

Kevin Huang
CSS 537
Assignment 2: Network Firewalls
02/12/2022

Introduction

For this assignment, I learned about how firewalls work and also how to set up simple firewalls for a network. I first explored LKM and Netfilter and was able to set up a simple firewall with hooks. Next, I learned about iptables and how to use it to set up a firewall. To build on this, I learned about connection tracking, how to implement stateful firewalls, how to limit network traffic, and load balancing.

Task 1: Implementing a Simple Firewall

Task 1.A: Implement a Simple Kernel Module

For this task, I learned how the Loadable Kernel Module (LKM) lets us add new modules to the kernel at runtime. I compiled the hello.c that was provided and I was able to successfully load a module (using insmod) and then remove it (using rmmod) from the kernel. This was verified through the messages that were displayed in the syslog file (using dmesg) when the module was loaded and then removed.

Screenshot 1.A.1: make to compile a loadable kernel module

```
[02/07/22] seed@VM:~/.../kernel_module$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/Documents/assignment2/Files/kernel_module modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M] /home/seed/Documents/assignment2/Files/kernel_module/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/seed/Documents/assignment2/Files/kernel_module/hello.o
see include/linux/module.h for more information
  CC [M] /home/seed/Documents/assignment2/Files/kernel_module/hello.mod.o
  LD [M] /home/seed/Documents/assignment2/Files/kernel_module/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
```

Screenshot 1.A.2: insert the hello.ko module, list all modules, remove the hello.ko module, and check the messages

```
[02/07/22] seed@VM:~/.../kernel_module$ sudo insmod hello.ko
[02/07/22] seed@VM:~/.../kernel_module$ lsmod | grep hello
hello                16384  0
[02/07/22] seed@VM:~/.../kernel_module$ sudo rmmod hello
[02/07/22] seed@VM:~/.../kernel_module$ dmesg
```

Screenshot 1.A.3: dmesg shows the "Hello World!" and "Bye-bye World!" messages

```
[ 1557.684383] hello: module license 'unspecified' taints kernel.
[ 1557.684384] Disabling lock debugging due to kernel taint
[ 1557.684411] hello: module verification failed: signature and/or required key
missing - tainting kernel
[ 1557.684665] Hello World!
[ 1580.890866] Bye-bye World!.
```

Task 1.B: Implement a Simple Firewall Using Netfilter

In task 1.B, I implemented firewalls using Netfilter and also hooks. For task #1, I need to implement a firewall to block requests to Google's DNS server. I tested if I can set up a firewall to block UDP packets to Google's DNS server (8.8.8.8). I first checked that the request does go through normally without the firewall. I then compiled the seedFilter.c program that was provided and inserted the module. Now I tried to send the same request again but received a "connection timed out". This is a good indication that the firewall is working. I also checked the syslog and found that the packets to 8.8.8.8 were indeed being dropped.

Screenshot 1.B.1: DNS query to Google without firewall rule

```
[02/08/22] 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
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 53824
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.example.com.                IN      A

;; ANSWER SECTION:
www.example.com.                19232   IN      A      93.184.216.34

;; Query time: 16 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Tue Feb 08 00:11:37 EST 2022
;; MSG SIZE rcvd: 60
```

Screenshot 1.B.2: After compiling seedFilter, the module is inserted and another attempt to DNS query to Google but this time connection timed out

```
[02/08/22] seed@VM:~/.../packet_filter$ sudo insmod seedFilter.ko
[02/08/22] seed@VM:~/.../packet_filter$ lsmod | grep seedFilter
seedFilter                16384  0
[02/08/22] 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
```

Screenshot 1.B.3: using dmesg, the syslog shows UDP packets to 8.8.8.8 were dropped

```
[ 2856.826888] Registering filters.
[ 2877.194144] *** LOCAL_OUT
[ 2877.194145] 127.0.0.1 --> 127.0.0.1 (UDP)
[ 2877.194271] *** LOCAL_OUT
[ 2877.194271] 10.0.2.5 --> 8.8.8.8 (UDP)
[ 2877.194275] *** Dropping 8.8.8.8 (UDP), port 53
[ 2882.193128] *** LOCAL_OUT
[ 2882.193130] 10.0.2.5 --> 8.8.8.8 (UDP)
[ 2882.193140] *** Dropping 8.8.8.8 (UDP), port 53
[ 2887.195825] *** LOCAL_OUT
[ 2887.195827] 10.0.2.5 --> 8.8.8.8 (UDP)
[ 2887.195837] *** Dropping 8.8.8.8 (UDP), port 53
[ 2959.992648] *** LOCAL_OUT
[ 2959.992649] 10.0.2.5 --> 91.189.89.199 (UDP)
[ 2961.633728] *** LOCAL_OUT
[ 2961.633730] 10.0.2.5 --> 192.168.1.1 (UDP)
[ 2961.652613] *** LOCAL_OUT
[ 2961.652614] 127.0.0.1 --> 127.0.0.53 (UDP)
[ 2961.652711] *** LOCAL_OUT
[ 2961.652712] 10.0.2.5 --> 192.168.1.1 (UDP)
[ 2961.671184] *** LOCAL_OUT
[ 2961.671185] 127.0.0.53 --> 127.0.0.1 (UDP)
[ 2962.300954] The filters are being removed.
```


For task #2, I needed to attach the printInfo function to all of the netfilter hooks. I hooked the printInfo function to each of the netfilter hooks (pre_routing, local_in, forward, local_out, and post_routing) by creating and registering 5 hooks in the seedFilter.c program. I inserted the module and did the dig command again to Google's DNS server and observed in the syslog 4 of the 5 hook functions invoked. The first hook that was invoked was NF_INET_LOCAL_OUT, this was triggered because the local machine (10.0.2.5) made a request out to 8.8.8.8. Then, the NF_INET_POST_ROUTING hook was triggered once the request is sent out through the physical medium. When the 10.0.2.3 machine receives the packet from the physical layer, the NF_IP_PRE_ROUTING is triggered. Then, the NF_IP_LOCAL_IN is triggered when the packet is sent up to 10.0.2.3 for processing. Lastly, the NF_IP_FORWARD hook will be invoked if the packet is supposed to be forwarded to a different machine. I triggered this by using 10.9.0.5 machine to ping 192.168.60.5. Since the packet needs to go through the router, the router will be forwarding the packet and I can see that it successfully triggered the hook.

Screenshot 1.B.4: Creating hooks for all 5 netfilter hooks

```
77     hook1.hook = printInfo;
78     hook1.hooknum = NF_INET_LOCAL_OUT;
79     hook1.pf = PF_INET;
80     hook1.priority = NF_IP_PRI_FIRST;
81     nf_register_net_hook(&init_net, &hook1);
91     hook2.hook = printInfo;
92     hook2.hooknum = NF_INET_PRE_ROUTING;
93     hook2.pf = PF_INET;
94     hook2.priority = NF_IP_PRI_FIRST;
95     nf_register_net_hook(&init_net, &hook2);
96
97     hook3.hook = printInfo;
98     hook3.hooknum = NF_INET_LOCAL_IN;
99     hook3.pf = PF_INET;
100    hook3.priority = NF_IP_PRI_FIRST;
101    nf_register_net_hook(&init_net, &hook3);
102
103    hook4.hook = printInfo;
104    hook4.hooknum = NF_INET_FORWARD;
105    hook4.pf = PF_INET;
106    hook4.priority = NF_IP_PRI_FIRST;
107    nf_register_net_hook(&init_net, &hook4);
108
109    hook5.hook = printInfo;
110    hook5.hooknum = NF_INET_POST_ROUTING;
111    hook5.pf = PF_INET;
112    hook5.priority = NF_IP_PRI_FIRST;
113    nf_register_net_hook(&init_net, &hook5);
114
```

Screenshot 1.B.5: using the dig command for testing

```
[02/08/22] 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
;; Got answer:
;; ->HEADER<- opcode: QUERY, status: NOERROR, id: 16947
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
```

Screenshot 1.B.6: syslog shows successful hooks

```
[ 4631.938967] 20:12:06.596598 timesync vgsvcTimeSyncWorker: Radical guest time
change: 52 153 334 564 000ns (GuestNow=1 644 351 126 596 589 000 ns GuestLast=1
644 298 973 262 025 000 ns fSetTimeLastLoop=true )
[ 5440.204436] Registering filters.
[ 5524.570745] *** LOCAL_OUT
[ 5524.570747] 10.0.2.5 --> 10.0.2.3 (UDP)
[ 5524.570758] *** POST_ROUTING
[ 5524.570759] 10.0.2.5 --> 10.0.2.3 (UDP)
[ 5524.574261] *** PRE_ROUTING
[ 5524.574262] 10.0.2.3 --> 10.0.2.5 (UDP)
[ 5524.574266] *** LOCAL_IN
[ 5524.574266] 10.0.2.3 --> 10.0.2.5 (UDP)
```

Screenshot 1.B.7: sys shows successful FORWARD hook when 10.9.0.5 pings 192.168.60.5

```
[26066.238170] *** PRE_ROUTING
[26066.238172] 10.9.0.5 --> 192.168.60.5 (ICMP)
[26066.238183] *** FORWARD
[26066.238184] 10.9.0.5 --> 192.168.60.5 (ICMP)
[26066.238186] *** POST_ROUTING
[26066.238186] 10.9.0.5 --> 192.168.60.5 (ICMP)
```

For task #3, I needed to add two more hooks for the following cases: preventing other computers to ping the VM and preventing other computers to telnet into the VM. I used the code that was provided to us for the telnet hook function and I wrote a similar hook function for ICMP. I fired up the containers and used 10.9.0.5 machine to send out requests. I first pinged Google's DNS server (8.8.8.8) to make sure pings were working. I then tried pinging the VM (10.9.0.1). Checking the syslog, I see that the ICMP packets were dropped. I then tried telnet into 192.168.60.6 and it was working properly so then I tried telnet into the VM and checking syslog again I can see those packets were also successfully dropped. I have succeeded in writing two hooks for preventing pings and telnet requests into the VM.

Screenshot 1.B.8: hook function for blocking ICMP to 10.9.0.1

```
16 unsigned int icmpFilter(void *priv, struct sk_buff *skb,
17                          const struct nf_hook_state *state) {
18     struct iphdr *iph;
19     u32 ip_addr;
20     char ip[16] = "10.9.0.1";
21     in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);
22     iph = ip_hdr(skb);
23
24     if (iph->protocol == IPPROTO_ICMP && iph->daddr == ip_addr) {
25         printk(KERN_INFO "Dropping ICMP packet to %d.%d.%d.%d\n",
26                ((unsigned char *)&iph->daddr)[0],
27                ((unsigned char *)&iph->daddr)[1],
28                ((unsigned char *)&iph->daddr)[2],
29                ((unsigned char *)&iph->daddr)[3]);
30         return NF_DROP;
31     } else {
32         return NF_ACCEPT;
33     }
34 }
```

Screenshot 1.B.9: Machine 10.9.0.5 sends out ping requests to 8.8.8.8 and 10.9.0.1

```
root@ec37fe01db0d:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=55 time=24.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=55 time=15.0 ms
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 15.011/19.530/24.050/4.519 ms
root@ec37fe01db0d:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
^C
--- 10.9.0.1 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3050ms
```


Screenshot 1.B.10: Syslog file shows pings to 10.9.0.1 were dropped

```
[15098.036411] Registering filters.  
[15107.021859] Dropping ICMP packet to 10.9.0.1  
[15108.024679] Dropping ICMP packet to 10.9.0.1  
[15109.048331] Dropping ICMP packet to 10.9.0.1  
[15110.072253] Dropping ICMP packet to 10.9.0.1  
[15111.072253] Dropping ICMP packet to 10.9.0.1
```

Screenshot 1.B.11: hook function for blocking telnet to 10.9.0.1

```
36 unsigned int telnetFilter(void *priv, struct sk_buff *skb,  
37                          const struct nf_hook_state *state) {  
38     struct iphdr *iph;  
39     struct tcphdr *tcph;  
40     u32 ip_addr;  
41     char ip[16] = "10.9.0.1";  
42     in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);  
43  
44     iph = ip_hdr(skb);  
45     tcph = (void *)iph+iph->ihl*4;  
46  
47     if (iph->protocol == IPPROTO_TCP && iph->daddr == ip_addr) {  
48         printk(KERN_INFO "Dropping telnet packet to %d.%d.%d.%d\n",  
49             ((unsigned char *)&iph->daddr)[0],  
50             ((unsigned char *)&iph->daddr)[1],  
51             ((unsigned char *)&iph->daddr)[2],  
52             ((unsigned char *)&iph->daddr)[3]);  
53         return NF_DROP;  
54     } else {  
55         return NF_ACCEPT;  
56     }  
57 }
```

Screenshot 1.B.12: Machine 10.9.0.5 sends out telnet requests to 192.168.60.6 and 10.9.0.1

```
root@ec37fe01db0d:/# telnet 192.168.60.6
Trying 192.168.60.6...
Connected to 192.168.60.6.
Escape character is '^]'.
^]
telnet> quit
Connection closed.
root@ec37fe01db0d:/# telnet 10.9.0.1
Trying 10.9.0.1...
^C
```

Screenshot 1.B.13: Syslog file shows telnet requests to 10.9.0.1 were dropped

```
[15107.021859] Dropping ICMP packet to 10.9.0.1
[15108.024679] Dropping ICMP packet to 10.9.0.1
[15109.048331] Dropping ICMP packet to 10.9.0.1
[15110.072253] Dropping ICMP packet to 10.9.0.1
[15423.221279] Dropping telnet packet to 10.9.0.1
[15424.245926] Dropping telnet packet to 10.9.0.1
[15426.262248] Dropping telnet packet to 10.9.0.1
[15430.326595] Dropping telnet packet to 10.9.0.1
[02/08/22] seed@VM:~/.../packet_filter$
```


Task 2: Experimenting with Stateless Firewall Rules

Task 2.A: Protecting the Router

In this part, I learned about iptables and how it can be used for firewalls. For this task, I learned how to protect the router by using iptables to set up rules so that only pings can access the router machine. I applied to iptables commands that were provided. I then pinged and telnet into the router's 10.9.0.11 interface and only the ping went through. I also did the same thing to the 192.168.60.11 interface and only the ping went through. This shows that only pings are able to access the router machine.

Screenshot 2.A.1: executed iptables commands on the router container

```
root@3b685f768d39:/# iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
root@3b685f768d39:/# iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
root@3b685f768d39:/# iptables -P OUTPUT DROP
root@3b685f768d39:/# iptables -P INPUT DROP
```

Screenshot 2.A.2: 10.9.0.5 machine pinged and then telnet into router 10.9.0.11 interface

```
root@ec37fe01db0d:/# 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=0.064 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.055 ms
^C
--- 10.9.0.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1036ms
rtt min/avg/max/mdev = 0.055/0.059/0.064/0.004 ms
root@ec37fe01db0d:/# telnet 10.9.0.11
Trying 10.9.0.11...
^C
```

Screenshot 2.A.3: 10.9.0.5 machine pinged and then telnet into router 192.168.60.11 interface

```
root@ec37fe01db0d:/# ping 192.168.60.11
PING 192.168.60.11 (192.168.60.11) 56(84) bytes of data.
64 bytes from 192.168.60.11: icmp_seq=1 ttl=64 time=0.057 ms
64 bytes from 192.168.60.11: icmp_seq=2 ttl=64 time=0.057 ms
^C
--- 192.168.60.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1017ms
rtt min/avg/max/mdev = 0.057/0.057/0.057/0.000 ms
root@ec37fe01db0d:/# telnet 192.168.60.11
Trying 192.168.60.11...
^C
_
```

Screenshot 2.A.4: Restored the filter table to its original state for the router machine

```
root@3b685f768d39:/# iptables -F
root@3b685f768d39:/# iptables -P OUTPUT ACCEPT
root@3b685f768d39:/# iptables -P INPUT ACCEPT
```

Task 2.B: Protecting the Internal Network

In this part, I needed to set up firewall rules on the router to protect the 192.168.60.0/24 internal network. For this task, I used iptables to set up firewall rules. The rules are as follows:

1. Outside hosts cannot ping internal hosts.
2. Outside hosts can ping the router.
3. Internal hosts can ping outside hosts.
4. All other packets between the internal and external networks should be blocked.

To implement #1, I made a rule that any ICMP requests coming from eth0 (outside) are dropped. Also, I made another for any ICMP replies coming out of eth0 will also be dropped.

For #2, I made rules stating that ICMP requests with INPUT to the router are allowed and same with ICMP replies from the router through eth0 are allowed.

For #3, I made rules stating that any ICMP requests going into eth1 (from inside) are allowed and ICMP replies going out of eth1 are also allowed.

For #4, I updated INPUT, OUTPUT, and FORWARD to drop all other packets.

I printed the filter table out and it all looked correct. I tested the rules by doing the following:

- Used 10.9.0.5 (outside host) to ping 192.168.60.5 (inside host) and the packets were dropped showing successful application of rule #1
- Used 10.9.0.5 (outside host) to ping 10.9.0.11 (router) and the router responded showing successful application of rule #2
- Used 192.168.60.5 (inside host) to ping 10.9.0.5 (outside host) and ICMP requests and replies went through showing successful application of rule #3
- Used 192.168.60.5 (inside host) to telnet into 10.9.0.5 (outside host) and was not able to connect showing successful application of rule #4
- Used 10.9.0.5 (outside host) to telnet into 10.9.0.11 (router) and was not able to connect showing successful application of rule #4
- Used 10.9.0.5 (outside host) to telnet into 192.168.60.5 (inside host) and was not able to connect showing successful application of rule #4
- Used 192.168.60.5 (inside host) to telnet into 192.168.60.11 (router) and was not able to connect showing successful application of rule #4

Screenshot 2.B.1: rules for outside host not able to ping internal hosts

```
root@3b685f768d39:/# iptables -A FORWARD -i eth0 -o eth1 -p icmp --icmp-type echo-request -j DROP
root@3b685f768d39:/# iptables -A FORWARD -i eth1 -o eth0 -p icmp --icmp-type echo-reply -j DROP
```

Screenshot 2.B.2: rules for allowing outside host to ping router

```
root@3b685f768d39:/# iptables -A INPUT -i eth0 -p icmp --icmp-type echo-request -j ACCEPT
root@3b685f768d39:/# iptables -A OUTPUT -o eth0 -p ICMP --icmp-type echo-reply -j ACCEPT
```

Screenshot 2.B.3: rules for internal hosts to ping outside hosts

```
root@3b685f768d39:/# iptables -A FORWARD -i eth1 -o eth0 -p icmp --icmp-type echo-request -j ACCEPT
root@3b685f768d39:/# iptables -A FORWARD -i eth0 -o eth1 -p icmp --icmp-type echo-reply -j ACCEPT
```

Screenshot 2.B.4: rules for blocking all other packets

```
root@3b685f768d39:/# iptables -P OUTPUT DROP
root@3b685f768d39:/# iptables -P INPUT DROP
root@3b685f768d39:/# iptables -P FORWARD DROP
```

Screenshot 2.B.5: filter table after all rules applied

```
root@3b685f768d39:/# iptables -t filter -L -n
Chain INPUT (policy DROP)
target     prot opt source                destination            icmptype 8
ACCEPT     icmp -- 0.0.0.0/0              0.0.0.0/0              icmptype 8

Chain FORWARD (policy DROP)
target     prot opt source                destination            icmptype 8
DROP       icmp -- 0.0.0.0/0              0.0.0.0/0              icmptype 0
DROP       icmp -- 0.0.0.0/0              0.0.0.0/0              icmptype 8
ACCEPT     icmp -- 0.0.0.0/0              0.0.0.0/0              icmptype 0
ACCEPT     icmp -- 0.0.0.0/0              0.0.0.0/0              icmptype 0

Chain OUTPUT (policy DROP)
target     prot opt source                destination            icmptype 0
ACCEPT     icmp -- 0.0.0.0/0              0.0.0.0/0              icmptype 0
```

Screenshot 2.B.6: External host pinging internal host

```
root@ec37fe01db0d:/# 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 ---
5 packets transmitted, 0 received, 100% packet loss, time 4081ms
```

Screenshot 2.B.7: External host pinging router

```
root@ec37fe01db0d:/# 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=0.058 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.057 ms
^C
--- 10.9.0.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1009ms
rtt min/avg/max/mdev = 0.057/0.057/0.058/0.000 ms
```

Screenshot 2.B.8: Internal host pinging external host

```
root@c366d503094d:/# 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=0.073 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=63 time=0.069 ms
^C
--- 10.9.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1032ms
rtt min/avg/max/mdev = 0.069/0.071/0.073/0.002 ms
```


Screenshot 2.B.9: Internal host telnet into external host

```
root@c366d503094d:/# telnet 10.9.0.5
Trying 10.9.0.5...
^C
```

Screenshot 2.B.10: External host telnet into router

```
root@ec37fe01db0d:/# telnet 10.9.0.11
Trying 10.9.0.11...
^C
```

Screenshot 2.B.11: External host telnet into internal host

```
root@ec37fe01db0d:/# telnet 192.168.60.5
Trying 192.168.60.5...
^C
```

Screenshot 2.B.12: Internal host telnet into router

```
root@c366d503094d:/# telnet 192.168.60.11
Trying 192.168.60.11...
^C
```

Task 2.C: Protecting Internal Servers

In this part, I learned how to protect the TCP servers inside the internal network. For this task, I used iptables to set up firewall rules. The rules are as follows:

1. Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts.
2. Outside hosts cannot access other internal servers.
3. Internal host can access all the internal servers.
4. Internal hosts cannot access external servers.
5. Connection tracking mechanism is not allowed.

To satisfy these conditions, the first rule I did was to make a rule to allow external hosts to only send TCP packets with destination IP address 192.168.60.5 and destination port 23 (telnet). Also, I made a rule to allow only 192.168.60.5 to send TCP packets externally from its source port 23. This satisfies condition #1. I then set everything else to be dropped. Since this is the case, condition #2 is satisfied because no other internal host or server can be accessed if it's not 192.168.60.5's port 23. Condition #3 is automatically met because the internal hosts do not have to go through the router to access other internal servers. By dropping everything else, condition #4 is met since packets can only go out through 192.168.60.5 port 23. And condition #5 is satisfied since I did not use any connection tracking mechanism.

I tested the rules with the following scenarios:

- Used 10.9.0.5 (external host) to telnet into 192.168.60.5 (internal host) and was able to establish a connection showing successful application of rule #1.
- Used 10.9.0.5 (external host) to telnet into 192.168.60.6 (other internal host) and was not able to connect showing successful application of rule #2.
- Used 192.168.60.5 (internal host) to telnet into 192.168.60.6 (other internal host) and was able to connect showing successful application of rule #3.
- Used 192.168.60.5 (internal host) to telnet into 10.9.0.5 (external host) and was not able to connect showing successful application of rule #4.

Screenshot 2.C.1: adding the appropriate rules to the filter table

```
root@3b685f768d39:/# iptables -A FORWARD -i eth0 -o eth1 -p tcp -d 192.168.60.5 --dport 23 -j ACCEPT
root@3b685f768d39:/# iptables -A FORWARD -i eth1 -o eth0 -p tcp -s 192.168.60.5 --sport 23 -j ACCEPT
root@3b685f768d39:/# iptables -P OUTPUT DROP
root@3b685f768d39:/# iptables -P INPUT DROP
root@3b685f768d39:/# iptables -P FORWARD DROP
```

Screenshot 2.C.2: filter table after rules are applied

```
root@3b685f768d39:/# iptables -t filter -L -n
Chain INPUT (policy DROP)
target      prot opt source                destination

Chain FORWARD (policy DROP)
target      prot opt source                destination      tcp dpt:23
ACCEPT     tcp  --  0.0.0.0/0             192.168.60.5    tcp dpt:23
ACCEPT     tcp  --  192.168.60.5         0.0.0.0/0       tcp spt:23

Chain OUTPUT (policy DROP)
target      prot opt source                destination
```

Screenshot 2.C.3: external host telnet into 192.168.60.5

```
root@ec37fe01db0d:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.

^]
telnet> █
```

Screenshot 2.C.4: external host attempting to telnet into internal host that is not 192.168.60.5

```
root@ec37fe01db0d:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
```

Screenshot 2.C.5: internal host connecting to another internal host

```
root@c366d503094d:/# telnet 192.168.60.6
Trying 192.168.60.6...
Connected to 192.168.60.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
87b6cbaa6d4c login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)
```

Screenshot 2.C.6: internal host attempting to connect to external host

```
root@c366d503094d:/# 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

In this task, I explored connection tracking and how to set up a stateful firewall. I utilized conntrack to observe how iptables track connections.

ICMP experiment: In this part, I observed the connection tracking information on the router for ICMP packets. I used 10.9.0.5 machine to ping 192.168.60.5 and then checked the connection tracker on the router. I see that it tracked the request and also the reply. It has information logged such as source and destination addresses, type and code for the ICMP packet. The ICMP connection state was kept for about 30 seconds.

Screenshot 3.A.1: 10.9.0.5 pinged 192.168.60.5

```
root@ec37fe01db0d:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.080 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.081 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.066 ms
^C
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2065ms
rtt min/avg/max/mdev = 0.066/0.075/0.081/0.006 ms
```

Screenshot 3.A.2: Connection tracker on router

```
root@3b685f768d39:/# conntrack -L
icmp      1 15 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=68 src=192.168.60.5 dst=10.9.0.5 type=0
code=0 id=68 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@3b685f768d39:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
```

UDP experiment: In this part, I observed the connection tracking information on the router for UDP packets. I started a netcat UDP server on 192.168.60.5 and then sent out a UDP packet on 10.9.0.5. The connection tracker on the router tracked the connection for about 30 seconds. It has address and port information (which the ICMP experiment did not) and also shows if a message was unreplied since I had not replied from the server. This log time was similar to the ICMP experiment.

Screenshot 3.A.3: Started a netcat UDP server on 192.168.60.5

```
root@c366d503094d:/# nc -lu 9090
hello
^C
```

Screenshot 3.A.4: Sent out UDP packet on 10.9.0.5

```
root@ec37fe01db0d:/# nc -u 192.168.60.5 9090
hello
^C
```

Screenshot 3.A.5: Connection tracker on router showing information for the UDP packet

```
root@3b685f768d39:/# conntrack -L
udp      17 4 src=10.9.0.5 dst=192.168.60.5 sport=58639 dport=9090 [UNREPLIED] src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=58639 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@3b685f768d39:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
```

TCP experiment: In this part, I observed the connection tracking information on the router for TCP packets. I started a netcat TCP server on 192.168.60.5 and then sent out a TCP packet on 10.9.0.5. The connection tracker on the router had different on there this time. It showed “ESTABLISHED” and the time on the log was really long, about 12 hours it looked like. When I exited the program, the log changed to “TIME_WAIT” state. And now it looks like it stayed on the tracker for 2 minutes before getting dropped off. This behavior makes sense because for UDP and ICMP packets, they are connectionless so it’s not guarantee that the packets are delivered and you don’t know if the hosts at each end are still there. You don’t want to keep the connection open for too long since the hosts might not be there anymore. For TCP, it’s connection oriented so the connection will stay open for a long time until the hosts closes the connection. Once the TCP closing process is initiated, the tracker goes to a TIME_WAIT state which correlates to the TCP FIN procedure.

Screenshot 3.A.6: Started a netcat TCP server on 192.168.60.5

```
root@c366d503094d:/# nc -l 9090
hello
root@c366d503094d:/#
```

Screenshot 3.A.7: Sent out TCP packet on 10.9.0.5

```
root@ec37fe01db0d:/# nc 192.168.60.5 9090
hello
^C
```

Screenshot 3.A.8: Connection tracker on router showing information for the “ESTABLISHED” TCP connection

```
root@3b685f768d39:/# conntrack -L
tcp      6 431996 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=41798 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=41798 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

Screenshot 3.A.9: Connection tracker on router showing information for the “TIME_WAIT” TCP connection after program was closed

```
tcp      6 111 TIME_WAIT src=10.9.0.5 dst=192.168.60.5 sport=41798 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=41798 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

Task 3.B: Setting Up a Stateful Firewall

For this task, I will be rewriting the firewall rules in Task 2.C but implementing a stateful firewall with rules based on connections. The updated rules are as follows:

1. Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts.
2. Outside hosts cannot access other internal servers.
3. Internal host can access all the internal servers.
4. Internal hosts can access external servers.

I used the provided example rule to accept packets from already established connections. To begin the connection, I made a rule to allow SYN packets to go to 192.168.60.6 port 23. The other change to the rules is that internal hosts can now access external servers. To implement this, I allowed SYN packets to go out externally.

I tested the rules with the following scenarios:

- Used 10.9.0.5 (external host) to telnet into 192.168.60.5 (internal host) and was able to establish a connection showing successful application of rule #1.
- Used 10.9.0.5 (external host) to telnet into 192.168.60.6 (other internal host) and was not able to connect showing successful application of rule #2.
- Used 192.168.60.5 (internal host) to telnet into 192.168.60.6 (other internal host) and was able to connect showing successful application of rule #3.
- Used 192.168.60.5 (internal host) to telnet into 10.9.0.5 (external host) and able to connect showing successful application of rule #4.

Screenshot 3.B.1: adding rules to the filter table

```
root@3b685f768d39:/# iptables -A FORWARD -p tcp -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
root@3b685f768d39:/# iptables -A FORWARD -p tcp -i eth0 -o eth1 -d 192.168.60.5 --dport 23 --syn -m conntrack --ctstate NEW -j ACCEPT
root@3b685f768d39:/# iptables -A FORWARD -p tcp -i eth1 -o eth0 --syn -m conntrack --ctstate NEW -j ACCEPT
root@3b685f768d39:/# iptables -P FORWARD DROP
root@3b685f768d39:/# iptables -P OUTPUT DROP
root@3b685f768d39:/# iptables -P INPUT DROP
```

Screenshot 3.B.2: filter table after all the rules have been added

```
root@3b685f768d39:/# iptables -t filter -L -n
Chain INPUT (policy DROP)
target    prot opt source                destination
Chain FORWARD (policy DROP)
target    prot opt source                destination
ACCEPT    tcp  --  0.0.0.0/0             0.0.0.0/0             ctstate RELATED,ESTABLISHED
ACCEPT    tcp  --  0.0.0.0/0             192.168.60.5          tcp dpt:23 flags:0x17/0x02 ctstate NEW
ACCEPT    tcp  --  0.0.0.0/0             0.0.0.0/0             tcp flags:0x17/0x02 ctstate NEW
Chain OUTPUT (policy DROP)
target    prot opt source                destination
```


Screenshot 3.B.3: external host successfully telnet into 192.168.60.5

```
root@ec37fe01db0d:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
^]
telnet> quit
Connection closed.
```

Screenshot 3.B.4: external host unable to telnet into other internal host

```
root@ec37fe01db0d:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
```

Screenshot 3.B.5: internal host successfully telnet into other internal host

```
root@c366d503094d:/# telnet 192.168.60.6
Trying 192.168.60.6...
Connected to 192.168.60.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
87b6cbaa6d4c login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)
```

Screenshot 3.B.6: internal host successfully telnet into external host

```
root@c366d503094d:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
ec37fe01db0d login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)
```

Thinking about how to write these rules without using connection tracking mechanism, I think it will be harder to implement. For the case of allowing internal hosts to access external servers, I can write a rule to allow SYN packets to go out externally. And then for the return external packets, I can write a rule to check for the SYN ACK flags. A disadvantage to this is since we are not checking for established connections, an attacker can send a fake SYN ACK packet for a request that doesn't exist. An advantage for not using connection tracking mechanisms could be less overhead. For rules that use connection tracking mechanisms, the advantage of using is that you have extra security against unsolicited replies to requests. A disadvantage could be that the connection tracker has limited space and that space can run out.

Task 4: Limiting Network Traffic

For Task 4, I learned how to limit the number of packets that can pass through the firewall. I applied the provided rules to the router. In the first scenario where both rules were used, after the initial 5 packets, the amount of packets that went through afterwards were throttled. Packet 6 was dropped, packets 8-12 were dropped, etc. In the second scenario where only the first rule was used, the rule did not work. The packets were through like normal. This shows that the second rule is needed. This is because the default policy without the second rule is to accept the packet. After the threshold is met, the firewall does the default policy which is to keep forwarding the packets. We need the second rule to change the action to dropping the packets after the first rule is met.

Screenshot 4.1: Adding the rules to the filter table

```
root@3b685f768d39:/# iptables -A FORWARD -s 10.9.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT
root@3b685f768d39:/# iptables -A FORWARD -s 10.9.0.5 -j DROP
```

Screenshot 4.2: Filter table after both rules are applied

```
root@3b685f768d39:/# iptables -t filter -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source                destination

Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
ACCEPT     all  --  10.9.0.5                0.0.0.0/0             limit: avg 10/min burst 5
DROP       all  --  10.9.0.5                0.0.0.0/0

Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination
```

Screenshot 4.3: Ping results with both rules active

```
root@ec37fe01db0d:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.096 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.084 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.066 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.071 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.055 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.081 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.067 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=63 time=0.064 ms
64 bytes from 192.168.60.5: icmp_seq=25 ttl=63 time=0.065 ms
64 bytes from 192.168.60.5: icmp_seq=31 ttl=63 time=0.065 ms
64 bytes from 192.168.60.5: icmp_seq=37 ttl=63 time=0.067 ms
64 bytes from 192.168.60.5: icmp_seq=43 ttl=63 time=0.078 ms
^C
--- 192.168.60.5 ping statistics ---
45 packets transmitted, 12 received, 73.3333% packet loss, time 45060ms
rtt min/avg/max/mdev = 0.055/0.071/0.096/0.010 ms
```

Screenshot 4.4: filter table with only first rule active

```
root@3b685f768d39:/# iptables -t filter -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source                destination

Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
ACCEPT     all  --  10.9.0.5              0.0.0.0/0              limit: avg 10/min burst 5

Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination
```

Screenshot 4.5: Ping results with only first rule active

```
root@ec37fe01db0d:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.076 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.068 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.068 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.075 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.069 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=0.047 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.068 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=0.069 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=0.066 ms
64 bytes from 192.168.60.5: icmp_seq=10 ttl=63 time=0.069 ms
64 bytes from 192.168.60.5: icmp_seq=11 ttl=63 time=0.068 ms
64 bytes from 192.168.60.5: icmp_seq=12 ttl=63 time=0.068 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.070 ms
^C
--- 192.168.60.5 ping statistics ---
13 packets transmitted, 13 received, 0% packet loss, time 12278ms
rtt min/avg/max/mdev = 0.047/0.067/0.076/0.006 ms
```


Task 5: Load Balancing

For this task, I learned load balancing, another application of iptables. There are two modes that we can do: random and nth. To set up the experiment, I ran 'nc -luk 8080' on all three internal hosts (192.168.60.5, 192.168.60.6, and 192.168.60.7). I started off by testing the nth mode. I inputted the given rule into the nat table. This rule should route every 3 UDP packets going to 8080 to 192.168.60.5. Then, I used 10.9.0.5 to send 4 UDP packet to the router's 8080 port. I expected 192.168.60.5 to receive 2 UDP packets and that is exactly what I observed. Afterwards, I added two additional rules for the other two internal servers. The only thing I changed was the IP address since I want to keep the occurrence the same – every 3. I used 10.9.0.5 to send 3 UDP packets to the router's 8080 port and each of the three internal servers received 1 UDP packet.

Screenshot 5.1: Start the servers on each of the three internal machines

```
root@c366d503094d:/# nc -luk 8080
TX errors 0 dropped 0 over
root@87b6cbaa6d4c:/# nc -luk 8080
root@7a8b06717219:/# nc -luk 8080
```

Screenshot 5.2: Inputting the first nth mode rule

```
root@3b685f768d39:/# 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
```

Screenshot 5.3: nat table after rule is active

```
root@3b685f768d39:/# iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target      prot opt source                destination            udp dpt:8080 statistic mode nth
DNAT        udp  --  0.0.0.0/0              0.0.0.0/0              every 3 to:192.168.60.5:8080
```

Screenshot 5.3: 10.9.0.5 sending 4 UDP packets

```
root@ec37fe01db0d:/# echo hello | nc -u 10.9.0.11 8080
^C
root@ec37fe01db0d:/# echo hello | nc -u 10.9.0.11 8080
root@ec37fe01db0d:/# ^C
root@ec37fe01db0d:/# echo hello | nc -u 10.9.0.11 8080
root@ec37fe01db0d:/# echo hello | nc -u 10.9.0.11 8080
```

Screenshot 5.4: 192.168.60.5 received 2 UDP packets

```
root@c366d503094d:/# nc -luk 8080
hello
hello
```

Screenshot 5.5: Adding 2 more rules for the other 2 internal servers

```
root@3b685f768d39:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth
--every 3 --packet 0 -j DNAT --to-destination 192.168.60.6:8080
root@3b685f768d39:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth
--every 3 --packet 0 -j DNAT --to-destination 192.168.60.7:8080
```

Screenshot 5.6: nat table after all 3 rules are active

```
root@3b685f768d39:/# iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source                destination              udp dpt:8080 statistic mode nth
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 3 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 3 to:192.168.60.7:8080
```

Screenshot 5.7: 10.9.0.5 sends 3 UDP packets

```
root@ec37fe01db0d:/# echo hello | nc -u 10.9.0.11 8080
^C
root@ec37fe01db0d:/# echo hello | nc -u 10.9.0.11 8080
^C
root@ec37fe01db0d:/# echo hello | nc -u 10.9.0.11 8080
^C
```

Screenshot 5.8: Each of the three internal servers received 1 packet each

```
root@8root@7a8broot@c366d503094d:/# nc -luk 8080
hellohellohello
```

After testing the nth mode, I went on to test the random mode. Since there are three internal servers that I want to load balance, I want the probability for each to be one third. I wrote the rules for the three with 0.33 (estimated) probability. I used 10.9.0.5 to send 13 UDP packets and the packets were distributed with 6, 5, and 2 on the machines. Since this is a small sample size, the probability does not appear to be the same but with a larger sample size the probability for each internal server should be the same.

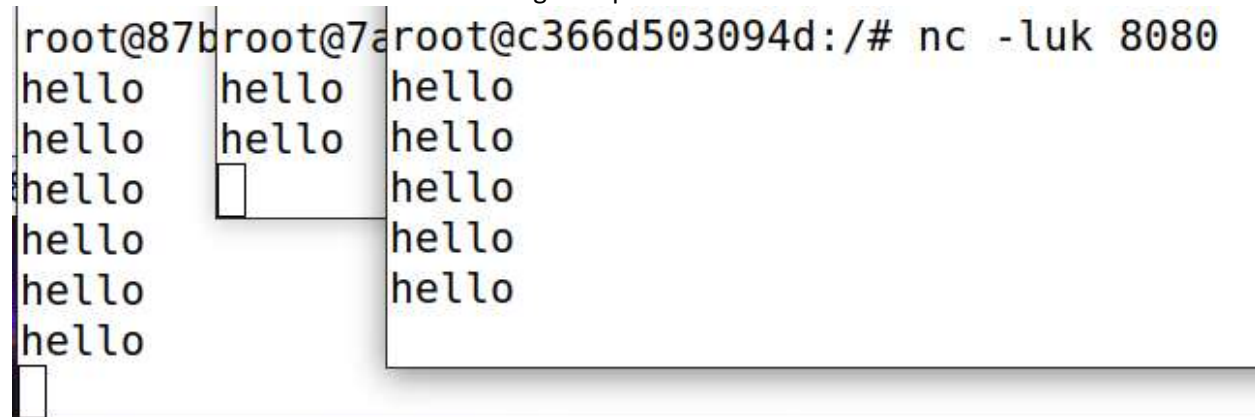
Screenshot 5.9: Adding rules for using random mode

```
root@3b685f768d39:/# 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
root@3b685f768d39:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.33 -j DNAT --to-destination 192.168.60.6:8080
root@3b685f768d39:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.33 -j DNAT --to-destination 192.168.60.7:8080
```

Screenshot 5.10: nat table after all 3 rules are active

```
root@3b685f768d39:/# iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source                destination              udp dpt:8080 statistic mode random
DNAT       udp  --  0.0.0.0/0              0.0.0.0/0                udp dpt:8080 statistic mode random
dom probability 0.330000000007 to:192.168.60.5:8080
DNAT       udp  --  0.0.0.0/0              0.0.0.0/0                udp dpt:8080 statistic mode random
dom probability 0.330000000007 to:192.168.60.6:8080
DNAT       udp  --  0.0.0.0/0              0.0.0.0/0                udp dpt:8080 statistic mode random
dom probability 0.330000000007 to:192.168.60.7:8080
```

Screenshot 5.11: Results for 10.9.0.5 sending UDP packets with random mode



```
root@87b:root@7a:root@c366d503094d:/# nc -luk 8080
hello      hello      hello
hello      hello      hello
hello      [ ]         hello
hello      [ ]         hello
hello      [ ]         hello
hello      [ ]         hello
hello      [ ]         [ ]
[ ]        [ ]        [ ]
```

Conclusion/Discussion

This assignment was a good introduction into firewalls, netfilter, and iptables. Before this, I did not know how firewall rules were implemented. The tasks in the assignment showed me how easy it is to set up firewall rules. I can see that these are powerful tools and you have to be very careful when working with these tools since they can cause system crashes or change Docker containers settings. I look forward to exploring more applications of iptables on my own.