;
    int set_optmax;
    int (*set)(struct sock *sk, int optval, void __user *user, unsigned int len);
    int (*compat_set)(struct sock *sk, int optval, void __user *user, unsigned int len);
    int get_optmin;
    int get_optmax;
    int (*get)(struct sock *sk, int optval, void __user *user, int *len);
    int (*compat_get)(struct sock *sk, int optval, void __user *user, int *len);
    /* Use the module struct to lock set/get code in place */
    struct module *owner;
};

```

- ❑ list: 链表的头指针。
- ❑ pf: 协议族选项。
- ❑ set_optmin: 设置 sockopt 命令匹配范围的最小值。
- ❑ set_optmax: 设置 sockopt 命令匹配范围的最大值。
- ❑ set: 设置函数实现，此函数对应用层的 setsockopt() 函数进行响应，按照应用层传入的数据进行设置。
- ❑ get_optmin: 获取 sockopt 命令匹配范围的最小值。

- ❑ `get_optmax`: 获取 `sockopt` 命令匹配范围的最大值。
- ❑ `get`: 获取函数实现，此函数对应用层的 `getsockopt()` 函数进行响应，将当前内核中的设置返回给应用层。

之前学过了套接字选项，它可以设置套接字的超时时间、接收缓冲区大小等来控制 socket。在 netfilter 中，`nf_register_sockopt` 和 `nf_unregister_sockopt` 函数是在 socket 的选项控制上挂接钩子函数，使得用户可以注册自己的 `opt` 函数，处理特殊的 socket 控制。

`sockopt` 函数注册函数将按照用户的指定实现特定的 `sockopt` 命令响应函数。当要注册 `set` 函数时，根据 `set_optmin` 和 `setoptmax` 来判断某个 `sockopt` 调用是否由本函数响应；当为 `get` 函数时，则判断 `get_optmin` 和 `getoptmax` 范围。

这里 `set` 和 `get` 分别调用的函数是 `nf_sockopt_set` 和 `nf_sockopt_get` 函数。

4) 注册/注销 钩子和套接字选项

`nf_register_sockopt` 和 `nf_unregister_sockopt` 这两个注册和注销扩展套接字选项的函数可以直接用。

但是由于版本问题，`nf_register_hook` 和 `nf_unregister_hook` 这两个注册和注销钩子函数在高版本中就没有了，或者说换函数了。所以需要自己写这两个函数：

```

/*注册钩子函数，高版本没有所以自己写*/
static int nf_register_hook(struct nf_hook_ops *reg)
{
    struct net *net, *last;
    int ret;

    rtnl_lock();
    for_each_net(net)
    {
        ret = nf_register_net_hook(net, reg);
        if(ret && ret != -ENOENT)
            goto rollback;
    }
    rtnl_unlock();
    return 0;

    rollback:
    last = net;
    for_each_net(net)
    {
        if(net == last)
            break;
        nf_unregister_net_hook(net, reg);
    }
    rtnl_unlock();
    return ret;
}

```

高版本中变为了 nf_register_net_hook 函数，并且参数还不一样。

以及注销钩子函数如下：

```

/*取消注册钩子函数，高版本没有所以自己写*/
static void nf_unregister_hook(struct nf_hook_ops *reg)
{
    struct net *net;

    rtnl_lock();
    for_each_net(net)
    {
        nf_unregister_net_hook(net, reg);
    }
    rtnl_unlock();
}

```

高版本中变为了 nf_unregister_net_hook 函数，并且参数还不一样。

5) 挂在 LOCAL_IN 的钩子调用如下函数：

总函数代码如下：

```

/* 在LOCAL_IN挂接钩子 */
static unsigned int nf_hook_in(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
    struct iphdr *iph = ip_hdr(skb);
    unsigned int src_ip = iph->saddr;
    unsigned int des_ip = iph->daddr;
    struct tcphdr *tcph = NULL;
    struct udphdr *udph = NULL;

    if(IS_BANDIP(b_status))
    {
        if(b_status.band_ip == iph->saddr)
        {
            printk(KERN_ALERT "DROP one packet from %d.%d.%d.%d to this host\n" ,
                    (src_ip&0xff000000)>>0,
                    (src_ip&0xff0000ff)>>8,
                    (src_ip&0x00ff0000)>>16,
                    (src_ip&0xff000000)>>24);

            return NF_DROP;
        }
    }

    switch(iph->protocol)
    {
        /*IP协议类型*/
        case IPPROTO_TCP:
            /*TCP协议*/
            /*丢弃禁止端口的TCP数据*/
            if(IS_BANDPORT_TCP(b_status))
            {
                tcph = tcp_hdr(skb);
                /*获得TCP头*/
                if(tcph->dest == b_status.band_port.port) /*端口匹配*/
                {
                    printk(KERN_ALERT "drop one tcp packet from %d.%d.%d.%d to the port %d\n" ,
                            (src_ip&0xff000000)>>0,
                            (src_ip&0xff0000ff)>>8,
                            (src_ip&0x00ff0000)>>16,
                            (src_ip&0xff000000)>>24,
                            ntohs(tcph->dest));

                    return NF_DROP; /*丢弃该数据*/
                }
            }
            break;

        case IPPROTO_UDP:
            /*UDP协议*/
            /*丢弃UDP数据*/
            if(IS_BANDPORT_UDP(b_status)) /*设置了丢弃UDP协议*/
            {
                udph = udp_hdr(skb);
                /*UDP头部*/
                if(udph->dest == b_status.band_port.port) /*UDP端口判定*/
                {
                    printk(KERN_ALERT "drop one udp packet from %d.%d.%d.%d to the port %d\n" ,
                            (src_ip&0xff000000)>>0,
                            (src_ip&0xff0000ff)>>8,
                            (src_ip&0x00ff0000)>>16,
                            (src_ip&0xff000000)>>24,
                            ntohs(udph->dest));

                    return NF_DROP; /*丢弃该数据*/
                }
            }
            break;
        default:
            break;
    }

    return NF_ACCEPT;
}

```

首先实现第一个功能：

禁止某个 IP 地址数据包的接收；

在这个钩子上，需要判断本地接收的数据包的原发送地址是否为已经禁止主机的目的地址。如果数据包的原发送地址是已经禁止的主机 IP 地址，则将这个包抛弃。并在系统日志中打印如下信息：

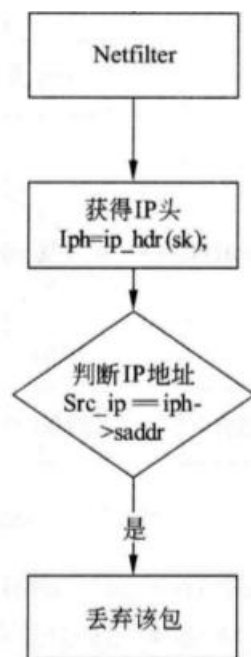
```

if(IS_BANDIP(b_status))
{
    if(b_status.band_ip == iph->saddr)
    {
        printk(KERN_ALERT "DROP one packet from %d.%d.%d.%d to this host\n",
                (src_ip&0xff000000)>>0,
                (src_ip&0xff0000ff)>>8,
                (src_ip&0x00ff0000)>>16,
                (src_ip&0x000000ff)>>24);

        return NF_DROP;
    }
}

```

也可以用下面这个流程图来表示：



接下来实现第二个功能：

禁止某个端口的数据响应；

钩子 LOCAL_IN 用于过滤发往本机的数据包。对于这些数据包，要根据不同的协议进行处理。

口如果为 TCP 协议的数据包，将目的 IP 地址端口变量 `tcph->dest` 与禁止端口变量 `bstatus.band port.port` 进行比较，如果相同，则把这个数据包丢弃。

口如果为 UDP 协议的数据包，与 TCP 协议数据包的操作相同，将目的 IP 地址端口变量 `tcph->dest` 与禁止端口变量 `b_status.band_port.port` 进行比较，如果相同则把这个数据包丢弃。

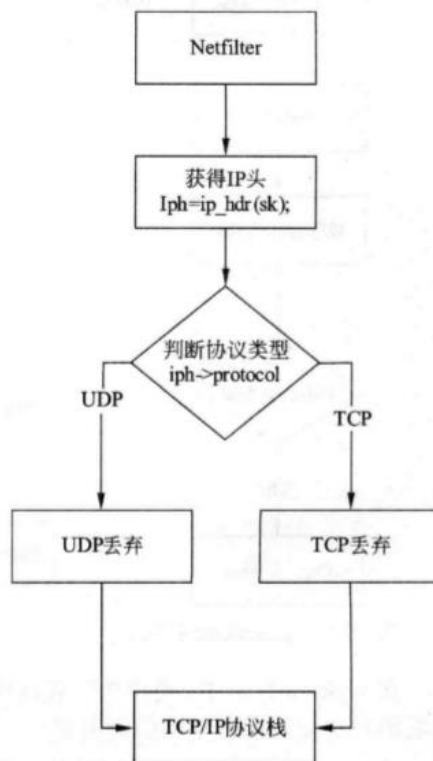
```

switch(iph->protocol)                                /*IP协议类型*/
{
    case IPPROTO_TCP:                                /*TCP协议*/
        /*丢弃禁止端口的TCP数据*/
        if(IS_BANDPORT_TCP(b_status))
        {
            tcp_h = tcp_hdr(skb);                    /*获得TCP头*/
            if(tcp_h->dest == b_status.band_port.port) /*端口匹配*/
            {
                printk(KERN_ALERT "drop one tcp packet from %d.%d.%d.%d to the port %d\n" ,
                    (src_ip&0xff000000)>>0,
                    (src_ip&0xff0000ff)>>8,
                    (src_ip&0x00ff0000)>>16,
                    (src_ip&0xff000000)>>24,
                    ntohs(tcp_h->dest));
                return NF_DROP;                        /*丢弃该数据*/
            }
        }
        break;
    case IPPROTO_UDP:                                /*UDP协议*/
        /*丢弃UDP数据*/
        if(IS_BANDPORT_UDP(b_status))                 /*设置了丢弃UDP协议*/
        {
            udph = udp_hdr(skb);                      /*UDP头部*/
            if(udph->dest == b_status.band_port.port) /*UDP端口判定*/
            {
                printk(KERN_ALERT "drop one udp packet from %d.%d.%d.%d to the port %d\n" ,
                    (src_ip&0xff000000)>>0,
                    (src_ip&0xff0000ff)>>8,
                    (src_ip&0x00ff0000)>>16,
                    (src_ip&0xff000000)>>24,
                    ntohs(udph->dest));
                return NF_DROP;                        /*丢弃该数据*/
            }
        }
        break;
    default:
        break;
}

```

并在系统日志中打印上述信息。

也可以用下面这个流程图来表示:



6) 挂在 LOCAL_OUT 的钩子调用如下函数：

总代码如下：

```
/* 在LOCAL_OUT上挂接钩子 */
static unsigned int nf_hook_out(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
    struct iphdr *iph = ip_hdr(skb);
    unsigned int src_ip = iph->saddr;
    unsigned int des_ip = iph->daddr;

    switch (iph->protocol)
    {
        case IPPROTO_ICMP:
            if (IS_BANDPING(b_status))
            {
                printk(KERN_ALERT "DROP one ICMP packet from %d.%d.%d.%d to %d.%d.%d.%d\n" ,
                    (src_ip&0xff000000)>>0,
                    (src_ip&0xff0000ff)>>8,
                    (src_ip&0x00ff0000)>>16,
                    (src_ip&0xff000000)>>24,
                    (des_ip&0xff000000)>>0,
                    (des_ip&0xff0000ff)>>8,
                    (des_ip&0x00ff0000)>>16,
                    (des_ip&0x00ff000000)>>24);

                return NF_DROP;
            }
            break;
        default:
            break;
    }

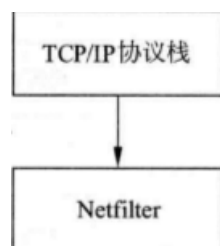
    return NF_ACCEPT;
}
```

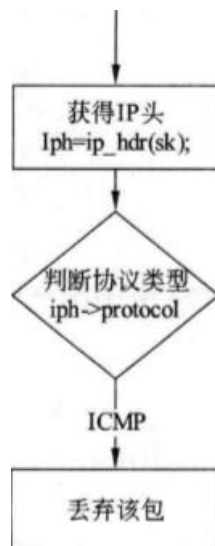
这个函数实现第三个功能：

禁止 Ping 发送；

钩子 LOCALOUT 对从本地发出的数据包进行过滤。在这个钩子上，判断发送出去的数据包类型是不是 ICMP 类型的，如果是，那么就抛弃掉。因为 ping 程序发送的包就是 ICMP 类型的包，这样做就是禁止了 ping 发送。

也可以用下面这个流程图来表示：





7) 动态配置实现

也就是写 `nf_sockopt_set` 和 `nf_sockopt_get` 函数。

动态配置的实现采用了注册私有 `sockopt` 的方法，使用 API 函数 `nf_register_sockopt0` 在 `IPRAW` 层注册一个私有的 `sockopt` 处理钩子函数，利用其中的回调函数 `set` 和 `get` 来实现与用户层的交互。

在 `getsockopt` 的扩展中，本例中的实现在判断 `cmd` 命令合法后，直接将数据复制到用户空间，没有进行 `cmd` 字的匹配。

在 `setsockopt` 的扩展中，先将用户输入的参数复制到内核空间，然后进行相应的设置。

如果为 IP 地址禁止发送的命令，则设置其对应参数；如果为端口禁止的命令，则查看相应的协议，根据 `UDP` 和 `TCP` 的不同来设置协议类型和端口；如果为 `PING` 回显禁止就设置此布尔类型变量。`sockopt` 扩展的流程图如图：

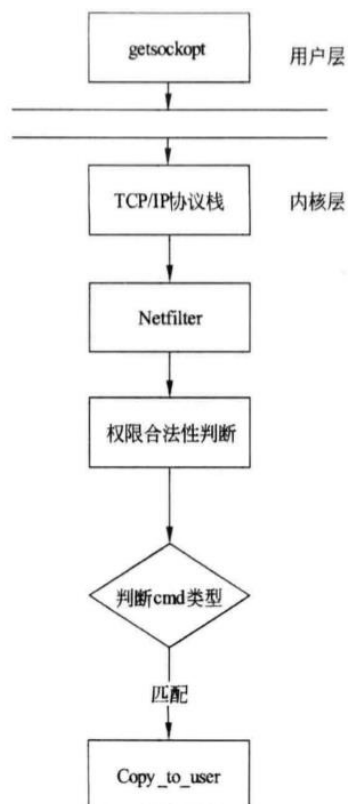


图 17.11 getsockopt 扩展

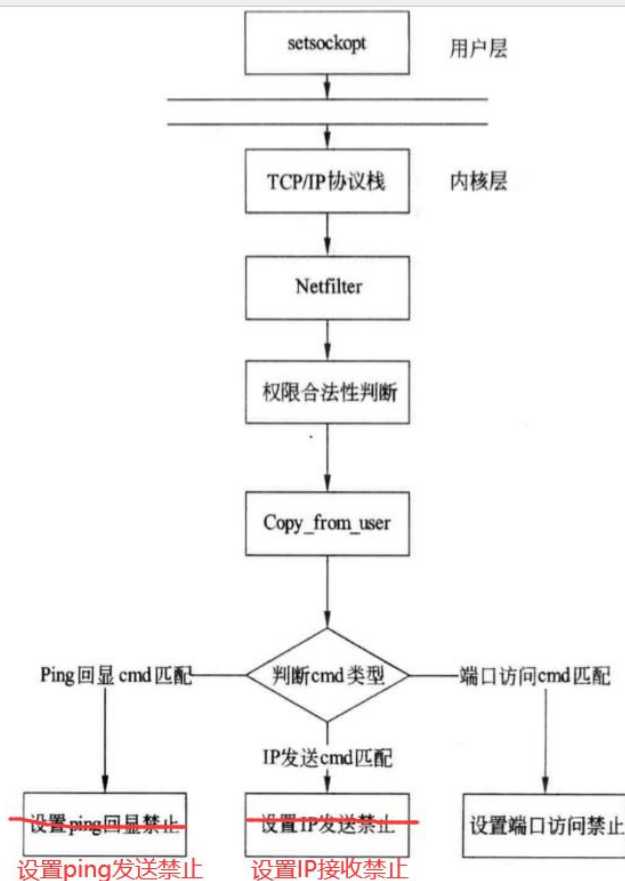


图 17.12 setsockopt 扩展

扩展的 sockopt 设置函数 nf_sockopt_set。

这个函数先检查是否用户有权限使用扩展命令，通常只有 root 用户才有操作此命令的权限。然后调用 copy_from_user () 函数将用户空间的数据复制到内核空间中，判断用户输入的命令选项类型进行进一步的处理：

- 当命令为 SOE_BANDIP 时，表示设置禁止主机的 IP。这时，将控制变量 b_status.band ip 设置为用户输入的 IP 地址。
- 当命令为 SOE_BANDPORT 时，表示禁止端口。根据用户的协议不同，将控制变量 b_status.band_port.protocol 分别设置为 IPPROTO_TCP 和 IPPROTO_UDP;并同时 will 控制变量 b_status.band port.port 设置为用户输入的端口值。
- 当命令为 SOE_BANDPING 时，表示禁止 ping 操作。这时，将控制变量 b_status.band ping 设置为 1，表示禁止 ping 操作。

代码如下：

```

/* nf_sock 选项扩展操作*/
static int
nf_sockopt_set(struct sock *sock,
               int cmd,
               void __user *user,
               unsigned int len)
{
    int ret = 0;
    struct band_status status;

    /* 权限检查 */
    if(!capable(CAP_NET_ADMIN))           /*没有足够权限*/
    {
        ret = -EPERM;
        goto ERROR;
    }
    /* 从用户空间复制数据*/
    ret = copy_from_user(&status, user, len);
    if(ret != 0)                          /*复制数据失败*/
    {
        ret = -EINVAL;
        goto ERROR;
    }

    /* 命令类型 */
    switch(cmd)
    {
        case SOE_BANDIP:                  /*禁止IP协议*/
            /* 设置禁止IP协议 */
            if(IS_BANDIP(status))          /*设置禁止IP协议*/
                b_status.band_ip = status.band_ip;
            else                            /*取消禁止*/
                b_status.band_ip = 0;

            break;
    }
}

```

```

case SOE_BANDPORT:                                /*禁止端口*/
/* 设置端口禁止和相关的协议类型 */
if(IS_BANDPORT_TCP(status))                        /*禁止TCP*/
{
    b_status.band_port.protocol = IPPROTO_TCP;
    b_status.band_port.port = status.band_port.port;
}
else if(IS_BANDPORT_UDP(status))                  /*禁止UDP*/
{
    b_status.band_port.protocol = IPPROTO_UDP;
    b_status.band_port.port = status.band_port.port;
}
else                                              /*其他*/
{
    b_status.band_port.protocol = 0;
    b_status.band_port.port = 0;
}

break;
case SOE_BANDPING:                                /*禁止ping*/
if(IS_BANDPING(status))                          /*禁止PING*/
{
    b_status.band_ping = 1;
}
else                                              /*取消禁止*/
{
    b_status.band_ping = 0;
}

break;
default:                                          /*其他为错误命令*/
ret = -EINVAL;
break;
}

ERROR:
return ret;
}

```

扩展获取 sockopt 选项的函数 `nf_sockopt_get`。

这个函数根据用户输入的命令是否为 `SOE_BANDIP`、`SOE_BANDPORT` 或者 `SOE_BANDPING`，将变量 `bstatus` 的值复制给用户。

代码如下：

```

/* nf sock 操作扩展命令操作*/
static int
nf_sockopt_get(struct sock *sock,
               int cmd,
               void __user *user,
               unsigned int len)
{
    int ret = 0;

    /* 权限检查*/
    if(!capable(CAP_NET_ADMIN))           /*没有权限*/
    {
        ret = -EPERM;
        goto ERROR;
    }

    /* 将数据从内核空间复制到用户空间 */
    switch(cmd)
    {
        case SOE_BANDIP:
        case SOE_BANDPORT:
        case SOE_BANDPING:
            /*复制数据*/
            ret = copy_to_user(user, &b_status, len);
            if(ret != 0)                       /*复制数据失败*/
            {
                ret = -EINVAL;
                goto ERROR;
            }
            break;
        default:
            ret = -EINVAL;
            break;
    }

ERROR:
    return ret;
}

```

实际上在下面测试中只用到了 set 函数，而没有用到这个 get 函数，因为只需要向内核中传参数即可，不需要读取内核中的参数。

8) makefile 如下:

```

target := netfilter
obj-m := $(target).o
KERNELDIR = /lib/modules/$(shell uname -r)/build

.PHONY: all install uninstall clean

all:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

install:
    insmod $(target).ko

uninstall:
    rmmod $(target).ko

clean:
    rm -rf *.o *.mod *.cmd *.mod.c *.ko *.order
    rm -rf Module.symvers *.cmd .tmp_versions

test:test.c
    gcc -o $@ $^

```

3. 写测试程序测试 2 写的模块。

首先说明一下，我的虚拟机的 ip 为 192.168.222.129:

```
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$ ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    ether 02:42:94:62:61:4e txqueuelen 0 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.222.129 netmask 255.255.255.0 broadcast 192.168.222.255
    inet6 fe80::c56b:3ce6:a715:7efa prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:16:4b:a4 txqueuelen 1000 (Ethernet)
    RX packets 24776 bytes 17009340 (17.0 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 9156 bytes 2291138 (2.2 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 1139 bytes 134498 (134.4 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1139 bytes 134498 (134.4 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

我主机的 ip 为 192.168.222.1:

以太网适配器 VMware Network Adapter VMnet8:

```
连接特定的 DNS 后缀 . . . . . :
本地链接 IPv6 地址. . . . . : fe80::7477:e634:e264:a5db%14
IPv4 地址 . . . . . : 192.168.222.1
子网掩码 . . . . . : 255.255.255.0
默认网关. . . . . :
```

1) 禁止某个 IP 地址数据包的接收

首先还是引入头文件，并且定义这个结构体：

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>

#define SOE_BANDIP 0X6001
#define SOE_BANDPORT 0X6002
#define SOE_BANDPING 0X6003

#define NF_SUCCESS 0
#define NF_FAILURE 1

struct nf_bandport
{
    unsigned short protocol;

    unsigned short port;
};

struct band_status
{
    unsigned int band_ip;

    struct nf_bandport band_port;

    unsigned char band_ping;
};

```

之后进入主函数：

```

int main()
{
    struct band_status b_status;
    b_status.band_ip = inet_addr("192.168.222.1");
    socklen_t len = sizeof(b_status);

    int fd = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
    if(fd == -1)
    {
        perror("create socket error\n");
        exit(-1);
    }

    if(setsockopt(fd, IPPROTO_IP, SOE_BANDIP, &b_status, len) == -1)
    {
        perror("set sock opt error\n");
        printf("errno: %d\n", errno);
        exit(-1);
    }

    return 0;
}

```

最重要的就是红框内，这个 b_status 就是要传入到内核中的数据，把禁止的 ip 设为我主机的 ip，其他的用户空间的操作很简单，就是用 socket 打开

相关协议类型的 socket，直接调用 set/getsockopt() 函数就可以进行操作了。在第 12 章中有详细的介绍，读者可以参考自己来编写，要注意建立 socket 时使用 RAW 类型。并且注意选项为 SOE_BANDIP。

2) 允许某个 IP 地址数据包的接收

代码和第一个基本相同，唯一不同的如下：

```
struct band_status b_status;  
b_status.band_ip = 0;
```

设置为 0，就说明不禁止。

3) 禁止 ping 发送

代码和第一个基本相同，唯两不同的如下：

```
struct band_status b_status;  
b_status.band_ping = 1;
```

这里值设置为 1，说明要禁止 ping。

同时：

```
if(setsockopt(fd, IPPROTO_IP, SOE_BANDPING, &b_status, len) == -1)  
{  
    perror("set sock opt error\n");  
    printf("errno: %d\n", errno);  
    exit(-1);  
}
```

这个选项需要改一下。

4) 允许 ping 发送

代码和上面一个基本相同，唯一不同的如下：

```
struct band_status b_status;  
b_status.band_ping = 0;
```

这里值设置为 0，说明允许 ping。

5) 禁止某个端口的数据响应

代码和第一个基本相同，唯两不同的如下：

```

struct band_status b_status;
b_status.band_port.port = htons(80);
b_status.band_port.protocol = IPPROTO_TCP;

```

这里要设置端口号和协议。另外：

```

if(setsockopt(fd, IPPROTO_IP, SOE_BANDPORT, &b_status, len) == -1)
{
    perror("set sock opt error\n");
    printf("errno: %d\n", errno);
    exit(-1);
}

```

这里也要改一下。

6) 允许某个端口的数据响应

代码和上面一个基本相同，唯一不同的如下：

```

struct band_status b_status;
b_status.band_port.port = 0;
b_status.band_port.protocol = 0;

```

端口和协议都为 0，说明不禁止某个端口的数据响应

三、实验过程分析

（详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格、绘制曲线、波形等）

小题分：

1. gcc 编译器问题，写好代码后，直接 make 时会报以下错误：

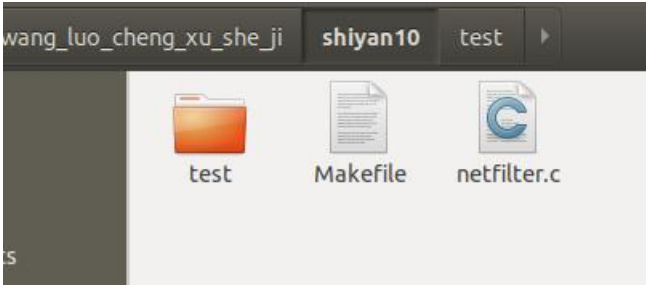


```

dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10/1
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/1$ make
make -C /lib/modules/4.15.0-106-generic/build M=/home/dc/wang_luo_cheng_xu_she_ji/shiyan10/1 modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-106-generic'
CC [M] /home/dc/wang_luo_cheng_xu_she_ji/shiyan10/1/netfilter.o
gcc: error: unrecognized command line option '-fstack-protector-strong'
scripts/makefile.build:337: recipe for target '/home/dc/wang_luo_cheng_xu_she_ji/shiyan10/1/netfilter.o' failed
make[2]: *** [/home/dc/wang_luo_cheng_xu_she_ji/shiyan10/1/netfilter.o] Error 1
Makefile:1577: recipe for target '_module_/home/dc/wang_luo_cheng_xu_she_ji/shiyan10/1' failed
make[1]: *** [_module_/home/dc/wang_luo_cheng_xu_she_ji/shiyan10/1] Error 2
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-106-generic'
Makefile:8: recipe for target 'all' failed
make: *** [all] Error 2

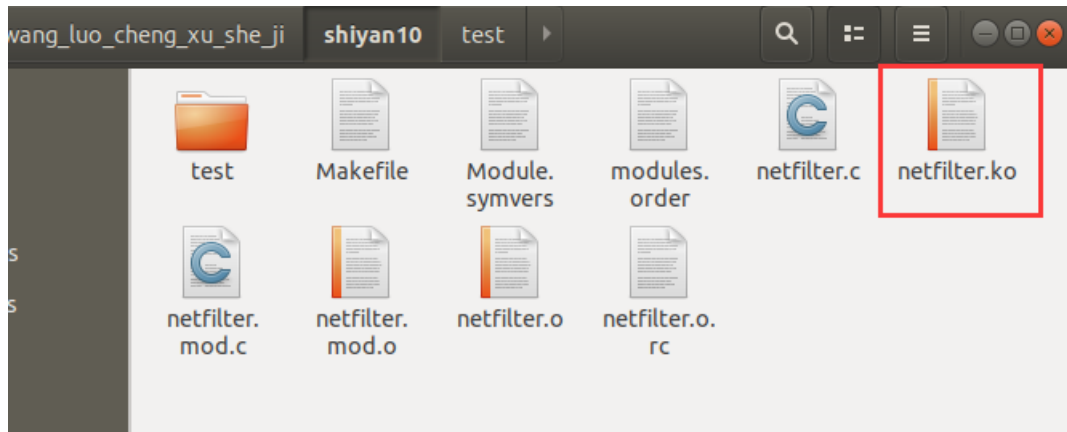
```

解决如下：

<p>进入到/usr/src/linux-headers-4.15.0-106-generic 目录下，修改 Makefile 文件：</p> <pre># This selects the stack protector compiler flag. Testing it is delayed # until after .config has been reprocessed, in the prepare-compiler-check # target. ifdef CONFIG_CC_STACKPROTECTOR_REGULAR stackp-flag := -fstack-protector stackp-name := REGULAR else ifdef CONFIG_CC_STACKPROTECTOR_STRONG # stackp-flag := -fstack-protector-strong # stackp-name := STRONG endif # Force off for distro compilers that enable stack protector by default. stackp-flag := \$(call cc-option, -fno-stack-protector) endif endif # Find arch-specific stack protector compiler sanity-checking script. ifdef CONFIG_CC_STACKPROTECTOR stackp-path := \$(src tree)/scripts/gcc-\$(SRCARCH)_\$(BITS)-has-stack-protector.sh stackp-check := \$(wildcard \$(stackp-path)) endif KBUILD_CFLAGS += \$(stackp-flag)</pre> <p>在这 5 句话之前添加注释，即可解决此问题。</p> <p>2. 在写 ip 禁止程序时，需要使用 inet_addr 函数把 ip 字符串转为整数。</p> <pre>struct band_status b_status; b_status.band_ip = inet_addr("192.168.222.1");</pre> <p>3. 在写端口禁止程序时，需要使用 htons 把数字转为网络字节序。</p> <pre>struct band_status b_status; b_status.band_port.port = htons(80); b_status.band_port.protocol = IPPROTO_TCP;</pre>	
四、实验结果总结 (对实验结果进行分析，完成思考题目，总结实验的新的体会，并提出实验的改进意见)	小题分：
<p>实验结果：</p> <p>首先在这个目录下使用 make:</p> 	

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ make
```

执行之后会生成一个驱动模块：



然后加载进来：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ sudo insmod netfilter.ko
[sudo] password for dc:
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$
```

可以使用 lsmod 查看当前模块：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ lsmod
```

Module	Size	Used by
netfilter	16384	0
btrfs	1138688	0
zstd_compress	163840	1 btrfs
xor	24576	1 btrfs
raid6_pq	114688	1 btrfs
ufs	77824	0
qnx4	16384	0
hfsplus	106496	0
hfs	57344	0

使用 dmesg 查看系统日志：

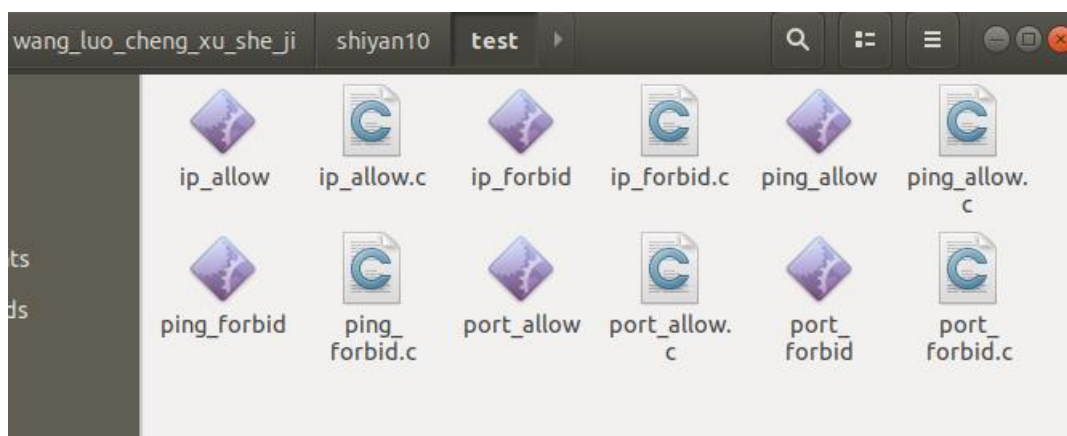
```
dc@ubuntu: ~
File Edit View Search Terminal Help
dc@ubuntu:~$ dmesg
```

```
[23663.962561] 2017301510052-zbw netfilter init successfully
dc@ubuntu:~$
```

可以看到成功加载进来。

下面依次测试题目几个功能：

首先依次把代码编译成可执行程序：



1) 禁止某个 IP 地址数据包的接收

这里设置的是禁止接收主机（192.168.222.1）发送的数据。

首先没操作之前，主机是可以 Ping 虚拟机（192.168.222.129）的：

```
C:\Users\12943>ping 192.168.222.129

正在 Ping 192.168.222.129 具有 32 字节的数据:
来自 192.168.222.129 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.222.129 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.222.129 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.222.129 的回复: 字节=32 时间<1ms TTL=64

192.168.222.129 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 0ms, 平均 = 0ms
```

然后执行 ip_forbid 程序：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyang10/test
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyang10/test$ sudo ./ip_forbid
[sudo] password for dc:
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyang10/test$
```

这时再向虚拟机发送数据包：

```
C:\Users\12943>ping 192.168.222.129

正在 Ping 192.168.222.129 具有 32 字节的数据:
请求超时。
请求超时。

192.168.222.129 的 Ping 统计信息:
    数据包: 已发送 = 2, 已接收 = 0, 丢失 = 2 (100% 丢失),
Control-C
^C
C:\Users\12943>
```

发现 ping 不通。

同时查看系统日志：

```
[17372.188945] 2017301510052-zbw netfilter init successfully
[23663.962561] 2017301510052-zbw netfilter init successfully
[24069.763803] DROP one packet from 192.168.222.1 to this host
[24074.440465] DROP one packet from 192.168.222.1 to this host
dc@ubuntu:~$
```

可以看到已经禁止接收主机 ip 的数据包。

2) 允许某个 IP 地址数据包的接收

执行 ip_allow 程序：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10/test
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$ sudo ./ip_forbid
[sudo] password for dc:
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$ sudo ./ip_allow
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$
```

这时再向虚拟机发送数据包：

```
C:\Users\12943>ping 192.168.222.129

正在 Ping 192.168.222.129 具有 32 字节的数据:
来自 192.168.222.129 的回复: 字节=32 时间=1ms TTL=64
来自 192.168.222.129 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.222.129 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.222.129 的回复: 字节=32 时间<1ms TTL=64

192.168.222.129 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 1ms, 平均 = 0ms
```

虚拟机又可以接收主机发送的数据包了。

3) 禁止 ping 发送

首先没操作之前，虚拟机先 ping 主机试试：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ ping 192.168.222.1
PING 192.168.222.1 (192.168.222.1) 56(84) bytes of data.
64 bytes from 192.168.222.1: icmp_seq=1 ttl=128 time=0.528 ms
64 bytes from 192.168.222.1: icmp_seq=2 ttl=128 time=0.660 ms
64 bytes from 192.168.222.1: icmp_seq=3 ttl=128 time=0.623 ms
64 bytes from 192.168.222.1: icmp_seq=4 ttl=128 time=0.532 ms
64 bytes from 192.168.222.1: icmp_seq=5 ttl=128 time=0.514 ms
64 bytes from 192.168.222.1: icmp_seq=6 ttl=128 time=0.531 ms
64 bytes from 192.168.222.1: icmp_seq=7 ttl=128 time=0.684 ms
^C
--- 192.168.222.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6147ms
rtt min/avg/max/mdev = 0.514/0.581/0.684/0.072 ms
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$
```

可以 ping 通。
现在执行 ping_forbid 程序：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10/test
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$ sudo ./ping_forbid
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$
```

再次 ping 主机试试：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ ping 192.168.222.1
PING 192.168.222.1 (192.168.222.1) 56(84) bytes of data.
64 bytes from 192.168.222.1: icmp_seq=1 ttl=128 time=0.528 ms
64 bytes from 192.168.222.1: icmp_seq=2 ttl=128 time=0.660 ms
64 bytes from 192.168.222.1: icmp_seq=3 ttl=128 time=0.623 ms
64 bytes from 192.168.222.1: icmp_seq=4 ttl=128 time=0.532 ms
64 bytes from 192.168.222.1: icmp_seq=5 ttl=128 time=0.514 ms
64 bytes from 192.168.222.1: icmp_seq=6 ttl=128 time=0.531 ms
64 bytes from 192.168.222.1: icmp_seq=7 ttl=128 time=0.684 ms
^C
--- 192.168.222.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6147ms
rtt min/avg/max/mdev = 0.514/0.581/0.684/0.072 ms
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ ping 192.168.222.1
PING 192.168.222.1 (192.168.222.1) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
^C
--- 192.168.222.1 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3070ms

dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$
```


发现 ping 被禁止了。

同时查看系统日志：

```
[23663.962561] 2017301510052-zbw netfilter init successfully
[24069.763803] DROP one packet from 192.168.222.1 to this host
[24074.440465] DROP one packet from 192.168.222.1 to this host
[24193.869116] DROP one packet from 192.168.222.1 to this host
[24477.618228] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1
[24478.641128] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1
[24479.664881] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1
[24480.688687] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1
dc@ubuntu:~$
```

可以看到发向主机的 ping 数据被丢掉了。

4) 允许 ping 发送

执行 ping_allow 函数：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10/test
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$ sudo ./ping_forbid
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$ sudo ./ping_allow
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$
```

再次 ping 主机试试：

```

dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ ping 192.168.222.1
PING 192.168.222.1 (192.168.222.1) 56(84) bytes of data.
64 bytes from 192.168.222.1: icmp_seq=1 ttl=128 time=0.528 ms
64 bytes from 192.168.222.1: icmp_seq=2 ttl=128 time=0.660 ms
64 bytes from 192.168.222.1: icmp_seq=3 ttl=128 time=0.623 ms
64 bytes from 192.168.222.1: icmp_seq=4 ttl=128 time=0.532 ms
64 bytes from 192.168.222.1: icmp_seq=5 ttl=128 time=0.514 ms
64 bytes from 192.168.222.1: icmp_seq=6 ttl=128 time=0.531 ms
64 bytes from 192.168.222.1: icmp_seq=7 ttl=128 time=0.684 ms
^C
--- 192.168.222.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6147ms
rtt min/avg/max/mdev = 0.514/0.581/0.684/0.072 ms
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ ping 192.168.222.1
PING 192.168.222.1 (192.168.222.1) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
^C
--- 192.168.222.1 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3070ms

dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ ping 192.168.222.1
PING 192.168.222.1 (192.168.222.1) 56(84) bytes of data.
64 bytes from 192.168.222.1: icmp_seq=1 ttl=128 time=1.00 ms
64 bytes from 192.168.222.1: icmp_seq=2 ttl=128 time=1.66 ms
64 bytes from 192.168.222.1: icmp_seq=3 ttl=128 time=2.16 ms
64 bytes from 192.168.222.1: icmp_seq=4 ttl=128 time=0.646 ms
64 bytes from 192.168.222.1: icmp_seq=5 ttl=128 time=0.446 ms
64 bytes from 192.168.222.1: icmp_seq=6 ttl=128 time=0.538 ms
64 bytes from 192.168.222.1: icmp_seq=7 ttl=128 time=0.544 ms
^C
--- 192.168.222.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6091ms
rtt min/avg/max/mdev = 0.446/1.001/2.166/0.615 ms
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$

```

又可以 ping 通了。

5) 禁止某个端口的数据响应

这个测试的是 80 端口，可以先安装 apache，查看当前开放端口：

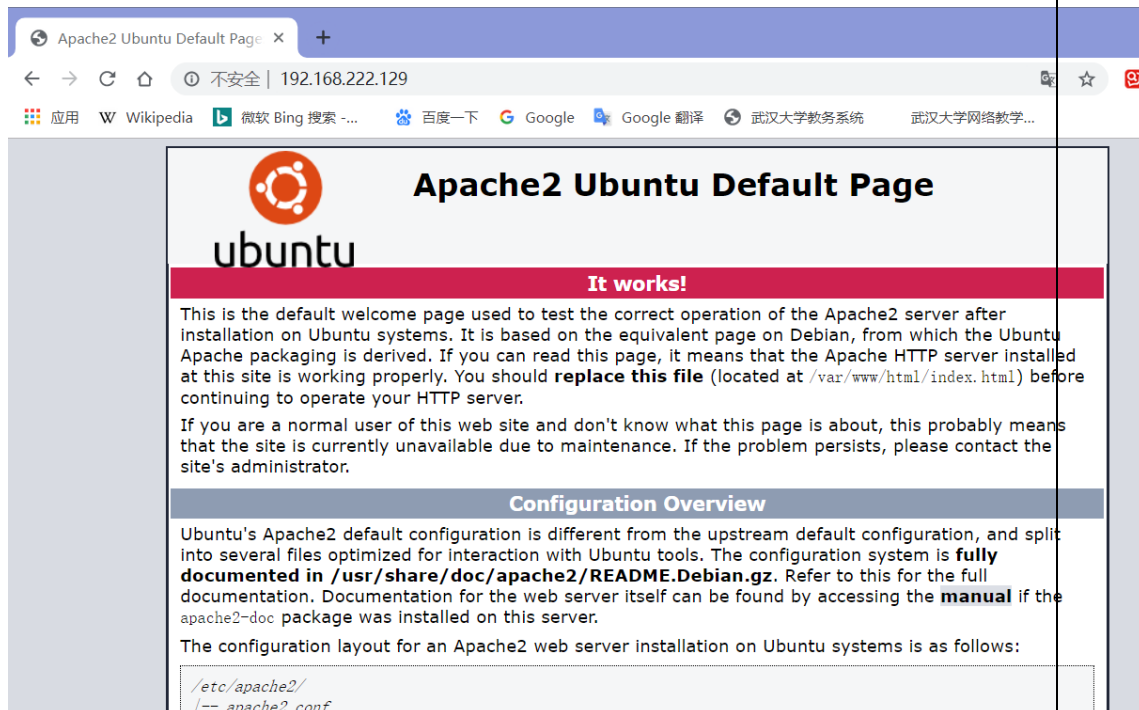
```

dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ netstat -aptn
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.53:53          0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22            0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:631         0.0.0.0:*               LISTEN      -
tcp        0      0 192.168.222.129:58562 35.163.169.36:443      ESTABLISHED 7836/firefox
tcp6       0      0 :::80                 :::*                   LISTEN      -
tcp6       0      0 :::22                 :::*                   LISTEN      -
tcp6       0      0 :::1:631              :::*                   LISTEN      -

```

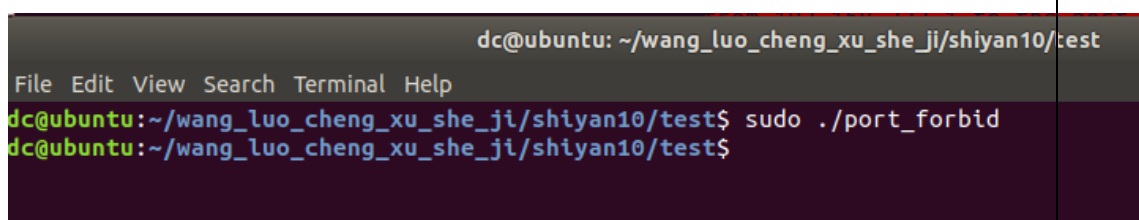
80 端口已开放。并且是 tcp 协议，所以写测试程序的时候用的也是 tcp。

用主机访问试试：

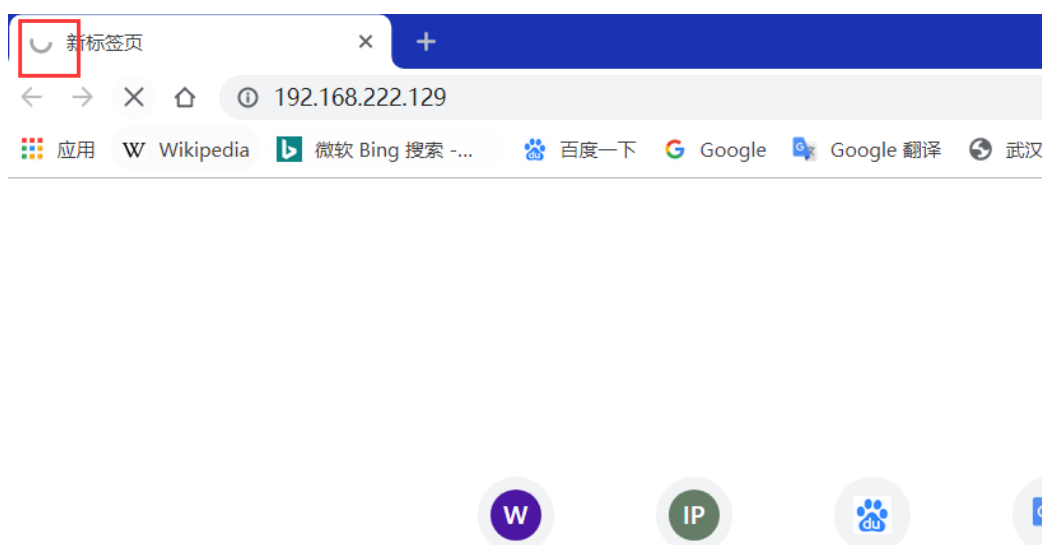


可以访问。

执行 port_forbid 程序：



主机再次访问：



发现一直在加载，就是访问不了。

查看系统日志：

```
dc@ubuntu: ~  
File Edit View Search Terminal Help  
[23663.962561] 2017301510052-zbw netfilter init successfully  
[24069.763803] DROP one packet from 192.168.222.1 to this host  
[24074.440465] DROP one packet from 192.168.222.1 to this host  
[24193.869116] DROP one packet from 192.168.222.1 to this host  
[24477.618228] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1  
[24478.641128] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1  
[24479.664881] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1  
[24480.688687] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1  
[25052.317067] drop one tcp packet from 192.168.222.1 to the port 80  
[25052.317331] drop one tcp packet from 192.168.222.1 to the port 80  
[25052.564608] drop one tcp packet from 192.168.222.1 to the port 80  
[25053.312237] drop one tcp packet from 192.168.222.1 to the port 80  
[25053.313032] drop one tcp packet from 192.168.222.1 to the port 80  
[25053.568453] drop one tcp packet from 192.168.222.1 to the port 80  
[25055.311866] drop one tcp packet from 192.168.222.1 to the port 80  
[25055.312900] drop one tcp packet from 192.168.222.1 to the port 80  
[25055.565593] drop one tcp packet from 192.168.222.1 to the port 80  
[25059.312531] drop one tcp packet from 192.168.222.1 to the port 80  
[25059.313597] drop one tcp packet from 192.168.222.1 to the port 80  
[25059.566259] drop one tcp packet from 192.168.222.1 to the port 80  
[25067.312732] drop one tcp packet from 192.168.222.1 to the port 80  
[25067.313648] drop one tcp packet from 192.168.222.1 to the port 80  
[25067.566455] drop one tcp packet from 192.168.222.1 to the port 80  
dc@ubuntu:~$
```

可以看到访问该端口的数据包都被丢掉了。

6) 允许某个端口的数据响应

执行 port_allow 程序：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10/test  
File Edit View Search Terminal Help  
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$ sudo ./port forbid  
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$ sudo ./port_allow  
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10/test$
```

主机再次访问：



发现又可以访问了。

最后卸载模块：

```
dc@ubuntu: ~/wang_luo_cheng_xu_she_ji/shiyan10
File Edit View Search Terminal Help
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ sudo rmmod netfilter
[sudo] password for dc:
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$
```

可以用 lsmod 查看当前模块：

```
dc@ubuntu:~/wang_luo_cheng_xu_she_ji/shiyan10$ lsmod
Module              Size  Used by
btrfs               1138688  0
zstd_compress       163840  1 btrfs
xor                  24576  1 btrfs
raid6_pq            114688  1 btrfs
ufs                  77824  0
qnx4                 16384  0
hfsplus             106496  0
hfs                  57344  0
minix                32768  0
ntfs                 102400  0
msdos                20480  0
jfs                  188416  0
xfs                  1204224  0
```

发现原来第一个模块就是我们加载的模块被成功卸载。

查看系统日志：

```
dc@ubuntu: ~  
File Edit View Search Terminal Help  
[23663.962561] 2017301510052-zbw netfilter init successfully  
[24069.763803] DROP one packet from 192.168.222.1 to this host  
[24074.440465] DROP one packet from 192.168.222.1 to this host  
[24193.869116] DROP one packet from 192.168.222.1 to this host  
[24477.618228] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1  
[24478.641128] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1  
[24479.664881] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1  
[24480.688687] DROP one ICMP packet from 192.168.222.129 to 192.168.222.1  
[25052.317067] drop one tcp packet from 192.168.222.1 to the port 80  
[25052.317331] drop one tcp packet from 192.168.222.1 to the port 80  
[25052.564608] drop one tcp packet from 192.168.222.1 to the port 80  
[25053.312237] drop one tcp packet from 192.168.222.1 to the port 80  
[25053.313032] drop one tcp packet from 192.168.222.1 to the port 80  
[25053.568453] drop one tcp packet from 192.168.222.1 to the port 80  
[25055.311866] drop one tcp packet from 192.168.222.1 to the port 80  
[25055.312900] drop one tcp packet from 192.168.222.1 to the port 80  
[25055.565593] drop one tcp packet from 192.168.222.1 to the port 80  
[25059.312531] drop one tcp packet from 192.168.222.1 to the port 80  
[25059.313597] drop one tcp packet from 192.168.222.1 to the port 80  
[25059.566259] drop one tcp packet from 192.168.222.1 to the port 80  
[25067.312732] drop one tcp packet from 192.168.222.1 to the port 80  
[25067.313648] drop one tcp packet from 192.168.222.1 to the port 80  
[25067.566455] drop one tcp packet from 192.168.222.1 to the port 80  
[25235.959374] 2017301510052-zbw netfilter clean successfully  
dc@ubuntu: ~$
```

执行了卸载的函数，成功卸载。

实验体会：

这次的实验是关于 netfilter 的，虽然网络安全课程做过这方面的实验，不过做的都是关于 iptables 表的实验，这个钩子函数的调用，模块的实验还是没有用到的。这次的实验不难，主要的问题是书上的例子都太老了，一些函数都不能用了，在高版本内核中，这个函数都换了名字而且改了参数，所以需要自己从网上重新找资料来写代码，这是十分令人头疼的，不过好在最后成功完成了这个实验，收获满满。

对 netfilter 了解更深了，对其中钩子函数的使用也熟悉了，学到了很多。