Firewall Exploration Lab

Task 0: Lab setup

SEED Labs – Firewall Exploration Lab 2

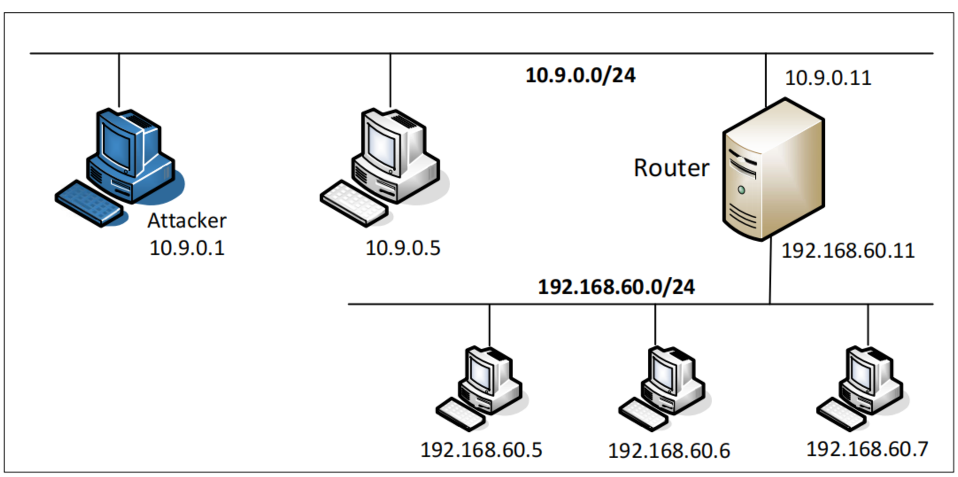


Figure 1: Lab setup

Task 1: Implementing a Simple Firewall

Task 1.A: Implement a Simple Kernel Module

1. Comple the kernel mode ($ make)

```
[03/24/23]seed@VM:~/.../kernel_module$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/Firewall/Files/kernel_module modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M]  /home/seed/Firewall/Files/kernel_module/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/seed/Firewall/Files/kernel_module/hello.mod.o
  LD [M]  /home/seed/Firewall/Files/kernel_module/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
[03/24/23]seed@VM:~/.../kernel_module$
```

2. We can see some files are created. ($ ls)

```
[03/24/23]seed@VM:~/.../kernel_module$ ls
hello.c  hello.ko  hello.mod  hello.mod.c  hello.mod.o  hello.o  Makefile  modules.order  Module.symvers
[03/24/23]seed@VM:~/.../kernel_module$
```

3. insert a model ($ sudo insmod hello.ko) and check it exist ($ lsmod | grep hello)

```
[03/24/23]seed@VM:~/.../kernel_module$ sudo insmod hello.ko
[03/24/23]seed@VM:~/.../kernel_module$ lsmod | grep hello
hello                  16384  0
[03/24/23]seed@VM:~/.../kernel_module$
```

4.insert a model ($ sudo rmmod **hello**) and check it exist ($ lsmod | grep hello)

```
[03/24/23]seed@VM:~/.../kernel_module$ sudo rmmod hello
[03/24/23]seed@VM:~/.../kernel_module$ lsmod | grep hello
[03/24/23]seed@VM:~/.../kernel_module$
```

5.check the message ($ dmesg | tail -2)

```
[03/24/23]seed@VM:~/.../kernel_module$ dmesg | tail -2
[ 1694.758083] Hello World!
[ 2061.402033] Bye-bye World!.
[03/24/23]seed@VM:~/.../kernel_module$
```

Task 1.B: Implement a Simple Firewall Using Netfilter

▼ Task 1.B.0 BackGround & Code

Netfilter & Hook

Netfilter checks if any kernel module has registed a callback function at this hook. Each registed modual will be caled and they are free to analyze or manipulate the pakcet. The hook function will return five possoble values

Each funtion has priority , **lower valuehas higher priority** to call.

NF_ACCEPT	Let the packet go
NF_DROP	Discard the packet
NF_QUEUE	pass the packet to user_space via nf_queue
NF_STOLEN	Infrom the the filter to forget about the packet(for futher analysis)
NF_REPEAT	let net filter call the modual again

IPV4 five hooks :

NF_INET_PRE_ROUTING: Called before any routing.

For Host:

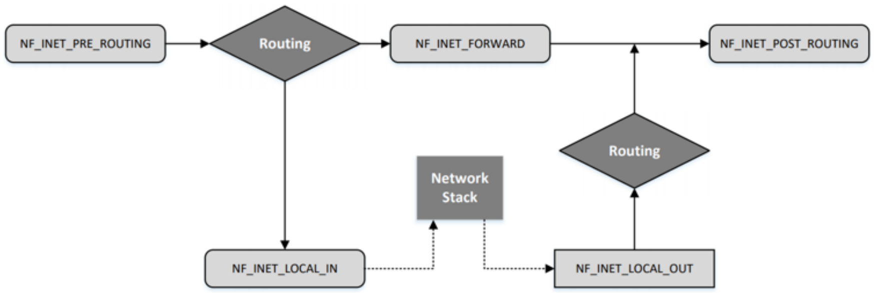
NF_INET_LOCAL_IN: If the packet for the host,**before send to the network** stack and eventually consumed by the stack.

NF_INET_LOCAL_OUT: packet **genterate by the local host.** This is the **First hook** for the pakets on their way out of the host.

Router to other:

NF_INET_FORWARD: Packet forward ot other hosts reach this hook.

NF_INET_POST_ROUTING: When a packet,forward or generated is **going out of the host,** it will pass this hook



code 1 Register hook functions to the netfilter:

The `registerFilter` function is called when the module is loaded

```
int registerFilter(void) {
//Prints a message indicating that filters are being registered.
   printk(KERN_INFO "Registering filters.\n");
//
   hook1.hook = printInfo;//hook_function
   hook1.hooknum = NF_INET_LOCAL_OUT; // hook_point
   hook1.pf = PF_INET; //IPV4 protocal
   hook1.priority = NF_IP_PRI_FIRST;//the highest priority
// Registers hook1 with the Netfilter framework
//&init_net is a pointer to the net structure that represents the initial network name
sp   //ace in the Linux kernel.
   nf_register_net_hook(&init_net, &hook1);
/****************************************************/
   hook2.hook = blockUDP;
   hook2.hooknum = NF_INET_POST_ROUTING;
   hook2.pf = PF_INET;
   hook2.priority = NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net, &hook2);

   return 0;
}
//This function removes the two network filters that were registered using the Netfilt
er framework
void removeFilter(void) {
   printk(KERN_INFO "The filters are being removed.\n");
// Unregisters hook1 and hook2 with the Netfilter framework
   nf_unregister_net_hook(&init_net, &hook1);
   nf_unregister_net_hook(&init_net, &hook2);
}

module_init(registerFilter);
```

```
module_exit(removeFilter);

MODULE_LICENSE("GPL");
```

code2: Hook functions:

This is a function in the Linux Netfilter framework that processes a network packet and prints out information

```
unsigned int printInfo(void *priv, struct sk_buff *skb,
                 const struct nf_hook_state *state)
{
   struct iphdr *iph;//a pointer to an IP header struct (iph)
   char *hook; //a pointer to a character array (hook)
   char *protocol; // another pointer to a character array (protocol)

//switch statement determines which hook the packet is being processed
   switch (state->hook){
     case NF_INET_LOCAL_IN:     hook = "LOCAL_IN";     break;
     case NF_INET_LOCAL_OUT:    hook = "LOCAL_OUT";    break;
     case NF_INET_PRE_ROUTING:  hook = "PRE_ROUTING";  break;
     case NF_INET_POST_ROUTING: hook = "POST_ROUTING"; break;
     case NF_INET_FORWARD:      hook = "FORWARD";      break;
     default:                   hook = "IMPOSSIBLE";   break;
   }
   printk(KERN_INFO "*** %s\n", hook); // Print out the hook info

   iph = ip_hdr(skb);//extracts the IP header from the skb struct and stores it in the
iph variable.
//determines the protocol of the packet based on the protocol field in the IP header.
   switch (iph->protocol){
     case IPPROTO_UDP:  protocol = "UDP";   break;
     case IPPROTO_TCP:  protocol = "TCP";   break;
     case IPPROTO_ICMP: protocol = "ICMP";  break;
     default:           protocol = "OTHER"; break;

   }
   // Print out the IP addresses and protocol
   printk(KERN_INFO "    %pI4  --> %pI4 (%s)\n",
                    &(iph->saddr), &(iph->daddr), protocol);

   return NF_ACCEPT; //packet should be accepted and allowed to continue through the n
etwork stack.
}
```

code3:

This code defines a another callback function called `blockUDP`

```
unsigned int blockUDP(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
   struct iphdr *iph;
   struct udphdr *udph;

   u16  port   = 53;
   char ip[16] = "8.8.8.8";
   u32  ip_addr;

//The function then checks if the skb (socket buffer) is null, and if so, it returns N
F_ACCEPT (i.e., the packet is allowed to pass through)
   if (!skb) return NF_ACCEPT;
//retrieves a pointer to the iph (Internet Protocol Header) structure from the given s
kb (socket buffer) structure.
   iph = ip_hdr(skb);
   // Convert the IPv4 address from dotted decimal to 32-bit binary
   in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);

   if (iph->protocol == IPPROTO_UDP) {//if the protocol of the packet is UDP
       udph = udp_hdr(skb);//sets udph to point to the UDP header of the packet

//if the destination IP address of the packet matches the ip_addr variable and if the
 destination port is set to port (which is currently set to 53, the port used by DNS).
       if (iph->daddr == ip_addr && ntohs(udph->dest) == port){
            printk(KERN_WARNING "*** Dropping %pI4 (UDP), port %d\n", &(iph->daddr), p
ort);
            return NF_DROP; //DROP
        }
   }
   return NF_ACCEPT; //PASS
}
```

overall

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/if_ether.h>
#include <linux/inet.h>


static struct nf_hook_ops hook1, hook2;
```

```
unsigned int blockUDP(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
   struct iphdr *iph;
   struct udphdr *udph;

   u16  port   = 53;
   char ip[16] = "8.8.8.8";
   u32  ip_addr;

//The function then checks if the skb (socket buffer) is null, and if so, it returns N
F_ACCEPT (i.e., the packet is allowed to pass through)
   if (!skb) return NF_ACCEPT;
//retrieves a pointer to the iph (Internet Protocol Header) structure from the given s
kb (socket buffer) structure.
   iph = ip_hdr(skb);
   // Convert the IPv4 address from dotted decimal to 32-bit binary
   in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);

   if (iph->protocol == IPPROTO_UDP) {//if the protocol of the packet is UDP
       udph = udp_hdr(skb);//sets udph to point to the UDP header of the packet

//if the destination IP address of the packet matches the ip_addr variable and if the
 destination port is set to port (which is currently set to 53, the port used by DNS).
       if (iph->daddr == ip_addr && ntohs(udph->dest) == port){
            printk(KERN_WARNING "*** Dropping %pI4 (UDP), port %d\n", &(iph->daddr), p
ort);
            return NF_DROP; //DROP
        }
   }
   return NF_ACCEPT; //PASS
}
unsigned int printInfo(void *priv, struct sk_buff *skb,
                 const struct nf_hook_state *state)
{
   struct iphdr *iph;//a pointer to an IP header struct (iph)
   char *hook; //a pointer to a character array (hook)
   char *protocol; // another pointer to a character array (protocol)

//switch statement determines which hook the packet is being processed
   switch (state->hook){
     case NF_INET_LOCAL_IN:     hook = "LOCAL_IN";     break;
     case NF_INET_LOCAL_OUT:    hook = "LOCAL_OUT";    break;
     case NF_INET_PRE_ROUTING:  hook = "PRE_ROUTING";  break;
     case NF_INET_POST_ROUTING: hook = "POST_ROUTING"; break;
     case NF_INET_FORWARD:      hook = "FORWARD";      break;
     default:                   hook = "IMPOSSIBLE";   break;
   }
   printk(KERN_INFO "*** %s\n", hook); // Print out the hook info

   iph = ip_hdr(skb);//extracts the IP header from the skb struct and stores it in the
iph variable.
//determines the protocol of the packet based on the protocol field in the IP header.
   switch (iph->protocol){
```

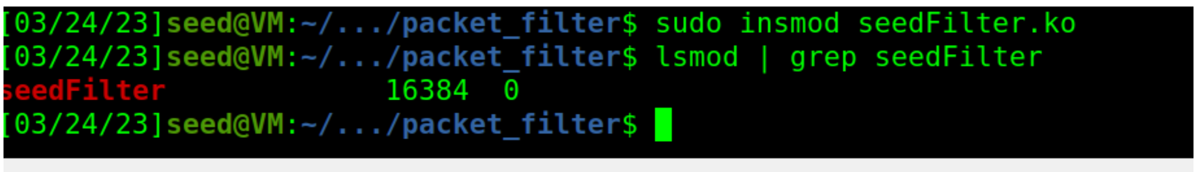
```
      case IPPROTO_UDP:  protocol = "UDP";   break;
      case IPPROTO_TCP:  protocol = "TCP";   break;
      case IPPROTO_ICMP: protocol = "ICMP";  break;
      default:           protocol = "OTHER"; break;

   }
   // Print out the IP addresses and protocol
   printk(KERN_INFO "    %pI4  --> %pI4 (%s)\n",
                    &(iph->saddr), &(iph->daddr), protocol);

   return NF_ACCEPT; //packet should be accepted and allowed to continue through the n
etwork stack.
}

int registerFilter(void) {
//Prints a message indicating that filters are being registered.
   printk(KERN_INFO "Registering filters.\n");
//
   hook1.hook = printInfo;//hook_function
   hook1.hooknum = NF_INET_LOCAL_OUT; // hook_point
   hook1.pf = PF_INET; //IPV4 protocal
   hook1.priority = NF_IP_PRI_FIRST;//the highest priority
// Registers hook1 with the Netfilter framework
//&init_net is a pointer to the net structure that represents the initial network name
sp   //ace in the Linux kernel.
   nf_register_net_hook(&init_net, &hook1);
/****************************************************/
   hook2.hook = blockUDP;
   hook2.hooknum = NF_INET_POST_ROUTING;
   hook2.pf = PF_INET;
   hook2.priority = NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net, &hook2);

   return 0;
}
//This function removes the two network filters that were registered using the Netfilt
er framework
void removeFilter(void) {
   printk(KERN_INFO "The filters are being removed.\n");
// Unregisters hook1 and hook2 with the Netfilter framework
   nf_unregister_net_hook(&init_net, &hook1);
   nf_unregister_net_hook(&init_net, &hook2);
}

module_init(registerFilter);
module_exit(removeFilter);

MODULE_LICENSE("GPL");
```

Task 1.B.1

1. **compile the code (**$ make):

```
[03/24/23]seed@VM:~/.../packet_filter$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/Firewall/Files/packet_filter modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M]  /home/seed/Firewall/Files/packet_filter/seedFilter.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/seed/Firewall/Files/packet_filter/seedFilter.mod.o
  LD [M]  /home/seed/Firewall/Files/packet_filter/seedFilter.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
[03/24/23]seed@VM:~/.../packet_filter$ ls
Makefile  modules.order  Module.symvers  seedFilter.c  seedFilter.ko  seedFilter.mod  seedFilter.mod.c  seedFilter.mod.o  seedFilter.o
[03/24/23]seed@VM:~/.../packet_filter$
```

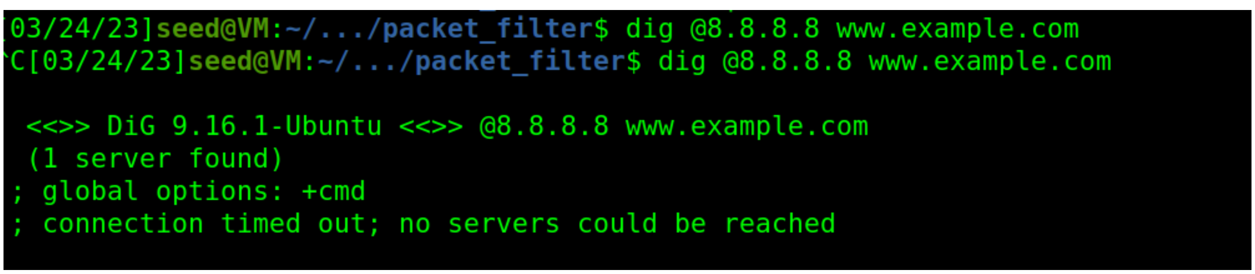
2. **install model**
   a. $ sudo insmod seedFilter.ko
   b. lsmod | grep seedFilter



```
[03/24/23]seed@VM:~/.../packet_filter$ sudo insmod seedFilter.ko
[03/24/23]seed@VM:~/.../packet_filter$ lsmod | grep seedFilter
seedFilter             16384  0
[03/24/23]seed@VM:~/.../packet_filter$
```

3. test firewall : dig @8.8.8.8 www.example.com

   After couple seconds, message showed up,which meaning we cannot reach 8.8.8.8. Our firewall works.



```
[03/24/23]seed@VM:~/.../packet_filter$ dig @8.8.8.8 www.example.com
^C[03/24/23]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
```

4. check the dmesg ($dmesg):

   The 8.8.8.8 UDP packets are droped here

```
[17473.513128] Registering filters.
[17587.136359] *** LOCAL_OUT
[17587.136361]     127.0.0.1  --> 127.0.0.1 (UDP)
[17587.137037] *** LOCAL_OUT
[17587.137038]     10.0.2.5  --> 8.8.8.8 (UDP)
[17587.137045] *** Dropping 8.8.8.8 (UDP), port 53
[17592.132227] *** LOCAL_OUT
[17592.132229]     10.0.2.5  --> 8.8.8.8 (UDP)
[17592.132237] *** Dropping 8.8.8.8 (UDP), port 53
[17619.016001] *** LOCAL_OUT
[17619.016002]     10.0.2.5  --> 66.253.214.16 (UDP)
[17627.930395] *** LOCAL_OUT
[17627.930397]     127.0.0.1  --> 127.0.0.1 (UDP)
[17627.930873] *** LOCAL_OUT
[17627.930874]     10.0.2.5  --> 8.8.8.8 (UDP)
[17627.930878] *** Dropping 8.8.8.8 (UDP), port 53
[17632.928720] *** LOCAL_OUT
[17632.928722]     10.0.2.5  --> 8.8.8.8 (UDP)
[17632.928730] *** Dropping 8.8.8.8 (UDP), port 53
[17636.644639] *** LOCAL_OUT
[17636.644781]     10.0.2.5  --> 54.187.101.67 (TCP)
[17636.645099] *** LOCAL_OUT
[17636.645100]     10.0.2.5  --> 54.187.101.67 (TCP)
[17637.933648] *** LOCAL_OUT
[17637.933650]     10.0.2.5  --> 8.8.8.8 (UDP)
[17637.933657] *** Dropping 8.8.8.8 (UDP), port 53
[17705.087088] *** LOCAL_OUT
[17705.087090]     10.0.2.5  --> 91.189.94.4 (UDP)
[17709.030992] *** LOCAL_OUT
[17709.030993]     10.0.2.5  --> 10.0.2.3 (UDP)
[17728.914954] *** LOCAL_OUT
[17728.914956]     10.0.2.5  --> 66.253.214.16 (UDP)
[17728.934318] *** LOCAL_OUT
```

5. remove the filter to avoid crash ($ sudo rmmod seedFilter)

```
[03/24/23]seed@VM:~/.../packet_filter$ sudo rmmod seedFilter
[03/24/23]seed@VM:~/.../packet_filter$ sudo rmmod seedFilter
rmmod: ERROR: Module seedFilter is not currently loaded
[03/24/23]seed@VM:~/.../packet_filter$ ^C
```

Task 1.B.2

I make the follwing modifications:

1. Regist 3 extra hooks
2. For each hook , set printInfo as callback function
3. Add 3 extra removeRules

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/if_ether.h>
#include <linux/inet.h>


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

unsigned int printInfo(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
   struct iphdr *iph;
   char *hook;
   char *protocol;

   switch (state->hook){
     case NF_INET_LOCAL_IN:     hook = "LOCAL_IN";     break;
     case NF_INET_LOCAL_OUT:    hook = "LOCAL_OUT";    break;
     case NF_INET_PRE_ROUTING:  hook = "PRE_ROUTING";  break;
     case NF_INET_POST_ROUTING: hook = "POST_ROUTING"; break;
     case NF_INET_FORWARD:      hook = "FORWARD";      break;
     default:                   hook = "IMPOSSIBLE";   break;
   }
   printk(KERN_INFO "*** %s\n", hook); // Print out the hook info

   iph = ip_hdr(skb);
   switch (iph->protocol){
     case IPPROTO_UDP:  protocol = "UDP";   break;
     case IPPROTO_TCP:  protocol = "TCP";   break;
     case IPPROTO_ICMP: protocol = "ICMP";  break;
     default:           protocol = "OTHER"; break;

   }
   // Print out the IP addresses and protocol
   printk(KERN_INFO "    %pI4  --> %pI4 (%s)\n",
                    &(iph->saddr), &(iph->daddr), protocol);

   return NF_ACCEPT;
}

int registerFilter(void) {
   printk(KERN_INFO "Registering filters.\n");
   hook1.hook=printInfo;
   hook1.hooknum=NF_INET_PRE_ROUTING;
   hook1.pf=PF_INET;
   hook1.priority=NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net,&hook1);
```

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

   hook3.hook=printInfo;
   hook3.hooknum=NF_INET_LOCAL_OUT;
   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_POST_ROUTING;
   hook5.pf=PF_INET;
   hook5.priority=NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net,&hook5);
    return 0;
}

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);


}

module_init(registerFilter);
module_exit(removeFilter);

MODULE_LICENSE("GPL");
```

After re-compile, I **ping 8.8.8.8 -c 1** then use (**$ demsg)** to check the message

```
[21373.101121] *** LOCAL_OUT
[21373.101122]      10.0.2.5  --> 8.8.8.8 (ICMP)
[21373.101129] *** POST_ROUTING
[21373.101130]      10.0.2.5  --> 8.8.8.8 (ICMP)
[21373.121245] *** PRE_ROUTING
[21373.121246]      8.8.8.8  --> 10.0.2.5 (ICMP)
[21373.121255] *** LOCAL_IN
[21373.121255]      8.8.8.8  --> 10.0.2.5 (ICMP)
[03/24/23]seed@VM:~/.../packet_filter$
```

Step 1: The host at 10.0.2.5 generates an ICMP packet, which is then hooked by **LOCAL_OUT**.

Step 2: Since 8.8.8.8 is not on the local network, the host routes the packet to 8.8.8.8, which is then hooked by **POST_ROUTING**.

Step 3: After reaching 8.8.8.8, it sends an ICMP reply to 10.0.2.5, which is then hooked by **PRE_ROUTING**.

Step 4: Since 10.0.2.5 is the host IP address, the packet goes through **LOCAL_IN** and then goes to the host.

Regarding forwarding, when we turn on forwarding, the kernel simply forwards the packet to another interface and sends it out as a router. We can observe that the FORWARDING function will be triggered.

After experimentation, to avoid crashes, remove the seedFilter.

$ sudo rmmod seedFilter

```
[03/24/23]seed@VM:~/.../packet_filter$ sudo rmmod seedFilter
[03/24/23]seed@VM:~/.../packet_filter$ sudo rmmod seedFilter
rmmod: ERROR: Module seedFilter is not currently loaded
[03/24/23]seed@VM:~/.../packet_filter$
```

Task1.B.3

1. Register two hooks on the NF_INET_PRE_ROUTING.
2. Callback function 1: Drop the ICMP packet whose destination is 10.9.0.1 (host).

3. Callback function 2: Drop the TCP packet whose destination is 10.9.0.1 (host).

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/if_ether.h>
#include <linux/inet.h>


static struct nf_hook_ops hook1, hook2;


unsigned int blockICMP(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
   struct iphdr *iph;
   //struct udphdr *udph; // don't care port

   //u16  port   = 53;
   char ip[16] = "10.9.0.1";
   u32  ip_addr; //32-bit 10.9.0.1

   if (!skb) return NF_ACCEPT;

   iph = ip_hdr(skb);
   // Convert the IPv4 address from dotted decimal to 32-bit binary
   in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);

   if (iph->protocol == IPPROTO_ICMP) {
       //udph = udp_hdr(skb);
       //check the destination of the packet, if it is the host(10.9.0.1) drop it
       if (iph->daddr == ip_addr){
            printk(KERN_WARNING "*** Dropping packet %pI4 (ICMP)\n", &(iph->saddr));
            return NF_DROP;
        }
   }
   return NF_ACCEPT;
}


unsigned int blockTelnet(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
   struct iphdr *iph;
   struct tcphdr *tcph;
```

```
   u16  port   = 23;//telnet port
   char ip[16] = "10.9.0.1";
   u32  ip_addr;

   if (!skb) return NF_ACCEPT;

   iph = ip_hdr(skb);
   // Convert the IPv4 address from dotted decimal to 32-bit binary
   in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);
        //telnet is TCP protocol
   if (iph->protocol == IPPROTO_TCP) {
       tcph = tcp_hdr(skb);
       // if the destination ip address is 10.9.0.1 and port is 23 and it is a tcp packet
 drop it.
       if (iph->daddr == ip_addr && ntohs(tcph->dest) == port){
            printk(KERN_WARNING "*** Dropping %pI4 (TCP/Telnet), port %d\n", &(iph->sadd
r), port);
            return NF_DROP;
        }
   }
   return NF_ACCEPT;
}

int registerFilter(void) {
   printk(KERN_INFO "Registering filters.\n");
   hook1.hook=blockICMP;
   hook1.hooknum=NF_INET_PRE_ROUTING;
   hook1.pf=PF_INET;
   hook1.priority=NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net,&hook1);

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

   return 0;
}

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);
}

module_init(registerFilter);
module_exit(removeFilter);

MODULE_LICENSE("GPL");
```

test:

Build the Lab-enviroment, then go to the 10.9.0.5

ping 10.9.0.1 and telnet 10.9.0.1: they all fails

```
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
^C
--- 10.9.0.1 ping statistics ---
9 packets transmitted, 0 received, 100% packet loss, time 8177ms

root@210bb0f432fe:/# telnet 10.9.0.1
Trying 10.9.0.1...
^C
```

Now,go to the VM use **$ dmesg. We can find all packets are sucessfully droped.**

```
Registering filters.
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping packet 10.9.0.5 (ICMP)
*** Dropping 10.9.0.5 (TCP/Telnet), port 23
*** Dropping 10.9.0.5 (TCP/Telnet), port 23
*** Dropping 10.9.0.5 (TCP/Telnet), port 23
```

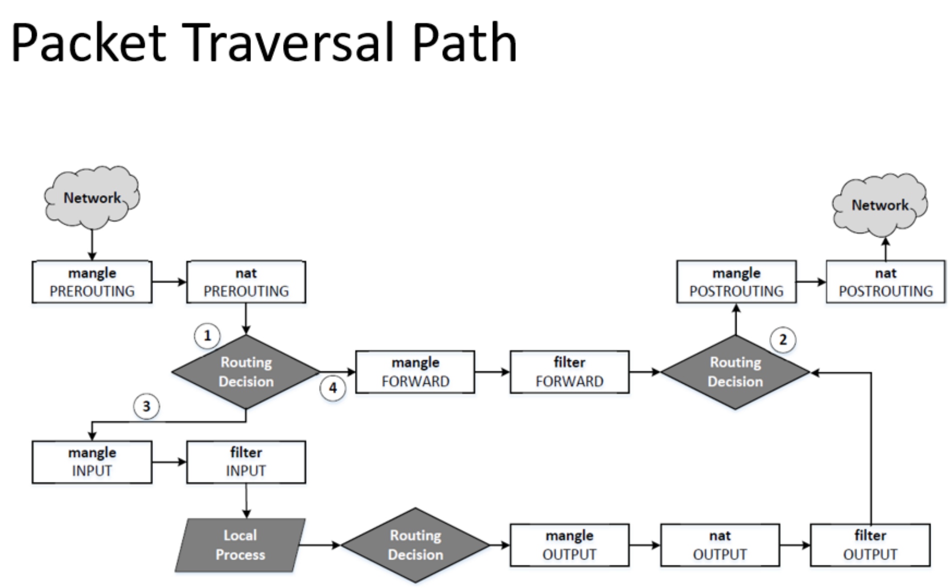
last step remove the filter to avoid crash ($ sudo rmmod seedFilter)

```
[03/24/23]seed@VM:~/.../packet_filter$ sudo rmmod seedFilter
rmmod: ERROR: Module seedFilter is not currently loaded
[03/24/23]seed@VM:~/.../packet_filter$
```

Task 2: Experimenting with Stateless Firewall Rules

▼ Background

Network paket traversal through iptables



iptables and chains

Table	Chain	Functionality
filter	INPUT FORWARD OUTPUT	packet filtering
nat	PREROUTING INPUT OUTPUT POSTROUTING	modify source or destination network address
mangle	PREROUTING INPUT FORWARD OUTPUT POSTROUTING	Packet content modification

iptables general format:

```
iptables -t <table> -<operation> <chain>  <rule>   -j <target>
         ---------- -------------------- -------  -----------
            Table          Chain            Rule      Action
```

Task 2.A: Protecting the Router

1. Set iptables
    a. Add rules let all icmp(ping) packets in/out

    -A INPUT -p icmp --icmp-type echo-request -j ACCEPT:

    defalt table: filter

    Append a rule on INPUT chain: **A** INPUT

    the rule is protocal icmp and some details: **-p icmp** --icmp-type echo-request

    For packet meet the rule take action ACCEPT: **-j ACCEPT**

    -**A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT:**

    defalt table: filter

    Append a rule on **OUTPUT** chain: **A OUTPUT**

    the rule is protocal icmp and some details: **-p icmp --icmp-type echo-reply**

    For packet meet the rule take action ACCEPT: **-j ACCEPT**

    b. other packets all droped (default rule)

    -P OUTPUT DROP

    defalt table: filter

    add default rule on OUTPUT chain, which is drop these unacceptable(other packets except (ICMP) pakcets: **-P OUTPUT DROP**

    -P INPUT DROP:

    defalt table: filter

    add default rule on **INPUT** chain, which is drop these unacceptable(other packets except (ICMP) pakcets: **-P INPUT DROP.**

```
root@a22b48d19234:/# iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
root@a22b48d19234:/# iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
root@a22b48d19234:/# iptables -P OUTPUT DROP
root@a22b48d19234:/# iptables -P INPUT DROP
root@a22b48d19234:/# iptables -L INPUT
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     icmp --  anywhere             anywhere             icmp echo-request
root@a22b48d19234:/# iptables -L OUTPUT
Chain OUTPUT (policy DROP)
target     prot opt source               destination
ACCEPT     icmp --  anywhere             anywhere             icmp echo-reply
```

2. Test:

   a. go to 10.9.0.5 ping 10.9.0.11 (router), we can find ping is **work as normal.**

```
root@280b3db87991:/# 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=0.191 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.043 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.041 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.044 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=64 time=0.042 ms
^C
--- 10.9.0.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4085ms
rtt min/avg/max/mdev = 0.041/0.072/0.191/0.059 ms
```

   b. go to 10.9.0.5 telnet 10.9.0.11 (router), we can find time out here.

```
root@280b3db87991:/# telnet 10.9.0.11
Trying 10.9.0.11...
telnet: Unable to connect to remote host: Connection timed out
root@280b3db87991:/#
```

3. Cleanup:

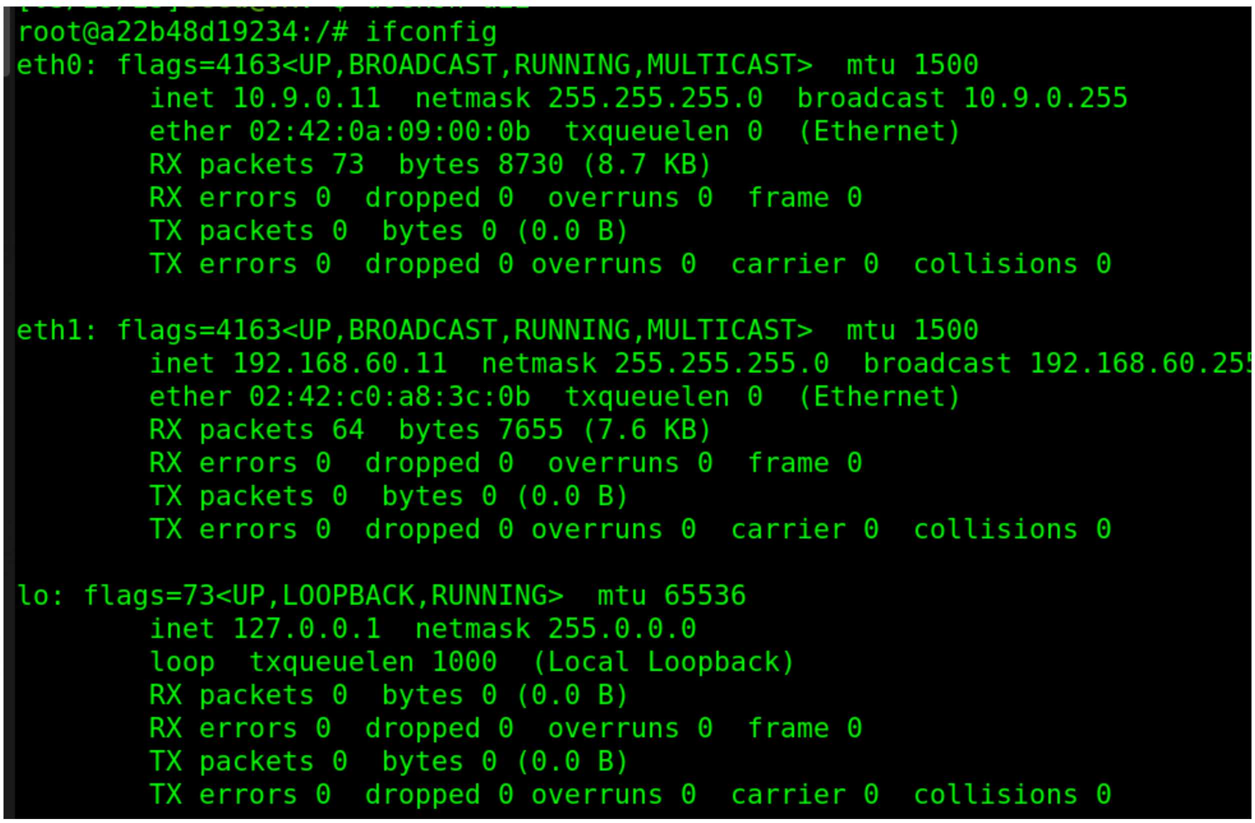
```
root@a22b48d19234:/# iptables -F
root@a22b48d19234:/# iptable -P OUTPUT ACCEPT
bash: iptable: command not found
root@a22b48d19234:/# iptables -P OUTPUT ACCEPT
root@a22b48d19234:/# iptables -P INPUT ACCEPT
root@a22b48d19234:/#
```

Task 2.B: Protecting the Internal Network

(restart the docker, otherwise,with same code, cannot block the telnet)

1. go to router and use ($ ifconfig), we can get the ip for the eh0 and eth1 interface.

interface	ip
eth0(External Network)	10.9.0.11/24
eth1 (Internal Network)	192.168.60.11/24



2. set iptables:

   icmp packet from outside:

   a. A **icmp request** Packet comes from outside (though eth0):

      destionation is the Router (10.9.0.11) accept the packet

iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-request -d 10.9.0.11 -j ACCEPT

destionation is the Internal Net (192.168.60.0/24) drop the packet

iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-request -d 192.168.60.0/24 -j ACCEPT

b. A icmp reply Packets comes from outside (though eth0)

   This means that if we send an ICMP request from the router or from the internal network, we should accept the reply packets.

   destionation is the Router (10.9.0.11), accept the reply packet

   iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-reply -d 10.9.0.11 -j ACCEPT

   destionation is the Internal Net (192.168.60.0/24) accept the reply packet

   iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-reply -d 192.168.60.0/24 -j ACCEPT

icmp packet send out from router or Internal Net:

all of them should be accept:

from router:

iptables -A FORWARD -o eth0 -p icmp --icmp-type echo-request -s 10.9.0.11 -j ACCEPT

from Internal net:

iptables -A FORWARD -o eth0 -p icmp --icmp-type echo-request -s 192.168.60.0/24 -j ACCEPT

other packet :

all drop :

iptables -P FORWARD DROP

3. **test**

   a. outside cannot ping internal host:

   host A (10.9.0.5 ping 192.168.60.5)

```
root@835790efc549:/# ping 192.168.60.5 -c 1
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.

--- 192.168.60.5 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

root@835790efc549:/#
```

   b. outside can ping router

   host A (10.9.0.51 ping 10.9.0.11)

```
--- 10.9.0.11 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.053/0.053/0.053/0.000 ms
root@835790efc549:/#
```

   c. Internal host can ping outside hosts

   host1(192.168.60.5) ping 10.9.0.5

```
root@61f890973a88:/# ping 10.9.0.5 -c 1
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=0.060 ms

--- 10.9.0.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.060/0.060/0.060/0.000 ms
```

d. All other packets should be blocked

Internal host cannot telnet outside hosts

```
root@61f890973a88:/# telnet 10.9.0.5
Trying 10.9.0.5...
telnet: Unable to connect to remote host: Connection timed out
root@61f890973a88:/#
```

Task 2.C: Protecting Internal Servers

(restart the docker, otherwise,with same code, cannot block the telnet)

1. go to router and use ($ ifconfig), we can get the ip for the eh0 and eth1 interface.

interface	ip
eth0(External Network)	10.9.0.11/24
eth1 (Internal Network)	192.168.60.11/24

```
root@a22b48d19234:/# ifconfig
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 73  bytes 8730 (8.7 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

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.60.11  netmask 255.255.255.0  broadcast 192.168.60.25
        ether 02:42:c0:a8:3c:0b  txqueuelen 0  (Ethernet)
        RX packets 64  bytes 7655 (7.6 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

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        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
```

2. **set iptables:**

iptables -n -L FORWARD

Since Telnet is a protocol that uses TCP (Transmission Control Protocol) as its underlying transport protocol, it will reply **packet to the external host** which connect the server.

Based on the requirement, the host (192.168.60.5/port 23) is the only “hole”, that we can communicate with outside. so:

telnet packets from outside:

iptables -A FORWARD -i eth0 -p tcp --dport 23 -d 192.168.60.5 -j ACCEPT

telnet packets from inside(server reply):

iptables -A FORWARD -o eth0 -p tcp --sport 23 -s 192.168.60.5 -j ACCEPT

all other cases drop:

iptables -P FORWARD DROP

test:

1. **Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts:**

   From hostA(10.9.0.5) telnet 192.168.60.5 , 192.168.60.6, 192.168.60.7 we can get **only 192.168.60.5 can get connet.**

```
root@050d980a306a:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
```

```
root@050d980a306a:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
root@050d980a306a:/# telnet 192.168.60.7
Trying 192.168.60.7...
^C
root@050d980a306a:/#
```

2. **Outside hosts cannot access other internal servers**

```
root@050d980a306a:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
root@050d980a306a:/# telnet 192.168.60.7
Trying 192.168.60.7...
^C
root@050d980a306a:/#
```

3.Internal hosts can access all the internal servers.

host1:192.168.60.5

```
root@eae90bf44007:/# telnet 192.168.60.6
Trying 192.168.60.6...
Connected to 192.168.60.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
715410c47d55 login: Connection closed by foreign host.
root@eae90bf44007:/# telnet 192.168.60.7
Trying 192.168.60.7...
Connected to 192.168.60.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
524ba12ba54c login: ^CConnection closed by foreign host.
root@eae90bf44007:/#
```

host2: 192.168.60.6

```
oot@715410c47d55:/# telnet 192.168.60.5
rying 192.168.60.5...
onnected to 192.168.60.5.
scape character is '^]'.
buntu 20.04.1 LTS
ae90bf44007 login: ^CConnection closed by foreign host.
oot@715410c47d55:/# telnet 192.168.60.7
rying 192.168.60.7...
onnected to 192.168.60.7.
scape character is '^]'.
buntu 20.04.1 LTS
24ba12ba54c login: ^CConnection closed by foreign host.
oot@715410c47d55:/#
```

host3: 192.168.60.7

```
[03/25/23]seed@VM:~$ st host3/192.168.60.7
[03/25/23]seed@VM:~$ telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
^C^C



Ubuntu 20.04.1 LTS




eae90bf44007 login: ^CConnection closed by foreign host.
[03/25/23]seed@VM:~$ telnet 192.168.60.6
Trying 192.168.60.6...
Connected to 192.168.60.6.
Escape character is '^]'.
```

4.Internal hosts cannot access external servers

host1:192.168.60.5

```
root@715410c47d55:/# ^C
root@715410c47d55:/# telnet 10.9.0.5
Trying 10.9.0.5...
^C
root@715410c47d55:/#
```

```
root@eae90bf44007:/# ^C
root@eae90bf44007:/# telnet 10.9.0.5
Trying 10.9.0.5...
^C
```

Task 3: Connection Tracking and Stateful Firewall

Task 3.A: Experiment with the Connection Tracking

ICMP experiment

go to 10.9.0.5 ping 192.168.60.5

Then $ conntrack -L

```
oot@ec585d128fa0:/#  conntrack -L
cmp     1 27 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=33 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=33 mark=0 use=1
onntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

The number "1" represents the protocol number for the connection tracking entry

The number "27" represents the timeout value(27s) for the connection tracking entry.

This entry represents an ICMP ping (type 8) request sent from source IP address 192.168.60.11 to destination IP address 192.168.60.5, with a packet identifier (ID) of 33. The entry also shows the reply packet (type 0) sent from 192.168.60.5 back to 192.168.60.11 with the same ID.

The `mark` field in the entry is set to 0, indicating that no special marking has been applied to the connection.

The `use` field shows that this connection tracking entry has been used once, which means that the connection has been established and closed.

UDP experiment

```
// On 192.168.60.5, start a netcat UDP server
# nc -lu 9090
// On 10.9.0.5, send out UDP packets
# nc -u 192.168.60.5 9090
<type something, then hit return>
```

go to 10.9.0.5 Then $ conntrack -L

```
oot@ec585d128fa0:/#  conntrack -L
udp      17 6 src=192.168.60.11 dst=192.168.60.5 sport=34412 dport=9090 [UNREPLIED] src=192.168.60.5 dst=192.168.60.11 sport=9090 dport=34412
 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
oot@ec585d128fa0:/# ^C
oot@ec585d128fa0:/# ^C
oot@ec585d128fa0:/#
```

The number "17" represents the protocol number for the connection tracking entry

The number "6" represents the timeout value(6s) for the connection tracking entry.

The flow entry is a UDP connection between two IP addresses: 192.168.60.11 and 192.168.60.5.

The source IP address (src) is 192.168.60.11, and the destination IP address (dst) is 192.168.60.5. The source port (sport) is 34412, and the destination port (dport) is 9090. The connection is currently in an **UNREPLIED** state, which means that the connection has been initiated but no response has been received yet.

The reverse connection from 192.168.60.5 to 192.168.60.11 is also shown, with the source and destination IP addresses and port numbers reversed. The mark and use fields are additional metadata about the connection tracking entry.

TCP experiment

```
// On 192.168.60.5, start a netcat TCP server
# nc -l 9090
// On 10.9.0.5, send out TCP packets
# nc 192.168.60.5 9090
<type something, then hit return>
```

```
^C
root@ec585d128fa0:/#  conntrack -L
tcp      6 117 TIME_WAIT src=192.168.60.11 dst=192.168.60.5 sport=45390 dport=9090 src=192.168.60.5 dst=192.168.60.11 sport=9090 dport=45390
[ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@ec585d128fa0:/#
```

the `tcp 6 117 TIME_WAIT` part indicates that the flow entry is for a TCP (Transmission Control Protocol) connection, with protocol number 6. The connection is currently in the TIME_WAIT state, which is a normal part of the TCP connection termination process.

The source IP address (src) is 192.168.60.11, and the destination IP address (dst) is 192.168.60.5. The source port (sport) is 45390, and the destination port (dport) is 9090. The reverse connection from 192.168.60.5 to 192.168.60.11 is also shown, with the source and destination IP addresses and port numbers reversed.

The `[ASSURED]` tag indicates that this flow entry has been marked as assured, which means that the connection has been seen passing in both directions and is considered to be an active connection. The `mark` field is used for setting up firewall rules based on the connection, and the `use` field indicates the number of packets that have been seen for the connection.

The `117` value represents the timeout value for this connection in seconds. In this case, the timeout value is 117 seconds, which means that if no new packets are seen for this connection within the next 117 seconds, it will be removed from the connection tracking table.

Task 3.B: Setting Up a Stateful Firewall

▼ ctstate

`--ctstate` option is used with the `conntrack` command to filter the output based on the connection state of the network connections being tracked.

NEW	The connection is in the process of being established
ESTABLISHED	The connection is established and data can flow between the hosts
RELATED	The connection is related to another connection that has already been established.
INVALID	The connection is in an invalid state
UNTRACKED	The connection is untracked by the conntrack module
SNAT	The connection is being SNATed (Source Network Address

	Translation)
DNAT	The connection is being DNATed (Destination Network Address Translation)
SNAT|DNAT	The connection is being both SNATed and DNATed

1. **set iptables:**

if(tcp)

{

if(not establish telnet connection)

{

1. **telnet packets from outside (eth0):**

   Accept incoming SYN packets to 192.168.60.5, it only allows to telnet 192.168.60.5::23

```
iptables -A FORWARD -p tcp -i eth0 -d 192.168.60.5 --dport 23 --syn -m con
ntrack --ctstate NEW -j ACCEPT
```

2. **telnet packet from inside (eth1):**

   accept all SYN packets from internal to outside in telnet way, it allows internal to access external.

```
iptables -A FORWARD -p tcp -i eth1 --dport 23 --syn -m conntrack --ct
state NEW -j ACCEPT
```

   After router to eth0, send out.

```
iptables -A FORWARD -p tcp -o eth0 --dport 23 --syn -m conntrack --ct
state NEW -j ACCEPT
```

}

else //build connection

{

accept TCP packets belonging to an existing connection, it allows existing telnet to communicate.

```
iptables -A FORWARD -p tcp -m conntrack --ctstate ESTABLISHED,RELATED
 -j ACCEPT
```

}

}

else (not tcp)

{

all drop

```
iptables -P FORWARD DROP
```

}

Final rules:

```
root@ec585d128fa0:/# iptables -n -L FORWARD --line-number
Chain FORWARD (policy DROP)
num  target     prot opt source               destination
1    ACCEPT     tcp  --  0.0.0.0/0            192.168.60.5         tcp dpt:23 flags:0x17/0x02 ctstate NEW
2    ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:23 flags:0x17/0x02 ctstate NEW
3    ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0            ctstate RELATED,ESTABLISHED
4    ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:23 flags:0x17/0x02 ctstate NEW
root@ec585d128fa0:/#
```

2. test:

1.**Outside can only access the telnet server on 192.168.60.5**

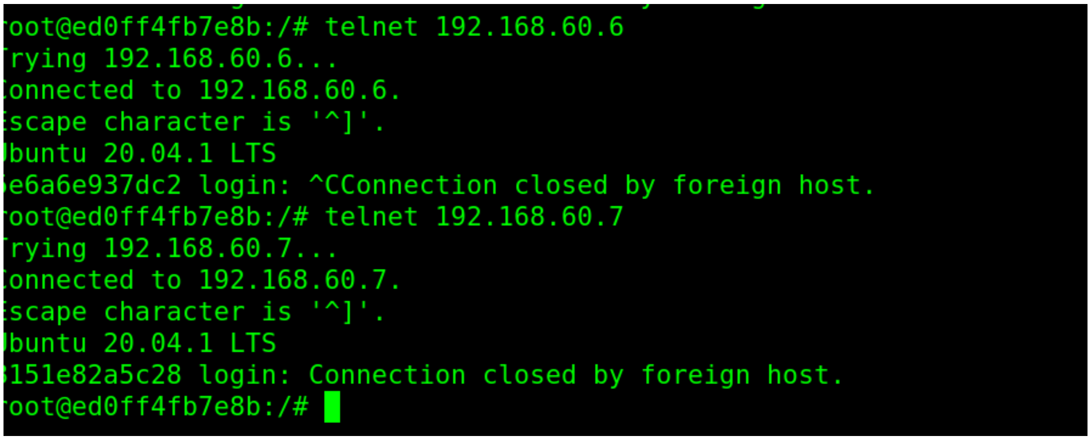
```
root@858e26ec3ff0:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
^M
Ubuntu 20.04.1 LTS


ed0ff4fb7e8b login: ^CConnection closed by foreign host.
root@858e26ec3ff0:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
root@858e26ec3ff0:/# telnet 192.168.60.7
Trying 192.168.60.7...
^C
root@858e26ec3ff0:/# 
```

2.**Outside can not access the telnet server on 192.168.60.6 or 192.168.60.6**

```
root@858e26ec3ff0:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
^M
Ubuntu 20.04.1 LTS


ed0ff4fb7e8b login: ^CConnection closed by foreign host.
root@858e26ec3ff0:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
root@858e26ec3ff0:/# telnet 192.168.60.7
Trying 192.168.60.7...
^C
root@858e26ec3ff0:/# 
```

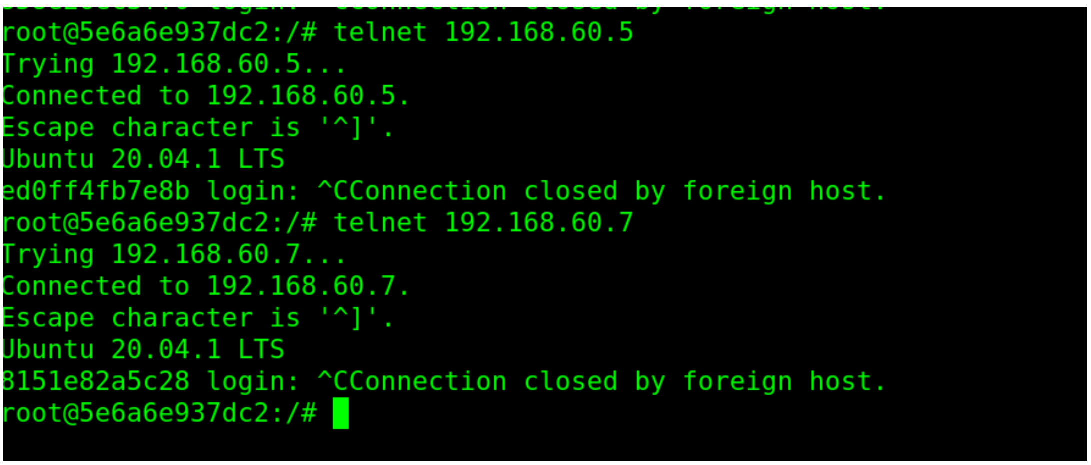
3.Internal host can access all interal servers:

192.168.60.5:



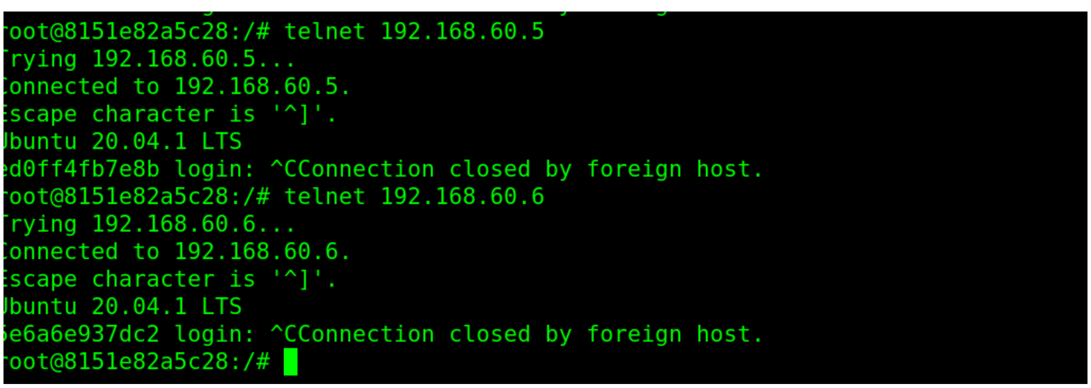
```
oot@ed0ff4fb7e8b:/# telnet 192.168.60.6
rying 192.168.60.6...
onnected to 192.168.60.6.
scape character is '^]'.
buntu 20.04.1 LTS
e6a6e937dc2 login: ^CConnection closed by foreign host.
oot@ed0ff4fb7e8b:/# telnet 192.168.60.7
rying 192.168.60.7...
onnected to 192.168.60.7.
scape character is '^]'.
buntu 20.04.1 LTS
151e82a5c28 login: Connection closed by foreign host.
oot@ed0ff4fb7e8b:/#
```

192.168.60.6:



```
root@5e6a6e937dc2:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
ed0ff4fb7e8b login: ^CConnection closed by foreign host.
root@5e6a6e937dc2:/# telnet 192.168.60.7
Trying 192.168.60.7...
Connected to 192.168.60.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
8151e82a5c28 login: ^CConnection closed by foreign host.
root@5e6a6e937dc2:/#
```

192.168.60.7:



```
oot@8151e82a5c28:/# telnet 192.168.60.5
rying 192.168.60.5...
onnected to 192.168.60.5.
scape character is '^]'.
buntu 20.04.1 LTS
d0ff4fb7e8b login: ^CConnection closed by foreign host.
oot@8151e82a5c28:/# telnet 192.168.60.6
rying 192.168.60.6...
onnected to 192.168.60.6.
scape character is '^]'.
buntu 20.04.1 LTS
e6a6e937dc2 login: ^CConnection closed by foreign host.
oot@8151e82a5c28:/#
```

4.internal can cess all exteral servers

192.168.60.5:

```
root@ed0ff4fb7e8b:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
858e26ec3ff0 login:
```

192.168.60.6:

```
root@5e6a6e937dc2:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
858e26ec3ff0 login:
```

192.168.60.7:

```
root@8151e82a5c28:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
858e26ec3ff0 login:
```

analysis of connection tracking mechanism vs staleless mechanism

reference:https://www.cdw.com/content/cdw/en/articles/security/stateful-versus-stateless-firewalls.html

The advantage of using connection tracking mechanism is that it simplifies the firewall rules and allows the firewall to automatically keep track of the connection state for each connection. This means that the firewall can dynamically allow or block traffic based on the connection state without having to explicitly specify rules for each connection. This approach is also more efficient as it reduces the number of rules needed to manage the firewall.

The disadvantage of using connection tracking mechanism is that it can potentially introduce security risks if the connection state is not properly managed.

The advantage of staleless mechanism is that it provides more fine-grained control over the traffic that is allowed or blocked. This approach is also more secure as it avoids the potential risks associated with connection tracking.

The disadvantage of staleless mechanism is that it can be more complex to manage, especially for larger networks with many connections. It also requires more explicit rules to be specified, which can increase the risk of errors and misconfigurations.

Task 4: Limiting Network Traffific

The first command adds a rule to the FORWARD chain that allows traffic from the source IP address 10.9.0.5 to pass through the firewall, but limits it to 10 packets per minute with a burst of up to 5 packets. This is done using the "-m limit" option, which allows you to set packet rate limits

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

The second command adds a rule to the same chain that drops all traffic from the same source IP address.

```
iptables -A FORWARD -s 10.9.0.5 -j DROP
```

With the second rule

Go to 10.9.0.5 and ping 192.168.60.5 -c 20.

We found 60% packet loss. The reason is that only 10 packets are allowed to pass per minute. For these "selected" packets, we let them continue to go, then we can get an icmp_reply. For these "unselected" packets, we can use the second rule, **iptables -A FORWARD -s 10.9.0.5 -j DROP**, to drop the packet.

```
[03/25/23]seed@VM:~$ docksh 166
root@166396e7625f:/# ping 192.168.60.5 -c 20
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=0.147 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.053 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.052 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.051 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.053 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.051 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.052 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=63 time=0.052 ms

--- 192.168.60.5 ping statistics ---
20 packets transmitted, 8 received, 60% packet loss, time 19496ms
rtt min/avg/max/mdev = 0.051/0.063/0.147/0.031 ms
```

Without the second rule

Delete the second rule, then go to 10.9.0.5 and ping 192.168.60.5 -c 20.

We found 0% packet loss. The reason is that only 10 packets are allowed to pass per minute. For the "selected" packets, we let them continue to go, and we can get an icmp_reply. For the "unselected" packets, we use the default rule, which is to accept them. So these "unselected" packets can continue to go and reach the destination.

```
--- 192.168.60.5 ping statistics ---
20 packets transmitted, 8 received, 60% packet loss, time 19448ms
rtt min/avg/max/mdev = 0.050/0.056/0.079/0.008 ms
root@166396e7625f:/# ping 192.168.60.5 -c 20
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=0.070 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.050 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.073 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.055 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.051 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=0.051 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.150 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=0.063 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=0.051 ms
64 bytes from 192.168.60.5: icmp_seq=10 ttl=63 time=0.048 ms
64 bytes from 192.168.60.5: icmp_seq=11 ttl=63 time=0.049 ms
64 bytes from 192.168.60.5: icmp_seq=12 ttl=63 time=0.050 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.051 ms
64 bytes from 192.168.60.5: icmp_seq=14 ttl=63 time=0.051 ms
64 bytes from 192.168.60.5: icmp_seq=15 ttl=63 time=0.048 ms
64 bytes from 192.168.60.5: icmp_seq=16 ttl=63 time=0.056 ms
64 bytes from 192.168.60.5: icmp_seq=17 ttl=63 time=0.055 ms
64 bytes from 192.168.60.5: icmp_seq=18 ttl=63 time=0.050 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=63 time=0.049 ms
64 bytes from 192.168.60.5: icmp_seq=20 ttl=63 time=0.067 ms

--- 192.168.60.5 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 19466ms
rtt min/avg/max/mdev = 0.048/0.059/0.150/0.022 ms
root@166396e7625f:/#
```

Task 5: Load Balancing

Using the nth mode (round-robin)

The following three rules aim to ensure that all three internal hosts receive an equal number of packets using the nth mode.

The first rule picks the first packet out of every three packets whose destination port is 8080, and changes its destination to 192.168.60.5:8080.

```
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
```

Since the first rule has already picked the first packet, the second rule only needs to consider the remaining two packets. The second rule picks the first packet out of every two packets whose destination port is 8080, and changes its destination to 192.168.60.6:8080.

```
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
```

The third rule picks all packets whose destination port is 8080 and changes their destination to 192.168.60.7:8080. Since the first two rules have already picked the first two packets in a round, the last packet must go through this rule. Therefore, it only needs to pick up all packets whose destination port is 8080.

```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 1 --packet 0 -j DNAT --to-destination 192.168.60.7:8080
```

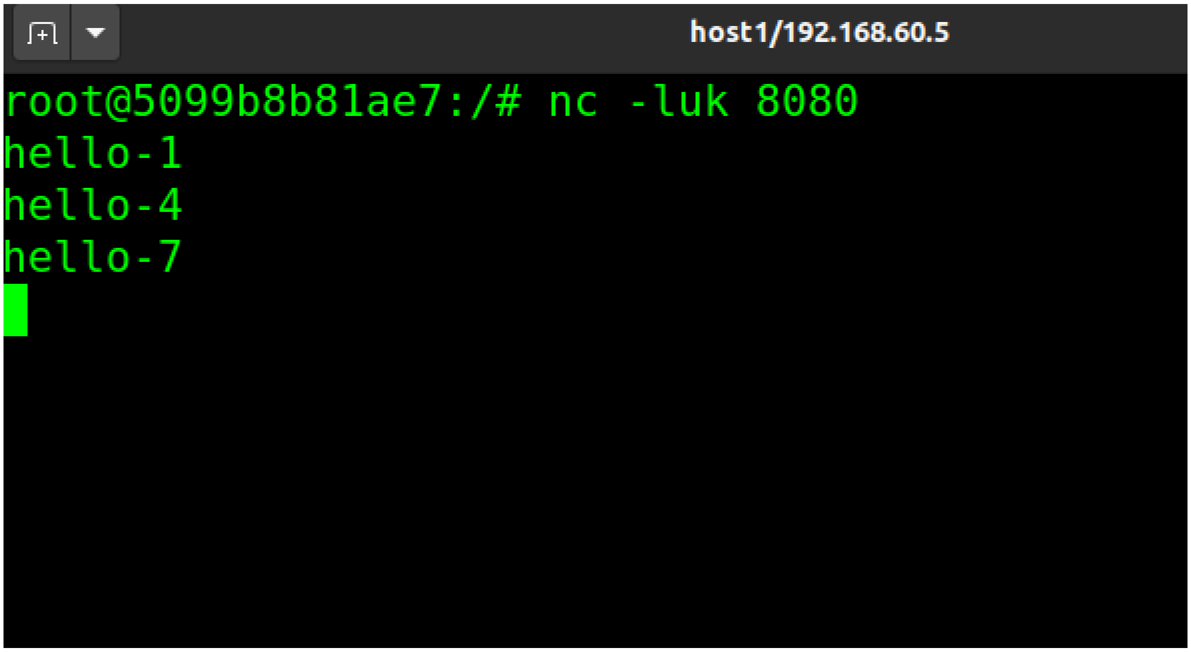
TEST:

I sent 9 packets to 10.9.0.11:8080. We can see that each of host 1, 2, and 3 received the packets in a round-robin fashion.

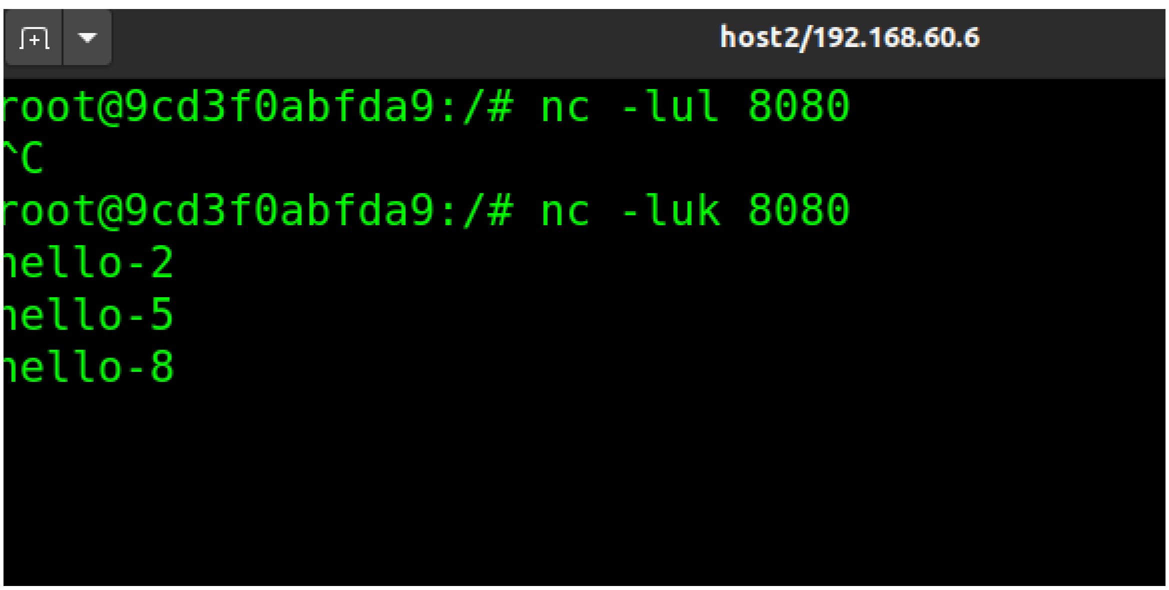
hostA: 10.9.0.5

```
root@a3b0fa271a15:/# echo hello-1 | nc -u 10.9.0.11 8080
^C
root@a3b0fa271a15:/# echo hello-2 | nc -u 10.9.0.11 8080
^C
root@a3b0fa271a15:/# echo hello-3 | nc -u 10.9.0.11 8080
^C
root@a3b0fa271a15:/# echo hello-4 | nc -u 10.9.0.11 8080
^C
root@a3b0fa271a15:/# echo hello-5 | nc -u 10.9.0.11 8080
^C
root@a3b0fa271a15:/# echo hello-6 | nc -u 10.9.0.11 8080
^C
root@a3b0fa271a15:/# echo hello-7 | nc -u 10.9.0.11 8080
^C
root@a3b0fa271a15:/# echo hello-8 | nc -u 10.9.0.11 8080
^C
root@a3b0fa271a15:/# echo hello-9 | nc -u 10.9.0.11 8080
^C
```

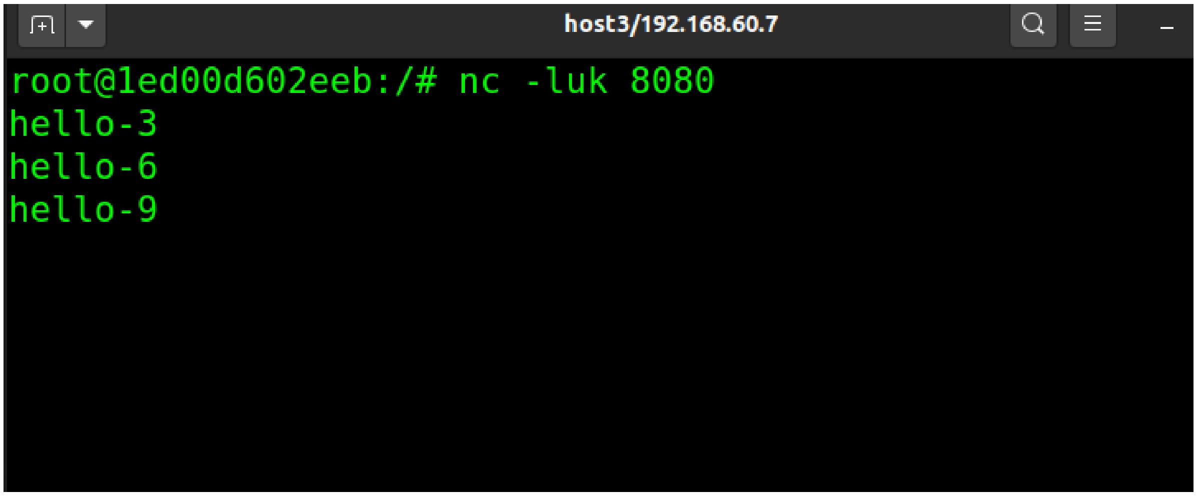
host 1: 192.168.60.5



host 2: 192.168.60.6



host3: 192.168.60.7



Using the random mode

To distribute traffic equally among servers, set the overall probability to 1/3. The first rule is to select a matching packet with a 0.33 probability.

```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probab
ility
0.33 -j DNAT --to-destination 192.168.60.5:8080
```

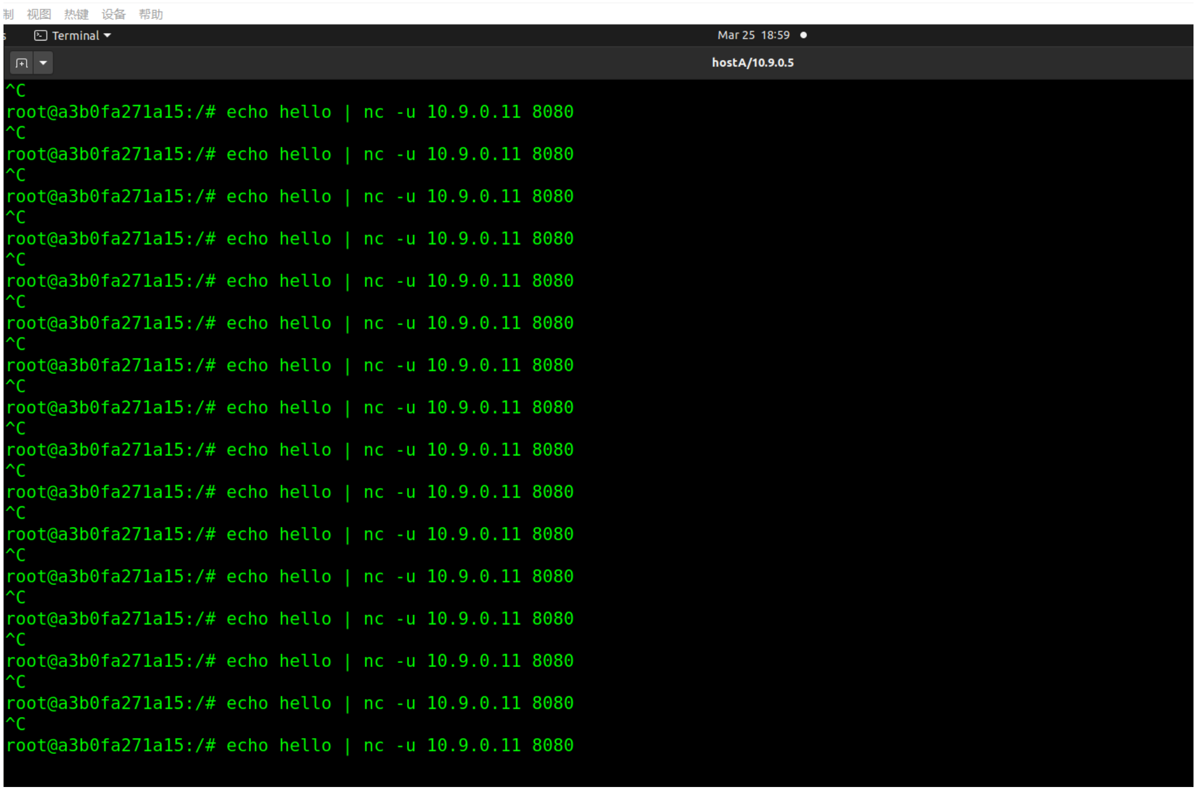
To ensure equal traffic distribution among servers, set the overall probability to 1/3. This leaves 2/3 probability remaining after the first rule. To achieve 1/3 probability, set the probability to 1/2, as 2/3 * 1/2 = 1/3.

```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probab
ility 0.5
-j DNAT --to-destination 192.168.60.6:8080
```

After applying the first and second rules, the probability of left is 1/3. So we set the probability to 1.

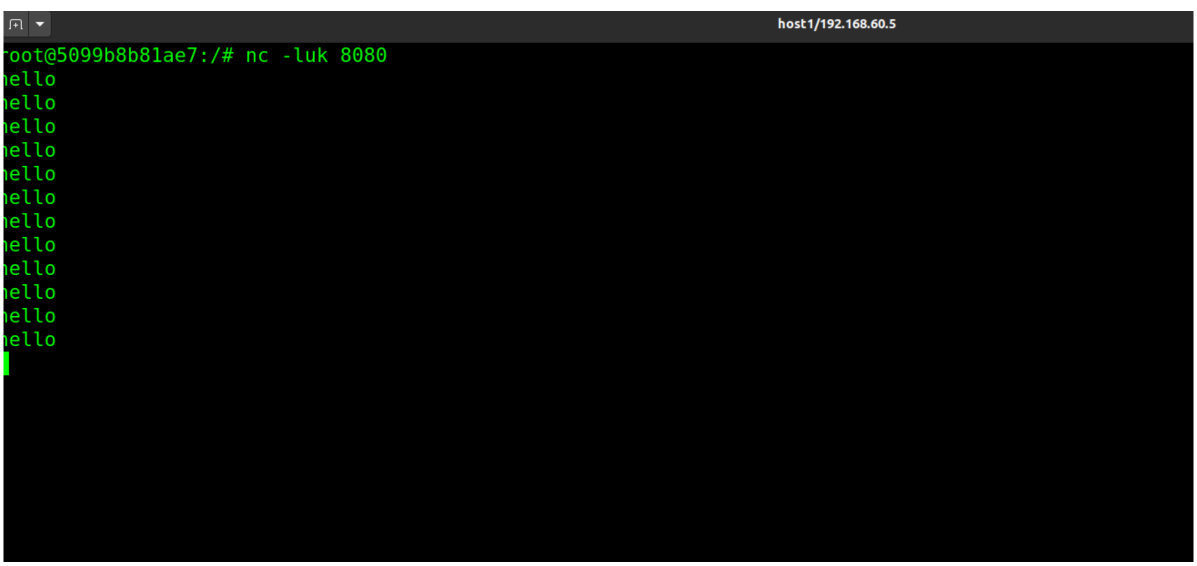
```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probab
ility 1
-j DNAT --to-destination 192.168.60.7:8080
```

10.9.0.5: I sent 47 packets to 10.9.0.11. The possibility of reaching the destination is equal for all three servers, indicating that the rule is working as expected.



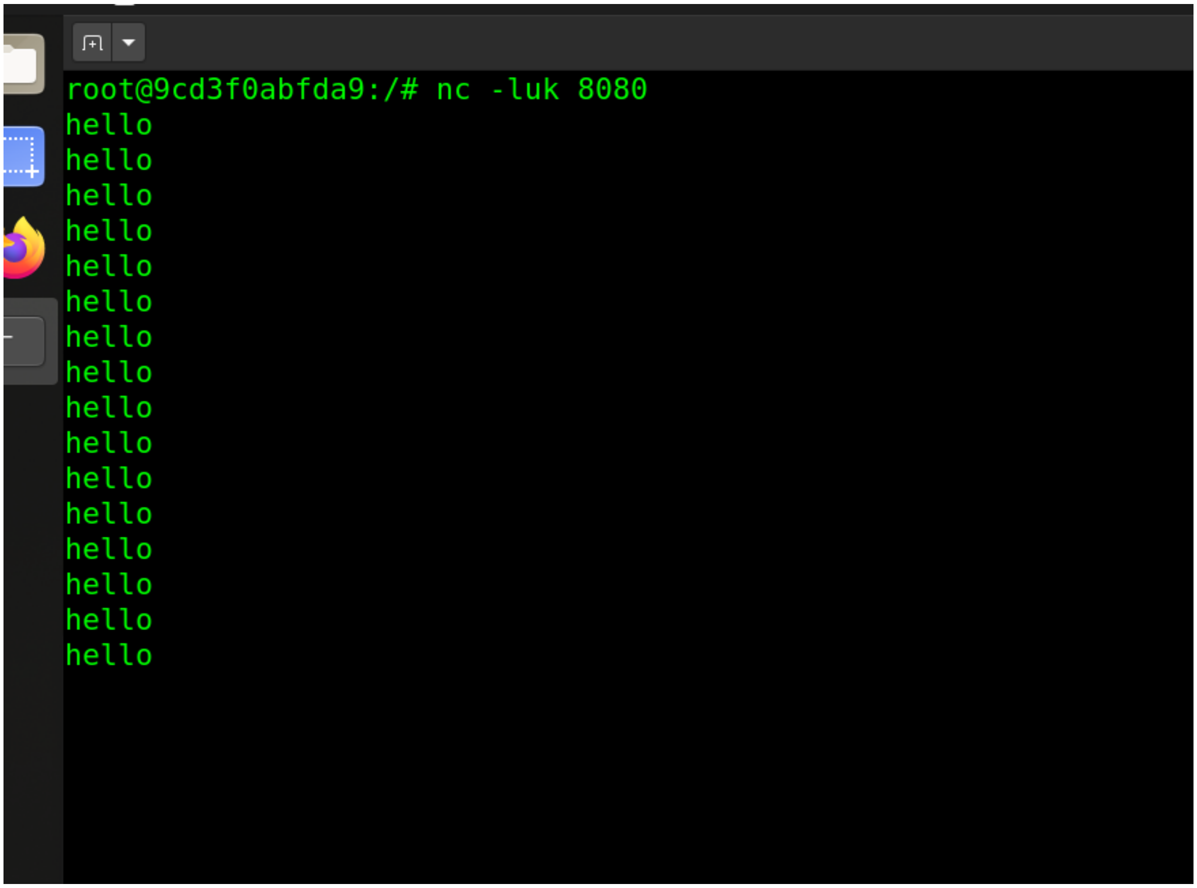
192.168.60.5:

12/47around to 28%



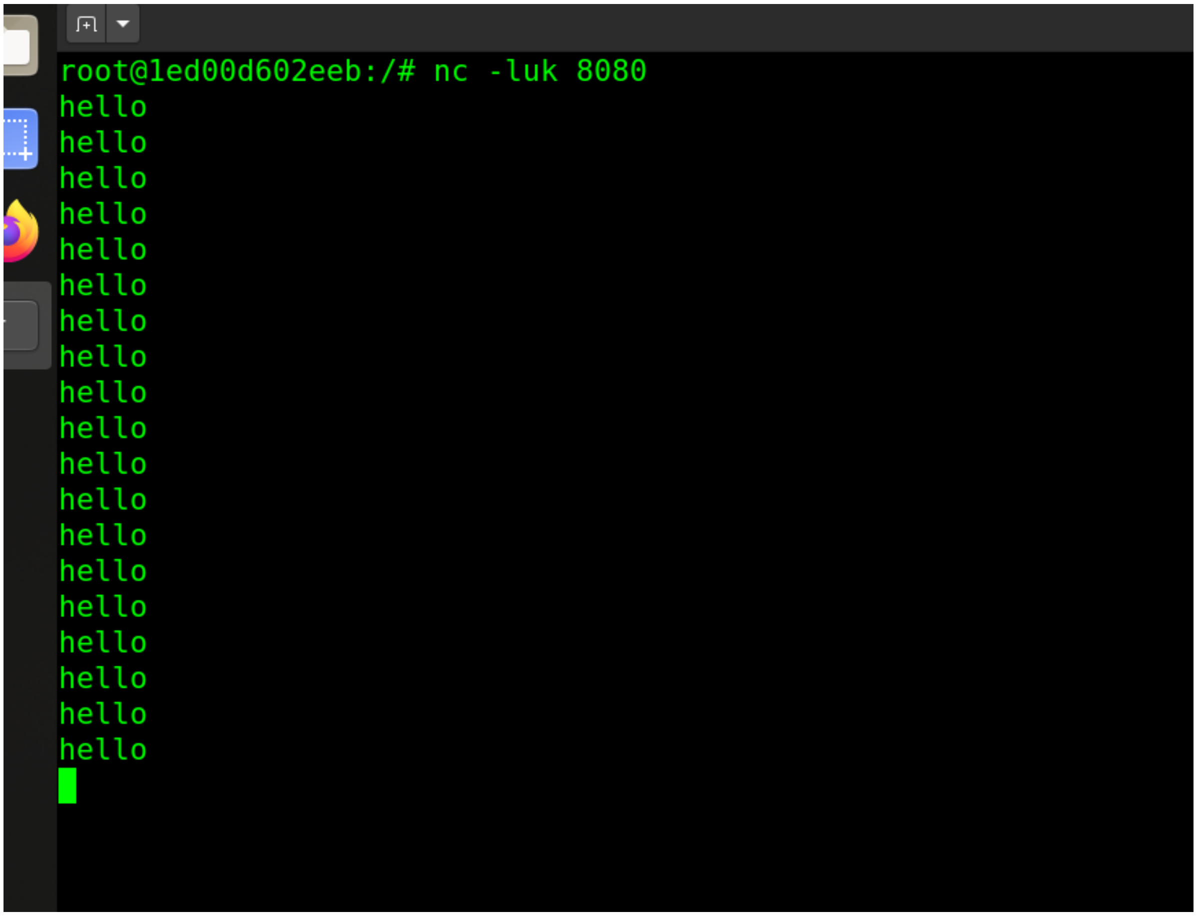
192.168.60.6:

16/47 around to 35%



192.168.60.7:

19/47 arount to 37%


root@1ed00d602eeb:/# nc -luk 8080
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello