Ethical Hacking: Firewall Security Lab - Report

Authors: David Polzoni (2082157) & Eugenio Caripoti (2079454)

Date: 26/10/2023, A.Y. 2023-2024

Task 1.A: Implement a Simple Kernel Module

Code snippet to implement basic example of a Linux kernel module:

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

int initialization(void)
{
    printk(KERN_INFO "Hello World!\n");
    return 0;
}

void cleanup(void)
{
    printk(KERN_INFO "Bye-bye World!.\n");
}

module_init(initialization);
module_exit(cleanup);
```

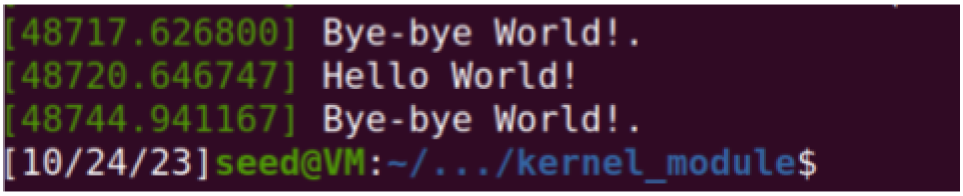
Makefile to compile `hello.c`:

```
obj-m += hello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Bash commands to manage KLM:

```
$ sudo insmod hello.ko # inserting a module
$ lsmod | grep hello # list modules
$ sudo rmmod hello # remove the module
$ dmesg # check the messages
```

Result of the experiment:



```
[48717.626800] Bye-bye World!.
[48720.646747] Hello World!
[48744.941167] Bye-bye World!.
[10/24/23]seed@VM:~/.../kernel_module$
```

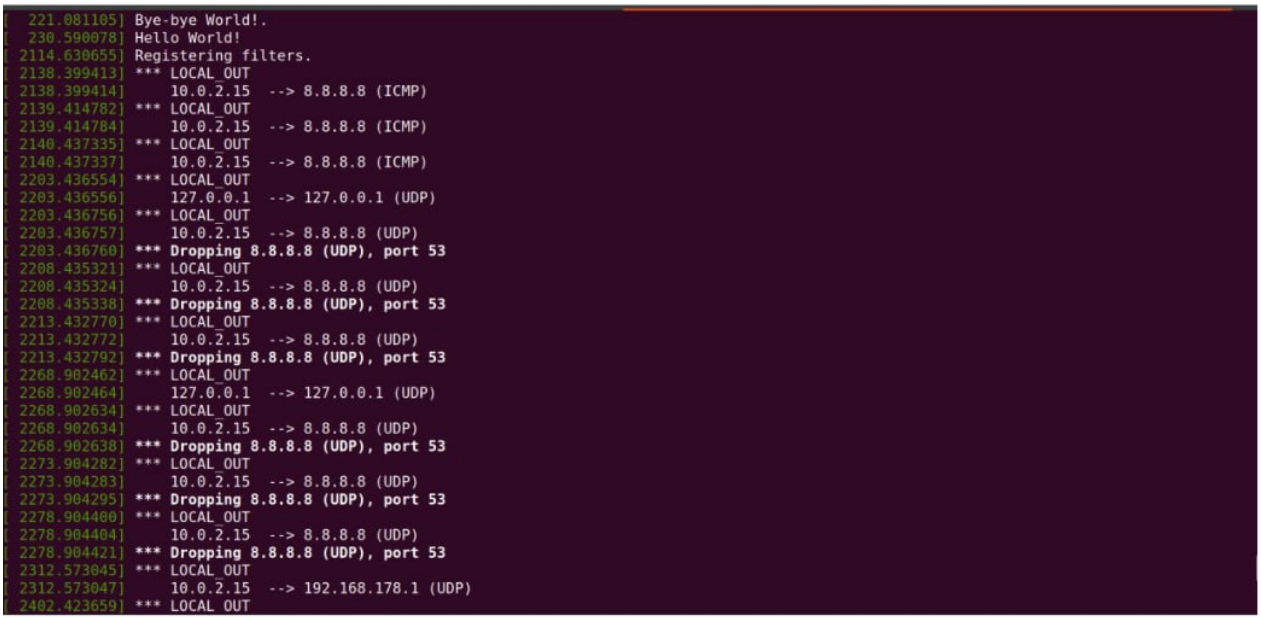
Task 1.B: Implement a Simple Firewall Using Netfilter

1. Compile the sample code using the provided Makefile. Load it into the kernel, and demonstrate that the firewall is working as expected. You can use the following command to generate UDP packets to 8.8.8.8, which is Google's DNS server. If your firewall works, your request will be blocked; otherwise, you will get a response. dig @8.8.8.8 www.example.com

Code snippet to filter **UDP traffic**:

```
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;

   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_UDP) {
       udph = udp_hdr(skb);
       if (iph->daddr == ip_addr && ntohs(udph->dest) == port){
            printk(KERN_WARNING "*** Dropping %pI4 (UDP), port %d\n", &(iph->daddr), port);
            return NF_DROP;
        }
   }
   return NF_ACCEPT;
}
```

Result of the experiment:



```
[  221.081105] Bye-bye World!.
[  230.590078] Hello World!
[ 2114.630655] Registering filters.
[ 2138.399413] *** LOCAL_OUT
[ 2138.399414]     10.0.2.15  --> 8.8.8.8 (ICMP)
[ 2139.414782] *** LOCAL_OUT
[ 2139.414784]     10.0.2.15  --> 8.8.8.8 (ICMP)
[ 2140.437335] *** LOCAL_OUT
[ 2140.437337]     10.0.2.15  --> 8.8.8.8 (ICMP)
[ 2203.436554] *** LOCAL_OUT
[ 2203.436556]     127.0.0.1  --> 127.0.0.1 (UDP)
[ 2203.436756] *** LOCAL_OUT
[ 2203.436757]     10.0.2.15  --> 8.8.8.8 (UDP)
[ 2203.436760] *** Dropping 8.8.8.8 (UDP), port 53
[ 2208.435321] *** LOCAL_OUT
[ 2208.435324]     10.0.2.15  --> 8.8.8.8 (UDP)
[ 2208.435338] *** Dropping 8.8.8.8 (UDP), port 53
[ 2213.432770] *** LOCAL_OUT
[ 2213.432772]     10.0.2.15  --> 8.8.8.8 (UDP)
[ 2213.432792] *** Dropping 8.8.8.8 (UDP), port 53
[ 2268.902462] *** LOCAL_OUT
[ 2268.902464]     127.0.0.1  --> 127.0.0.1 (UDP)
[ 2268.902634] *** LOCAL_OUT
[ 2268.902634]     10.0.2.15  --> 8.8.8.8 (UDP)
[ 2268.902638] *** Dropping 8.8.8.8 (UDP), port 53
[ 2273.904282] *** LOCAL_OUT
[ 2273.904283]     10.0.2.15  --> 8.8.8.8 (UDP)
[ 2273.904295] *** Dropping 8.8.8.8 (UDP), port 53
[ 2278.904400] *** LOCAL_OUT
[ 2278.904404]     10.0.2.15  --> 8.8.8.8 (UDP)
[ 2278.904421] *** Dropping 8.8.8.8 (UDP), port 53
[ 2312.573045] *** LOCAL_OUT
[ 2312.573047]     10.0.2.15  --> 192.168.178.1 (UDP)
[ 2402.423659] *** LOCAL_OUT
```

2. Using your experiment results to help explain at what condition will **each of the hook function be invoked**.

When a network packet is recieved by a network device, it initially goes through the Prerouting hook. This is where the routing decision is made. The kernel determines whether the packet is intended for a local process, such as a listening socket on the system, or whether it should be forwarded as if the system is functioning as a router. In the first scenario, the packet proceeds through the Input hook and is then delivered to the local process. If the packet is meant for forwarding, it goes through the Forward hook and subsequently the Postrouting hook before being transmitted via a network device. For packets generated locally, like those originating from a client or server process, they must first pass through the Output hook and then the Postrouting hook before being dispatched via a network device.

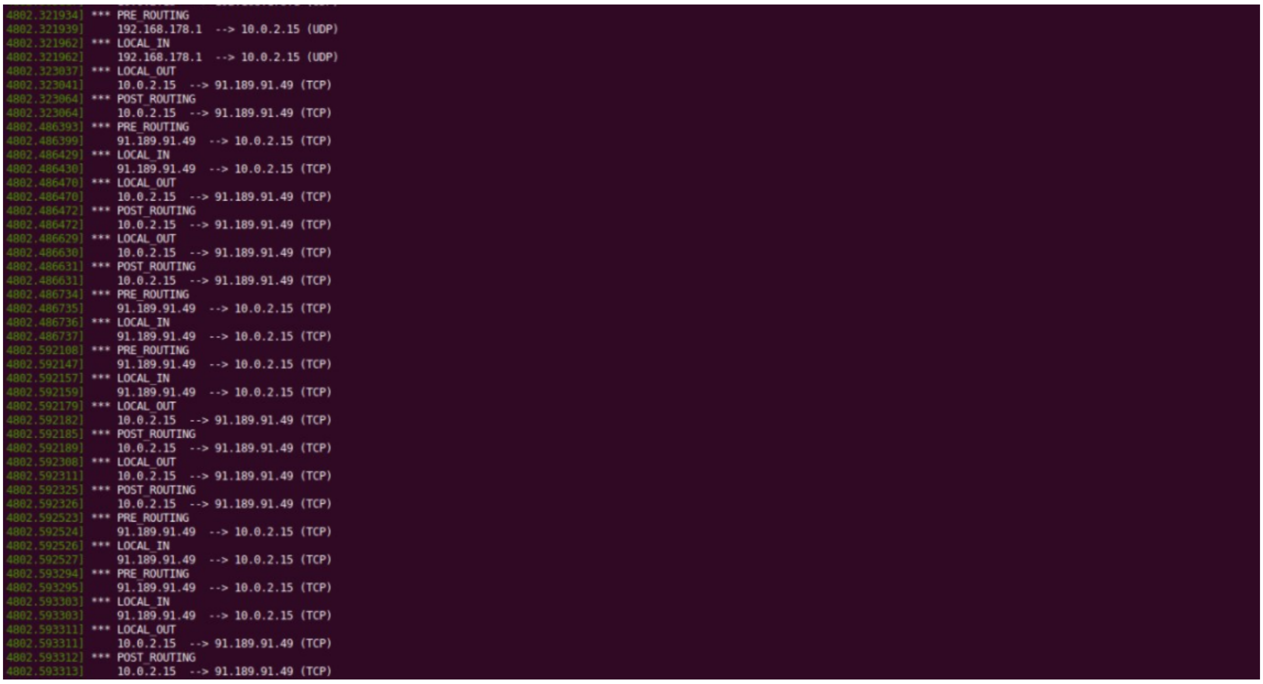
```
NF_INET_PRE_ROUTING
NF_INET_LOCAL_IN
NF_INET_FORWARD
NF_INET_LOCAL_OUT
NF_INET_POST_ROUTING
```

Code snippet to print information about the hooks "routing" process:

```
static struct nf_hook_ops hook_1, hook_2, hook_3, hook_4, hook_5;

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");

   hook_1.hook = printInfo;
   hook_1.hooknum = NF_INET_PRE_ROUTING;
   hook_1.pf = PF_INET;
   hook_1.priority = NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net, &hook_1);

   hook_2.hook = printInfo;
   hook_2.hooknum = NF_INET_LOCAL_IN;
   hook_2.pf = PF_INET;
   hook_2.priority = NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net, &hook_2);

   hook_3.hook = printInfo;
```

```
    hook_3.hooknum = NF_INET_FORWARD;
    hook_3.pf = PF_INET;
    hook_3.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook_3);

    hook_4.hook = printInfo;
    hook_4.hooknum = NF_INET_LOCAL_OUT;
    hook_4.pf = PF_INET;
    hook_4.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook_4);

    hook_5.hook = printInfo;
    hook_5.hooknum = NF_INET_POST_ROUTING;
    hook_5.pf = PF_INET;
    hook_5.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook_5);

    return 0;
}

void removeFilter(void) {
    printk(KERN_INFO "The filters are being removed.\n");
    nf_unregister_net_hook(&init_net, &hook_1);
    nf_unregister_net_hook(&init_net, &hook_2);
    nf_unregister_net_hook(&init_net, &hook_3);
    nf_unregister_net_hook(&init_net, &hook_4);
    nf_unregister_net_hook(&init_net, &hook_5);
}
```

The ordered flow of hooks described above can be verified using the output logs of the Loadable Kernel Module (LKM):



```
4802.321934] *** PRE_ROUTING
4802.321939]     192.168.178.1  --> 10.0.2.15 (UDP)
4802.321962] *** LOCAL_IN
4802.321962]     192.168.178.1  --> 10.0.2.15 (UDP)
4802.323037] *** LOCAL_OUT
4802.323041]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.323064] *** POST_ROUTING
4802.323064]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.486393] *** PRE_ROUTING
4802.486399]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.486429] *** LOCAL_IN
4802.486430]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.486470] *** LOCAL_OUT
4802.486470]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.486472] *** POST_ROUTING
4802.486472]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.486629] *** LOCAL_OUT
4802.486630]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.486631] *** POST_ROUTING
4802.486631]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.486734] *** PRE_ROUTING
4802.486735]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.486736] *** LOCAL_IN
4802.486737]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.592108] *** PRE_ROUTING
4802.592147]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.592157] *** LOCAL_IN
4802.592159]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.592179] *** LOCAL_OUT
4802.592182]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.592185] *** POST_ROUTING
4802.592189]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.592308] *** LOCAL_OUT
4802.592311]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.592325] *** POST_ROUTING
4802.592326]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.592523] *** PRE_ROUTING
4802.592524]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.592526] *** LOCAL_IN
4802.592527]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.593294] *** PRE_ROUTING
4802.593295]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.593303] *** LOCAL_IN
4802.593303]     91.189.91.49  --> 10.0.2.15 (TCP)
4802.593311] *** LOCAL_OUT
4802.593311]     10.0.2.15  --> 91.189.91.49 (TCP)
4802.593312] *** POST_ROUTING
4802.593313]     10.0.2.15  --> 91.189.91.49 (TCP)
```

3. **LKM blocks ping (ICMP)** and **Telnet** packets.

Code snippet to block ICMP traffic, including ICMP echo-request used for pinging:

```
static struct nf_hook_ops hook_1, hook_2, hook_3, hook_4, hook_5, hook_6;

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

```
   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);

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

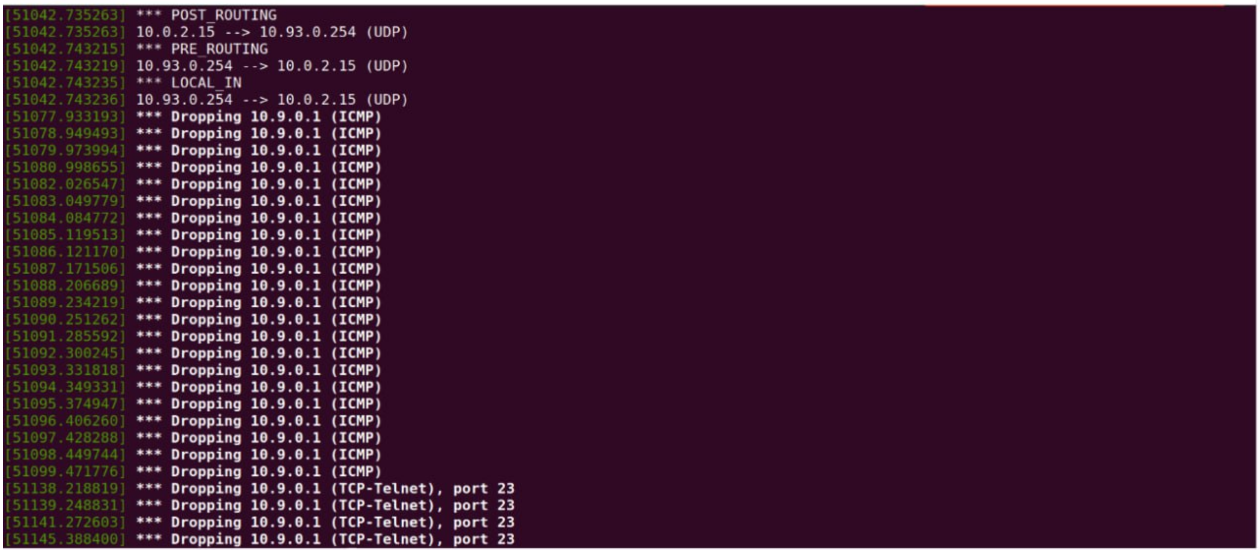
Code snippet to block Telnet traffic:

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

   u16  port = 23;
   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);

   if (iph->protocol == IPPROTO_TCP) {
       tcph = tcp_hdr(skb);
       if (iph->daddr == ip_addr && ntohs(tcph->dest) == port){
            printk(KERN_WARNING "*** Dropping %pI4 (TCP-Telnet), port %d\n", &(iph->daddr), port);
            return NF_DROP;
        }
   }
   return NF_ACCEPT;
}
```

Bash commands execution on 10.9.0.5:

```
ping 10.9.0.1
telnet 10.9.0.1
```

Result of the experiment:



```
[51042.735263] *** POST_ROUTING
[51042.735263] 10.0.2.15 --> 10.93.0.254 (UDP)
[51042.743215] *** PRE_ROUTING
[51042.743219] 10.93.0.254 --> 10.0.2.15 (UDP)
[51042.743235] *** LOCAL_IN
[51042.743236] 10.93.0.254 --> 10.0.2.15 (UDP)
[51077.933193] *** Dropping 10.9.0.1 (ICMP)
[51078.949493] *** Dropping 10.9.0.1 (ICMP)
[51079.973994] *** Dropping 10.9.0.1 (ICMP)
[51080.998655] *** Dropping 10.9.0.1 (ICMP)
[51082.026547] *** Dropping 10.9.0.1 (ICMP)
[51083.049779] *** Dropping 10.9.0.1 (ICMP)
[51084.084772] *** Dropping 10.9.0.1 (ICMP)
[51085.119513] *** Dropping 10.9.0.1 (ICMP)
[51086.121170] *** Dropping 10.9.0.1 (ICMP)
[51087.171506] *** Dropping 10.9.0.1 (ICMP)
[51088.206689] *** Dropping 10.9.0.1 (ICMP)
[51089.234219] *** Dropping 10.9.0.1 (ICMP)
[51090.251262] *** Dropping 10.9.0.1 (ICMP)
[51091.285592] *** Dropping 10.9.0.1 (ICMP)
[51092.300245] *** Dropping 10.9.0.1 (ICMP)
[51093.331818] *** Dropping 10.9.0.1 (ICMP)
[51094.349331] *** Dropping 10.9.0.1 (ICMP)
[51095.374947] *** Dropping 10.9.0.1 (ICMP)
[51096.406260] *** Dropping 10.9.0.1 (ICMP)
[51097.428288] *** Dropping 10.9.0.1 (ICMP)
[51098.449744] *** Dropping 10.9.0.1 (ICMP)
[51099.471776] *** Dropping 10.9.0.1 (ICMP)
[51138.218819] *** Dropping 10.9.0.1 (TCP-Telnet), port 23
[51139.248831] *** Dropping 10.9.0.1 (TCP-Telnet), port 23
[51141.272603] *** Dropping 10.9.0.1 (TCP-Telnet), port 23
[51145.388400] *** Dropping 10.9.0.1 (TCP-Telnet), port 23
```

Task 2.A: Protecting the Router

1. Please report your observation and explain the purpose for each rule.

```
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
```

The previous command appends the rule to the end of the INPUT chain within the filter table. This rule pertains to the ICMP protocol and specifically focuses on echo-request packets. If a packet matches this rule, the target action is to allow it to pass through.

```
iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
```

The above command appends the rule to the end of the OUTPUT chain within the filter table. This rule is designed for the ICMP protocol, specifically targeting echo-reply packets. When a packet matches this rule, the intended action is to allow it to pass through.

```
iptables -P OUTPUT DROP
```

The previous command set the policy for the built-in (non-user-defined) OUTPUT chain to the given target, filter table. The policy target is to DROP.

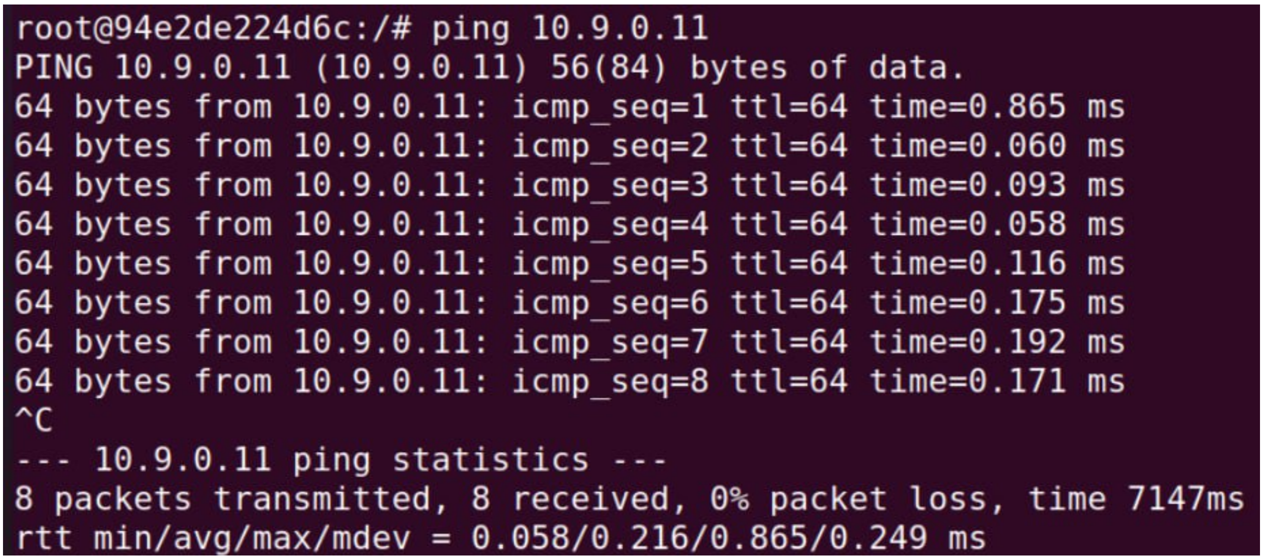
```
iptables -P INPUT DROP
```

The above command set the policy for the built-in (non-user-defined) INPUT chain to the given target, filter table. The policy target is to DROP.

2. **Can you ping the router?**

Yes, it's possible. The initial two commands executed on the router establish rules for the INPUT and OUTPUT chains. These rules permit the passage of ICMP echo-requests and ICMP echo-replies packets. As iptables entries are processed sequentially, these particular rules will take effect prior to the default behavior.

Ping result:

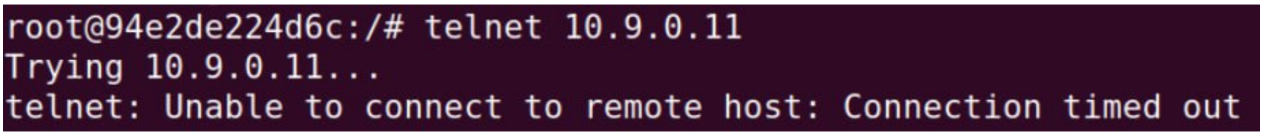


```
root@94e2de224d6c:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.865 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.060 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.093 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.058 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=64 time=0.116 ms
64 bytes from 10.9.0.11: icmp_seq=6 ttl=64 time=0.175 ms
64 bytes from 10.9.0.11: icmp_seq=7 ttl=64 time=0.192 ms
64 bytes from 10.9.0.11: icmp_seq=8 ttl=64 time=0.171 ms
^C
--- 10.9.0.11 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7147ms
rtt min/avg/max/mdev = 0.058/0.216/0.865/0.249 ms
```

3. **Can you telnet into the router?**

No, it can't be done. The reason is that the final two commands executed on the router establish a default behavior for the INPUT and OUTPUT chains, instructing them to discard any packets that do not match any of the previously defined rules in the tables.

Telnet result:



```
root@94e2de224d6c:/# telnet 10.9.0.11
Trying 10.9.0.11...
telnet: Unable to connect to remote host: Connection timed out
```

Task 2.B: Protecting the Internal Network

Rules to implement in order to protect the internal network.

1. **Outside** hosts **cannot** ping internal hosts.

2. **Outside hosts can ping the router**: no actions required since pinging the router do not involve the FORWARD, but we implement the following (**similar to the previous task**).

```
# accept all the traffic exchanged between the internal network and the router
iptables -A INPUT -s 192.168.60.0/24 -j ACCEPT
iptables -A OUTPUT -d 192.168.60.0/24 -j ACCEPT

# accept the ping packets directed to the router
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT

# block all the other packets directed toward the router and going out from the router
iptables -P OUTPUT DROP
iptables -P INPUT DROP
```

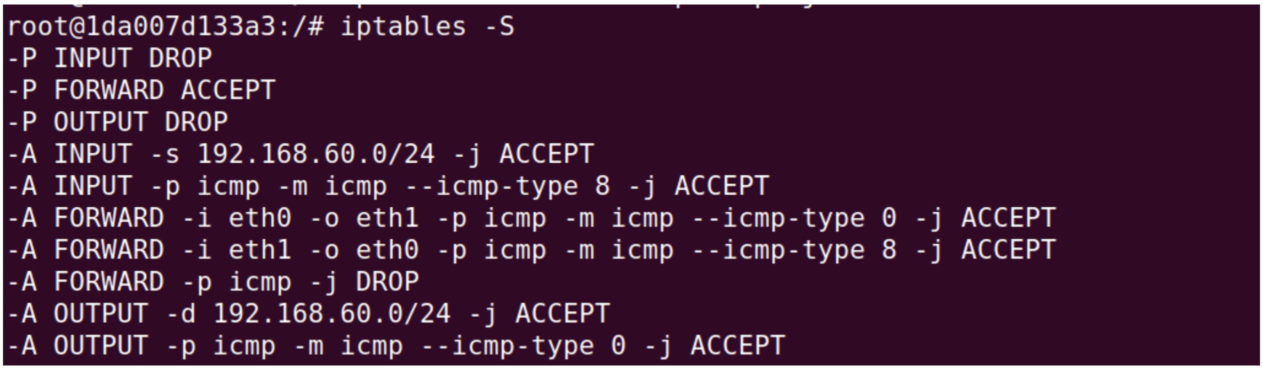
3. **Internal** hosts **can** ping outside hosts.

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

4. All other packets between the internal and external networks should be **blocked**.

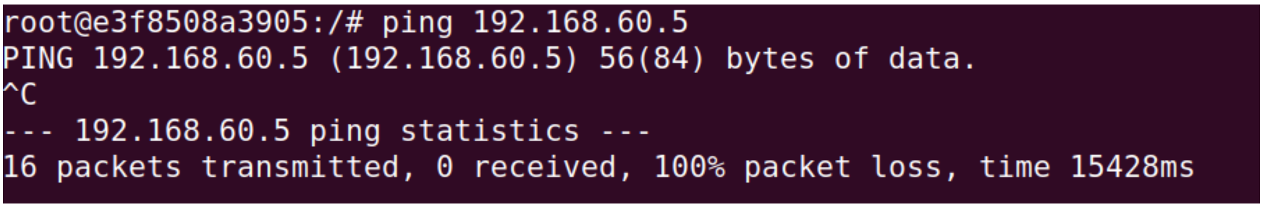
```
iptables -A FORWARD -p icmp -j DROP
```

Results of the experiment:



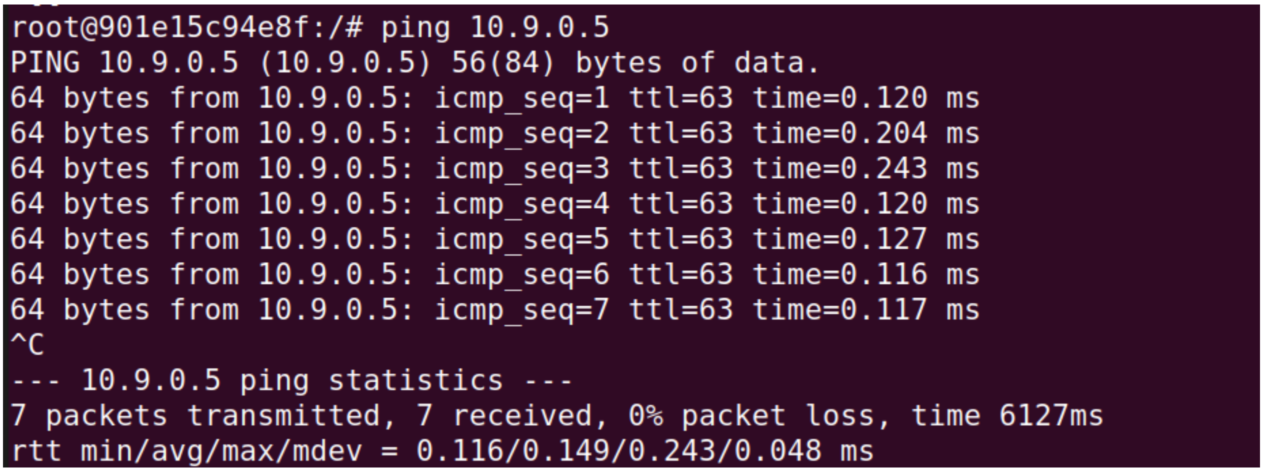
```
root@1da007d133a3:/# iptables -S
-P INPUT DROP
-P FORWARD ACCEPT
-P OUTPUT DROP
-A INPUT -s 192.168.60.0/24 -j ACCEPT
-A INPUT -p icmp -m icmp --icmp-type 8 -j ACCEPT
-A FORWARD -i eth0 -o eth1 -p icmp -m icmp --icmp-type 0 -j ACCEPT
-A FORWARD -i eth1 -o eth0 -p icmp -m icmp --icmp-type 8 -j ACCEPT
-A FORWARD -p icmp -j DROP
-A OUTPUT -d 192.168.60.0/24 -j ACCEPT
-A OUTPUT -p icmp -m icmp --icmp-type 0 -j ACCEPT
```

Host A (10.9.0.5) pinging 192.168.60.5:



```
root@e3f8508a3905:/# 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 ---
16 packets transmitted, 0 received, 100% packet loss, time 15428ms
```

Host A (192.168.60.5) pinging 10.9.0.5:



```
root@901e15c94e8f:/# 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=0.120 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=63 time=0.204 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=63 time=0.243 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=63 time=0.120 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=63 time=0.127 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=63 time=0.116 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=63 time=0.117 ms
^C
--- 10.9.0.5 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6127ms
rtt min/avg/max/mdev = 0.116/0.149/0.243/0.048 ms
```

Task 2.C: Protecting Internal Server

Rules to implement in order to protect the internal network.

1. All the internal hosts run a **telnet** server (listening to port 23). Outside hosts **can only access** the telnet server on **192.168.60.5**, not the other internal hosts.

```
iptables -A FORWARD -i eth0 -o eth1 -p tcp --dport 23 -d 192.168.60.5 -j ACCEPT
```

2. Outside hosts cannot access other internal servers.

```
iptables -A FORWARD -i eth0 -p tcp --dport 23 -j DROP
```

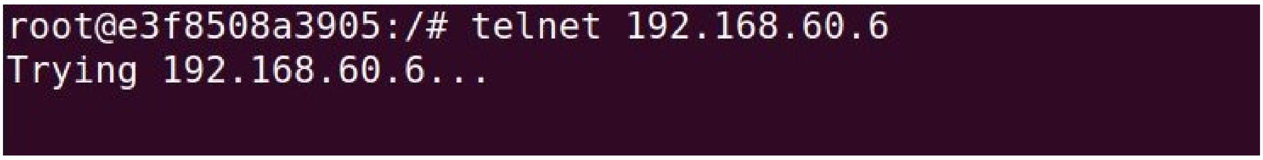
3. Internal hosts can access all the internal servers: it works by default.

4. Internal hosts **cannot** access **external servers.**

```
iptables -A FORWARD -o eth0 -p tcp --dport 23 -j DROP
```

5. In this task, the connection tracking mechanism is not allowed. It will be used later.

Result of the experiment:



```
root@e3f8508a3905:/# telnet 192.168.60.6
Trying 192.168.60.6...
```

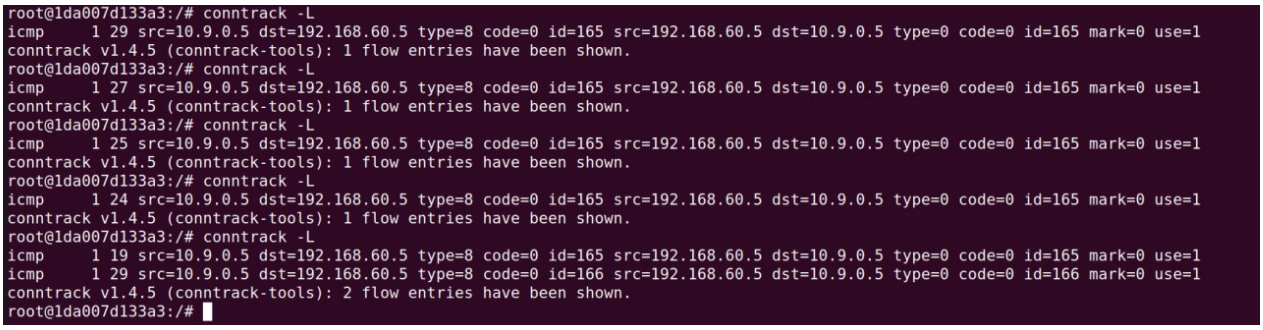
Task 3.A: Experiment with the Connection Tracking (Optional)

1. **ICMP experiment**: run the following command and check the connection tracking information on the router. Describe your observation. How long is the ICMP connection state be kept?

After 30 seconds, the ICMP state is removed from the tracking mechanism, provided no additional pings are transmitted from 10.9.0.5. The time remaining until the connection is deleted is indicated as the third value in the returned output. The second value denotes the protocol identifier used, and the remaining values provide information about the header fields of the IP/ICMP packets exchanged. The mark metadata is a customizable parameter that functions as a classification identifier for each connection.

```
icmp 1 29 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=50
   src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=50 mark=0 use=1

conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

ICMP tracking result:



```
root@1da007d133a3:/# conntrack -L
icmp     1 29 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=165 src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=165 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@1da007d133a3:/# conntrack -L
icmp     1 27 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=165 src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=165 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@1da007d133a3:/# conntrack -L
icmp     1 25 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=165 src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=165 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@1da007d133a3:/# conntrack -L
icmp     1 24 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=165 src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=165 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@1da007d133a3:/# conntrack -L
icmp     1 19 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=165 src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=165 mark=0 use=1
icmp     1 29 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=166 src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=166 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 2 flow entries have been shown.
root@1da007d133a3:/# 
```

2. **UDP experiment**: run the following command and check the connection tracking information on the router. Describe your observation. How long is the UDP connection state be kept?

Connection tracking begins when 10.9.0.5 sends a UDP packet. If 192.168.60.5 does not respond with a UDP packet, the UDP state is removed after 30 seconds (marked as UNREPLIED). However, if 192.168.60.5 responds with UDP packets, the connection is marked as ASSURED, and the UDP state is removed after 120 seconds. When traffic occurs in both directions, the UNREPLIED flag is erased and reset. Entries without traffic in both directions are marked as ASSURED and will not be erased even when the maximum tracked connections are reached, unlike non-assured connections.

```
udp 17 27 src=10.9.0.5 dst=192.168.60.5 sport=35944 dport=9090 [UNREPLIED]
   src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=35944 mark=0 use=1

conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

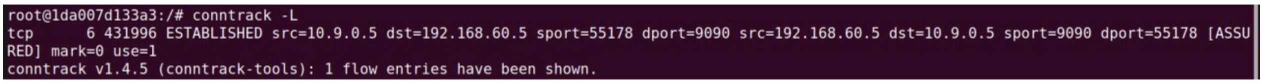
```
udp 17 115 src=10.9.0.5 dst=192.168.60.5 sport=54197 dport=9090
   src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=54197 [ASSURED] mark=0 use=1

conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

3. **TCP experiment**: run the following command and check the connection tracking information on the router. Describe your observation. How long is the TCP connection state be kept?

The TCP connection state is removed from tracking after 432000 seconds (5 days) if no further TCP packets are exchanged. Unlike the previous scenario, the connection is tracked as soon as it's initiated between the two machines, without the need for additional TCP packet exchanges.

```
tcp 6 431997 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=37564 dport=9090
   src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=37564 [ASSURED] mark=0 use=1

conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

TCP tracking result:



```
root@1da007d133a3:/# conntrack -L
tcp      6 431996 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=55178 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=55178 [ASSU
RED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

Task 3.B: Setting Up a Stateful Firewall (Optional)

Please rewrite the firewall rules in Task 2.C, but this time, we will add a rule allowing **internal hosts to visit any external server** (this was not allowed in Task 2.C).

1. All the internal hosts run a telnet server (listening to port 23). Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts.

```
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
```

2. Outside hosts cannot access other internal servers.

```
iptables -A FORWARD -p tcp -i eth0 --dport 23 -j DROP
```

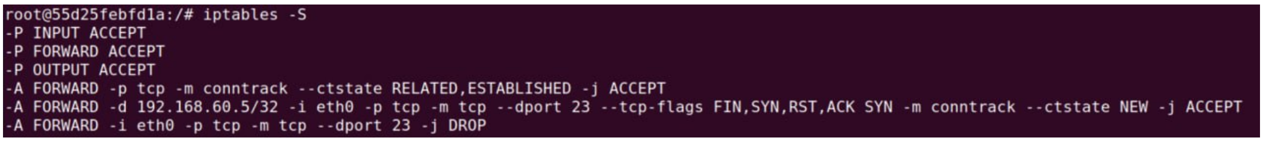
3. Internal hosts can access all the internal servers: it works by default.

4. Internal hosts can access external servers: it works by default.

After setting up the rules using the connection tracking mechanism, consider an alternative approach without relying on connection tracking (actual implementation not required). Then, compare these two rule sets and elucidate the advantages and disadvantages of each approach. Once this task is completed, ensure that all rules are cleared.

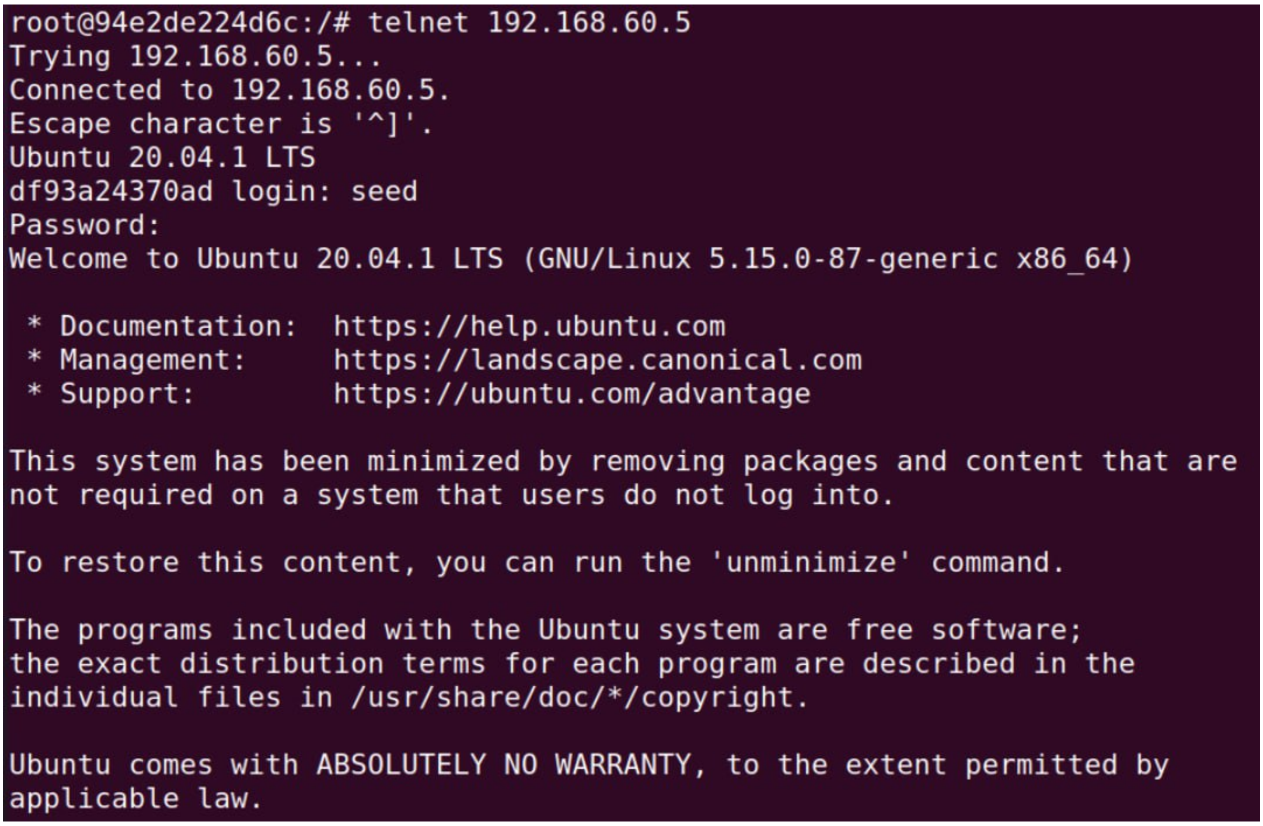
Without the connection tracking mechanism, the firewall rules for this task are simpler compared to those utilizing connection tracking (specifically, rule 4 in task 2.C can be omitted) since there's no need to consider the protocol's state. However, for more complex tasks, this approach can result in less accurate, less secure, or more complex firewall rules. Stateless firewalls can only allow packets of established connections by permitting specific types of packets within those connections for certain hosts, specific ports, and with specific characteristics. This can require numerous rules from the administrator, or alternatively, allowing all kinds of packets, which is not entirely secure. The connection tracking mechanism provides an easier and context-aware way to handle complex rules.

Results of the experiment:



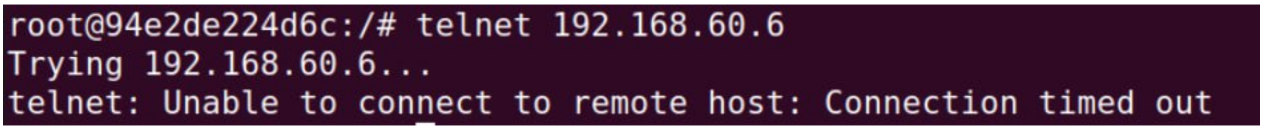
```
root@55d25febfd1a:/# iptables -S
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT
-A FORWARD -p tcp -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -d 192.168.60.5/32 -i eth0 -p tcp -m tcp --dport 23 --tcp-flags FIN,SYN,RST,ACK SYN -m conntrack --ctstate NEW -j ACCEPT
-A FORWARD -i eth0 -p tcp -m tcp --dport 23 -j DROP
```

Host A (10.9.0.5) telnet 192.168.60.5:



```
root@94e2de224d6c:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
df93a24370ad login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.15.0-87-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

Host A (10.9.0.5) telnet 192.168.60.6:



```
root@94e2de224d6c:/# telnet 192.168.60.6
Trying 192.168.60.6...
telnet: Unable to connect to remote host: Connection timed out
```

Task 4: Limiting Network Traffic (Optional)

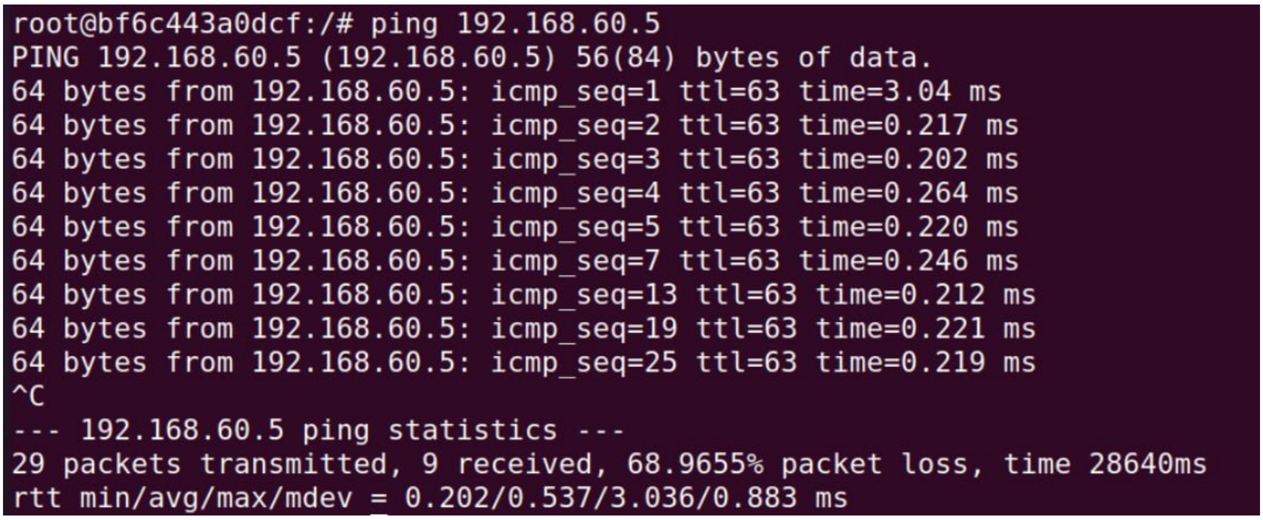
1. Please run the following commands on router, and then ping 192.168.60.5 from 10.9.0.5. Describe your observation. Please conduct the experiment with and without the second rule, and then explain whether the second rule is needed or not, and why.

```
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
```

Using only the first rule is essentially equivalent to having no rule at all, as the router's default behavior allows all packets to pass through.

However, when both commands are employed, the first time this rule is encountered by an ICMP packet, it will be allowed through. This is due to the default burst of 5, permitting the passage of the first five packets. Subsequently, a waiting period of 6 seconds (60/10 seconds) will ensue before another packet is accepted by this rule. During this time, any packets arriving at the router will be dropped, thanks to the second command. Additionally, for every 6 seconds that elapse without a packet match, one burst will be restored. If no packets trigger the rule for 30 seconds, the burst will be completely recharged, returning to its initial state.

Result of the experiment:



```
root@bf6c443a0dcf:/# 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=3.04 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.217 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.202 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.264 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.220 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.246 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.212 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=63 time=0.221 ms
64 bytes from 192.168.60.5: icmp_seq=25 ttl=63 time=0.219 ms
^C
--- 192.168.60.5 ping statistics ---
29 packets transmitted, 9 received, 68.9655% packet loss, time 28640ms
rtt min/avg/max/mdev = 0.202/0.537/3.036/0.883 ms
```

Task 5: Load Balancing (Optional)

1. Using the nth mode (round-robin). Please provide some explanation for the rules.

```
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
```

Initially, the counter for the first rule is at zero. If an initial UDP packet is transmitted the router's port 8080, it is forwarded to 192.168.60.5. This is because the packet counter is set to zero, and this rule is the first in the NAT table, making it the first to be executed. Subsequently, the counter for this rule increments to one.

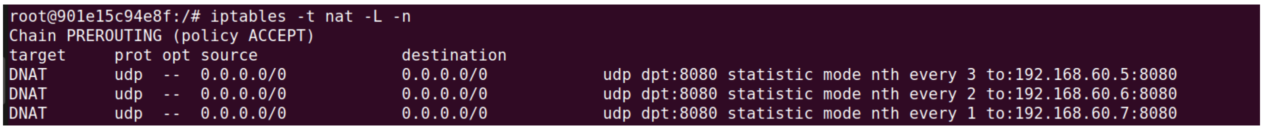
```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth
         --every 2 --packet 0 -j DNAT --to-destination 192.168.60.6:8080
```

When the second rule is active, sending a second UDP packet to the router's port 8080 will result in it being directed to 192.168.60.6. Simultaneously, the counter for this rule increases to 1, while the counter for the first rule increments to 2.

```
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
```

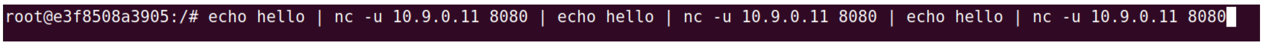
When the third rule is configured, sending a UDP packet to the router's port 8080 will result in the third packet of the three being directed to 192.168.60.7.

Results of the experiment:

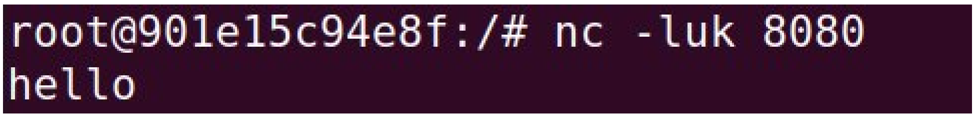


```
root@901e15c94e8f:/# iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode nth every 3 to:192.168.60.5:8080
DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode nth every 2 to:192.168.60.6:8080
DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode nth every 1 to:192.168.60.7:8080
```

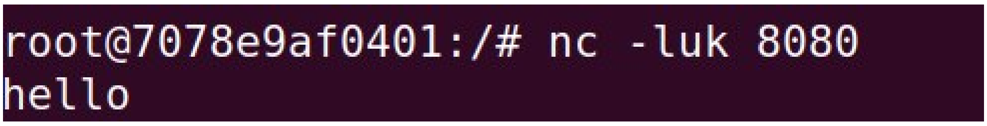
Sending 3 `echo hello` packets to UDP servers:



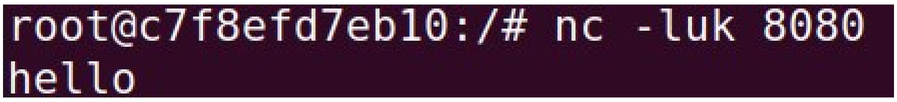
```
root@e3f8508a3905:/# echo hello | nc -u 10.9.0.11 8080 | echo hello | nc -u 10.9.0.11 8080 | echo hello | nc -u 10.9.0.11 8080
```



```
root@901e15c94e8f:/# nc -luk 8080
hello
```



```
root@7078e9af0401:/# nc -luk 8080
hello
```



```
root@c7f8efd7eb10:/# nc -luk 8080
hello
```

As can be seen from above, each host receives 1 out of the 3 packets sent.

2. Using the random mode. Please provide some explanation for the rules.

It's important to underline that the rules will be executed in the order they were added.

```
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
```

When using the first rule, sending a UDP packet to the router's port 8080 means that there is a 0.33 probability that a matching packet will be selected and forwarded to 192.168.60.5.

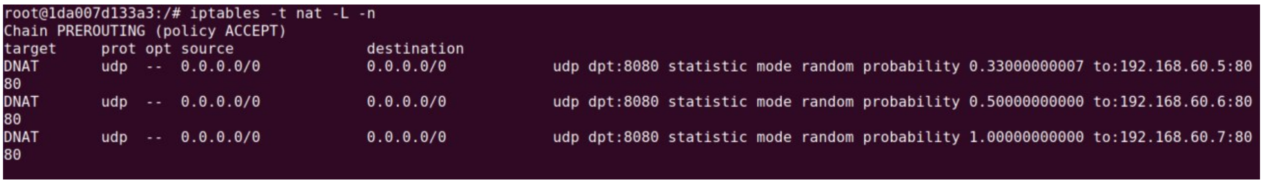
```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random
         --probability 0.50 -j DNAT --to-destination 192.168.60.6:8080
```

When the second rule is applied, sending a UDP packet to the router's 8080 port results in a 0.50 probability that a matching packet will be selected and forwarded to 192.168.60.6.

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

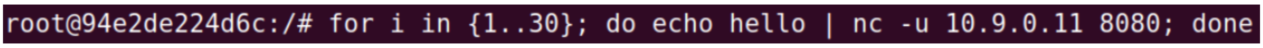
When the third rule is in effect, sending a UDP packet to the router's 8080 port ensures a probability 1 that a matching packet will be selected and forwarded to 192.168.60.7.

Results of the experiment:

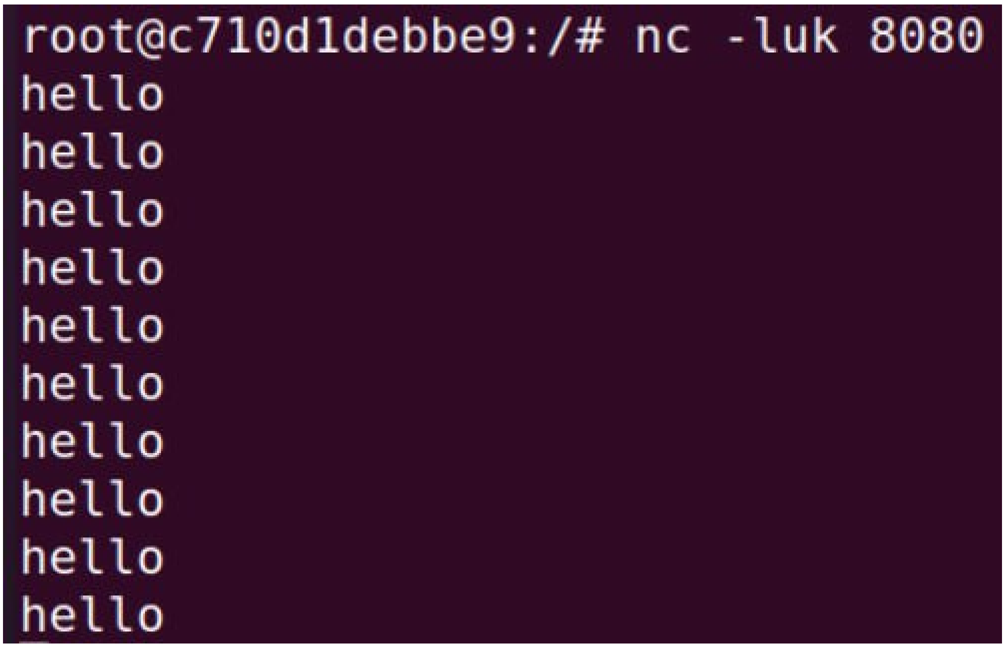


```
root@1da007d133a3:/# iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 0.33000000007 to:192.168.60.5:80
80
DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 0.50000000000 to:192.168.60.6:80
80
DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 1.00000000000 to:192.168.60.7:80
80
```

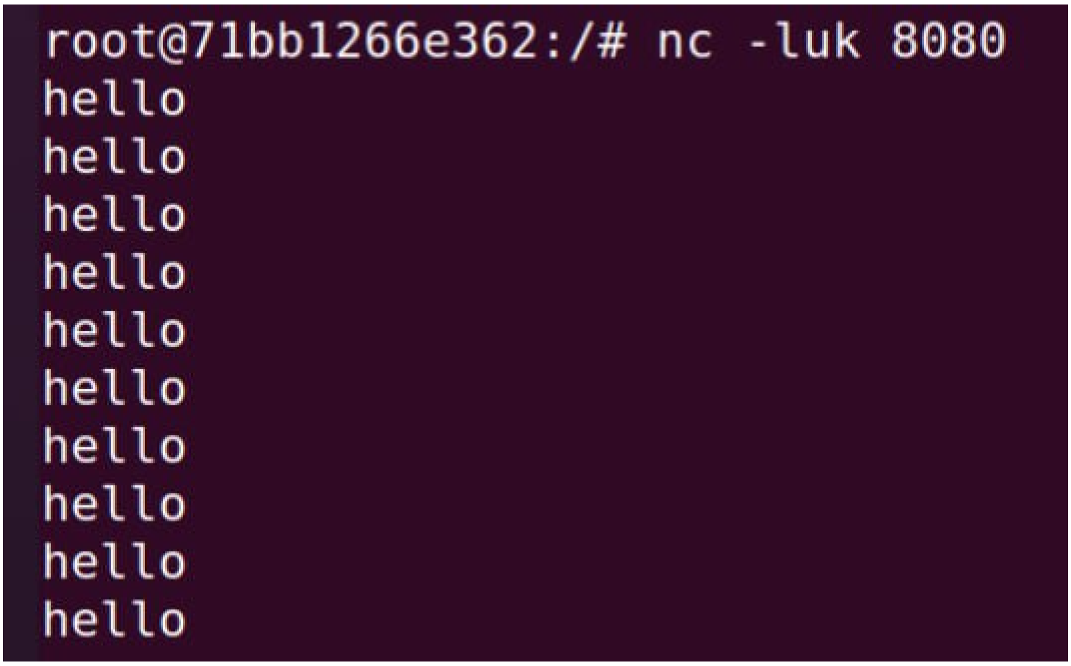
Sending 30 `echo hello` packets to UDP servers:



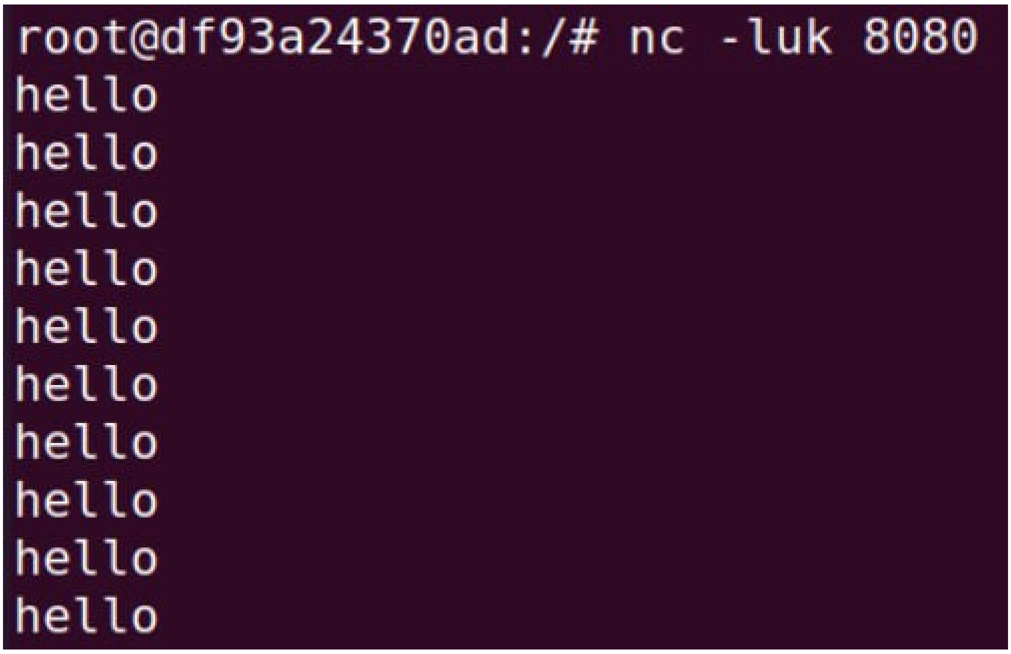
```
root@94e2de224d6c:/# for i in {1..30}; do echo hello | nc -u 10.9.0.11 8080; done
```



```
root@c710d1debbe9:/# nc -luk 8080
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```



```
root@71bb1266e362:/# nc -luk 8080
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```



```
root@df93a24370ad:/# nc -luk 8080
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
```

As evident from the above screenshots, we have been lucky as all three of them receive exactly 10 out of the 30 packets sent.