Firewall Exploration

57118224 邱龙

Task 1.A: Implement a Simple Kernel Module

将 kernel_module 文件夹拷贝到 home 目录下编译：

```
[07/24/21]seed@VM:~/kernel_module$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/kernel_module modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M]  /home/seed/kernel_module/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/seed/kernel_module/hello.o
see include/linux/module.h for more information
  CC [M]  /home/seed/kernel_module/hello.mod.o
  LD [M]  /home/seed/kernel_module/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
```

使用 sudo insmod hello.ko 命令加载模块并使用 lsmod | grep hello 命令查看:

```
[07/24/21]seed@VM:~/kernel_module$ sudo insmod hello.ko
[07/24/21]seed@VM:~/kernel_module$ lsmod | grep hello
hello                  16384  0
[07/24/21]seed@VM:~/kernel_module$
```

使用 sudo rmmod hello 命令删除模块，并使用 dmesg 命令输出消息：

```
[07/24/21]seed@VM:~/kernel_module$ sudo rmmod hello
[07/24/21]seed@VM:~/kernel_module$ dmesg
[    0.000000] Linux version 5.4.0-54-generic (buildd@lcy01-amd64-024) (gcc vers
ion 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)) #60-Ubuntu SMP Fri Nov 6 10:37:59 UTC
2020 (Ubuntu 5.4.0-54.60-generic 5.4.65)
```

结尾可以看到函数输出的消息：

```
[ 6026.129191] Hello World!
[ 6419.320120] Bye-bye World!.
```

Task 1.B: Implement a Simple Firewall Using Netfilter

1. 编译示例代码

编译示例代码前使用命令 dig @8.8.8.8 www.example.com 生成到谷歌服务器的 udp 数据包：

```
;; QUESTION SECTION:
;www.example.com.               IN      A

;; ANSWER SECTION:
www.example.com.        20140   IN      A       93.184.216.34
```

成功查询到 DNS 信息，说明数据包能够成功到达。

在 packet_filter 文件夹编译代码并加载到内核中：

```
[07/25/21]seed@VM:~/packet_filter$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/packet_filter modul
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M]  /home/seed/packet_filter/seedFilter.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/seed/packet_filter/seedFilter.mod.o
  LD [M]  /home/seed/packet_filter/seedFilter.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
[07/25/21]seed@VM:~/packet_filter$ sudo insmod seedFilter.ko
```

再次使用 dig 命令：

```
[07/25/21]seed@VM:~/packet_filter$ dig @8.8.8.8 www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> @8.8.8.8 www.example.com
; (1 server found)
;; global options: +cmd
;; connection timed out; no servers could be reached
```

发现连接超时，无法达到服务器。说明防火墙成功工作。

2. 将 printInfo 函数挂接到所有的 netfilter 的 hook 上

添加 hook：

```
static struct nf_hook_ops hook1, hook2, hook3, hook4, hook5;
```

修改 registerFilter 和 removeFilter 函数：

```
int registerFilter(void) {
    printk(KERN_INFO "Registering filters.\n");

    hook1.hook = printInfo;
    hook1.hooknum = NF_INET_LOCAL_IN;
    hook1.pf = PF_INET;
    hook1.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook1);

    hook2.hook = printInfo;
    hook2.hooknum = NF_INET_POST_ROUTING;
    hook2.pf = PF_INET;
    hook2.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook2);

    hook3.hook = printInfo;
    hook3.hooknum = NF_INET_PRE_ROUTING;
    hook3.pf = PF_INET;
    hook3.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook3);

    hook4.hook = printInfo;
    hook4.hooknum = NF_INET_FORWARD;
    hook4.pf = PF_INET;
    hook4.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook4);

    hook5.hook = printInfo;
    hook5.hooknum = NF_INET_LOCAL_OUT;
    hook5.pf = PF_INET;
    hook5.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook5);
```

```
void removeFilter(void) {
    printk(KERN_INFO "The filters are being removed.\n");
    nf_unregister_net_hook(&init_net, &hook1);
    nf_unregister_net_hook(&init_net, &hook2);
    nf_unregister_net_hook(&init_net, &hook3);
    nf_unregister_net_hook(&init_net, &hook4);
    nf_unregister_net_hook(&init_net, &hook5);
}
```

重新编译和加载（加载前删除以前的模块）后，我们可以 ping 下百度地址 202.108.22.5，如何 dmesg 查看输出：

```
[ 4436.423383] *** LOCAL_OUT
[ 4436.423388]     192.168.161.135  --> 202.108.22.5 (ICMP)
[ 4436.423417] *** POST_ROUTING
[ 4436.423418]     192.168.161.135  --> 202.108.22.5 (ICMP)
```

可以看到本地的数据包发送时先是调用了 NF_INET_LOCAL_OUT,然后是 NF_INET_POST_ROUTING。

```
[ 4436.448710] *** PRE_ROUTING
[ 4436.448741]     202.108.22.5  --> 192.168.161.135 (ICMP)
[ 4436.448756] *** LOCAL_IN
[ 4436.448758]     202.108.22.5  --> 192.168.161.135 (ICMP)
```

可以看到接收到发送给本地的数据包时先是调用 NF_INET_PRE_ROUTING，然后是 NF_INET_LOCAL_IN。

结合我们的实验现象以及查阅相关资料可知：NF_INET_PRE_ROUTING 在数据包进入系统，进行 ip 校验后调用。然后进入路由代码，如果数据包是发送给本机，则调用 NF_INET_LOCAL_IN；如果数据包需要转发则调用 NF_INET_FORWARD。本地产生数据包时先调用 NF_IP_LOCAL_OUT ，进行路由选择处理，然后调用 NF_IP_POST_ROUTING 后发送出去。

3. 实现另外两个 hook

再添加两个 hook 函数，代码如下：

```
unsigned int blockICMP(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
    struct iphdr *iph;
    if (!skb) return NF_ACCEPT;
    iph = ip_hdr(skb);

    if (iph->protocol == IPPROTO_ICMP) {
        printk(KERN_WARNING "*** Dropping %pI4 (ICMP)\n", &(iph->daddr));
        return NF_DROP;
    }
    return NF_ACCEPT;
}
```

这个函数负责阻止 ICMP 报文。

```
unsigned int blockTCP(void *priv, struct sk_buff *skb,
                      const struct nf_hook_state *state)
{
    struct iphdr *iph;
    struct tcphdr *tcph;
    u16 port = 23;
    if (!skb) return NF_ACCEPT;
    iph = ip_hdr(skb);
    if (iph->protocol == IPPROTO_TCP) {
        tcph = tcp_hdr(skb);
        if (ntohs(tcph->dest) == port){
            printk(KERN_WARNING "*** Dropping %pI4 (TCP), port %d\n",
&(iph->daddr), port);
            return NF_DROP;
         }
    }
    return NF_ACCEPT;
}
```

这个函数负责阻止目的端口为 23 的 TCP 报文。

在 registerFilter 函数中添加两个 hook 的调用（removeFilter 函数中也要有对应的注销 hook，代码与前面 task1.B.2 类似，不再介绍）：

```
  hook6.hook = blockICMP;
    hook6.hooknum = NF_INET_LOCAL_IN;
    hook6.pf = PF_INET;
    hook6.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook6);

    hook7.hook = blockTCP;
    hook7.hooknum = NF_INET_LOCAL_IN;
    hook7.pf = PF_INET;
    hook7.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook7);
```

前面我们已经讨论过什么时候调用了什么 hook，这里我们要阻止特定数据包发给虚拟机，所以我们使用 NF_INET_LOCAL_IN 在数据包发给本机时调用我们的 hook 函数，阻止特定数据包接收。

在容器 10.9.0.5 中 ping 我们虚拟机的 ip 地址并进行 Telnet 连接：

```
root@7da19bac6fee:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
^C
--- 10.9.0.1 ping statistics ---
15 packets transmitted, 0 received, 100% packet loss, time 14343ms
```

```
root@7da19bac6fee:/# telnet 10.9.0.1
Trying 10.9.0.1...
```

发现 ping 和 Telnet 都无法成功。Dmesg 命令查看内核输出：

```
[ 7789.794464] *** Dropping 10.9.0.1 (ICMP)
[ 7790.817494] *** PRE_ROUTING
[ 7790.817500]     10.9.0.5  --> 10.9.0.1 (ICMP)
[ 7790.817537] *** PRE_ROUTING
[ 7790.817538]     10.9.0.5  --> 10.9.0.1 (ICMP)
[ 7790.817549] *** Dropping 10.9.0.1 (ICMP)
[ 7828.199491] *** Dropping 10.9.0.1 (TCP), port 23
[ 7829.219696] *** PRE_ROUTING
[ 7829.219701]     10.9.0.5  --> 10.9.0.1 (TCP)
[ 7829.219750] *** PRE_ROUTING
[ 7829.219751]     10.9.0.5  --> 10.9.0.1 (TCP)
[ 7829.219765] *** Dropping 10.9.0.1 (TCP), port 23
[ 7831.233761] *** PRE_ROUTING
[ 7831.233771]     10.9.0.5  --> 10.9.0.1 (TCP)
[ 7831.233871] *** PRE_ROUTING
[ 7831.233873]     10.9.0.5  --> 10.9.0.1 (TCP)
[ 7831.233905] *** Dropping 10.9.0.1 (TCP), port 23
```

发现发送给 10.9.0.1 的 ICMP 数据包和目的地址为 23 的 TCP 数据包成功被阻止。说明我们防火墙设置成功。

Task 2: Experimenting with Stateless Firewall Rules

Task 2.A: Protecting the Router

路由器容器上执行下列 iptables 命令：

```
iptables -A INPUT    -p icmp --icmp-type echo-request    -j ACCEPT
iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
iptables -P OUTPUT DROP ¥ Set default rule for OUTPUT
iptables -P INPUT    DROP ¥ Set default rule for INPUT
```

pdf 上 request 和 reply 写反了，需要交换过来。

在 10.9.0.5 主机上 ping 路由器以及 Telnet 连接：

```
root@fddc2b5bcf64:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.074 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.051 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.104 ms
^C
--- 10.9.0.11 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2029ms
rtt min/avg/max/mdev = 0.051/0.076/0.104/0.021 ms
root@fddc2b5bcf64:/# telnet 10.9.0.11
Trying 10.9.0.11...
```

发现可以 ping 通，但是 Telnet 无法连接。这是因为我们在 iptables 规则中允许 icmp 请求报文输入，允许 icmp 响应报文输出，所以可以 ping 通。其他报文都不允许通过，所以无法 Telnet 成功。

Task 2.B: Protecting the Internal Network

查看路由器接口：

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.11  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:0b  txqueuelen 0  (Ethernet)
        RX packets 167  bytes 17018 (17.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 18  bytes 1372 (1.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.60.11  netmask 255.255.255.0  broadcast 192.168.60.255
        ether 02:42:c0:a8:3c:0b  txqueuelen 0  (Ethernet)
        RX packets 65  bytes 6982 (6.9 KB)
        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
```

可以看到外部接口为 eth0,内部接口为 eth1。

防火墙要求如下：

1. 外部主机无法 ping 内部主机。
2. 外部主机可以 ping 通路由器。
3. 内部主机可以 ping 外部主机。
4. 应该阻止内部和外部网络之间的所有其他数据包。

设 iptables -P FORWARD DROP 阻止内部和外部之间所有数据包。但是路由器可以 ping 通。

可设 iptables -A FORWARD -p icmp --icmp-type echo-request -i eth1 -j ACCEPT 允许转发内部网络的 ICMP 请求报文。

可设 iptables -A FORWARD -p icmp --icmp-type echo-reply -i eth0 -j ACCEPT 允许转发外部网络的 ICMP 响应报文。这样就能使得内部主机可以 ping 外部主机，但是外部主机的请求报文无法转发，使得外部主机无法 ping 通内部主机。

所以我们执行如下规则：

```
iptables -A FORWARD -p icmp --icmp-type echo-request -i eth1 -j ACCEPT
iptables -A FORWARD -p icmp --icmp-type echo-reply -i eth0 -j ACCEPT
iptables -P FORWARD DROP
```

在外部主机 10.9.0.5 中 ping 内部主机 192.168.60.5

```
root@fddc2b5bcf64:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9231ms
```

无法 ping 通。

在外部主机 10.9.0.5 中 ping 路由器地址 10.9.0.11

```
root@fddc2b5bcf64:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.087 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.103 ms
^C
--- 10.9.0.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1005ms
rtt min/avg/max/mdev = 0.087/0.095/0.103/0.008 ms
```

可以 ping 通。

在内部主机 192.168.60.5 中 ping 外部主机 10.9.0.5

```
root@c822a6d7ec54:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=63 time=0.105 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=63 time=0.080 ms
^C
--- 10.9.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1004ms
rtt min/avg/max/mdev = 0.080/0.092/0.105/0.012 ms
```

可以 ping 通。

在内部主机 192.168.60.5 中向外部主机 10.9.0.5 建立 Telnet

```
root@c822a6d7ec54:/# telnet 10.9.0.5
Trying 10.9.0.5...
```

无法建立 Telnet。

在外部主机 10.9.0.5 中向内部主机 192.168.60.5 建立 Telnet

```
root@fddc2b5bcf64:/# telnet 192.168.60.5
Trying 192.168.60.5...
```

无法建立 Telnet。
说明我们防火墙设置成功。

Task 2.C: Protecting Internal Servers

过滤规则如下：

```
iptables -A FORWARD   -i eth0 -p tcp -d 192.168.60.5 --dport 23 -j ACCEPT
iptables -A FORWARD   -o eth0 -p tcp -s 192.168.60.5 --sport 23 -j ACCEPT
iptables -A FORWARD   -i eth1 -p tcp -s 192.168.60.0/24 --dport 23 -j ACCEPT
iptables -A FORWARD   -o eth1 -p tcp -s 192.168.60.0/24 --sport 23 -j ACCEPT
iptables -P FORWARD DROP
```

外部主机 10.9.0.5 访问内部主机 192.168.60.5：

```
root@fddc2b5bcf64:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
c822a6d7ec54 login: seed
Password:
```

能够成功访问。

外部主机 10.9.0.5 访问其他内部主机 192.168.60.6：

```
root@fddc2b5bcf64:/# telnet 192.168.60.6
Trying 192.168.60.6...
```

无法访问。

内部主机 192.168.60.5 访问内部主机 192.168.60.6 和 192.168.60.7：

```
root@c822a6d7ec54:/# telnet 192.168.60.6
Trying 192.168.60.6...
Connected to 192.168.60.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
ec5b9c159941 login:
root@c822a6d7ec54:/# telnet 192.168.60.7
Trying 192.168.60.7...
Connected to 192.168.60.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
3a3d0dd207e1 login:
```

都可以成功访问。

内部主机 192.168.60.5 访问外部主机 10.9.0.5：

```
root@c822a6d7ec54:/# telnet 10.9.0.5
Trying 10.9.0.5...
```

无法访问。

符合要求，防火墙规则设置成功。

Task 3: Connection Tracking and Stateful Firewall

Task 3.A: Experiment with the Connection Tracking

（1）ICMP 实验

在主机 10.9.0.5 上执行命令 ping 192.168.60.5：

```
root@fddc2b5bcf64:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.089 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.145 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.231 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.063 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.400 ms
```

然后执行 conntrack -L 命令查看追踪信息：

```
[07/26/21]seed@VM:~/.../Labsetup$ docksh fd
root@fddc2b5bcf64:/# conntrack -L
icmp     1 29 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=52 src=192.168.60.5
 dst=10.9.0.5 type=0 code=0 id=52 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

发现 icmp 连接时间为 29 秒。

（2）udp 实验

在主机 192.168.60.5 上执行 nc -lu 9090，然后在 10.9.0.5 主机上执行 nc -u 192.168.60.5 9090 命令。在 10.9.0.5 主机上随便输入字符后快速执行 conntrack -L 命令。

```
root@fddc2b5bcf64:/# nc -u 192.168.60.5 9090
aa
```

```
root@c822a6d7ec54:/# nc -lu 9090
aa
```

```
root@fddc2b5bcf64:/# conntrack -L
udp      17 29 src=10.9.0.5 dst=192.168.60.5 sport=37208 dport=9090 [UNREPLIED]
src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=37208 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

可以看到 udp 连接的时间也是 29 秒。

（3）Tcp 实验

在主机 192.168.60.5 上执行 nc -l 9090，然后在 10.9.0.5 主机上执行 nc 192.168.60.5 9090 命令。在 10.9.0.5 主机上随便输入字符后快速执行 conntrack -L 命令。

```
root@fddc2b5bcf64:/# nc 192.168.60.5 9090
aaa
```

```
root@c822a6d7ec54:/# nc -l 9090
aaa
```

```
tcp      6 431999 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=56870 dport=90
90 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=56870 [ASSURED] mark=0 use=1
```

发现 tcp 连接保持时间约为 432000 秒。

```
root@fddc2b5bcf64:/# conntrack -L
tcp      6 119 TIME_WAIT src=10.9.0.5 dst=192.168.60.5 sport=56882 dport=9090 sr
c=192.168.60.5 dst=10.9.0.5 sport=9090 dport=56882 [ASSURED] mark=0 use=1
```

当结束连接时，看到 tcp 的连接时间为 119 秒。

Task 3.B: Setting Up a Stateful Firewall

使用连接追踪机制编写规则：

```
iptables -A FORWARD -p tcp -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -p tcp -i eth0 -d 192.168.60.5 --dport 23 --syn -m conntrack --ctstate NEW -j ACCEPT
iptables -A FORWARD -p tcp -i eth1 -s 192.168.60.0/24 --dport 23 --syn -m conntrack --ctstate NEW -j ACCEPT
iptables -P FORWARD DROP
```

外部主机 10.9.0.5 访问内部主机 192.168.60.5：

```
root@fddc2b5bcf64:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
c822a6d7ec54 login:
```

能够成功访问。

外部主机 10.9.0.5 访问其他内部主机 192.168.60.6：

```
root@fddc2b5bcf64:/# telnet 192.168.60.6
Trying 192.168.60.6...
```

无法访问。

内部主机 192.168.60.5 访问内部主机 192.168.60.6：

```
root@c822a6d7ec54:/# telnet 192.168.60.6
Trying 192.168.60.6...
Connected to 192.168.60.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
ec5b9c159941 login:
```

可以成功访问。

内部主机 192.168.60.5 访问外部主机 10.9.0.5：

```
root@c822a6d7ec54:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
fddc2b5bcf64 login:
```

访问成功。符合要求，防火墙规则设置成功。

不使用连接跟踪机制的过滤规则检查比较简单，优点是处理速度快，缺点是不会考虑数据包的上下文，因此会导致不准确、不安全或复杂的防火墙规则；使用连接跟踪机制的过滤规则对数据包的状态进行检查，优点是能建立有状态防火墙，安全性更高，更准确，缺点是无法对数据包内容进行识别。

Task 4: Limiting Network Traffic

限制可以通过防火墙的数据包数量，规则如下：

```
iptables -A FORWARD -s 10.9.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT
iptables -A FORWARD -s 10.9.0.5 -j DROP
```

在 10.9.0.5 主机上 ping 196.168.60.5，结果如下：

```
root@fddc2b5bcf64:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.171 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.145 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.162 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.193 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.229 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.125 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.176 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=63 time=0.243 ms
64 bytes from 192.168.60.5: icmp_seq=25 ttl=63 time=0.211 ms
64 bytes from 192.168.60.5: icmp_seq=31 ttl=63 time=0.190 ms
64 bytes from 192.168.60.5: icmp_seq=37 ttl=63 time=0.182 ms
64 bytes from 192.168.60.5: icmp_seq=43 ttl=63 time=0.194 ms
64 bytes from 192.168.60.5: icmp_seq=48 ttl=63 time=0.096 ms
^C
--- 192.168.60.5 ping statistics ---
50 packets transmitted, 13 received, 74% packet loss, time 50165ms
rtt min/avg/max/mdev = 0.096/0.178/0.243/0.038 ms
```

可以发现 ping 的过程中丢了很多包，导致收到包的速率降低，通过防火墙的数据包数量被限制。

不使用第二条规则时：

```
iptables -A FORWARD -s 10.9.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT
```

重新 ping 196.168.60.5，结果如下：

```
root@fddc2b5bcf64:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.098 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.107 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.237 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.203 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.201 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=0.083 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.066 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=0.064 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=0.205 ms
^C
--- 192.168.60.5 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8199ms
rtt min/avg/max/mdev = 0.064/0.140/0.237/0.065 ms
```

此时没有出现丢包，数据包没有得到限制。

实验结果说明需要第二条规则。这是因为如果不设置第二条规则，所有数据包默认 ACCEPT，不会发生丢包，也就不能限制数据包数量。

Task 5: Load Balancing

Using the nth mode (round-robin)

使用 nth 模式实现负载平衡,规则如下：

```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 3 --packet 0  -j DNAT --to-destination 192.168.60.5:8080
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 2 --packet 0  -j DNAT --to-destination 192.168.60.6:8080
iptables -t nat -A PREROUTING -p udp --dport 8080  -j DNAT --to-destination 192.168.60.7:8080
```

这里第一条规则将每三个数据包中的第一个发送给 192.168.60.5，然后第二条规则让剩下的两个中的数据包中的第一个发给 192.168.60.6，第三条规则让剩下的一个数据包发给 192.168.60.7。这样就使得三个数据包平均分给三个主机。

结果如下：

```
root@fddc2b5bcf64:/# echo 1 | nc -u 10.9.0.11 8080
^C
root@fddc2b5bcf64:/# echo 2 | nc -u 10.9.0.11 8080
^C
root@fddc2b5bcf64:/# echo 3 | nc -u 10.9.0.11 8080
^C
root@fddc2b5bcf64:/# echo 4 | nc -u 10.9.0.11 8080
^C
root@fddc2b5bcf64:/# echo 5 | nc -u 10.9.0.11 8080
^C
root@fddc2b5bcf64:/# echo 6 | nc -u 10.9.0.11 8080
^C
root@fddc2b5bcf64:/# echo 7 | nc -u 10.9.0.11 8080
^C
root@fddc2b5bcf64:/# echo 8 | nc -u 10.9.0.11 8080
^C
root@fddc2b5bcf64:/# echo 9 | nc -u 10.9.0.11 8080
^C
```

```
root@c822a6d7ec54:/# nc -luk 8080
1
4
7
```

```
root@ec5b9c159941:/# nc -luk 8080
2
5
8
```

```
root@3a3d0dd207e1:/# nc -luk 8080
3
6
9
```

在 10.9.0.5 主机上发送了 9 个数据包，三个主机按顺序接收到了数据包，最后都收到三个数据包。

Using the random mode

随机模式下的规则如下:

```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.33 -j DNAT --to-destination 192.168.60.5:8080
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.50 -j DNAT --to-destination 192.168.60.6:8080
iptables -t nat -A PREROUTING -p udp --dport 8080 -j DNAT --to-destination 192.168.60.7:8080
```

第一条规则以 1/3 的概率将数据包发给 192.168.60.5，如果第一条没有捕获，则以 1/2 的概率发给 192.168.60.6, 如果第二条也没捕获，则发给 192.168.60.7，这样三个主机收到数据包的概率都是 1/3。

10.9.0.5 主机上发送 30 个数据包，结果如下：

```
root@c822a6d7ec54:/# nc -luk 8080
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

```
root@ec5b9c159941:/# nc -luk 8080
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

```
root@3a3d0dd207e1:/# nc -luk 8080
hello
hello
hello
hello
hello
hello
hello
hello
```

三个数据包分别收到 12、10、8 个数据报，概率基本一致，数据包总数越大，数量越相同。实验成功。