Network Security Lab 5: Firewall

NetID : ds7615

Task 1: Implementing a Simple Firewall

This new module enables us to extend the functionalities of the kernel, without rebuilding the kernel or even rebooting the computer.
The LabSetup folder comes with a sample code to create a simple kernel module.
Lets build the module with **make** command.

```
root@ubuntu-seed-labs:~/Labsetup/Files# cd kernel_module/
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# ls
Makefile  hello.c
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# nano hello.c
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# nano Makefile
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# make
make -C /lib/modules/5.4.0-122-generic/build M=/root/Labsetup/Files/kernel_modu
le modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-122-generic'
  CC [M]  /root/Labsetup/Files/kernel_module/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /root/Labsetup/Files/kernel_module/hello.mod.o
  LD [M]  /root/Labsetup/Files/kernel_module/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-122-generic'
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# ls
Makefile        hello.c   hello.mod    hello.mod.o  modules.order
Module.symvers  hello.ko  hello.mod.c  hello.o
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module#
```

Once the module is compiled. It generates files with various extension. .ko file is the one of our interest.

Task 1.A: - Implement a Simple Kernel Module

```
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module#
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# sudo insmod hello.ko
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# lsmod | grep hello
hello                  16384  0
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# modinfo hello.ko
filename:       /root/Labsetup/Files/kernel_module/hello.ko
license:        GPL
srcversion:     717A72281ACFAA8385B33A8
depends:
retpoline:      Y
name:           hello
vermagic:       5.4.0-122-generic SMP mod_unload modversions
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module# sudo rmmod hello
root@ubuntu-seed-labs:~/Labsetup/Files/kernel_module#
```

In the above image after building the module. We are inserting the module in to the list of kernel processes, so it can run automatically with other processes.

```
[2809142.187883] hello: module verifi
y missing - tainting kernel
[2809142.188247] Hello World!
[2809220.567738] Bye-bye World!.
[2809270.401469] Hello World!
[2809325.501606] Bye-bye World!.
root@ubuntu-seed-labs:~/Labsetup/File
```

The hello program when starts the execution prints **Hello world** into the kernel logs. And when it gets removed. It prints **Bye Bye World.**

Task 1.B: - Implement a Simple Firewall Using Net filter

In this task, we will write our packet filtering program as an LKM, and then insert it into the packet process- ing path inside the kernel. This cannot be easily done in the past before netfilter was introduced into Linux.

Subtask 1: Hooking to Netfilter.
Labsetup comes with a sample packet filter program. Lets test it out to observe its behavior.
Before that, Lets look at how ping and dig command works.

```
root@ubuntu-seed-labs:~# ping www.example.com
PING www.example.com (93.184.216.34) 56(84) bytes of data.
64 bytes from 93.184.216.34 (93.184.216.34): icmp_seq=1 ttl=58 time=2.19 ms
64 bytes from 93.184.216.34 (93.184.216.34): icmp_seq=2 ttl=58 time=1.49 ms
64 bytes from 93.184.216.34 (93.184.216.34): icmp_seq=3 ttl=58 time=1.46 ms
64 bytes from 93.184.216.34 (93.184.216.34): icmp_seq=4 ttl=58 time=1.46 ms
^C
--- www.example.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.460/1.650/2.188/0.310 ms
root@ubuntu-seed-labs:~#
```

```
root@ubuntu-seed-labs:~/Labsetup/Files/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: 42070
;; 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.        13237   IN      A       93.184.216.34

;; Query time: 4 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Sun Dec 17 04:38:48 UTC 2023
;; MSG SIZE  rcvd: 60
```

In the above images ping and DNS request for example.com got a successful response.
Let look at the code of the seedFilter.

Below images shown is the code for seed filter which makes use of NetFilter.
Netfilter is created to enable authorized users to manipulate packets efficiently. This is accomplished through the integration of hooks within the Linux kernel. These hooks are strategically placed, including along the paths of incoming and outgoing packets. To manipulate incoming packets, users can connect their programs, embedded within Loadable Kernel Modules (LKMs), to the relevant hooks. When an incoming packet is received, the connected program is triggered. This program has the authority to determine whether the packet should be blocked and also has the capability to modify the packet as needed.

The program contains 2 hooks one to block UDP packets and another is to print information.

```c
#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/icmp.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;

   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;
}
```

```c
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_LOCAL_OUT;
   hook1.pf = PF_INET;
   hook1.priority = NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net, &hook1);

   hook2.hook = blockUDP;
   hook2.hooknum = NF_INET_POST_ROUTING;
   hook2.pf = PF_INET;
```

```
   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");
```

In order to see this filter in action we have to compile it and insert it in to kernel process.

```
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# sudo insmod seedFilter.ko
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# lsmod | grep seedFilter
seedFilter             16384  0
```

```
[3190214.785098]     162.243.190.66  --> 159.203.72.46 (TCP)
[3190214.785110] *** LOCAL_IN
[3190214.785111]     162.243.190.66  --> 159.203.72.46 (TCP)
[3190214.787805] Registering filters.
[3190214.795114] *** LOCAL_OUT
[3190214.795117]     159.203.72.46  --> 162.243.190.66 (TCP)
[3190214.795119] *** LOCAL_OUT
[3190214.795120]     159.203.72.46  --> 162.243.190.66 (TCP)
[3190214.795161] *** POST_ROUTING
```

When we insert the program into kernel process, it prints out **Filter Registered** into kernel logs.

```
root@ubuntu-seed-labs:~/Labsetup/Files/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

root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter#
```

```
[2814612.676621] *** LOCAL_OUT
[2814612.676622]     159.203.72.46  --> 8.8.8.8 (UDP)
[2814612.676630] *** Dropping 8.8.8.8 (UDP), port 53
[2814612.821370] *** LOCAL_OUT
[2814612.821398]     159.203.72.46  --> 212.70.149.74 (TCP)
[2814615.079066] *** LOCAL_OUT
[2814615.079098]     159.203.72.46  --> 212.70.149.74 (TCP)
[2814615.984896] *** LOCAL_OUT
[2814615.984930]     159.203.72.46  --> 212.70.149.74 (TCP)
[2814616.079807] *** LOCAL_OUT
[2814616.079831]     159.203.72.46  --> 212.70.149.74 (TCP)
[2814616.793386] *** LOCAL_OUT
[2814616.793424]     159.203.72.46  --> 162.243.190.66 (TCP)
[2814617.678358] *** LOCAL_OUT
[2814617.678360]     159.203.72.46  --> 8.8.8.8 (UDP)
[2814617.678376] *** Dropping 8.8.8.8 (UDP), port 53
[2814617.692570] *** LOCAL_OUT
[2814617.692574]     159.203.72.46  --> 162.243.190.66 (TCP)
[2814617.692775] *** LOCAL_OUT
[2814617.692777]     159.203.72.46  --> 162.243.190.66 (TCP)
```

In the above images we can see that. When we try to request DNS information. Server could not be reached and in the kernel logs we observe UDP packers dropping because of the SeedFilter.

```
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# sudo rmmod seedFilter
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter#
```

Lets remove the seedFilter from the kernel process.

```
root@ubuntu-seed-labs:~# dmesg | tail
[2815078.602887]     159.203.72.46  --> 162.243.190.66 (TCP)
[2815078.621578] *** LOCAL_OUT
[2815078.621580]     159.203.72.46  --> 162.243.190.66 (TCP)
[2815080.918675] *** LOCAL_OUT
[2815080.918697]     159.203.72.46  --> 45.142.182.77 (TCP)
[2815082.403023] *** LOCAL_OUT
[2815082.403027]     159.203.72.46  --> 162.243.190.66 (TCP)
[2815082.405691] *** LOCAL_OUT
[2815082.405693]     159.203.72.46  --> 162.243.190.66 (TCP)
[2815082.415583] The filters are being removed.
root@ubuntu-seed-labs:~#
```

It leaves a trace in kernel logs saying the filter is removed.

Subtask 2: Adding more hooks.

We copy the template from seed Filter and try to expand it by adding more hooks at different different stages of packet transmission.

```c
#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/icmp.h>
#include <linux/if_ether.h>
#include <linux/inet.h>


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


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;
}
```

```c
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;
}
```

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

   // NF_INET_PRE_ROUTING
   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);


   //NF_INET_LOCAL_IN
   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);

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

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

   //NF_INET_POST_ROUTING
   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 "seedPrint : 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");
```

Above image shows the code of the seedPrint file. There are 5 hooks in total each waiting on each stage of packet transmission like Pre routing and Post routing.

```
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# sudo insmod seedPrint.ko
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# lsmod | grep seedPrint
seedPrint              16384  0
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# modinfo seedPrint.ko
filename:       /root/Labsetup/Files/packet_filter/seedPrint.ko
license:        GPL
srcversion:     36F55E4266D1573DB0B559B
depends:
retpoline:      Y
name:           seedPrint
vermagic:       5.4.0-122-generic SMP mod_unload modversions
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter#
```

Lets insert it in to kernel to test it out.

```
root@ubuntu-seed-labs:~# dmesg
[3192748.883045] seedPrint : Registering filters.
[3192748.890735] *** LOCAL_OUT
[3192748.890739]     159.203.72.46  --> 198.211.111.194 (TCP)
[3192748.890749] *** POST_ROUTING
[3192748.890750]     159.203.72.46  --> 198.211.111.194 (TCP)
[3192748.902483] *** PRE_ROUTING
[3192748.902487]     198.211.111.194  --> 159.203.72.46 (TCP)
[3192748.902500] *** LOCAL_IN
[3192748.902501]     198.211.111.194  --> 159.203.72.46 (TCP)
[3192748.928004] *** PRE_ROUTING
[3192748.928035]     198.211.111.194  --> 159.203.72.46 (TCP)
[3192748.928050] *** LOCAL_IN
[3192748.928052]     198.211.111.194  --> 159.203.72.46 (TCP)
[3192750.837719] *** PRE_ROUTING
[3192750.837754]     162.243.188.66  --> 159.203.72.46 (TCP)
[3192750.837771] *** LOCAL_IN
[3192750.837774]     162.243.188.66  --> 159.203.72.46 (TCP)
[3192750.841438] *** PRE_ROUTING
[3192750.841441]     162.243.188.66  --> 159.203.72.46 (TCP)
[3192750.841453] *** LOCAL_IN
[3192750.841453]     162.243.188.66  --> 159.203.72.46 (TCP)
[3192750.841534] *** LOCAL_OUT
[3192750.841534]     159.203.72.46  --> 162.243.188.66 (TCP)
[3192750.841537] *** POST_ROUTING
[3192750.841538]     159.203.72.46  --> 162.243.188.66 (TCP)
root@ubuntu-seed-labs:~#
```

In the below image we again try getting DNS information using dig command.
In the kernel logs we can observe all the packets with their respective stage of the process is very nicely organized.

```
root@ubuntu-seed-labs:~/Labsetup/Files/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: 13946
;; 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.        4332    IN      A       93.184.216.34

;; Query time: 0 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Thu Dec 21 14:12:10 UTC 2023
;; MSG SIZE  rcvd: 60
```

```
root@ubuntu-seed-labs:~# dmesg | grep UDP -B1
[3193100.089816] *** LOCAL_OUT
[3193100.089820]     127.0.0.1  --> 127.0.0.1 (UDP)
[3193100.089833] *** POST_ROUTING
[3193100.089834]     127.0.0.1  --> 127.0.0.1 (UDP)
[3193100.089851] *** PRE_ROUTING
[3193100.089852]     127.0.0.1  --> 127.0.0.1 (UDP)
[3193100.089853] *** LOCAL_IN
[3193100.089854]     127.0.0.1  --> 127.0.0.1 (UDP)
[3193100.090046] *** LOCAL_OUT
[3193100.090047]     159.203.72.46  --> 8.8.8.8 (UDP)
[3193100.090054] *** POST_ROUTING
[3193100.090055]     159.203.72.46  --> 8.8.8.8 (UDP)
[3193100.092943] *** PRE_ROUTING
[3193100.092946]     8.8.8.8  --> 159.203.72.46 (UDP)
[3193100.092961] *** LOCAL_IN
[3193100.092962]     8.8.8.8  --> 159.203.72.46 (UDP)
```

```
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# sudo rmmod seedPrint
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter#
```

When Lets remove the filter as our experiment is working as expected.

```
[3193616.887066] *** LOCAL_OUT
[3193616.887067]     159.203.72.46  --> 198.211.111.194 (TCP)
[3193616.887070] *** POST_ROUTING
[3193616.887071]     159.203.72.46  --> 198.211.111.194 (TCP)
[3193616.888505] *** LOCAL_OUT
[3193616.888509]     159.203.72.46  --> 198.211.111.194 (TCP)
[3193616.888517] *** POST_ROUTING
[3193616.888518]     159.203.72.46  --> 198.211.111.194 (TCP)
[3193616.901778] seedPrint : The filters are being removed.
root@ubuntu-seed-labs:~#
```

Subtask 3: Block ICMP and TCP

In this lab we extend the hook functionality to block all kinds of packets. And observe its behavior.

Before diving into the experiment let's make sure all the basic functionalities we will use later on for testing are working properly.
In the below image we can see that ping and telnet are working properly.

```
root@ubuntu-seed-labs:~# docker exec -it d670ad1dc5de  /bin/bash
root@d670ad1dc5de:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.150 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.095 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.091 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.085 ms
64 bytes from 10.9.0.1: icmp_seq=5 ttl=64 time=0.072 ms
^C
--- 10.9.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4101ms
rtt min/avg/max/mdev = 0.072/0.098/0.150/0.026 ms
root@d670ad1dc5de:/#
```

```
root@d670ad1dc5de:/#
root@d670ad1dc5de:/# telnet 10.9.0.1
Trying 10.9.0.1...
Connected to 10.9.0.1.
Escape character is '^'.
Ubuntu 20.04.1 LTS
ubuntu-seed-labs login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-122-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.

seed@833fe3104c4a:~$ exit
logout
Connection closed by foreign host.
root@d670ad1dc5de:/#
```

```c
#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/icmp.h>
#include <linux/if_ether.h>
#include <linux/inet.h>


static struct nf_hook_ops hook1, hook2, hook3, hook4;

// block udp to 8.8.8.8:53
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;
}
```

The code applies all different kinds of hooks in Pre routing stage it self.

```c
// bock ping to 10.9.0.1
unsigned int blockICMP(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
   struct iphdr *iph;
   struct icmphdr *icmph;

   //u16  port   = 53;
   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) {
       icmph = icmp_hdr(skb);
       if (iph->daddr == ip_addr && icmph->type == ICMP_ECHO){
            printk(KERN_WARNING "*** Dropping %pI4 (ICMP)", &(iph->daddr));
            return NF_DROP;
        }
   }
   return NF_ACCEPT;
}

// bock telnet to 10.9.0.1:23
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
   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);
```

```c
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;
}
```

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

   hook1.hook = printInfo;
   hook1.hooknum = NF_INET_LOCAL_OUT;
   hook1.pf = PF_INET;
   hook1.priority = NF_IP_PRI_FIRST;
   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);

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

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


   return 0;
}

void removeFilter(void) {
   printk(KERN_INFO "seedBlock : 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);

}

module_init(registerFilter);
module_exit(removeFilter);

MODULE_LICENSE("GPL");
```

Lets compile the code using make.

```
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# make
make -C /lib/modules/5.4.0-122-generic/build M=/root/Labsetup/Files/packet_filter modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-122-generic'
  CC [M]  /root/Labsetup/Files/packet_filter/seedBlock.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /root/Labsetup/Files/packet_filter/seedBlock.mod.o
  LD [M]  /root/Labsetup/Files/packet_filter/seedBlock.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-122-generic'
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# ls
Makefile        modules.order  seedBlock.ko   seedBlock.mod.c  seedBlock.o   seedPrint.c
Module.symvers  seedBlock.c    seedBlock.mod  seedBlock.mod.o  seedFilter.c
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter#
```

Below image shows command to insert a program into the kernel .

```
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# insmod seedBlock.ko
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# lsmod | grep seedBlock
seedBlock              16384  0
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter# modinfo seedBlock.ko
filename:       /root/Labsetup/Files/packet_filter/seedBlock.ko
license:        GPL
srcversion:     66C9B2BA7A9D1C273D53F5A
depends:
retpoline:      Y
name:           seedBlock
vermagic:       5.4.0-122-generic SMP mod_unload modversions
root@ubuntu-seed-labs:~/Labsetup/Files/packet_filter#
```

When we try to ping after enabling seedblock. We can see that ICMP packets are dropping

```
root@d670ad1dc5de:/# 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 ---
200 packets transmitted, 0 received, 100% packet loss, time 203755ms

root@d670ad1dc5de:/#
```

```
[3213340.122570]     159.203.72.46  --> 104.28.238.182 (TCP)
[3213340.233007] *** Dropping 10.9.0.1 (ICMP)
[3213341.257043] *** Dropping 10.9.0.1 (ICMP)
[3213342.280977] *** Dropping 10.9.0.1 (ICMP)
[3213343.304986] *** Dropping 10.9.0.1 (ICMP)
[3213344.319355] *** LOCAL_OUT
[3213344.319359]     159.203.72.46  --> 162.243.188.66 (TCP)
[3213344.328980] *** Dropping 10.9.0.1 (ICMP)
[3213344.493296] *** LOCAL_OUT
[3213344.493323]     159.203.72.46  --> 172.104.138.223 (TCP)
[3213345.013241] *** LOCAL_OUT
[3213345.013270]     159.203.72.46  --> 198.211.111.194 (TCP)
[3213345.352960] *** Dropping 10.9.0.1 (ICMP)
[3213346.376925] *** Dropping 10.9.0.1 (ICMP)
[3213347.387053] *** LOCAL_OUT
[3213347.387084]     159.203.72.46  --> 162.243.190.66 (TCP)
[3213347.400940] *** Dropping 10.9.0.1 (ICMP)
[3213347.992620] *** LOCAL_OUT
[3213347.992649]     159.203.72.46  --> 77.90.185.189 (TCP)
[3213348.424914] *** Dropping 10.9.0.1 (ICMP)
[3213349.448924] *** Dropping 10.9.0.1 (ICMP)
[3213350.472911] *** Dropping 10.9.0.1 (ICMP)
[3213351.496815] *** Dropping 10.9.0.1 (ICMP)
[3213351.752772] *** LOCAL_OUT
[3213351.752798]     159.203.72.46  --> 78.117.166.161 (TCP)
```

Same trend can be seen even in case of telnet connection where TCP packets are dropping.

```
root@d670ad1dc5de:/# telnet 10.9.0.1
Trying 10.9.0.1...
^C
root@d670ad1dc5de:/#
```

```
[3213485.635051]     159.203.72.46  --> 198.211.111.194 (TCP)
[3213485.638310] *** Dropping 10.9.0.1 (TCP), port 23
[3213485.638683] *** LOCAL_OUT
[3213485.638685]     159.203.72.46  --> 198.211.111.194 (TCP)
[3213485.734926] *** LOCAL_OUT
[3213485.734948]     159.203.72.46  --> 103.130.215.106 (TCP)
[3213485.927573] *** LOCAL_OUT
[3213485.927681]     159.203.72.46  --> 103.130.215.106 (TCP)
[3213486.305492] *** LOCAL_OUT
[3213486.305520]     159.203.72.46  --> 103.130.215.106 (TCP)
[3213486.312485] *** LOCAL_OUT
[3213486.312488]     159.203.72.46  --> 103.130.215.106 (TCP)
[3213486.663477] *** Dropping 10.9.0.1 (TCP), port 23
[3213486.735528] *** LOCAL_OUT
[3213486.735550]     159.203.72.46  --> 103.130.215.106 (TCP)
[3213487.062303] *** LOCAL_OUT
[3213487.062328]     159.203.72.46  --> 103.130.215.106 (TCP)
[3213487.062928] *** LOCAL_OUT
[3213487.062930]     159.203.72.46  --> 103.130.215.106 (TCP)
[3213487.253317] *** LOCAL_OUT
[3213487.253345]     159.203.72.46  --> 45.79.109.4 (TCP)
[3213487.394136] *** LOCAL_OUT
[3213487.394163]     159.203.72.46  --> 103.130.215.106 (TCP)
[3213488.003500] *** LOCAL_OUT
[3213488.003503]     159.203.72.46  --> 170.64.202.72 (TCP)
[3213488.217974] *** LOCAL_OUT
[3213488.217978]     159.203.72.46  --> 170.64.202.72 (TCP)
[3213488.679418] *** Dropping 10.9.0.1 (TCP), port 23
[3213489.173926] *** LOCAL_OUT
```

Conclusion: The accomplishment of Task 1 showcased the effective deployment of a packet-filtering firewall in the Linux kernel, leveraging Netfilter for streamlined packet manipulation. The implementation took into account considerations for container configurations and virtual IP addresses to ensure a cohesive and easily manageable setup. The successful deployment of a basic kernel module underscored the capabilities of Loadable Kernel Modules, allowing for dynamic expansion of kernel functionality without the necessity for system reboot or kernel rebuilding. This knowledge sets the groundwork for further exploration into integrating packet filtering features within a firewall. However, it's essential to note the potential challenge of system destabilization due to the involvement of low-level kernel operations, necessitating careful handling and thorough testing.

Task 2: Experimenting with Stateless Firewall Rules

In this assignment, we will utilize iptables to establish a firewall. The iptables firewall serves the dual purpose of not only filtering packets but also affecting modifications to them. To efficiently manage these firewall rules catering to different objectives, iptables employs a hierarchical structure consisting of tables, chains, and rules. Multiple tables exist, each specifying the primary function of the rules, such as packet filtering in the filter table or packet modification in the nat or mangle tables.

```
root@d670ad1dc5de:/# 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=63 time=0.161 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=63 time=0.126 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=63 time=0.129 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=63 time=0.181 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=63 time=0.149 ms
64 bytes from 10.9.0.11: icmp_seq=6 ttl=63 time=0.130 ms
^C
--- 10.9.0.11 ping statistics ---
28 packets transmitted, 28 received, 0% packet loss, time 27655ms
rtt min/avg/max/mdev = 0.122/0.155/0.225/0.028 ms
root@d670ad1dc5de:/#
```

```
root@d670ad1dc5de:/#
root@d670ad1dc5de:/# telnet 10.9.0.11
Trying 10.9.0.11...
Connected to 10.9.0.11.
Escape character is '^'.
Ubuntu 20.04.11 LTS
ubuntu-seed-labs login: seed
Password:
Welcome to Ubuntu 20.04.11 LTS (GNU/Linux 5.4.0-122-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.

seed@833fe3104c4a:~$ exit
logout
Connection closed by foreign host.
root@d670ad1dc5de:/#
```

In the above image we can see that ping and telnet functionality is working properly. This is important to make sure as after the experiment this functionality will be affected.

Task 2.A: Protecting the Router

In this task, we will set up rules to prevent outside machines from accessing the router machine, except ping.

```
root@495d9e857712:/# iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
root@495d9e857712:/# iptables -t filter -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     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 8

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@495d9e857712:/#
```

Above rule filters all the **icmp** request packets.

```
root@495d9e857712:/# iptables -A OUTPUT -p icmp --icmp-type echo-reply   -j ACCEPT
root@495d9e857712:/# iptables -t filter -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     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 8

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0            icmptype 0
root@495d9e857712:/#
```

Above rule filters all **icmp** reply packets

```
root@495d9e857712:/# iptables -P OUTPUT DROP
root@495d9e857712:/# iptables -P INPUT DROP
root@495d9e857712:/# iptables -t filter -L -n
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     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 8

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy DROP)
target     prot opt source               destination
ACCEPT     icmp --  0.0.0.0/0            0.0.0.0/0            icmptype 0
root@495d9e857712:/#
```

All the packets which does not match any of the previous rule will be dropped by the filter.

```
root@d670ad1dc5de:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
^C
--- 10.9.0.11 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5117ms

root@d670ad1dc5de:/#
```

Lets test it out by pinging a host. As you can see the connection got dropped

```
root@d670ad1dc5de:/#
root@d670ad1dc5de:/# telnet 10.9.0.11
Trying 10.9.0.11...
```

Similarly telnet connection was also not successful.

```
root@495d9e857712:/# iptables -F
root@495d9e857712:/# iptables -P OUTPUT ACCEPT
root@495d9e857712:/# iptables -P INPUT  ACCEPT
root@495d9e857712:/# iptables -t filter -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@495d9e857712:/#
```

Task 2.B: Protecting the Internal Network

For this assignment, we are required to establish firewall regulations on the router to safeguard the internal network with the IP range of 192.168.60.0/24. Our focus will be on utilizing the FORWARD chain to achieve this objective.

```
root@495d9e857712:/# iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-request -j DROP
root@495d9e857712:/# iptables -t filter -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

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

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@495d9e857712:/#
```

```
root@495d9e857712:/# iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-request -j ACCEPT
root@495d9e857712:/# iptables -A FORWARD -i eth1 -p icmp --icmp-type echo-request -j ACCEPT
```

```
root@495d9e857712:/# iptables -P FORWARD DROP
root@495d9e857712:/#
```

Similar to the previous task, even in this we are implementing rules to protect internal resources from external users. In this case an external host.
Any request coming to internal hosts from external hosts will be blocked.
Below is the image of IPTable after the rules are configured.

```
root@495d9e857712:/# iptables -L -n -v
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       icmp --  *      *       0.0.0.0/0            0.0.0.0/0            icmptype 8
    0     0 DROP       icmp --  eth0   *       0.0.0.0/0            0.0.0.0/0            icmptype 8
    0     0 ACCEPT     icmp --  eth0   *       0.0.0.0/0            0.0.0.0/0            icmptype 8
    0     0 ACCEPT     icmp --  eth1   *       0.0.0.0/0            0.0.0.0/0            icmptype 8

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
root@495d9e857712:/#
```

```
root@d670ad1dc5de:/# 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=63 time=0.161 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=63 time=0.126 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=63 time=0.129 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=63 time=0.181 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=63 time=0.149 ms
64 bytes from 10.9.0.11: icmp_seq=6 ttl=63 time=0.130 ms
^C
--- 10.9.0.11 ping statistics ---
28 packets transmitted, 28 received, 0% packet loss, time 27655ms
rtt min/avg/max/mdev = 0.122/0.155/0.225/0.028 ms
root@d670ad1dc5de:/#
```

```
root@d670ad1dc5de:/# 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 4100ms

root@d670ad1dc5de:/#
```

In the above two example we can see that internal servers can ping and connect to external servers. But same cannot be done by external hosts.

Task 2.C: Protecting Internal Servers

In this assignment, our goal is to secure TCP servers within the internal network (192.168.60.0/24). Specifically, we aim to restrict external access to telnet servers, permitting only connection to 192.168.60.5. Additionally, we need to prevent external hosts from accessing other internal servers while enabling internal hosts to communicate freely within the network, with restrictions on external server access and without utilizing connection tracking in this particular task.

```
root@495d9e857712:/# iptables -A FORWARD -i eth0 -d 192.168.60.5 -p tcp --dport 23 -j ACCEPT
root@495d9e857712:/# iptables -A FORWARD -i eth1 -s 192.168.60.5 -p tcp --sport 23 -j ACCEPT
root@495d9e857712:/# iptables -P FORWARD DROP
root@495d9e857712:/# iptables -L -n -v
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ACCEPT     tcp  --  eth0   *       0.0.0.0/0            192.168.60.5         tcp dpt:23
    0     0 ACCEPT     tcp  --  eth1   *       192.168.60.5         0.0.0.0/0            tcp spt:23

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
root@495d9e857712:/#
```

Above image shows the IP table after the Rule configuration.

In order to test the configured rule. We tried connecting to telnet server on 192.168.60.5 from host machine A. which is successfull. In the same way. net cat connection can also be established between the two.

```
root@12c556e9ef3e:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
12c556e9ef3e login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-122-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.
```

```
root@d670ad1dc5de:/#
root@d670ad1dc5de:/# nc -lu 9090
hello
trdfcgvugjkhbkjl
hgfvhjb
```

```
root@833fe3104c4a:/#
root@833fe3104c4a:/# nc -u 10.9.0.5 9090
hello
trdfcgvugjkhbkjl
hgfvhjb
```

Conclusion:

In Task 2 of the Firewall Exploration Lab, I learned to use iptables on a Linux system to set up firewall rules for packet filtering and modification. The hierarchical structure of tables, chains, and rules was explained, and examples were provided for adding, listing, and deleting rules.

In Task 2.A, I practiced protecting the router by allowing ping and setting default rules for INPUT and OUTPUT. In Task 2.B, I configured rules in the FORWARD chain to safeguard the internal network, controlling ICMP traffic between internal and external hosts. Finally,

In Task 2.C, I applied rules to protect internal TCP servers, restricting external access and controlling internal host communication. The lab emphasized the importance of careful rule management and provided insights into firewall configuration using iptables.

Task 3: Connection Tracking and Stateful Firewall

ICMP experiment: Run the following command and check the connection tracking information on the router. Describe your observation. How long can the ICMP connection state be kept?

We will send out ICMP packets to the host machine 192.168.60.5 using the ping command. Since we need it to continuously send packets. We will run it in the background by adding & character as shown in the picture below.

Then we execute the Conntrack -L command to track how many connections are active.

```
root@243fff5590bf:/# ping 192.168.60.5 &>/dev/null &
[1] 34
root@243fff5590bf:/# conntrack -L
icmp     1 29 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=17 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=17 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/#
```

From the above picture we can see that. 1 flow entry is active and it is sending ICMP packets.

Now our task is to end the ping job and observe for how long the Conntrack will hold the status even after termination of the job.

```
root@243fff5590bf:/# jobs
[1]+  Running                 ping 192.168.60.5 &> /dev/null &
root@243fff5590bf:/# conntrack -L
icmp     1 29 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=20 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=20 mark=0 use=1
icmp     1 29 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=18 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=18 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 2 flow entries have been shown.
root@243fff5590bf:/# kill %1
root@243fff5590bf:/# conntrack -L
icmp     1 23 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=20 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=20 mark=0 use=1
icmp     1 29 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=18 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=18 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 2 flow entries have been shown.
[1]+  Terminated              ping 192.168.60.5 &> /dev/null
root@243fff5590bf:/# conntrack -L
icmp     1 16 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=20 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=20 mark=0 use=1
icmp     1 29 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=18 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=18 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 2 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
icmp     1 5 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=20 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=20 mark=0 use=1
icmp     1 29 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=18 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=18 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 2 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
icmp     1 29 src=192.168.60.11 dst=192.168.60.5 type=8 code=0 id=18 src=192.168.60.5 dst=192.168.60.11 type=0 code=0 id=18 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/#
```

In the above screenshot we can see that
When we execute a jobs command to see what jobs are running. It shows a ping job running in the background.

We can see the same when we execute Conntrack as well.
Kill %1 kills the ping job.
We immediately execute Command Conntrack -L multiple times to see how long the state is stored.

As per our observation it holds the state till approximately 8 seconds.
In the last you can see the number of flow streams decreased by one.

UDP experiment: Run the following command and check the connection tracking information on the router. Describe your observation. How long can the UDP connection state be kept?

In order to send out UDP packets. Lets login to host 192.168.60.5 and initiate a Netcat connection.

```
root@ubuntu-seed-labs:~# docker exec -it 833fe3104c4a /bin/bash
root@833fe3104c4a:/# nc -lu 9090
```

Now lets login to external host A on 10.9.0.5 and connect to the Netcat server on 192.168.60.5 and try sending some UDP packets. This can be done by typing something on the terminal.

```
root@d670ad1dc5de:/# nc -u 192.168.60.5 9090
hello
hello
```

Same text will be reflected on the host machine as well.

```
root@833fe3104c4a:/# nc -lu 9090
hello
hello
```

If we execute the Conntrack -L command now, We can observe a flow stream of UDP packets.

```
root@243fff5590bf:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
udp      17 26 src=10.9.0.5 dst=192.168.60.5 sport=51348 dport=9090 [UNREPLIED] src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=51348 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/#
```

Let's end the Netcat server connection and observe Conntrack as to how long the connection state is retained. By executing the Conntrack -L command again and again, we can get a rough estimate of how long it will store the state.

```
root@243fff5590bf:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
udp      17 26 src=10.9.0.5 dst=192.168.60.5 sport=51348 dport=9090 [UNREPLIED] src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=51348 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
udp      17 22 src=10.9.0.5 dst=192.168.60.5 sport=51348 dport=9090 [UNREPLIED] src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=51348 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
udp      17 13 src=10.9.0.5 dst=192.168.60.5 sport=51348 dport=9090 [UNREPLIED] src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=51348 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@243fff5590bf:/#
```

Based on the above observation. We can say that Conntrack stores the state for roughly 8 to 10 seconds.

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?

To send TCP packets between two hosts. Lets initiate Netcat TCP server on host 192.168.60.5

```
root@833fe3104c4a:/#
root@833fe3104c4a:/# nc -l 9090
gfdgfchg
hgchgv
```

Now from host 10.9.0.5 we will connect to the TCP server to send out TCP packets.

```
root@d670ad1dc5de:/# nc 192.168.60.5 9090
gfdgfchg
hgchgv
```

As seen in the above pictures, The TCP packets have been successfully received by the Server.
Lets Observe Conntrack for the Flow stream information.

```
root@243fff5590bf:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
tcp      6 431994 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
tcp      6 431991 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
tcp      6 431989 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
tcp      6 431976 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
tcp      6 431925 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/#
```

In the above image, we can see that the status of TCP packets are ESTABLISHED. Meaning the TCP handshake was successful and the connection is formed and retained for data transfer.

To check how long the state will be held by Conntrack, we have to end the TCP connection from both the sides and Observe the Conntrack Results.

```
root@833fe3104c4a:/# nc -l 9090
gfdgfchg
hgchgv
^C
root@833fe3104c4a:/#
```

```
root@d670ad1dc5de:/# nc 192.168.60.5 9090
gfdgfchg
hgchgv
^C
root@d670ad1dc5de:/#
root@d670ad1dc5de:/#
```

```
root@243fff5590bf:/# conntrack -L
tcp      6 113 TIME_WAIT src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/#
root@243fff5590bf:/# conntrack -L
tcp      6 106 TIME_WAIT src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
tcp      6 81 TIME_WAIT src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
tcp      6 42 TIME_WAIT src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
tcp      6 10 TIME_WAIT src=10.9.0.5 dst=192.168.60.5 sport=40472 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=40472 [ASSURED] mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@243fff5590bf:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@243fff5590bf:/#
```

Once we ended the TCP connection. We can see the status of TCP packets now changed to TIME_WAIT from ESTABLISHED. It will wait for 113s seconds before discarding the state. As time goes we can observe this timer count getting decreased and finally becoming Zero. that is when the connection state is discarded from Conntrack.

5.2 Task 3.B: Setting Up a Stateful Firewall

Now we are ready to set up firewall rules based on connections. In the following example, the "-m conntrack" option indicates that we are using the conntrack module,

```
root@243fff5590bf:/#
root@243fff5590bf:/# iptables -A FORWARD -p tcp -i eth0 -d 192.168.60.5 --dport 23 --syn -m conntrack --ctstate NEW -j ACCEPT
root@243fff5590bf:/# iptables -A FORWARD -i eth1 -p tcp --syn -m conntrack --ctstate NEW -j ACCEPT
root@243fff5590bf:/# iptables -A FORWARD -p tcp -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
root@243fff5590bf:/# iptables -A FORWARD -p tcp -j DROP
root@243fff5590bf:/# iptables -P FORWARD ACCEPT
root@243fff5590bf:/#
```

The First rule allows TCP packets belonging to an existing connection to pass through But it does not cover the SYN packets, which do not belong to any established connection. Without it, we will not be able to create a connection in the first place.

Therefore second rule accepts incoming SYN packet

Finally, we will set the default policy on FORWARD to drop everything. This way, if a TCP packet is not accepted by the two rules above, they will be dropped.

Any packet other than TCP will not be affected by this filtering. This is ensured by the last rule.
Last rule also enables internal hosts to visit any external server .

```
root@243fff5590bf:/# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
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
ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0            ctstate RELATED,ESTABLISHED
DROP       tcp  --  0.0.0.0/0            0.0.0.0/0

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

In order to test the Firewall we just created, let's Initialize the telnet connection. This will inturn use TCP to connect and will establish a TCP connection.

```
root@833fe3104c4a:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
d670ad1dc5de login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-122-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.

seed@d670ad1dc5de:~$
```

```
root@d670ad1dc5de:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
833fe3104c4a login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-122-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.

seed@833fe3104c4a:~$
```

We can see that the connection was successful between host 10.9.0.5 and 192.168.60.5
This tests the Happy flow of the Firewall and makes sure it is not stopping any legit connections.

Lets try the negative scenario where the firewall should not let the External host A to connect to any other internal host other than 192.168.60.5

```
root@d670ad1dc5de:/#
root@d670ad1dc5de:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
root@d670ad1dc5de:/# telnet 192.168.60.7
Trying 192.168.60.7...
^C
root@d670ad1dc5de:/#
```

In the above image, when we tried to initiate a connection from Host A to internal server 192.168.60.6 and 192.168.60.7 Firewall is stopping the connection which is an expected behavior.

Now let's test the rule which enables any internal server to connect to external server.

```
root@d670ad1dc5de:/#
root@d670ad1dc5de:/# nc -lu 9090
hello
trdfcgvugjkhbkjl
hgfvhjb
```

A Netcat TCP server is initiated in the External server Hoest A.

```
root@833fe3104c4a:/#
root@833fe3104c4a:/# nc -u 10.9.0.5 9090
hello
trdfcgvugjkhbkjl
hgfvhjb
```

In the above image, From internal host 1. We are connecting to the Netcat server hosted on external host A. The connection was established successfully and all the input given on the client is reflected on the server. This completes testing of all the scenarios of the Firewall.

In the end. Lets go ahead and clear all the rules.

```
root@243fff5590bf:/# iptables -F
root@243fff5590bf:/# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@243fff5590bf:/#
```

Conclusion:

I gained knowledge of stateful firewalls and the significance of connection tracking to take into account the context of packets, particularly in TCP connections, from Task 3 of the Firewall Exploration Lab. After the introduction of the kernel's conntrack mechanism, I experimented to learn about connection tracking for the TCP, UDP, and ICMP protocols. I learned how to configure a stateful firewall using iptables in Task 3.B, concentrating on rules based on established connections and SYN packets. The assignment also made it necessary to contrast using the connection monitoring system with other approaches, emphasizing the benefits and drawbacks of each strategy. All things considered, the lab helped me better understand stateful firewall setups and how they affect network security.

Task 4: Limiting Network Traffic

In this task, we will use **limit** module to limit how many packets from 10.9.0.5 are allowed to get into the internal network.

```
root@243fff5590bf:/# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@243fff5590bf:/# iptables -A FORWARD -s 10.9.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT
root@243fff5590bf:/#
```

The above command restricts traffic at IP 10.9.0.5 and only allows 10 packets per minute. Our task is to find out if this rule is enough on its own to restrict the packet flow.

In order to test that. Lets ping 192.168.60.5 from 10.0.0.5

```
root@d670ad1dc5de:/# 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=0.161 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.126 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.129 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.181 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.149 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=0.130 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.173 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=0.122 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=0.171 ms
64 bytes from 192.168.60.5: icmp_seq=10 ttl=63 time=0.146 ms
64 bytes from 192.168.60.5: icmp_seq=11 ttl=63 time=0.141 ms
64 bytes from 192.168.60.5: icmp_seq=12 ttl=63 time=0.215 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.151 ms
64 bytes from 192.168.60.5: icmp_seq=14 ttl=63 time=0.189 ms
64 bytes from 192.168.60.5: icmp_seq=15 ttl=63 time=0.149 ms
64 bytes from 192.168.60.5: icmp_seq=16 ttl=63 time=0.191 ms
64 bytes from 192.168.60.5: icmp_seq=17 ttl=63 time=0.156 ms
64 bytes from 192.168.60.5: icmp_seq=18 ttl=63 time=0.214 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=63 time=0.144 ms
64 bytes from 192.168.60.5: icmp_seq=20 ttl=63 time=0.123 ms
64 bytes from 192.168.60.5: icmp_seq=21 ttl=63 time=0.225 ms
64 bytes from 192.168.60.5: icmp_seq=22 ttl=63 time=0.134 ms
64 bytes from 192.168.60.5: icmp_seq=23 ttl=63 time=0.140 ms
64 bytes from 192.168.60.5: icmp_seq=24 ttl=63 time=0.146 ms
64 bytes from 192.168.60.5: icmp_seq=25 ttl=63 time=0.140 ms
64 bytes from 192.168.60.5: icmp_seq=26 ttl=63 time=0.147 ms
64 bytes from 192.168.60.5: icmp_seq=27 ttl=63 time=0.134 ms
64 bytes from 192.168.60.5: icmp_seq=28 ttl=63 time=0.122 ms
^C
--- 192.168.60.5 ping statistics ---
28 packets transmitted, 28 received, 0% packet loss, time 27655ms
rtt min/avg/max/mdev = 0.122/0.155/0.225/0.028 ms
root@d670ad1dc5de:/#
```

From what we observe from the above image. Every packet is going through. This is because whichever packets did not pass the filter is still going through because we do not have a second rule to drop such packets which did not pass the first filter.

Lets add a second filter to drop packets.

```
root@243fff5590bf:/# iptables -A FORWARD -s 10.9.0.5 -j DROP
root@243fff5590bf:/# iptables -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
root@243fff5590bf:/#
```

After this lets test it by pinging 192.168.60.5 from 10.0.0.5

```
root@d670ad1dc5de:/# 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=0.184 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.161 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.154 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.143 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.160 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.153 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.142 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=63 time=0.141 ms
64 bytes from 192.168.60.5: icmp_seq=25 ttl=63 time=0.139 ms
64 bytes from 192.168.60.5: icmp_seq=31 ttl=63 time=0.171 ms
64 bytes from 192.168.60.5: icmp_seq=37 ttl=63 time=0.162 ms
64 bytes from 192.168.60.5: icmp_seq=43 ttl=63 time=0.157 ms
64 bytes from 192.168.60.5: icmp_seq=48 ttl=63 time=0.169 ms
^C
--- 192.168.60.5 ping statistics ---
49 packets transmitted, 13 received, 73.4694% packet loss, time 49148ms
rtt min/avg/max/mdev = 0.139/0.156/0.184/0.012 ms
root@d670ad1dc5de:/#
```

This time we can observe 73.4694 % packet loss. This is because all the packets which didn't go through the first rule are dropped in the second rule. Now the filter is successfully limiting the packet count.

Conclusion: I discovered how to use the limit module of iptables to restrict network traffic in Task 4 of the Firewall Exploration Lab. I was able to observe the results of restricting how many packets from 10.9.0.5 may reach the internal network by executing particular commands on the router. In order to compare the experiment's outcomes with and without the second rule, I had to decide whether the second rule was necessary and explain why it was relevant.

Task 5: Load Balancing

In this task, we will use IPtables to load balance three UDP servers running in the internal network. Let's first start the server on each of the hosts: 192.168.60.5, 192.168.60.6, and 192.168.60.7

Below image shows different host we will be using for the experiment.

```
root@ubuntu-seed-labs:~# docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED             STATUS          PORTS     NAMES
495d9e857712   handsonsecurity/seed-ubuntu:large    "bash -c ' ip route …"   About an hour ago   Up 51 minutes             hostA-10.9.0.5
12c556e9ef3e   handsonsecurity/seed-ubuntu:large    "bash -c ' ip route …"   About an hour ago   Up 51 minutes             host1-192.168.60.5
a9a97f9e3a97   handsonsecurity/seed-ubuntu:large    "bash -c ' ip route …"   About an hour ago   Up 51 minutes             host2-192.168.60.6
a3515a21cead   seed-router-image                    "bash -c ' ip route …"   About an hour ago   Up 51 minutes             seed-router
9d3dc846cc58   handsonsecurity/seed-ubuntu:large    "bash -c ' ip route …"   About an hour ago   Up 51 minutes             host3-192.168.60.7
```

Now Let's write IP table rules to distribute the traffic equally among all the 3 hosts.

```
root@a3515a21cead:/#
root@a3515a21cead:/# 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
root@a3515a21cead:/# 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
root@a3515a21cead:/# 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
root@a3515a21cead:/# iptables -L -n
```

Below image shows the NAT table after configuring the rules. All the rules are set at the pre-routing stage.

```
root@a3515a21cead:/# 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

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER_OUTPUT  all  --  0.0.0.0/0            127.0.0.11

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER_POSTROUTING  all  --  0.0.0.0/0            127.0.0.11

Chain DOCKER_OUTPUT (1 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            127.0.0.11           tcp dpt:53 to:127.0.0.11:46245
DNAT       udp  --  0.0.0.0/0            127.0.0.11           udp dpt:53 to:127.0.0.11:42343

Chain DOCKER_POSTROUTING (1 references)
target     prot opt source               destination
SNAT       tcp  --  127.0.0.11           0.0.0.0/0            tcp spt:46245 to::53
SNAT       udp  --  127.0.0.11           0.0.0.0/0            udp spt:42343 to::53
root@a3515a21cead:/#
```

Lets initiate the Netcat server in each of the hosts.

```
root@ubuntu-seed-labs:~# docker exec -it 12c556e9ef3e /bin/bash
root@12c556e9ef3e:/# nc -luk 8080
```

```
root@ubuntu-seed-labs:~# docker exec -it a9a97f9e3a97 /bin/bash
root@a9a97f9e3a97 :/# nc -luk 8080
```

```
root@ubuntu-seed-labs:~# docker exec -it 9d3dc846cc58 /bin/bash
root@9d3dc846cc58:/# nc -luk 8080
```

Lets send out packets from host machine A which is an external server.

```
root@ubuntu-seed-labs:~# docker exec -it 495d9e857712 /bin/bash
root@495d9e857712:/#
root@495d9e857712:/# echo hello | nc -u 10.9.0.11 8080
^C
root@495d9e857712:/# echo hello1 hello2 hello3 | nc -u 10.9.0.11 8080
^C
root@495d9e857712:/# echo hello1 hello2 hello3 1 | nc -u 10.9.0.11 8080
^C
root@495d9e857712:/# echo hello1 hello2 hello3 2 | nc -u 10.9.0.11 8080
^C
```

In the below image we can see that all the traffic coming from Host A has reached respective hosts via router and is equally distributed.

```
root@ubuntu-seed-labs:~# docker exec -it 12c556e9ef3e /bin/bash
root@12c556e9ef3e:/# nc -luk 8080
hello
```

```
root@ubuntu-seed-labs:~# docker exec -it 9d3dc846cc58 /bin/bash
root@9d3dc846cc58:/# nc -luk 8080
hello1 hello2 hello3 2
```

```
root@ubuntu-seed-labs:~# docker exec -it a9a97f9e3a97 /bin/bash
root@a9a97f9e3a97:/# nc -luk 8080
hello1 hello2 hello3 1
```

```
root@ubuntu-seed-labs:~# docker exec -it 12c556e9ef3e /bin/bash
root@12c556e9ef3e:/# nc -luk 8080
hello
hello1 hello2 hello3
```

After the experiment. We shall clear the IPTables and restore it back to default state before next experiment

```
root@a3515a21cead:/# iptables -t nat -D PREROUTING 1
root@a3515a21cead:/# iptables -t nat -D PREROUTING 2
root@a3515a21cead:/# iptables -t nat -D PREROUTING 1
root@a3515a21cead:/# iptables -t nat -L -n --line-numbers
Chain PREROUTING (policy ACCEPT)
num  target     prot opt source               destination

Chain INPUT (policy ACCEPT)
num  target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
num  target     prot opt source               destination
1    DOCKER_OUTPUT  all  --  0.0.0.0/0            127.0.0.11
```

Random mode :

Let's use a different mode to achieve load balancing. In the below image we are setting up IPTable rules such that for every 3 packets, every packet has 33% chance of going to the First host. Once the rule is surpassed, the remaining 2 packets have a 50% chance of reaching host 2. And remaining one packet will definitely reach host 3

```
root@a3515a21cead:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.333 -j DNAT --to-destination 192.168.60.5:8080
root@a3515a21cead:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.5 -j DNAT --to-destination 192.168.60.6:8080
root@a3515a21cead:/# iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 1 -j DNAT --to-destination 192.168.60.7:8080
root@a3515a21cead:/# iptables -t nat -L -n --line-numbers
Chain PREROUTING (policy ACCEPT)
num  target     prot opt source               destination
1    DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 0.33300000010 to:192.168.60.5:8080
2    DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 0.50000000000 to:192.168.60.6:8080
3    DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:8080 statistic mode random probability 1.00000000000 to:192.168.60.7:8080

Chain INPUT (policy ACCEPT)
num  target     prot opt source               destination
```

We have initiated the Net cat server in all 3 hosts

```
root@ubuntu-seed-labs:~# docker exec -it 12c556e9ef3e /bin/bash
root@12c556e9ef3e:/# nc -luk 8080
```

```
root@ubuntu-seed-labs:~# docker exec -it a9a97f9e3a97 /bin/bash
root@a9a97f9e3a97 :/# nc -luk 8080
```

```
root@ubuntu-seed-labs:~# docker exec -it 9d3dc846cc58 /bin/bash
root@9d3dc846cc58:/# nc -luk 8080
```

We login to Host A external machine and try connecting to router via Netcat so router can load balance the traffic based on rules.

```
root@ubuntu-seed-labs:~# docker exec -it 495d9e857712 /bin/bash
root@495d9e857712:/#
root@495d9e857712:/# echo abcd | nc -u 10.9.0.11 8080
^C
root@495d9e857712:/# echo efg | nc -u 10.9.0.11 8080
^C
root@495d9e857712:/# echo sdgfdgb | nc -u 10.9.0.11 8080
^C
root@495d9e857712:/# echo egshftgj | nc -u 10.9.0.11 8080
^C
```

As we can see from the images below. All the traffic is equally distributed.

```
root@ubuntu-seed-labs:~# docker exec -it 12c556e9ef3e /bin/bash
root@12c556e9ef3e:/# nc -luk 8080
abcd
egshftgj
```

```
root@ubuntu-seed-labs:~# docker exec -it a9a97f9e3a97 /bin/bash
root@a9a97f9e3a97:/# nc -luk 8080
efg
```

```
root@ubuntu-seed-labs:~# docker exec -it 9d3dc846cc58 /bin/bash
root@9d3dc846cc58:/# nc -luk 8080
sdgfdgb
```

Conclusion for Task 5:
Load balancing using the nth mode in a round-robin approach involves selecting every third packet in the sequence. In this mode, the specific packet labeled as packet 0 undergoes modification by changing its destination IP address and port to 192.168.60.5:8080. This modification ensures that one out of every three UDP packets directed to the router's 8080 port is redirected to 192.168.60.5.

Task 6: Write-Up:

In the course of exploring Firewalls through Tasks 1 to 5 in the lab, several key learnings emerged at a high level

Task 1: Constructing a Basic Firewall
I learned how to use Netfilter, more especially iptables, to create a packet-filtering firewall inside the Linux kernel. This job demonstrated the ability of Loadable Kernel Modules to dynamically extend kernel functionality, with a focus on container configurations and virtual IP addresses. The comprehension served as the cornerstone for other work with firewalls.

Task 2: Stateless Firewall Rules
This job extended the functionality of firewall rules by using iptables, a program that manipulates packets in addition to filtering them. The explanation of the hierarchical structure of tables, chains, and rules illustrated how the rules are arranged for various objectives. Using iptables commands, I learnt how to add, list, and remove rules with an emphasis on safeguarding the router and internal network.

Task 3: Connection Tracking and Stateful Firewall
Using the conntrack technique, Task 3 introduced the idea of stateful firewalls in order to better understand the limits of stateless firewalls. Experiments including connection tracking for TCP, UDP, and ICMP protocols shown how crucial context is for firewall rule formulation. The concept of improving security through connection tracking was further enhanced by setting up a stateful firewall with rules based on connections.

Task 4: Restricting Network Traffic
In order to limit the quantity of packets that travel through the firewall, the job introduced the iptables limit module. Through testing with packet limits from particular sources, I learned how to manage network traffic. Observing how the limit module affects things and how certain rules are needed gave us useful information for optimizing firewall configurations.

Task 5: Load Balancing

This task examined the use of iptables in load balancing, demonstrating the tool's adaptability. Using the statistic module, the experiment entailed dividing up UDP traffic among several servers. The exposure to load balancing illustrated the broader possibilities of iptables beyond conventional firewall features, even though the lab acknowledged its inability to cover all iptables uses.

intriguing Observations and Difficulties:
Task 3's switch from stateless to stateful firewall settings made an intriguing remark about the importance of context in packet filtering. The connection tracking research, particularly for TCP, UDP, and ICMP, offered useful understanding of the dynamic nature of network communication.

Comprehending the complexities of iptables commands and guaranteeing proper rule implementation were among the challenges encountered in the lab. In addition, careful analysis of the hierarchical structure and possible effects on network traffic was necessary when designing rules for certain tasks.
To sum up, the lab experience gave me a comprehensive grasp of firewall concepts, covering everything from simple packet filtering to complex stateful settings, traffic limitation, and even investigating supplementary uses like load balancing. The practical information gained from working through rule configuration issues and the hands-on aspect of the exercises improved understanding of network security concepts.